2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing

Fast PageRank Computation on a GPU Cluster

Arnon Rungsawang and Bundit Manaskasemsak
Massive Information & Knowledge Engineering
Department of Computer Engineering, Faculty of Engineering
Kasetsart University, Bangkok 10900, Thailand.
{arnon,un}@mikelab.net

Abstract

We investigate the use of graphics processing units (GPUs) in accelerating PageRank computation. We first introduce a compact web graph representation which requires much less memory allocation than a well-known compressed sparse row format. The web graph is then simply partition into smaller chunks to fit the GPUs' device memory. We propose a fast PageRank algorithm to run on the GPU cluster. The design of algorithm is general and does not constrain on any large web graph fitting to the limited size of device memory. In the experiments, we test our PageRank algorithm on a small GPU cluster, using a set of real web data. We compare the parallel PageRank computation utilizing GPUs with CPUs. The results show that the proposed PageRank computation on GPUs gives promising result.

1 Introduction

Due to the unceasing growth of the Internet, search engines nowadays still play an important role in discovering useful information from the World Wide Web. For a given user query, there are many resulting pages to be treated as an answer. Good page ranking mechanism is thus required to determine the authoritativeness of the web search result. PageRank has been known as the most effective ranking algorithm since the advent of Google [4]. It analyzes the link structure, i.e., the web graph, to obtain the important values of web pages. Since the link structure data is usually huge owing to the web's size, PageRank computation is still a challenging problem for today search engine industry.

Many researchers have proposed methods in accelerating PageRank computation. For example, in algorithmic way, Kamvar et al. [13, 14] have proposed an extrapolation method that accelerates the convergence of the power method by periodically subtracting off estimates of the non-principal eigenvectors from the current iteration run, and an adaptive method that ignores the re-computation of the already converged PageRank of the web pages. In a high-performance computing way, Manaskasemsak et al. [16, 17] have proposed to use cluster computing technology and a mixed MPI-thread programming scheme in parallel PageRank computation. Plimpton and Devine [19] provide an enable MapReduce functionality on a message-passing library that facilitates the writing of the PagaRank code on the Hadoop computing platform [1].

Recently, a cluster of manycore graphics processing units (GPUs) has been introduced as a new general purposed high-performance computing paradigm. Researchers then treat the PageRank computation as a kind of iterative sparse matrix-vector multiplication (SpMV) and use GPUs as massively parallel computing devices to solve the problem [5, 23]. In this paper, we also explore the parallel PageRank computation on a GPU cluster. The web graph is represented as three compact vectors, which are divided, distributed to each machine, and loaded as needed into the memory of each GPU device during the computation. For the experiments with real web data, the PageRank computation on a small GPU cluster can be achieved faster than those computation on the conventional CPU platform.

The rest of this paper is organized as follows. Section 2 provides an overview of PageRank algorithm and the general purpose computing on GPUs. Section 3 introduces data representation and proposes parallel PageRank computational algorithms on CPUs as well as GPUs. Section 4 reports the performance results. Section 5 discusses related work, and Section 6 concludes the paper.

2 Background

2.1 PageRank Algorithm

The basic concept of PageRank [18] is to estimate importance of web pages based on hyperlink structure analysis. A link from page u to page v implies that the author of u implicitly confers some importance to v. Furthermore, a

978-0-7695-4633-9/12 $26.00 © 2012 IEEE
DOI 10.1109/PDP.2012.78

IEEE computer society

page mostly referred by other important pages is also more important.

For a set of n web pages, we let $\vec{r}$ be an n-dimensional column vector representing the PageRank scores of those pages, which $||\vec{r}||_1 = 1$. The relationship over hyperlink structure (i.e., web graph) can be represented as a sparse matrix A, which each entry a_{ij} is defined by:

$$a_{ij} = \begin{cases} \frac{1}{o_j} & \text{if page } j \text{ has a link to page } i, \\ 0 & \text{otherwise,} \end{cases} \quad (1)$$

where o_j is the number of pages that page j points out, called out-degree. The PageRank computation is equivalent to solving the principal eigenvector $\vec{r}$ of the problem $\vec{r} = \lambda A\vec{r}$ with the dominant eigenvalue $\lambda = 1$ in an eigensystem.

