2015 IEEE 17th International Conference on High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), and 2015 IEEE 12th International Conf on Embedded Software and Systems (ICESS)

GPregel: A GPU-Based Parallel Graph Processing Model

Siyan Lai, Guangda Lai, Guojun Shen, Jing Jin, Xiaola Lin
School of Information Science and Technology, Sun Yat-sen University
Guangzhou 510006, China
e-mail: {laisy2,laigd,shengj, jinj5} @mail2.sysu.edu.cn and linxl@mail.sysu.edu.cn

***Abstract*—With the development of information technology, graph computing becomes an increasingly important tool for information processing. Recently, GPU has been adopted to accelerate various graph processing algorithms. However, since the architecture of GPU is very different from traditional computing model, the learning threshold for developing GPU-based applications is high. In this paper, we propose a GPU-based parallel graph processing system named GPregel. GPregel harnesses a lightweight complier to hide the underlying complexity of the parallel details and simplifies programming. It greatly reduces the difficulty in utilizing the GPU to solve graph computing problems. We also design a special storage model for BSP model running on GPU, which overcomes the execution divergence and irregular memory access by coarse-grained designs. Experiments demonstrate that GPregel can achieve high performance with little work for developers.**

Keywords—graph processing; GPGPU; GPU programming; parellel computing

I. INTRODUCTION

Graph is an important data structure that is widely applied to various science and engineering problems. Graph processing becomes an essential data analysis tool for big data since more and more data are generated. With the growth of the amount of information, the graph processing based on big data becomes increasingly important.

In general, the graph algorithm is iterative. Batch-execution features of their inherent mode of calculation include continually accessing the points and edges, and update their values under some conditions. Many graph algorithms are of fine-grained parallelism. Parallel computing is usually used to improve the performance. Previous studies [1-4] showed that many common graph algorithms can be parallelized based on *Bulk Synchronous Parallel* (BSP) model [5]. Google uses Pregel, a distributed BSP model graph processing system, to process significant portions of data every day. Several other systems are based on Pregel, including Giraph [6], GoldenOrb [7], GPS [8]. With the development of information technology, new paradigms such as social networks, big data, and graph search, graph computing becomes an important tool for information processing.

The *general-purpose computing on graphics processing units* (GPGPU) is extensively applied to solve compute-intensive problems by its excellent parallel computing capability. GPGPU uses a graphics processing unit (GPU), which typically handles computation only for computer graphics, to perform computation in a large number of applications traditionally handled by the CPU. In recent years, researchers have used GPU to deal with a number of graph processing problems and achieved satisfied results [9-12]. In particular, the GPU-based solution can improve computing efficiency significantly compared with the conventional CPU programming paradigm.

Researchers have begun to use the GPU to accelerate the graph computing application [9-12]. Harish designs and implements a number of commonly used in the GPU algorithm [7], including BFS, SSSP and APSP. Katz and others use diagonal method to block the adjacency matrix of graphs, accelerating realization of full source shortest path algorithm based on GPU [9]. Vineet et al. use CUDA implementation for graph of cutting pressure-heavy marker (push-Relabel) algorithm [10]. Hong et. al. proposed a Warp-unit method to parallelize the task (called virtual warp-centric), as well as techniques for using globally load balance queues and atomic operations [13]. Compared with Harish's work, warp-centric has the better speed when the topology of the graph when highly irregular. BFS is implemented algorithm in different ways [14, 15].

However, since the architecture of GPU is different from the traditional computing model, developing GPU-based applications is difficult. In addition, due to the complexity of the graph structure, developing a correct and even efficient GPU-based graph processing application is a challenge. Medusa adopted EVM (Edge-Message-Vertex) graph programming interfaces to address the issue [4]. The EVM is actually a different processing schema with Pregel because it processes the edges, messages and vertexes separately. Unfortunately, some algorithms that the operations of vertexes and edges are interdependent and inseparable may not be suitable to be processed in Medusa. In order to solve this problem, we propose a GPU-based parallel graph processing system named *GPregel* in this paper. GPregel is a Pregel-like system that utilizes GPU to accelerate graph computing. The system provides a simple API for developing graph processing applications, and it is optimized in many aspects for efficient GPU computing. Developers do not need to deal with the underlying details of GPU in GPregel. Using only C++ language can develop efficient parallel applications for graph processing. The developers who are familiar with Pregel can seamlessly switch to GPregel. The main features of the proposed GPregel are summarized as follows:

- GPregel provides a programming framework that not only can hide the underlying complexity of the parallel details, but also can simplify programming. As the API of GPregel is almost same to that of Pregel, GPregel and Pregel covered of algorithm range is consistent. Meanwhile, just like Pregel, GPregel also exploits coarse–grained processing schema. This main goal is achieved by designing a special compiler.

