Towards Exploiting GPUs for Fast PageRank Computation of Large-Scale Networks

Min-Soo Kim

Department of Information and Communication Engineering, DGIST
50-1, Sang-Ri, Hyeonpung-Myeon, Dalseong-Gun, Daegu 711-873, Korea
mskim@dgist.ac.kr

The size of network (or graph) is increasing, and so, a fast algorithm for network analysis is more important than ever. PageRank-style algorithm is one of the most important and fundamental algorithms for network analysis. Meanwhile, the paradigm of micro-architecture design of computer processors has been shifted to on-chip multi-core CPUs, and furthermore, many-core GPUs. The current fastest PageRank method is the one based on multi-core CPUs. In contrast, there is lack of studies on using many-core GPUs for networks analysis including PageRank yet due to difficulty to develop a GPU algorithm that efficiently manipulates complex and irregular data structures like complex networks. This paper proposes novel fast parallel PageRank methods that exploit the massive parallelism of GPU for large-scale networks. More specifically, the paper proposes the node-centric method computing PageRank in terms of nodes and the edge-centric method computing PageRank in terms of edges. They efficiently compute PageRank based on a GPU with compact data structures and concise kernel functions. Through extensive experiments, the paper shows the proposed methods outperform the state-of-the-art method by up to about two times.

Key Words: Complex networks, PageRank, multi-core CPU, many-core GPU, node-centric, edge-centric

1. INTRODUCTION

Complex network (or graph) perspective provides us with a clear way of analyzing the whole structure among entities [1]. A network is usually made up of a set of actors (i.e., nodes) and a set of dyadic relationships between two actors (i.e., edges). There are various kinds of complex networks in many fields such as computer networks, telecommunication, sociology, biology, and neuroscience, depending on the kind of nodes and edges. The complexity of networks is increasing, and their sizes are getting bigger and bigger [2], [3]. In order to analyze such a large-scale complex networks within a reasonable time or in a near real-time fashion, the fast algorithms for network analysis are more important than ever.

There are many kinds of algorithms of network analysis including computing PageRank, computing betweenness centrality, finding communities, finding connected components, and counting triangles (i.e., computing potential friends). PageRank [4] is one of the most important and fundamental algorithms among them. It assigns a numerical value to each node by considering directed or undirected links (i.e., edges) among nodes. Here, the node receiving much attention, i.e., many incoming edges, has a high numerical value, and the node of a higher PageRank value is usually regarded as a more important node in a network.

Meanwhile, the paradigm of micro-architecture design of CPUs has shifted to on-chip multi-core processors instead of increasing clock speed, and furthermore, on-chip many-core GPUs start to be used as potentially important co-processors for general-purpose computing. Since the speed of a single core of CPU or GPU does not increase much anymore, there is a certain limit to what single thread algorithms

could improve the performance of network analysis. Therefore, the importance of parallel algorithms and data structures exploiting multi cores of CPU or many cores of GPU is growing bigger and bigger according to the growth in the size of networks.

Recently, the memory-based parallel graph analysis software called Green-Marl [5] has been proposed to fully exploit multi cores of CPU for near real-time graph analysis in a single computer. Green-Marl generates a highly optimized C++ code for a given high-level algorithmic description. The multi-threaded code for PageRank generated by Green-Marl performs better than highly-tuned hand-coded implementation [5].

Compared with this state-of-the-art network analysis method using a multi-core CPU, there is lack of studies on using many-core GPUs for networks analysis including PageRank yet. In general, GPUs have a greater theoretical, i.e., potential computing power compared with CPUs, and such theoretical performance gap is widening more and more. There are already many cases that improve the performance based on GPUs [6], [7] that manipulate relatively simple and regular data structures such as matrix and image. However, it is nontrivial to develop a GPU algorithm that very efficiently manipulates complex and irregular data structures like complex networks. As a result, a PageRank method exploiting a many-core GPU that can outperform Green-Marl's PageRank method has not been proposed so far.

This paper proposes a novel fast parallel PageRank computation based on a many-core GPU that exploits the massive amount of thread-level parallelism of GPU for large-scale networks. For efficient processing on GPU, the paper proposes concise kernel algorithms with considering the SIMT (single instruction, multiple thread) model of a GPU and compact data structure with considering the bandwidth between main memory and GPU. More specifically, I propose the following two methods considering the structure of networks: the node-centric method and the edge-centric method. The node-centric method calculates PageRank values in terms of nodes, whereas the edge-centric one calculates PageRank values in terms of edges. Through extensive experiments, I show that our proposed method outperforms the state-of-the-art method by up to about two times.