Since the matrix A is generally sparse, the computation may encounter problems against correctness and convergence. However, these problems can be solved by adding artificial links with uniform probability from all to all pages. Let E be an $(n \times n)$-square matrix which all entries are $\frac{1}{n}$. Then, the modification matrix A' is defined as:

$$A' = \alpha A + (1 - \alpha)E, \quad (2)$$

where the coefficient α determines the weights of actual and artificial links. In practice, it is usually set to 0.85.

Note that these artificial links introduced by E have no impact on the sparsity of the matrix used in the computation. In particular, the matrix-vector multiplication $\vec{r} = \lambda A'\vec{r}$ can be implemented efficiently using the power method as shown in Algorithm 1. In the algorithm, the rank vector is repeatedly calculated until its residual error is less than a pre-defined threshold value ϵ.

Algorithm 1 : The power method of PageRank.

function $sequentialPR(A, n, \epsilon)$

1: $\vec{r}^{(0)} = \left[\frac{1}{n}\right]_{n\times 1}$, $\alpha = 0.85$, $t = 0$
2: **repeat**
3: $\quad \vec{r}^{(t+1)} = \alpha A\vec{r}^{(t)}$
4: $\quad c = ||\vec{r}^{(t+1)}||_1 - ||\vec{r}^{(t)}||_1$
5: $\quad \vec{r}^{(t+1)} = \vec{r}^{(t+1)} + c\left[\frac{1}{n}\right]_{n\times 1}$
6: $\quad t = t + 1$
7: **until** $||\vec{r}^{(t+1)} - \vec{r}^{(t)}||_1 < \epsilon$
8: **return** $\vec{r}^{(t+1)}$

2.2 General Purpose Computing on GPUs

GPUs differ from a general purpose CPU in their design for the single instruction multiple data (SIMD) paradigm. Due to the inherent parallelism, the GPUs have been previously used for accelerating graphics rendering pipelines. However, today's GPUs have been used as a hardware accelerator for various non-graphics applications since NVIDIA introduced compute unified device architecture (CUDA), such as matrix multiplication [3, 8, 15, 24], databases [9, 10, 11, 12], and scientific computation. The CUDA provides an C/C++-like interface for developers to program a GPU device for general purpose computation (GPGPU).

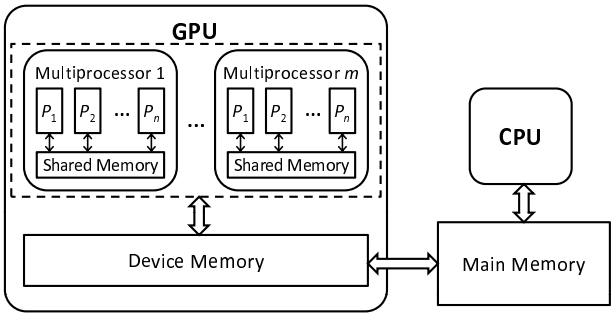


Figure 1. GPU as a manycore architecture.

Figure 1 shows the model of GPU as a manycore architecture. The GPU device consists of a set of streaming multiprocessors (SMs) which each one equips with a cluster of streaming processors (SPs). The CUDA exposes the massive thread parallelism and the fast interprocessor communication via the shared memory. The GPU also has an amount of device memory with high bandwidth and high access latency, which is visible to all threads on the device. In a CUDA program, developers can specify the dimensions of grid and block in order to identify a number of threads mapping addresses of the device memory. These threads concurrently execute instructions of a *kernel*, i.e., the parallel region of code. Since the device memory is separated from the host's memory (i.e., main memory), it needs to first transfer data from host to device before the execution.

3 Parallel PageRank on GPUs

To compute the PageRank vector for huge web graphs (i.e., expect to be a billion links or more) would require massive computing power as well as enormous amounts of memory. A parallelized version of PageRank is consequently investigated to end our goal. In this section, we first introduce a representation of web graph used in the computation. Then, we detail our parallel implementation exploring on both CPU and GPU clusters.