978-1-4799-8937-9/15 $31.00 © 2015 IEEE
DOI 10.1109/HPCC-CSS-ICESS.2015.184

- To accelerate GPregel on the GPU device, we design a special storage model that can well fit for the GPU memory architecture. It overcomes the execution divergence and irregular memory access by inheriting Pregel-like coarse-grained schema when BSP model runs on GPU. GPregel does not cover certain functions such as checking points and fault recovery to make the system simpler, so that we can focus on optimizing the system performance. Actually, single point of failure means GPU device is broken and the computation has to stop.

Extensive experiments have been conducted to evaluate the proposed system performance. The experimental results indicate that GPregel can achieve high performance with little work for developers. The brief API enables developer to fully utilize the computational power of GPU for graph computing. The developers do not need to deal with the details of GPU. The GPregel also exhibits high scalability as it can achieve satisfied performance extending to the algorithms in which the operation of vertexes and edges are interdependent and inseparable.

II. COMPUTATION MODEL

In this section, we present the computation model of the proposed GPregel. GPregel is a Pregel-like system and it has a homologous computation model with Pregel. We evolve the computation model to suit the features of GPU. The evolvement avoids some high cost operations to insure the expected performance without hurting the correctness of system.

We first introduce the definition of directed graph structure since it is the main process object for GPregel. A directed graph is described as G=<V, E>, where V and E indicate the set of vertexes and the set of edges of graph G, respectively. Fig. 1 shows a sample of directed graph with 6 vertexes and 10 directed edges. The reason why we focus on processing directed graph is that the relationship of the real world can be considered as a directed graph and an undirected graph is transformed into directed graph form easily.

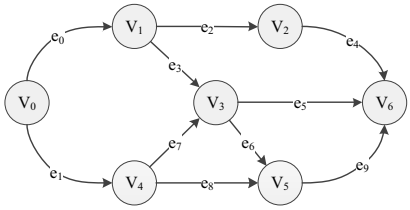

Fig. 1. A Directed Graph Sample

Similar to Pregel, GPregel also uses BSP model for message passing. Computation is consisted of many super-steps. At each super-step, GPregel calls a user-defined function, named *Compute()*, for each vertex. This function first reads the message sent to a vertex in the previous super-step, and then sends message to the current vertex's neighbors (for next super-step to be received), or mutates the value of the vertex and even its outgoing edges. In each super-step, for all vertexes, *Compute()* function executes concurrently. To guarantee all the *Compute()* function has finished its task, a barrier synchronization would be executed after each super-step.

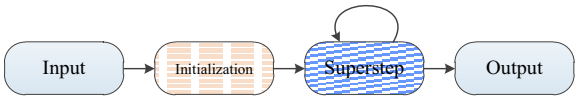

Fig. 2. The Architecture of GPregel

A typical GPregel computation model includes four main steps: input, initialization, a series of super-steps and output. The data of a graph is stored in file format. Input file has all the information of vertexes and edges for computing, while output file contains the result. GPregel supports multifile in input and output process. GPregel takes advantage of multi-thread to read or write files to improve the system throughput and avoid I/O bottleneck. For a more efficient way, it is easy to combine Google's Protocol Buffers or Facebook's Thrift to provide file compression and optimization [16, 17]. Initialization changes the input file to system storage structure by sorting and indexing the input data.

We use an example to illustrate the computation process in GPregel. As shown in Fig. 4, the given graph where each vertex has a value and we need to find out the minimal value of all its predecessor nodes. Message is sent along the edges. In each super-step, vertexes use the minimal value received from messages to update its current value. Only the vertex with a new value will send messages. When no further messages are created, the system changes to inactive state and the algorithm terminates.