The rest of this paper is organized as follows. Section 2 presents the preliminaries about many-core GPU and PageRank for the paper. Section 3 proposes the edge-centric method, and Section 4 the node-centric method. Section 5 presents the experimental results. Section 6 reviews related work including Green-Marl. Section 7 concludes the paper.

2. PRELIMINARIES

2.1 Many-core GPU

The modern many-core GPUs have a lot of cores which can be used for general-purpose computing on. For example, NVIDIA's TESLA K20X recently released has a total of 2,688 cores. In general, a GPU consists of multiple streaming multiprocessors, which again consists of multiple cores. In case of TESLA K20X, it has 15 streaming multiprocessors of 192 cores. A GPU contains its own device memory of up to 6GBytes, which is globally accessed by all cores. It is connected with CPU and main memory

through PCI bus interface. Even though a GPU has much more cores than a CPU, the speed of a GPU core is much slower than that of a CPU, and furthermore, there is a critical limitation that all threads currently running on a GPU only can perform the same instruction. That model is called Single Instruction, Multiple Thread (SIMT). Due to the SIMT model, a more sophisticated code may rather degrade the performance when running on a GPU.

The computation using GPUs is performed by calling the function called *kernel*. Here, the set of instructions described in the kernel is executed in all threads on a GPU. The GPU programming models such as CUDA and OpenCL usually allow users to identify a specific GPU thread in the kernel code with special variables for the number of blocks(shortly, *numBlocks*), the number of threads per block(shortly, *numThreads*), a block index(shortly, *blockIdx*), and a thread index within a block(shortly, *threadIdx*). A thread in the kernel code can identify its own ID by calculating $blockIdx \times numThreads + threadIdx$.

The current technology of GPU usually does not support the capability of dynamic memory allocation yet. Thus, a CPU thread should allocate the memory required for input data and output data on the device memory, and then, copy input data from the main memory to the device memory before calling the kernel function. After executing the kernel, the CPU thread should copy output data from the device memory to the main memory.

2.2 PageRank

This paper defines a social network G as (V, E) where V is a set of nodes, and E a set of edges. Edges could be directed or undirected. Then, the PageRank algorithm could be defined as (1), where $oldPR(w)$ is a function returning an old PageRank value of the node w, and $newPR(v)$ a function returning a new PageRank value of the node v, and $outDegree(w)$ a function returning the outdegrees of the node w. Also, df is a damping factor, where 0.85 is used in many cases, and $v.adj$ a set of adjacent nodes to the node v.

$$newPR(v) = df \cdot \Sigma_{w \in v.adj} (oldPR(w)/outDegree(w)) + (1-df) / |V| \quad (1)$$

Figure 1 shows a small example network and its input file following one of popular formats. There are ten nodes, and so, the initial PageRank value of each node is 1.0 / 10 = 0.1. After one iteration of PageRank computation, the PageRank value of the node 0, $newPR(v_0)$ becomes $0.85 \times (0.1/1 + 0.1/2 + 0.1/3 + 0.1/3) + 0.15/10 \approx 0.199$.

3. EDGE-CENTRIC GPU PAGERANK COMPUTATION

The edge-centric method for GPU PageRank computation regards an edge of network as a unit of updating the PageRank value. During a specific iteration of PageRank, each GPU thread takes responsibility for partial updating the PageRank value of the destination node of the corresponding edge. Section 3.1 presents the data structure of the edge-centric method, and Section 3.2 the corresponding algorithm.

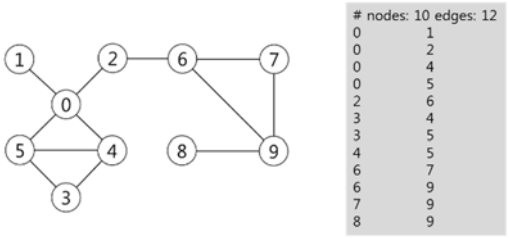

Figure 1. An example network and its input file.