3.1 Web Graph Representation

Suppose that each web page is first assigned by an ordinal number, we let n and l be the number of pages and links

$$A = \begin{pmatrix} 1/2 & 0 & 1/3 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1/3 & 1/2 \\ 1/2 & 0 & 1/3 & 1/2 \end{pmatrix}$$

$$\begin{aligned} \vec{w}_g\,[8] &= \{\ 0\ 2\ 1\ 2\ 3\ 0\ 2\ 3\ \} \\ \vec{w}_i\,[5] &= \{\ 0\ 2\ 2\ 5\ 8\ \} \\ \vec{w}_o\,[4] &= \{\ 2\ 1\ 3\ 2\ \} \end{aligned}$$

Figure 2. Example of a compact web graph representation.

On P_0: $b = 0, e = 1$

$$\begin{aligned} \vec{w}_g\,[2] &= \{\ 0\ 2\ \} \\ \vec{w}_i\,[3] &= \{\ 0\ 2\ 2\ \} \\ \vec{w}_o\,[4] &= \{\ 2\ 1\ 3\ 2\ \} \end{aligned}$$

On P_1: $b = 2, e = 3$

$$\begin{aligned} \vec{w}_g\,[6] &= \{\ 1\ 2\ 3\ 0\ 2\ 3\ \} \\ \vec{w}_i\,[3] &= \{\ 0\ 3\ 6\ \} \\ \vec{w}_o\,[4] &= \{\ 2\ 1\ 3\ 2\ \} \end{aligned}$$

Figure 3. Example of a partitioned web graph on two processors.

contained in the web graph, respectively. Rather representing the web graph as a sparse matrix described in Equation (1), we here propose the compressed sparse web graph (CSWG) format using the following three vectors. First, the vector $\vec{w}_g$ with l elements is used to store concatenated lists of source pages that link to each respective target page. Second, the vector $\vec{w}_i$, having $n+1$ elements, records the cumulative in-degrees used to indicate the start and end positions of the list in $\vec{w}_g$. Last, the vector $\vec{w}_o$, having n elements, stores the out-degrees. An example of a web graph and three representative vectors are given in Figure 2.

In the example, the matrix A contains eight non-zero entries which suppose to be the links from source-IDs {0, 2}, {1, 2, 3}, and {0, 2, 3} to target-ID 0, 2, and 3, respectively. Thus, those source-IDs are stored in $\vec{w}_g$, and the indices in $\vec{w}_g$ given by $\vec{w}_i$ are 0, 2, 2, 5, 8. Note that we repeat the indices of 2 twice since the target-ID 1 has zero in-degree and we indicate the end of index by 8. Finally, $\vec{w}_o$ simply stores out-degrees of each target.

Comparing to the well-known compressed sparse row (CSR) representation [2] which is generally used in algebraic problems, the proposed CSWG introduced for this specific purpose requires much less memory allocation. Obviously, CSWG requires $2n + l$ entries compared to CSR which needs $n + 2l$ entries, when l is normally much larger than n for the web graph matrices.

One simple way, though the problem of unbalanced computational workload occurs, is to partition the web graph by the number of processors, and let each processor to compute on the data of each assigned partition. For this, we simply divide the vectors $\vec{w}_g$ and $\vec{w}_i$ into several chunks with equal number of target pages (i.e., number of rows in A), except $\vec{w}_o$. Figure 3 shows an example of two partitions of the sample web graph depicted in Figure 2. The variables b and e denote the beginning- and ending-ID of target pages in each partition. Note that the sizes of $\vec{w}_g$ and $\vec{w}_i$ are determined by the number of links and the number of target pages (i.e., $e - b + 2$) assigned to that partition, respectively.

3.2 Implementation on a CPU Cluster

We modify the power method of PageRank, as shown in Algorithm 1, to obtain a parallelized version exploring on the CPU cluster as illustrated in Algorithm 2. Using MPI [22], this algorithm is run on each processor P_i with separate inputs $\vec{w}_g$, $\vec{w}_i$, and $\vec{w}_o$, determined by b and e, respectively.