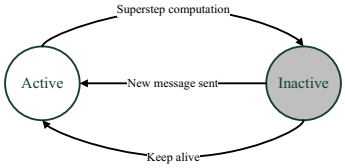

Fig. 3. Vertex State Machine

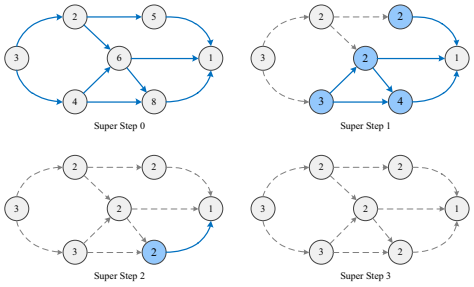

Fig. 4. An Example of Maximum Value. Dotted lines are messages. Shaded vertexes have voted to halt.

III. PROGRAMMING INTERFACE

The computational logic of the above example is simple, but developer will find it difficult to implement on the GPU. This is because developer need to manage the memory of GPU as well as configure the thread parameters, handle thread synchronization and elaborately design the data structure of a graph. The purpose of developing GPregel is to simplify the above operations.

GPregel simplifies the API to graph structure definition and *Compute()* function definition, which is more similar to Pregel's API and Pregel code can easily transplant to GPregel. Using GPregel for different graph algorithms, developers need to define different graph structure in our four structures as *Global*, *Vertex*, *Edge* and *Message.*

3.1 Graph Structure Definition

Developer can use a C++ style code to define the graph structure and some useful information in *Global*, *Vertex, Edge* and *Message* structures. The user defined value will be added to the system defined code after compilation. All the data about the graph are stored in input file. This file can be separated into many small files and GPregel makes use of multi-thread to read them in the beginning.

Global structure consists of the metadata and the number of vertex and edge and other global information. GPregel will define the number of vertex and edge by default, and developer can add the other values by defining them into *Global* struct.

Vertex structure consists of the property of each vertex. Similar to *Global, Vertex* struct has its own default member variables including vertex id, the number of in-edge and out-edge. Developer can define their own values in *Vertex* struct such as the value of a vertex and minimum value of its predecessor nodes.

Edge structure contains the directed-edge information. GPregel defines *from* and *to* member in advance, which represent the vertex id that the edge is coming from and pointing at, respectively. The same with the *Vertex* struct, developers can define their own member variables, such as the weight of an edge, and then use 'in' or 'out' key word to decorate them. GPregel will generate the *Get()* and *Set()* function accordingly.

Message structure is designed to contain the message information that will transmit in every super-step. Different from the structures we discussed above, GPregel does not define the default member variable for developer, because the message sent in each super-step varies in different algorithm. Developer do not need to use the "in" or "out" modifier for the member in *Message* struct.

3.2 Compute() Function Definition

Fig. 5 and Fig. 6 shows the structures that developers need to define for the minimum precursor example. GPregel compiler then will generate five data structures for developers, contain *Vertex, Edge, OutEdgeIterator, Message,* and *MessageIterator*. *Vertex* includes the API to access global information and all the members in one vertex. *Edge* includes the API to access the members in one directed edge. *OutEdgeIterator* includes the API to access all out-edges of a vertex. *Message* includes the API to create message in each super-step. *MessageIterator* includes the API to read the received message. It is worth noting that the Message constructor needs an edge to be a parameter, for message is bound to an edge. Also, GPregel limits one edge can only send one message and the last message will overwrite the previous messages.

Compute() function is defined in the Vertex struct as a member function, so developer can call all the other member functions in Vertex struct inside *Compute()*. The benefit is that developer can focus on current vertex and different vertexes will not disturb each other. Developers do not need to consider which vertex is being manipulated and the state of other vertexes. GPregel will pass the *MessageIterator* as a parameter to *Compute()* function and all APIs generated by GPregel. Developer can traverse the message iterator to get all the messages and handle them in one super-step. Here the minimum precursor example is used to illustrate how to write the *Compute()* function. As shown in Fig. 7, we traverse all the message sent to one vertex and set the minimum value in line 3~7. Then we send the minimum value to all out-edges with this vertex. After a series of super-steps, we can get the minimum precursor value of every vertex. Developer just need to write few lines of code to complete this task and never need to worry about the complexity of GPU and write code to manage the low level details. GPregel can greatly reduce the learning threshold and the cost for developing an efficiency parallel algorithm.

```
struct Vertex {
                //  the number of vertex
                    in int value;
                // the minimum of ancestor value
                out int min_ancestor_value = (~0U) >> 1;
                };
```