3.1 Data structure

It is well-known that it is important to reduce the amount of data to be transferred between main memory and the device memory of GPU for a better performance in GPU computation [8]. At the same time, the data structures should be appropriate for GPU computation, that is, simple enough to be processed in GPUs. Since the ALU and memory scheme of GPU is not as powerful as that of CPU, using relatively complex data structures such as a map and a nested structure (*e.g.*, vector of lists) is not efficiently supported by GPU.

For representing networks, among the three formats are widely used [9], the adjacency matrix format inherently has a space complexity of $O(|V|^2)$ although compression techniques can be applied for reducing the space, and so, it is mainly suitable for dense networks. Since the larger social networks tend to be relatively sparser in terms of the size of matrix (*i.e.*, $|V|^2$) [10], this paper follows either of the remaining two formats and especially adopt the format of array of edges for the edge-centric method. It has a space complexity of $O(|E|)$, where $|E| << |V|^2$ for large-scale networks, and also, the format is very simple so as to be easily processed in GPUs.

PageRank computation in (1) requires the information about the outdegrees of each node. Thus, the edge-centric method sends outdegree information (shortly, *outDegree*) as well as array of edges (shortly, *edgeArray*) to GPU. Figure 2 shows the *edgeArray* and *outDegree* for the example network in Figure 1. The size of *edgeArray* is $2|E|$, where each edge consists of a pair of source node ID and destination node ID, and that of *outDegree* is $|V|$. In addition to them, it is also necessary to send the array of old PageRank values (shortly, *oldPR*) and the array of new PageRank values (shortly, *newPR*) to GPU, each size of which is $|V|$. Therefore, the space complexity of all data sent to GPU in the edge-centric method becomes $O(2|E|+3|V|)$. The amount of space can be reduced much by compression, but it is beyond the scope of this paper.

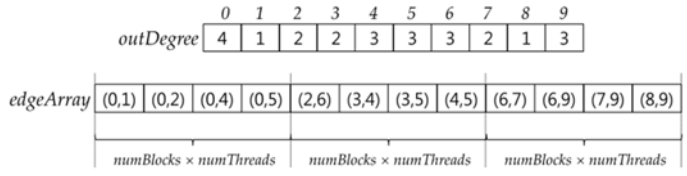

Figure 2. Example data structures for the edge-centric method.

3.2 Algorithms

Intuitively, the edge-centric method exploits the massive parallelism of GPU such that a large number of edges in *edgeArray* are processed simultaneously. In GPU, up to (*numBlocks* × *numThreads*) threads can run at the same time logically. Here, *numBlocks* and *numThreads* are specified by a user. For instance, assume that *numBlocks* and *numThreads* are two and two, respectively. Then, (*numBlocks* × *numThreads*) = 4, so each thread processes a total three edges in *edgeArray* of Figure 2 during a single iteration of PageRank.

The original PageRank equation in (1) is specified in terms of nodes, so it is necessary to calculate it in terms of edges for the edge-centric model. The edge-centric algorithm performs PageRank in the following two phases: (1) initialize *newPR* with the minimum PageRank value and (2) increase the values of *newPR* while scanning *edgeArray*. In the first phase, the minimum PageRank value (shortly, *minPR*) means the latter part of (1), *i.e.*, (1 − df) / |*V*|. In the second phase, for an edge (*w*, *v*), the algorithm increases *newPR*[*v*] by (*df* × *oldPR*[*w*] / *outDegree*[*w*]). If it updates all *newPR*[*v*] for all edges (*w*, *v*) ∈ *E* in such a way, then the final result is equivalent to a single iteration of the original PageRank computation.