Algorithm 2 : Parallel PageRank on each processor P_i exploring on the CPU cluster.

function $parallelPR_{CPU}(\vec{w}_g, \vec{w}_i, \vec{w}_o, n, b, e, \epsilon)$

1: $s = e - b + 1$
2: $\vec{r}_{host}^{(0)} = \left[\frac{1}{n}\right]_{n\times 1}$, $\alpha = 0.85$, $t = 0$
3: **repeat**
4: $\quad \vec{d}_{host} = [0]_{s\times 1}$
5: $\quad$ **for** $j = b$ **to** e **do**
6: $\quad\quad pos_{begin} = \vec{w}_i[j-b], pos_{end} = \vec{w}_i[j-b+1] - 1$
7: $\quad\quad$ **for** $k = pos_{begin}$ **to** pos_{end} **do**
8: $\quad\quad\quad p = \vec{w}_g[k]$
9: $\quad\quad\quad \vec{d}_{host}[j-b] = \vec{d}_{host}[j-b] + \alpha\left(\frac{\vec{r}_{host}[p]}{\vec{w}_o[p]}\right)$
10: $\quad\quad$ **end for**
11: $\quad$ **end for**
12: $\quad \vec{r}_{host}^{(t+1)} \leftarrow$ Call MPI to broadcast $\vec{d}_{host}$
13: $\quad c = ||\vec{r}_{host}^{(t+1)}||_1 - ||\vec{r}_{host}^{(t)}||_1$
14: $\quad \vec{r}_{host}^{(t+1)} = \vec{r}_{host}^{(t+1)} + c\left[\frac{1}{n}\right]_{n\times 1}$
15: $\quad t = t + 1$
16: **until** $||\vec{r}_{host}^{(t+1)} - \vec{r}_{host}^{(t)}||_1 < \epsilon$
17: **return** $\vec{r}_{host}^{(t+1)}$

In the Algorithm 2, the outer loop, i.e., line 3–16, controls iterative computation by inspecting the convergence of the rank vector. The middle loop, i.e., line 5–11, specifies each target page. For the innermost loop between line 7 and 10, all source pages corresponding to that target are enumerated and subsequently cumulated their scores divided

by out-degrees. In line 12, the portion of resulting rank vector of processor P_i is afterward synchronized with all other processors to obtain the whole rank vector $\vec{r}_{host}$. For the rest of statements, i.e., line 13–14, small probabilities of complete links are added to ensure rank convergence.

3.3 Implementation on a GPU Cluster

To utilize the GPU computing, we further illustrate an improvement of the previous parallel PageRank version as depicted in Algorithm 3.

As we can see in the main function $parallelPR_{GPU}()$ of the Algorithm 3, there are four common steps: (1) First allocating memory on the device (line 3), then (2) copying needed data in computation from host memory to device memory (line 4, 6, and 10), (3) launching the GPU's kernel function to process the data on the device (line 11), and finally (4) moving results back from device to host (line 15). More precisely, the outer loop between line 5 and 20 controls iterative computation. For the inner loop between line 8 and 14, if the assigned partitioned web graph is larger than the size of pre-allocated device memory (i.e., size of l elements of $\vec{g}_{dev}$, seeing line 3) then it will be divided again into smaller chunks. Each chunk is then loaded into the device memory before launching the kernel. Afterward, in line 15, the portion of resulting rank vector of processor P_i is copied back from the device to host. Similarly, for the rest of statements, rank vector synchronization and adding the probabilities of complete links process are yielded in line 16 and 17–18, respectively.

For the kernel function, $kernelPR()$, a concurrent thread will be assigned an identification, tid, and be contributed to compute a score of one target page. However, only a number of threads are assigned to execute. The loop in line 4–7 enumerates all source pages by cumulating their scores divided by out-degrees.

Algorithm 3 : Parallel PageRank on each processor P_i exploring on the GPU cluster.