Fig. 5. User-defined Vertex Construe

```
struct Message {
                // the ancestor value of vertex
                int value;
                };
```

Fig. 6. User-defined Message Construe

```
1.  __device__ void Compute(MessageIterator* msgs) {
2.   int min_val = (SuperStep() == 0 ? get_value() :
     get_min_ancestor_value());
3.   for (; !msgs->Done(); msgs->Next()) {
4.    if (msgs->get_value() < min_val) {
5.     min_val = msgs->get_value();
6.    }
7.   }
8.   if (min_val < get_min_ancestor_value()) {
9.    set_min_ancestor_value (min_val);
10.   for (OutEdgeIterator it = GetOutEdgeIterator(); !it.Done(); it.Next()) {
11.    Message msg(*it);
12.    msg.set_value(min_val);
13.    msg.Send();
```

Fig. 7. A Sample of Compute() Function

IV. SYSTEM DESIGN

In this section, we will discuss the system architecture and implementation of GPregel in detail. To pressure the high programmability of GPregel within the efficiency, GPregel goes deeply into the storage model and designs the internal algorithm elaborately. Developers can experience the convenience and high-performance while taking GPregel as a black box and do not need to consider the underlying architecture. First, we will describe the basic storage mode for GPregel, which is the key part of GPregel. Because GPU cannot communicate with hard disk and peripheral equipment directly, it is inevitable to use CPU for scheduling, so that it

will affect the system performance greatly if we store the data in peripheral equipment. Hence, we load all of the data into the memory of GPU. In this premise, how to organize the data becomes the key issue of GPregel. We then will explain how GPregel compiles the developer's code to work with CUDA.

4.1 Storage Model

The most important information is a graph is about the edges and vertexes. In GPregel, to realize the communication between different super-steps, we also need an extra *Message* structure to keep the message sent in each super−step. Firstly consider the *Vertex* structure. The members of different vertexes are stored in a multiple dimensions array. Each dimension represents one member in each vertex. Suppose *Vertex* struct has *k* members, which are *M1, M2, ..., Mk,* respectively. The storage structure is shown in Fig. 8. We use the example in Section II to show the storage model of *Vertex.* This kind of data structure can take full advantage of the Coalesced Memory Access [18] characteristic in CUDA. Assume there is an instruction that read for the *i*-th member of *Vertex.* It will be run in *Compute()* function by every thread and all this request will fall on the successive positions. So Coalesced Memory Access can be achieved effectively and therefore requests in a warp will combine into one, which helps to improve the utilization ratio of memory bandwidth. Such scheme can also be applied to struct Edge and Message.

After defined the storage structure of Vertex, Edge and Message, we also need to design the relationship to organize them to solve the graph algorithm. Fig. 9 shows the basic storage model of GPregel. At step 1, we set a fixed size message buffer, which is called it msg_recv. Every massage is mapped with a directed edge and all the messages are sorted according to the terminal vertex id. At step 2, there is an Edge array that is called Outgoing Edges. All this outgoing edges are sorted according to its initial vertex id. Also, in the *Vertexes* array, every vertex will set two subscripts in addition, which indicate the ranges of its outgoing edges in *Outgoing Edges* array. At step 3, set a sent message buffer, called *msg_send.* All messages sent over the *Outgoing Edges* will shuffle to the right place sorted by the terminal vertex id. The messages with the same receiver will be arranged together. At step 4, GPregel copies the *msg_send* to *msg_recv* automatically, and *Compute()* function can use such information in next super-step. As illustrated in Fig. 8, we can see the whole procedure of one super-step. Developers define the *Compute()* function worked at step 1 and step 2. Step 3 will run automatically in GPregel after message is sent in *Compute()* function. If the *Compute()* function has finished already, step 4 will also be run in GPregel automatically. There are many methods that need to implement the shuffle procedure at step 4, so we use a *Shuffle Map* to indicate which position from the outgoing message should be placed in *msg_send* buffer. Fig. 9 shows that the storage model using minimal predecessor example in Fig. 4.