Algorithm 1 and 2 present the kernel functions of the proposed edge-centric model. Algorithm 1 performs the first phase, *i.e.*, assigning *minPR* to every element of *newPR*. Algorithm 2 performs the second phase, *i.e.*, updating *newPR* while scanning *edgeArray*. In Algorithm 1, Lines 1, 2, 4, and 5 are for processing *newPR* in chunks, where the chunk size is (*numBlocks* × *numThreads*). The local variable *idx* indicates the index of the element in *newPR* that the current thread is processing. In this kernel, each GPU thread initializes the target *newPR* element with *minPR* (Line 3). Here, note that there is no race condition on *newPR* since the memory addresses to be updated are all different with each other. Algorithm 2 processes *edgeArray* in chunks (Lines 1, 2, 7, and 8) as in Algorithm 1. But, unlike Algorithm 1, the number of elements processed by Algorithm 2 is *numEdges*, not *numNodes* (Line 2). Line 4 indicates increasing *newPR*[*v*] by (*df* × *oldPR*[*w*] / *outDegree*[*w*]). Here, note that the algorithm must call an atomic operation for updating *newPR*[*v*] since there could be a race condition. For example, while thread *A* is increasing *newPR*[*v*] for an edge (*w*, *v*), thread *B* might be increasing *newPR*[*v*] for an edge (*u*, *v*) at the same time. Since there are a huge number of threads running (*e.g.*, one million), and the number of nodes are usually much smaller than that of edges, there is a high probability that a race condition occurs at each *newPR*[*v*]. The GPU programming model like CUDA allows us to use the *atomicAdd* function for the float (*i.e.*, single-precision) type. However, it may not support the *atomicAdd* function for the double-precision type. In case of using the double type for more precise PageRank values, the *atomicAdd* function for the double type can be implemented by using other atomic function like *atomicCAS*. If a network is not directed, but undirected, then, it is usually regarded as a bidirected one in PageRank computation, so the algorithm needs to increase *newPR*[*w*] by (*df* × *oldPR*[*v*] / *outDegree*[*v*]) as well (Line 6). Finally, note that the edge-centric method need to put a thread synchronization barrier after the kernel-2 (Algorithm 2), *i.e.*, after finishing one iteration of PageRank computation. Otherwise, some threads perform the (*i*+1)-th

iteration while other threads still perform the *i*-th iteration, and so the algorithm may get wrong results. The GPU programming model usually provides various kinds of barrier functions (*e.g.*, the *cudaDeviceSynchronize* function in CUDA).

Algorithm 1 Edge-centric PageRank GPU Kernel-1

```
Input:  (1) numNodes // # of edges
        (2) minPR    // minimum PageRank value
InOut:  (1) newPR    // new PageRank values
1:  idx = blockIdx * numThreads + threadIdx;
2:  while idx < numNodes do
3:    newPR[idx] = minPR;
4:    idx += numBlocks * numThreads;
5:  end while
```

Algorithm 2 Edge-centric PageRank GPU Kernel-2

```
Input:   (1) numEdges   // # of edges
         (2) df         // damping factor
         (3) outDegree // out-degree array
         (4) edgeArray // edge array
         (5) oldPR     // old PageRank values
InOut:   (1) newPR     // new PageRank values
1:  idx = blockIdx * numThreads + threadIdx;
2:  while idx < numEdges do
3:    (w, v) = edgeArray[idx];
4:    atomicAdd(newPR[v], df * oldPR[w] / outDegree[w]);
5:    // perform the below line only for undirected networks
6:    atomicAdd(newPR[w], df * oldPR[v] / outDegree[v]);
7:    idx += numBlocks * numThreads;
8:  end while
9:  return newPR;
```

4. NODE-CENTRIC GPU PAGERANK COMPUTATION

The node-centric method for GPU PageRank computation regards a node as a unit of updating the PageRank value. During a specific iteration of PageRank, each GPU thread takes responsibility for full updating the PageRank value of the corresponding node even in case the number of incoming edges is large (*i.e.*, $|v.adj|$ is large). Section 4.1 presents the data structure of the node-centric method, and Section 4.2 the corresponding algorithm.

4.1 Data structure

Among the formats addressed in Section 3.1, the node-centric method adopts the remaining format, *i.e.*, array of adjacency lists. That format has a good space complexity of $O(|E|+|V|)$ and is appropriate for the node-centric model since all IDs of incoming or outgoing nodes for a specific node are clustered together. However, the method cannot use a naïve implementation of that presentation such as a vector of lists since it is a nested structure, and also, requires runtime dynamic allocation internally on device memory. The current technology of GPU does not support nested structures and dynamic allocation well. So, I propose a new data structure implementing array of adjacency lists that can be efficiently processed on GPU.