function $parallelPR_{GPU}(\vec{w}_g, \vec{w}_i, \vec{w}_o, n, b, e, \epsilon)$

1: $s = e - b + 1$
2: $\vec{r}^{(0)}_{host} = \left[\frac{1}{n}\right]_{n\times 1}$, $\vec{d}_{host} = [0]_{s\times 1}$, $\alpha = 0.85$, $t = 0$
3: Allocate $\vec{r}_{dev}$ of n elements on device
Allocate $\vec{d}_{dev}$ of s elements on device
Allocate $\vec{o}_{dev}$ of n elements on device
Allocate $\vec{i}_{dev}$ of $s+1$ elements on device
Allocate $\vec{g}_{dev}$ as large l elements as possible on device
4: Copy $\vec{w}_o$ from host to $\vec{o}_{dev}$ on device
Copy $\vec{w}_i$ from host to $\vec{i}_{dev}$ on device
5: **repeat**
6: Copy $\vec{r}^{(t)}_{host}$ from host to $\vec{r}_{dev}$ on device
Set all elements of $\vec{d}_{dev}$ on device to 0
7: $b' = 0$
8: **for** $e' = 0$ **to** s **do**
9: **if** $(e' == s)$ OR $(\vec{w}_i[e'+1] - \vec{w}_i[b'] > l)$ **then**
10: Copy $\vec{w}_g$ at position $\vec{w}_i[b']$ to $\vec{w}_i[e'] - 1$ from host to fit to $\vec{g}_{dev}$ on device
11: Launch $kernelPR(...)$ with parameters: $\vec{r}_{dev}$, $\vec{d}_{dev}, \vec{o}_{dev}, \vec{i}_{dev}, \vec{g}_{dev}, b', e', \alpha$
12: $b' = e'$
13: **end if**
14: **end for**
15: Copy $\vec{d}_{dev}$ from device back to $\vec{d}_{host}$ on host
16: $\vec{r}^{(t+1)}_{host} \leftarrow$ Call MPI to broadcast $\vec{d}_{host}$
17: $c = ||\vec{r}^{(t+1)}_{host}||_1 - ||\vec{r}^{(t)}_{host}||_1$
18: $\vec{r}^{(t+1)}_{host} = \vec{r}^{(t+1)}_{host} + c\left[\frac{1}{n}\right]_{n\times 1}$
19: $t = t + 1$
20: **until** $||\vec{r}^{(t+1)}_{host} - \vec{r}^{(t)}_{host}||_1 < \epsilon$
21: **return** $\vec{r}^{(t+1)}_{host}$

/* launching on device only */
function $kernelPR(\vec{r}_{dev}, \vec{d}_{dev}, \vec{o}_{dev}, \vec{i}_{dev}, \vec{g}_{dev}, b', e', \alpha)$

1: $tid = (blockIdx.x * blockDim.x) + threadIdx.x$
2: **if** $tid < e' - b'$ **then**
3: $pos_{base} = \vec{i}_{dev}[b']$, $pos_{begin} = \vec{i}_{dev}[tid + b']$, $pos_{end} = \vec{i}_{dev}[tid + b' + 1] - 1$
4: **for** $k = pos_{begin}$ **to** pos_{end} **do**
5: $p = \vec{g}_{dev}[k - pos_{base}]$
6: $\vec{d}_{dev}[tid + b'] = \vec{d}_{dev}[tid + b'] + \alpha\left(\frac{\vec{r}_{dev}[p]}{\vec{o}_{dev}[p]}\right)$
7: **end for**
8: **end if**

4 Experimental Results and Discussion

4.1 Experimental Setup

We perform experiments on the Maeka cluster [21], which composes of 8 computing nodes. Each node is equipped with dual 2.4GHz Intel Xeon E5620 processors, 16GB main memory, and a Tesla M2050 GPU. All compute nodes are networked with a simple Gigabit Ethernet. In our implementation, we use GCC v4.1.2, OpenMPI v1.4.3 and NVIDIA CUDA toolkit v3.2 libraries.

We experiment using 3 real web graphs retrieved from the University of Florida sparse matrix collection repository [7]: `arabic-2005`, `it-2004`, and `sk-2005`, which compose of 22.7M, 41.3M, 50.6M pages and 640M, 1150.1M, 1949.4M links, respectively.

4.2 Results and Discussion

We compare parallel PageRank computations running with the same number of GPUs and CPUs. For all experimental web graphs, we can fit their CSWG representation