Vertexes

Vertex Indexes	0	1	2	3	4	5	6
id	0	1	2	3	4	5	6
in_edge_count	0	1	1	2	1	2	3
out_edge_count	2	2	1	2	2	1	0
value	3	2	5	6	4	8	1
min_ancestor_value	-	-	-	-	-	-	-

Fig. 8. A Sample of the Storage Model of Vertexes

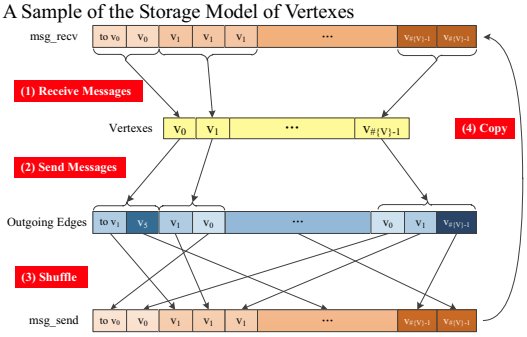

Fig. 9. A Sample of Basic Storage Model

4.2 Compiler Implementation

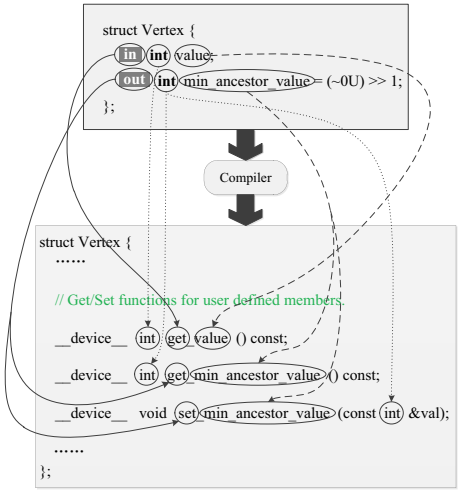

Fig. 10. The Flow of Compiler

To achieve good system performance, GPregel should provide brief API and avoid much repetitive work. However, it conflicts with the complex logic within the system. We design a compiler to tackle the problem. A complier helps to transform developers-defined data structure into system internal data structures, and generates corresponding API in *Compute()* function. Without such a compiler, developers would have to write much repetitive codes. In addition, compiler can be extended as system demands. Fig. 18 shows the implementation of GPregel's compiler. We use regular expression to do this. Take *Vertex* for example, we will have a regular expression to match and know this is a vertex. Another regular expression will match the member of *Vertex* and generate *Get()* and *Set()* function according to the key word "in" or "out". To make this compiler more efficiently, the logic code can be written by Python. Though compiler in GPregel is a lightweight compiler, it plays a significant role to reduce developer's work.

V. PERFORMANCE EVALUATION

We have conducted extensive experiments to evaluate performance and availability of GPregel. We first introduce test hardware and software environment，as well as test data sets. We then evaluate system performance in the following

two aspects: (1) the efficiency of general algorithms executing is compared between GPregel and CPU-based computing, (2) the performance is compared between GPregel and the existing GPU-based systems. These experiments and analysis indicate GPregel achieves its fundamental objectives, and developers can take advantage of the computing power of GPU efficiently in GPregel.

5.1 Environment and Test Data Sets of Experiments

We use NVIDIA's Tesla C2050 GPU and Intel's Xeon E5620 CPU in our experiments. The test data sets include R-MAT and Random type graph generated by GTgraph tool [2]. R-MAT is a sort of graph type as similar as Small-world network. Random is a sort of Erdős-Rényi random graph type. In addition, the test sets include WikiTalk and RoadNet-CA as well.

TABLE I. CHARACTERISTICS OF GRAPHS USED IN THE EXPERIMENTS

Graph	Vertexes ($\times 10^6$)	Edges ($\times 10^6$)	Max d	Avg d	σ
R-MAT	1.0	16.0	1742	16	32.9
Random	1.0	16.0	38	16	4.0
WikiTalk	2.4	5.0	100022	2.1	99.9
RoadNet-CA	2.0	5.5	12	2.8	1.0
BIP	4.0	16.0	19	4	5.1

In our experiment, three kinds of the execution time of GPregel system are be measured, including the time of pretreatment time-A, and the time of Compute() function executive time-B, and the time of operations in super-steps except Compute() function time-C (including the time of copy news array and the time of initiate each super-step).