The proposed data structure consists of two arrays: (1) *adjListArray* that concatenates all adjacency lists and (2) *ptrArray* that contains the offsets pointing to each adjacency list in *adjListArray*. Figure 3 shows the *adjListArray* and *ptrArray* for the example network in Figure 1. In the example, *ptrArray*[0] for the node 0 points to *adjListArray*[0], and the node 0 has four adjacent nodes 1, 2, 4, and 5, so they are stored successively between *adjListArray*[0] and *adjListArray*[3]. Next, *ptrArray*[1] for the node 1 points to *adjListArray*[3], and so on. The size of *ptrArray* is $|V|$, and that of *adjListArray* is $|E|$ for directed networks. If a network is undirected, both *v* and *w* for an edge (*w*, *v*) should be stored in *adjListArray* under this format. For instance, for the edge (1, 0), the node 1 is stored at *adjListArray*[0], and the node 0 stored at *adjListArray*[4]. So the size of *adjListArray* becomes $2|E|$. With considering the sizes of *oldPR* and *newPR*, the space complexity of all data sent to GPU in the node-centric method is $O(2|E|+3|V|)$, which is equal with that of the edge-centric method.

In the proposed data structure, the outdegrees of each node can be calculated using *ptrArray*. For example, the outdegree of the node 2 is calculated by (*ptrArray*[3] – *ptrArray*[2]), *i.e.*, 2. In order to calculate the outdegree of the last node, the node-centric method appends an extra element to *ptrArray* that contains the last offset of *adjListArray*. For example, the shaded element *ptrArray*[10] in Figure 3 contains 24.

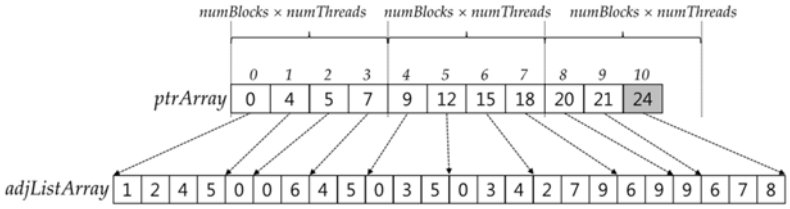

Figure 3. Example data structures for the node-centric method.

4.2 Algorithms

Intuitively, the node-centric method exploits the massive parallelism of GPU such that a large number of pointers in *ptrArray* are processed simultaneously. Here, since each element of *ptrArray* points an adjacency list in *adjListArray*, so each GPU thread should scan the specific entire adjacency list pointed by the corresponding element. Just like the edge-centric method, up to (*numBlocks* × *numThreads*) threads run logically at the same time. For example, if (*numBlocks* × *numThreads*) = 4, then, in Figure 3, each thread processes up to three adjacency lists during a single iteration of

PageRank. Specifically, the first thread would process the adjacency list {1, 2, 4, 5} pointed by *ptrArray*[0], {4, 5} pointed by *ptrArray*[4], and {9} pointed by *ptrArray*[8] in each PageRank iteration.

Algorithm 3 presents the kernel function of the node-centric model. Lines 1, 2, 10, and 11 are for processing *ptrArray* in chunks. The chunk size in Algorithm 3 is (*numBlocks* × *numThreads*) as in Algorithm 2. But, unlike Algorithm 2, it has an inner loop for processing *adjListArray* (Lines 4~8). After a thread enters the outer loop of Line 2, it first initializes the sum local variable, and then, calculates the PageRank value for the node *idx* within the inner loop. In the inner loop, it gets the offsets $\{w \mid ptrArray[idx] \leq w < ptrArray[idx+1]\}$ corresponding to incoming nodes of the node *idx*. Here, *w* is just an offset within *adjListArray*, not a node ID, so the thread gets the node ID *wID* in Line 5. Next, the thread gets the outdegree of the node *wID* by doing (*ptrArray*[*wID*+1] – *ptrArray*[*wID*]) in Line 6. Then, it increase sum by (*oldPR*[*wID*] / *wOutDegree*) according to (1). After finishing the inner loop, the algorithm calculate the new PageRank value for the node *idx*, *i.e.*, *newPR*[*idx*], in Line 9.