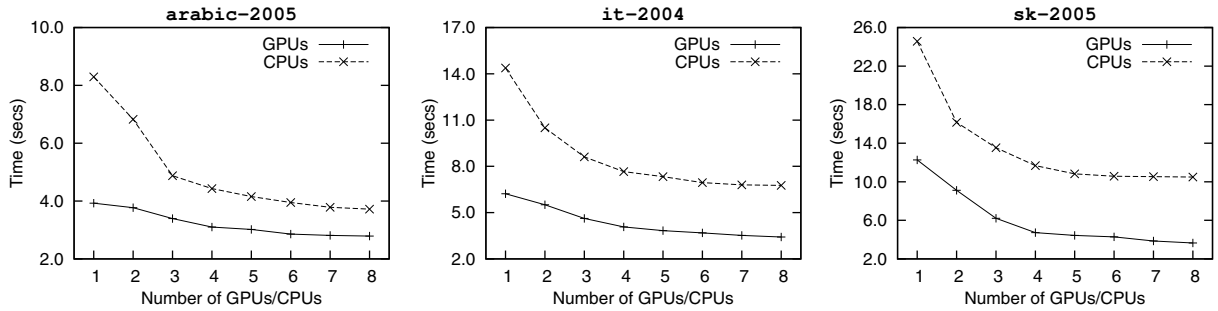


Figure 4. Average running time per iteration.

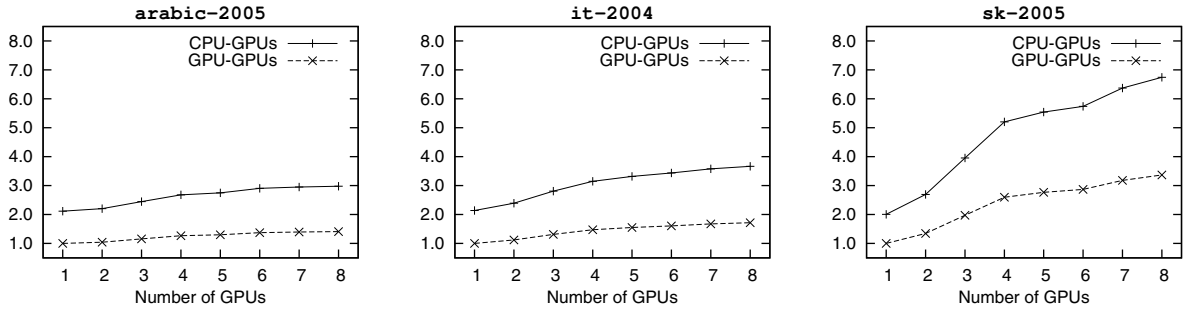


Figure 5. Speedup over one CPU and one GPU.

into any machine's memory, but not for the case of GPUs' device memory. The host memory of the machines in the cluster is then used as the web graph's temporary storage during the computation. Moreover, we run one MPI process for one CPU and one GPU in each computing node, and the GPUs' kernel in Algorithm 3 is run in double precision mode. Figure 4 illustrates the averaged running time per iteration after the PageRank scores have been converged.

From the figure, the running time on GPUs is much lower than that on CPUs. Both of them decrease quite significantly when increasing a small number of GPUs and CPUs (i.e., up to 4), and then gradually decrease since the effect of communication overhead (line 12 of Algorithm 2 and line 16 of the main function of Algorithm 3). However, as the whole web graph cannot be fit into GPUs' device memory, the overhead of repeated copying the rest of the partition between host and device memory is inevitable. This overhead has a significant effect on the computation on GPUs than the effect from random memory access on CPUs. Thus, we can see that as the number of GPUs/CPUs increases, the running time on GPUs decreases much slower. However, this running time becomes more significant decreasing as the size of web graph increases. This indicates that the parallel approach on GPUs will be scalable whenever the problem size (i.e., the size of web graph) is large.

Figure 5 concludes speedup performance calculated from the average running time per iteration of GPUs over one CPU (denoted "CPU-GPUs") and over one GPU (denoted "GPU-GPUs"). These results reveal that the GPU-GPUs speedups for `arabic-2005` and `it-2004` are quite poor. The reason is that when the number of GPUs increases, each assigned partition of those both web graphs is too small to busy the GPUs. Thus, the decrease in computation time obtained from GPU cores has not much impact when comparing with the time needed for copying data between host and device. However, the GPU-GPUs speedup for `sk-2005` which is a more dense graph than those of the previous two collections shows slightly better result. In addition, the CPU-GPUs results show that we preliminarily achieve 6.74x for this dense web collection.