5.2 Compare with CPU

Three computing-intensive algorithms, SSSP, BFS and PageRank, have been used in the experiment. PageRank algorithm enables us to effectively analyze the performance between GPU and CPU in computing-intensive algorithms. On the other hand, SSSP and BFS are data-intensive algorithms. We use both of computing-intensive algorithms and data-intensive algorithms to compare the efficiency of general algorithms between GPregel and CPU-based computing.

As illustrated in Fig. 11, in the execution time of the SSSP part, GPregel has a significant speedup than CPU-based computing except for R-MAT. The reason for this result is that R-MAT is highly irregular, and GPregel is not good at dealing with this type of the graphs. Combining with these two factors, the execution time in CPU-based computing and GPregel is similar. WikiTalk speedup is the best, almost 6 times.

In the experiments results on BSF, GPregel is ahead of the curve with the CPU-based computing except for RoadNet-CA, speedups of up to 6 times (Random). Because the BFS algorithm of the CPU-based computing is a linear algorithm and the diameter of RoadNet-CA is large, GPregel runs slower than CPU-based computing.

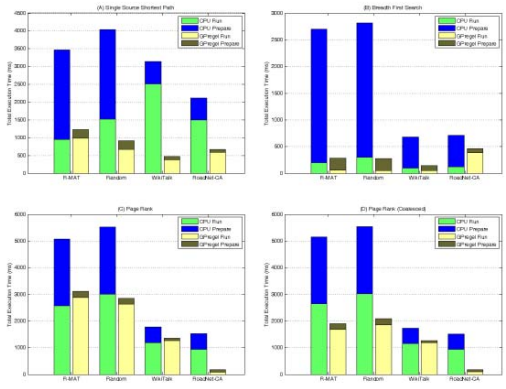

Fig. 11. Execution time of GPregel and CPU-based computing

In the experiment of PageRank algorithm, apart from RoadNet-CA, CPU-based computing and GPregel have similar execution time without using the coalesced memory accessing. Coalesce memory accessing can greatly improve the memory access performance of the GPU memory access. As a result, GPregel can speed up the computing strongly.

5.3 Compare with Existing Systems

In this section, we compare GPregel and other existent GPU-based systems. The existent GPU-based systems, we choose medusa and solutions from [9]. We call the solutions from [9] as Basic. Because Medusa is hardly deployed in our experiment environment, we use Basic as a reference to compare with Medusa. The execution time contains time-B and time-C. BFS algorithm is used in the experiment, and experimental results are shown in Fig. 12.

From the results, we can see that GPregel is a more stable system in SSSP condition. The efficiency does not change large with the type of graph. It is very important for a system according to the experimental results. The worst-case execution time of GPregel is not over 2.3 times longer than the Basic system (R-MAT). In WikiTalk data sets test, GPregel is even faster than Basics system.

Overall, we demonstrate that GPregel can achieve high performance with hiding underlying the GPU programming details. It is due to the enormous amount of optimization in GPregel.

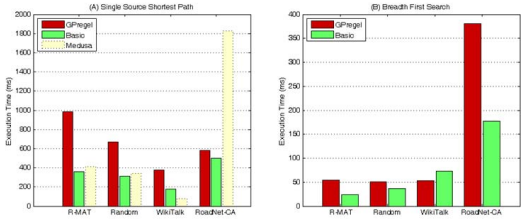

Fig. 12. Execution time of GPregel and existent systems

5.4 *Analysis of convenience*

Convenience is the fundamental goal in our GPregel system. However, Convenience is difficult to quantify. Here we use

certain indicators of GPregel system programming features, as shown in TABLE II.

We can get from TABLE II that GPregel greatly simplifies the calculation program design based on GPU by providing the brief API like Pregel to hides the GPU's parallel characteristics. For example, to achieve the SSSP and BFS algorithms, developers only need to write 14 lines and 10 lines of codes. However in complete implementation of [9], you need at least 62 and 58 lines of codes. Furthermore, implementation of each algorithm contains a coalition of 18 or more source files directory in Medusa.