Algorithm 3 Node-centric PageRank GPU Kernel

```
Input:   (1) numNodes  // # of nodes
         (2) df        // damping factor
         (3) ptrArray  // array for pointers
         (4) adjListArray // array for adjacency lists
         (5) oldPR     // old PageRank values
InOut:   (1) newPR     // new PageRank values
 1:  idx = blockIdx * numThreads + threadIdx;
 2:  while idx < numNodes do
 3:    sum = 0;
 4:    for each w, ptrArray[idx] ≤ w < ptrArray[idx+1] do
 5:      wID = adjListArray[w];
 6:      wOutDegree = ptrArray[wID+1] – ptrArray[wID];
 7:      sum += oldPR[wID] / wOutDegree;
 8:    end for
 9:    newPR[idx] = df * sum + (1 - df) / numNodes;
10:    idx += numBlocks * numThreads;
11:  end while
12:  return newPR;
```

5. EXPERIMENTAL RESULTS AND ANALYSIS

5.1 Experimental Setup

I evaluate the performance of the proposed edge-centric method and node-centric method based on a GPU, which are denoted as *GPU_edge* and *GPU_node*, respectively, compared with the state-of-the-art method, Green-Marl, based on CPU. When running Green-Marl with *n* threads, I denote it as *CPU_GM_<n>*. I also evaluate the performance of the basic single-thread algorithm of PageRank (shortly, *CPU_basic*) for reference. I measure a total elapsed time of ten PageRank iterations in all experiments. In case of *GPU_edge* and *GPU_node*, I measure the time so as to include the data transfer time between main memory and GPU.

For social networks, we use both synthetic datasets and real datasets. For synthetic datasets, we generate *small* (2 million nodes), *medium* (4 million nodes), and *large* (8 million nodes) networks by using the random network generator provided by Green-Marl. For each size, we again generate three kinds of networks: *sparse* (the number of edges is equal to that of nodes), *normal* (the number of edges is 5X larger than that of nodes), and *dense* (the number of edges is 10X lager than that of nodes). For real datasets, we use the road network of California (shortly, *roadNet-CA*), which consists of 1,965,206 nodes and 5,533,214 edges, and the citation network among US Patents (shortly, *cit-Patents*), which consists of 3,774,768 nodes and 16,518,948 edges.

I conduct all the experiments on the same PC with Intel i7 3940XM quad-core 3.0GHz, 32 GBytes main memory, and NVIDIA K5000M GPU of CUDA 1,344 cores. I choose the models of CPU and GPU such that they are released at almost the same time and have almost the same price. All experiments are performed on the same OS of SUSE Linux. Green-Marl is implemented in C++, and I also implement *CPU_basic* and our proposed two methods in C++. They all are compiled with the same optimized option of -O3. I use from one to eight execution threads for Green-Marl since the CPU used has four physical cores, and each core has two hyper threads. I also use one million GPU threads for the proposed methods, where *numBlocks* = 1,000, and *numThreads* = 1,000.

5.2 Synthetic Datasets

Figure 4 shows the elapsed time of each method for small synthetic data (2M nodes) by varying the density of the network. *CPU_basic* shows the worst performance, and Green-Marl using a single thread (*i.e.*, *CPU_GM_1*) outperforms it by up to 1.83 times, which means Green-Marl is a highly optimized method. When varying the number of threads of Green-Marl from one to eight, the elapsed time is decreasing gradually. Here, when changing from *CPU_GM_4* to *CPU_GM_8*, the elapsed time does not decrease much since logical cores by the hyper-threading option is not as very effective as physical cores. Among the proposed two methods, the node-centric method, *GPU_node* outperforms the other method, *GPU_edge*, by up to 2.82 times. It is because *GPU_edge* has extra overhead of using atomic operations and using two kernels. In addition, the number of iterations in the *for* loop of *GPU_edge* (= $|E|$) is usually much larger than that of *GPU_node* (= $|N|$). As a result, *GPU_edge* rather

shows worse performance than *CPU_GM_8*. However, *GPU_node* shows the best performance among all methods tested and outperforms *CPU_GM_8* by up to 1.96 times. The elapsed time for medium synthetic data (4M nodes) shows almost the same performance trend with Figure 4. Figure 5 shows the elapsed time for large synthetic data (8M nodes). This figure shows the performance trend by varying density in a different view.

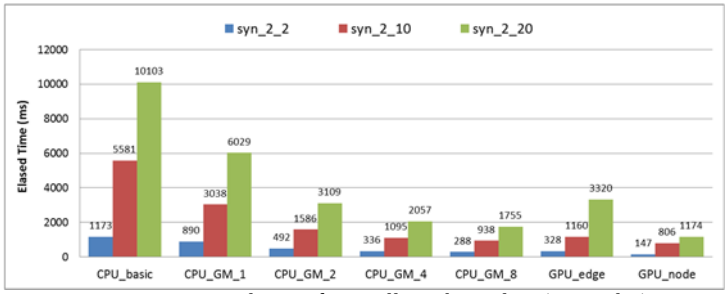

Figure 4. Varying density for small synthetic data (2M nodes).