5 Related Work

Designing a parallel algorithm using a GPU cluster can be an interesting challenge, particularly for iterative computing problem on a large size of sparse data structure like the PageRank's web graph. Mathematically, the PageRank computation can be treated as a numerical linear algebraic problem that computes the principal eigenvector of the sparse web graph matrix. Using the power method [18], the resulting eigenvector can then be calculated by iteratively

performing several SpMV operations. Since a sparse web graph matrix can be generally represented by the CSR format, many efficient SpMV kernels, such as those proposed in [3, 6, 8, 20], can therefore be explored and employed. To our knowledge, there currently are at least two research teams [5, 23] that propose the PageRank computing solutions using GPUs.

Wu et al. [23] first analyze the characteristic of sparse matrices that represent a link structure of the PageRank problem, and then design three different SpMV solvers in accordant with the number of non-zero entries in a row of the matrix. The group of rows with sparse non-zero entries will be computed by a SpMV solver using the small number of threads, while those with dense non-zero ones will be processed using the larger number of threads. They performed experiments using an AMD Radeon-based GPU machine, and reported a peak performance of 15x speedup on a small web collection that can be totally fit into GPU's device memory. One main difference between the work of Wu et al. [23] and ours is that they only report the feasibility of using one GPU to solve the PageRank computational problem of some small web graphs. For larger web graph, their proposed methodology is still questionable.

On the other side, Cevahir et al. [5] claim an efficient PageRank computation on an NVIDIA GPU cluster in which each node contains dual-GPUs. As recent GPUs have no direct connection between them, communication between GPUs are therefore coordinated by host CPUs via PCI-express bus and/or LAN connection. To minimize the communication cost between GPUs, they first exploit both the web pages' site information and the hypergraph partitioning technique to decompose the web graph matrix into smaller chunks and assign each chunk to each computing node in the cluster. Each chunk is again decomposed into two portions, each of which can be totally loaded into the device memory of each GPU equipped in a computing node. One MPI process [22] is then assigned for each GPU, and the CSR-vector-based SpMV kernel selected from [3] has been applied to solve the PageRank computational problem. However, this method cannot be applied instantly to large web data using a small size of GPU cluster. A large web graph must be first pre-processed using a hypergraph-based partitioning to obtain a smaller chunk of matrix partition, each chunk must be divided to fit into the device memory of a GPU. Then, the number of GPUs is presumed to be large enough to host the whole web graph matrix in device memory during the computation. Thus, to solve a PageRank computational problem with large web graphs, the method proposed by Cevahir et al. [5] needs a number of GPUs in a cluster to be available, in addition to those hypergraph based decompositional costs.

To overcome those limitations and to design a generalized PageRank algorithm on a GPU cluster, we here preliminarily explore a simple approach to partition the web graph based on the number of web pages and machines. After each partition is assigned to each machine, it has been divided again, by the CPU, into smaller chunks to fit the GPU's available memory. During the computation, the CPUs' host memory has been designed to host and buffer the rest of the partition. Thus, our approach in theoretical way is capable of computing the PagaRank of any large web graph, although we have only a small number of GPUs in our cluster setting. Although the presented preliminary results reveal only a speedup of 6.74x on eight GPUs over one CPU, comparing to a 10x speedup on a costly hundred GPUs over the same number of CPUs setting reported in [5] using a comparable size of web collections, the PageRank of all real web data sets explored in the experiments can be accomplished using just one or more GPUs setting.

6 Conclusions and Future Work

We have proposed a simple, specially designed and compact web graph representation, as well as presented an alternative parallel PageRank algorithm running on a small GPU cluster. To avoid spending very much time of partitioning the web data and balancing the communication overhead that might have between the processors, we simply divide the web graph into several chunks based on the number of web pages and processors. Then, each small chunk has been sent to GPUs for accelerating the PageRank computation. In the experiments, we compare the computational time on GPUs with CPU, and currently obtain up to 6.74 times faster using 8 GPUs. The proposed algorithm does not require to totally fit the whole web graph into GPUs' device memory during the computation. We however have to pay the overhead cost of copying data between GPUs' device memory and CPUs' host memory, and inevitably the additional synchronizational cost of the rank vector between the computing nodes during the computation.