TABLE II. CODING COMPLEXITY OF GPREGEL AND EXISTENT SYSTEMS

	Basic	Warp-centric	Medusa	GPregel
GPU code lines (SSSP)	61	N.A	13/11	14
GPU code lines (BFS)	58	76	9/7	10
GPU memory management	Yes	Yes	No	No
Kernel configuration	Yes	Yes	No	No
Parallel programming	Thread	Thread+Warp	No	No
Define workflow	Yes	Yes	Yes	No

From learning threshold, developers do not need to understand the GPU programming, memory management, or define the entire work process when GPregel is used, while other system require more or less of the knowledge. For example, after removing the duplicated code, Medusa system still needs to explicitly manage the memory, and also requires the developer to define a calculated process.

In fact, compared with the complete implementation and GPregel is not quite fair. The main problem is that the two have different optimization targets. GPregel's goal is to provide good programmability under acceptable performance. However, complete implementations typically do not take into account for ease of programming. In complete implementations, a special algorithm takes specific optimizations for high performance and these optimizations are not universally applicable usually. In summary, the main goal is to explain that GPregel not only greatly enhances programmability, but also provides high performance.

VI. CONCLUSIONS

Aiming at solving inefficient and difficult questions in the GPU-based parallel computing developing environment, we have proposed GPregel that is completed implementation of the parallel computing system based on GPU. GPregel is inspired by Pregel. The system provides the simplest user interface, contains many optimization algorithms based on GPU-specific features and particular figure, and presents multiple parallel algorithms. By designing the compiler based on regular expression and hiding the details of GPU programming and parallelism, it greatly reduces the burden on developers and improves development efficiency. Experimental results show that the GPregel greatly reduces the cost of developing a parallel graph based on GPU computation program.

The source code of GPregel is available at https://code.google.com/p/GPregel/.

ACKNOWLEDGMENT

This paper is supported by the National Natural Science Foundation of China under Grant No. 61472454.

REFERENCES

[1] M. DeLorimier, N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T.E. Uribe, Jr. T.F. Knight, and A. DeHon, GraphStep: A System Architecture for Sparse-Graph Algorithms, *null*, Year Published, pp. 143-151.

[2] D.A. Bader, and K. Madduri, GTgraph: A Synthetic Graph Generator Suite, *Atlanta*, 2006.

[3] G. Malewicz, M.H. Austern, A.J.C. Bik, J.C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, and G. Inc, Pregel: a system for large-scale graph processing, *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 135-146.

[4] J. Zhong, and B. He, Medusa: Simplified Graph Processing on GPUs, *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, no. 6, 2014, pp. 1543-1552.

[5] L.G. Valiant, A Bridging Model for Parallel Computation., *Communications of the ACM*, vol. 33, no. 8, 1990, pp. 103-111.

[6] Apache Incubator Giraph, http://incubator.apache.org/giraph/.

[7] GoldenOrb, http://www.raveldata.com/goldenorb/.

[8] S. Salihoglu, and J. Widom, GPS: A Graph Processing System *, *Stanford InfoLab*, 2013.

[9] Pawan Harish, and P.J. Narayanan, Accelerating Large Graph Algorithms on the GPU Using CUDA, *Lecture Notes in Computer Science*, 2007, pp. 197-208.

[10] G. He, H. Feng, C. Li, and H. Chen, Parallel simrank computation on large graphs with iterative aggregation, *KDD'10*, 2010.

[11] G.J. Katz, and J.T. Kider Jr, All-pairs shortest-paths for large graphs on the GPU, *In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, 2008, pp. 47-55.

[12] V. Vineet, and P.J. Narayanan, CUDA cuts: Fast graph cuts on the GPU, *in Computer Vision and Pattern Recognition Workshops. IEEE Computer Society*, Year Published, pp. 1-8.

[13] S. Hong, S. Kyun, K. Tayo, and O.K. Olukotun, Accelerating CUDA graph algorithms at maximum warp., *In PPoPP*, vol. 46, no. 8, 2011, pp. 267-276.

[14] L. Luo, M. Wong, and W. Hwu, An effective GPU implementation of breadth-first search, *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, Year Published, pp. 52-55.

[15] D. Merrill, M. Garland, and A. Grimshaw, High-Performance and Scalable GPU Graph Traversal, *In 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP ' 12*, vol. 47, no. 8, 2011, pp. 117-128.

[16] Protocol Buffers - Google's data interchange format, https://code.google.com/p/GPregel/.

[17] Apache thrift, http://thrift.apache.org/

[18] Nvidia, CUDA C Programming Guide version 4.0, *http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf [Online; accessed May 10 2011]. 63*, 2011.