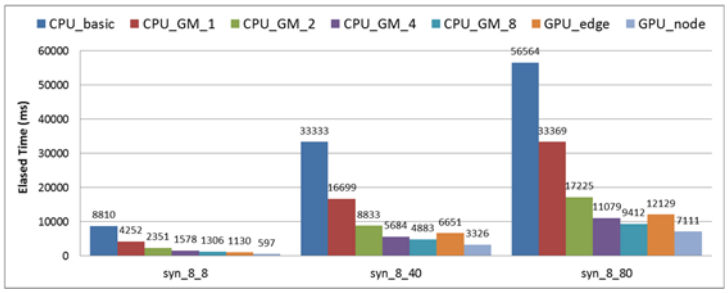

Figure 5. Varying density for large synthetic data (8M nodes).

5.3 Real Datasets

Figure 6 shows the elapsed time of each method for real datasets of readNet-CA and cit-Patents. CPU_basic still shows the worst performance, and the proposed method, GPU_node the best performance among all methods. Different from the results using synthetic datasets, the proposed method GPU_edge shows the second best performance, and CPU_GM_8 just the third best performance. It is because real networks have different distribution and topology from random networks.

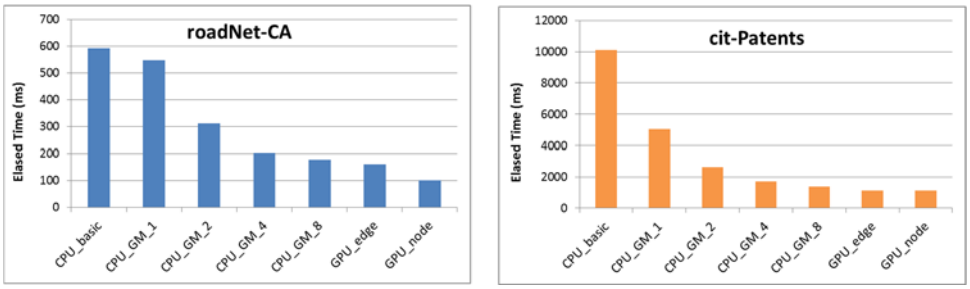

Figure 6. Elapsed time for real datasets.

6. RELATED WORK

There are many graph analysis methods that can compute PageRank for large-scale networks. They can be categorized into single-machine methods and distributed methods depending on the number of computers used. Each category can be again broken down into memory-based methods and disk-based methods depending on the storage type where the computation is performed.

The single-machine methods can perform PageRank computation efficiently on the strength of multi-core CPUs unless the size of data exceeds the size of storage. Nowadays, the size of main memory used in computers is getting larger and larger, and so, it is very common that a computer is equipped with several GBytes memory. That amount of memory can store sufficiently the social networks of ten million nodes and one hundred million, where most of networks have smaller size than it [11]. The single-machine methods also have a benefit that can be used in a wide-range of environments such as a small company, a small laboratory, and even mobile equipment, due to the small size and low price of the equipment. There are several major single-machine methods proposed recently. Neo4j [12] and GraphChi [13] are proposed for disk-based environments, and Green-Marl [5] is proposed for memory-based environments. Among them, Green-Marl shows the best performance, and so I have compared it with our proposed methods.

The distributed methods can perform PageRank computation for web-scale social networks larger than one billion nodes. But, they usually show the worse performance for smaller social networks due to overhead from network communication [13]. There are many methods proposed recently. PEGASUS [14], Pregel [15], GBase [16], and GraphLab [17] are proposed for disk-based distributed environments, and Trinity [18] is proposed for memory-based distributed environments.

7. CONCLUSION

This paper presented a fast, massive-parallel PageRank computation methods based on a many-core GPU. Specifically, the paper proposed the edge-centric method and the node-centric method. During a specific iteration of PageRank, a GPU thread in the former method partially updates the PageRank value the node, and so, needs to use atomic operations for solving a race condition, while that in the latter method fully

updates the PageRank value of the node. Between those two methods, the node-centric method outperforms the edge-centric method due to no overhead of using atomic operations and multiple kernels.