Though we obtain quite a promising preliminary result on a small cluster of GPUs, there are still much work to pursue. As we first just explore a simple partitioning method of the web graph matrix based on the number of its rows and the number of processors which we use in the cluster setting, the problem of unbalanced workload on the compute nodes has still not been well considered and investigated. In addition, we are also looking forward to spending much time on optimizing the kernel codes by trying some well-known robust SpMV routines, such as those reported in [3, 8], exploring the extrapolation and adaptive techniques as reported in [13, 14], decreasing the communication overhead needed to synchronize the rank vector between the MPI processes, further studying the use of CPUs' multi-threading capability, as well as the scalability issues using a set of larger web graphs.

Acknowledgement

This research is done under the support from Department of Computer Engineering, Faculty of Engineering, Kasetsart University. We thank to all research teams who help providing the experimental dataset. We also thank to anonymous reviewers for their comments and suggestions.

References

[1] Apache Hadoop. `http://hadoop.apache.org/`.

[2] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst. *Templates for the Solution of Algebraic Eigenvalue Problems: a Practical Guide*, chapter Common Issues: Sparse Matrix Storage Formats, pages 315–316. SIAM Publication, 2000.

[3] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009.

[4] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[5] A. Cevahir, C. Aykanat, A. Turk, B. B. Cambazoglu, A. Nukada, and S. Matsuoka. Efficient PageRank on GPU clusters. Technical Report 2010-HPC-128(21), IPSJ SIG, 2010.

[6] J. W. Choi, A. Singh, and R. W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *Proceedings of the 15th Symposium on Principles and Practice of Parallel Programming*, 2010.

[7] T. A. Davis and Y. F. Hu. The University of Florida sparse matrix collection. ACM Transactions on Mathematical Software, Nov. 2010, to be published, `http://www.cise.ufl.edu/research/sparse/matrices/`.

[8] M. Garland. Sparse matrix computations on manycore GPU's. In *Proceedings of the 45th Annual Design Automation Conference*, 2008.

[9] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: High performance graphics co-processor sorting for large database management. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2006.

[10] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2004.

[11] N. K. Govindaraju, N. Raghuvanshi, and D. Manocha. Fast and approximate stream mining of quantiles and frequencies using graphics processors. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2005.

[12] B. He, K. Yang, R. Fang, M. Lu, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational joins on graphics processors. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2008.

[13] S. D. Kamvar, T. H. Haveliwala, and G. H. Golub. Adaptive methods for the computation of PageRank. *Linear Algebra and Its Applications*, 386:51–65, 2004.

[14] S. D. Kamvar, T. H. Haveliwala, C. D. Manning, and G. H. Golub. Extrapolation methods for accelerating PageRank computation. In *Proceedings of the 12th International Conference on World Wide Web*, 2003.

[15] E. S. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, 2001.

[16] B. Manaskasemsak and A. Rungsawang. Parallel PageRank computation on a Gigabit PC cluster. In *Proceedings of the 18th International Conference on Advanced Information Networking and Applications*, 2004.

[17] B. Manaskasemsak, P. Uthayopas, and A. Rungsawang. A mixed MPI-thread approach for parallel page ranking computation. *OTM Confederated International Conferences, LNCS, Springer, Heidelberg*, 4276:1223–1233, 2006.

[18] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[19] S. J. Plimpton and K. D. Devine. MapReduce in MPI for large-scale graph algorithms. *Parallel Computing*, Elsevier B.V., Article in Press, 2011.

[20] Y. Shan, T. Wu, Y. Wang, B. Wang, Z. Wang, N. Xu, and H. Yang. FPGA and GPU implementation of large scale SpMV. In *Proceedings of the 8th Symposium on Application Specific Processors*, 2010.

[21] The Maeka. The GPU cluster of Kasetsart University, Bangkok, Thailand. `http://maeka.cpe.ku.ac.th/`.

[22] The Message Passing Interface (MPI). `http://www.mcs.anl.gov/research/projects/mpi/`.

[23] T. Wu, B. Wang, Y. Shan, F. Yan, Y. Wang, and N. Xu. Efficient PageRank and SpMV computation on AMD GPUs. In *Proceedings of the 39th International Conference on Parallel Processing*, 2010.

[24] X. Yang, S. Parthasarathy, and P. Sadayappan. Fast sparse matrix-vector multiplication on GPUs: Implications for graph mining. *Proceedings of the VLDB Endowment*, 4(4):231–242, 2011.