It is nontrivial to devise a GPU algorithm that outperforms the corresponding state-of-the-art CPU algorithm for complex irregular data structures like complex networks. Through extensive experiments on various kinds of synthetic datasets, I showed that the proposed method, especially, the node-centric method outperforms the state-of-the-art multi-threaded PageRank method of Green-Marl by up to about two times. This better performance of the node-centric method is due to its compact data structures and concise kernel function that efficiently exploits the massive parallelism of GPU. I will continue to explore the hybrid algorithms exploiting both CPUs and GPUs for network analysis in the future.

ACKNOWLEDGEMENTS

This work was supported by the IT R&D program of MSIP/KEIT [10041145, Self-Organized Software platform(SoSp) for Welfare Device] and the DGIST R&D Program of the Ministry of Education, Science and Technology of Korea[13-BD-0403].

REFERENCES

[1] Stanley Wasserman and Katherine Faust, “Social Network Analysis in the Social and Behavioral Sciences,” Social Network Analysis: Methods and Applications, pp. 1–27, Cambridge University Press. 1994.

[2] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, “Group formation in large social networks: membership, growth, and evolution,” in Proc. of the 12th ACM International Conference on Knowledge Discovery and Data Mining(KDD), pp. 44-54, Philadelphia, PA, USA, 2006.

[3] H. Kwak, C. Lee, H. Park, and S. Moon, “What is Twitter, a social network or a news media?,” in Proc. of the 19th International Conference on World Wide Web(WWW), pp. 591-600, Raleigh, North Carolina, USA, 2010.

[4] S. Brin, L. Page, R. Motwani, and T. Winograd, “The PageRank citation ranking: Bringing order to the Web,” Technical Report 1999-66, Stanford Digital Library Technologies Project, http://dbpubs.stanford.edu:8090/pub/1999-66, 1999.

[5] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, “Green-Marl: a DSL for easy and efficient graph analysis,” in Proc. of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS), pp. 349-362, London, England, UK, 2012.

[6] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey, “Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort,” in Proc. of the 2010 ACM SIGMOD International Conference on Management of Data(SIGMOD), pp. 351-362, Indianapolis, Indiana, USA, 2010.

[7] K. Wang, Y. Huai, R. Lee, F. Wang, X. Zhang, and J. H. Saltz, “Accelerating pathology image data cross-comparison on CPU-GPU hybrid systems,” in Proc. VLDB Endow., vol. 5, no. 11, pp. 1543-1554, 2012.

[8] R. Pearce, M. Gokhale, and N. Amato, “Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory,” in Proc. of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pp.1-11, 2010.

[9] Robert Sedgewick and Kevin Wayne, Algorithms, 4th Edition, Addison-Wesley, 2011.

[10] J. Leskovec, J. Kleinberg, and C. Faloutsos, “Graphs over time: densification laws, shrinking diameters and possible explanations,” in Proc. of the 11th ACM SIGKDD International Conference on Knowledge Discovery in Data Mining(KDD), pp. 177-187, Chicago, Illinois, USA, 2005.

[11] http://snap.stanford.edu/.

[12] http://www.neo4j.org/.
[13] A. Kyrola, G. Blelloch, and C. Guestrin, “GraphChi: large-scale graph computation on just a PC,” in Proc. of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI), pp. 31-46, Hollywood, CA, USA, 2012.
[14] U. Kang, C. E. Tsourakakis, and C. Faloutsos, “PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations,” in Proc. of the 2009 Ninth IEEE International Conference on Data Mining (ICDM), pp. 229-238, 2009.
[15] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, “Pregel: a system for large-scale graph processing,” in Proc. of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 135-146, Indianapolis, Indiana, USA, 2010.
[16] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos, “gbase: an efficient analysis platform for large graphs,” The VLDB Journal, vol. 21, no. 5, pp. 637-650, 2012.
[17] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, “Distributed GraphLab: a framework for machine learning and data mining in the cloud,” in Proc. VLDB Endow., vol. 5, no. 8, pp. 716-727, 2012.
[18] B. Shao, H. Wang, and Y. Li, “The Trinity Graph Engine,” Technical Report, Microsoft Research, 2012.