2013 IEEE International Conference on Cloud Computing Technology and Science

Towards GPU-Accelerated Large-Scale Graph Processing in the Cloud

Jianlong Zhong, Bingsheng He
Nanyang Technological University, Singapore

***Abstract*—Recently, we have witnessed that cloud providers start to offer heterogeneous computing environments. There have been wide interests in both clusters and cloud of adopting graphics processors (GPUs) as accelerators for various applications. On the other hand, large-scale graph processing is important for many data-intensive applications in the cloud. In this paper, we propose to leverage GPUs to accelerate large-scale graph processing in the cloud. Specifically, we develop an in-memory graph processing engine *G2* with three non-trivial GPU-specific optimizations. Firstly, we adopt fine-grained APIs to take advantage of the massive thread parallelism of the GPU. Secondly, G2 embraces a graph partition based approach for load balancing on heterogeneous CPU/GPU architectures. Thirdly, a runtime system is developed to perform transparent memory management on the GPU, and to perform scheduling for an improved throughput of concurrent kernel executions from graph tasks. We have conducted experiments on an Amazon EC2 virtual cluster of eight nodes. Our preliminary results demonstrate that 1) GPU is a viable accelerator for cloud-based graph processing, and 2) the proposed optimizations improve the performance of GPU-based graph processing engine. We further present the lessons learnt and open problems towards large-scale graph processing with GPU accelerations.**

***Keywords*-Large-scale graph processing, GPGPU, graph partitioning, cloud computing, GPU accelerations**

I. Introduction

Large-scale graph processing has become popular for various data-intensive applications on increasingly large web and social networks. Due to the ever increasing sizes of graphs, many applications host their graph processing tasks in the cloud with a large number of commodity servers [1], [2]. Many processing tasks are batch operations in which many vertices and/or edges of the graph are accessed within a task. Recent studies [1], [3], [4] have developed general programming engines for such batch graph processing tasks on the large-scale graph, and demonstrated very good performance and scalability in the cloud environment. However, existing studies mainly focus on upper-level software technologies for performance optimizations, and little work has been done on exploiting hardware architectural features of cloud computing. Therefore, in this paper, we investigate whether and how we can further improve the efficiency of those general graph processing engines with the consideration of hardware architectural features in the cloud.

Cloud has evolved into a platform with heterogeneous architectures. One of the representative heterogeneous architectures in the cloud is graphics processing units (GPUs). Amazon and Penguin have provided virtual machines with GPUs. In addition to the public cloud, two out of the top ten supercomputers are with GPUs integrated (according to Top500 list of June 2013). GPUs have become an effective accelerator for a wide range of applications from computation-intensive applications (e.g., [5], [6]) to data-intensive applications (e.g, [7], [8]). Compared with multicore CPUs, new-generation GPUs can have much higher computation power in terms of FLOPS and memory bandwidth. For example, an NVIDIA Tesla C2050 GPU can deliver the peak single precision floating point performance of over one Tera FLOPS, and memory bandwidth of 144 GB/s. In specific to graph processing, the GPU has been used as an accelerator for various graph processing applications [9], [10], [11], [12]. Encouraged by the significant performance improvement of graph processing by a single GPU, this paper attempts to integrate GPUs into distributed graph processing engines.

Most current graph engines (e.g., [1], [3], [4]) are based on (almost) the same programming model, which mainly adopts vertex-oriented APIs for users to develop their graph processing tasks. For example, Pregel [1] offers the user-defined API $Compute()$ executed on vertices, and executes $Compute()$ on all the vertices in parallel. This simple vertex-oriented programming model can be applicable to many graph processing operations. Thus, our research objective is to improve the efficiency of vertex-oriented programming model in the cloud with GPU accelerations.

Particularly, we develop the following key optimization techniques for the efficiency of GPU-accelerated vertex-oriented graph processing systems:

- *A GPU-based graph processing system architecture with fine-grained API support*: There are two key design decisions on our proposed system: 1) fine-grained API design, and 2) in-memory graph processing. The single vertex-based API design of current systems like Pregel [1] offers only coarse-grained parallelism, which is inefficient for the massive thread parallelism of the GPU. We decide to support fine-grained APIs at the granularity of individual vertices, edges and messages. The second decision on in-memory processing is to avoid the costly random accesses to the hard disk so that the GPU advantage can be maximized. This forms two levels of parallelism on heterogeneous systems:

978-0-7695-5095-4/13 $31.00 © 2013 IEEE
DOI 10.1109/CloudCom.2013.8

massive thread parallelism within a GPU/CPU and the parallelism across CPUs/GPUs.

- *Load balancing for heterogeneous platforms*: We are facing the cloud system with heterogeneous platforms where load balancing is critical for the overall performance. Thus, we develop a graph partitioning framework that leverages the popular of graph partitioning algorithm to adapt to the heterogeneity of the (virtual) cluster. Considering in-memory processing, we develop a simple and effective load-balanced approach for assigning graph partitions to different machines.
- *A runtime system for automatically optimizing the memory transfer between the main memory and the GPU memory*: A large-graph processing system usually has many concurrent graph processing tasks that can perform on the same graph or on different graphs. We develop a runtime system for automatically managing the data transfer between the main memory and the GPU memory, and for scheduling the GPU kernels with complementary resource usage for optimized throughput.

As a start, we develop a system prototype named G2 to realize all the GPU-based optimization techniques. G2 is based on MPI (Message Passing Interface), and integrates the vertex-based programming model and GPU accelerations. G2 offers the same programming interfaces as Medusa [12], a graph processing framework on the GPUs within a single machine. As a start, we have evaluated the efficiency of G2 with synthetic graphs in a virtual cluster of eight nodes on Amazon EC2. The preliminary results demonstrate that 1) each of the proposed techniques improves the effectiveness of the GPU acceleration; 2) putting them all together, G2 achieves a significant performance improvement of 50% over the CPU-only approach. To the best of our knowledge, G2 is the first attempt in building a GPU-accelerated large-scale graph processing engine. We believe that the experiences and methodologies in this study shed light on the design and implementation of next-generation graph processing engines.

Organization. The reminder of this paper is organized as follows. Section II introduces the background and related work. Section IV presents the system overview, followed by the design and implementation in Section IV. We present experimental results in Section V. We discuss the lessons and insights in Section VI, and conclude this paper in Section VII.

II. Background and Related Work

In this section, we introduce the background on the GPU and its applications in clusters and cloud computing. Next, we review the related work on general graph processing engines.

A. GPU

GPUs have rapidly evolved into a powerful accelerator for many applications, especially after CUDA was released by NVIDIA. This paper focuses on the design and implementation with NVIDIA CUDA. G2 takes advantage of the concurrent kernel execution capability of new-generation GPUs. With the introduction of CUDA, a GPU can be viewed as a many-core processor with a set of streaming multi-processors (SM). Each SM executes the instructions in the SIMD (Single Instruction Multiple Data) manner. The SMs are in turn executed in the SPMD manner. The program is called *kernel*.

In CUDA's abstraction, GPU threads are organized in a hierarchical configuration: usually 32 threads are firstly grouped into a warp; warps are further grouped into thread blocks. The CUDA runtime performs mapping and scheduling at the granularity of thread blocks. Each thread block is mapped and scheduled on an SM, and cannot be split among multiple SMs. Once a thread block is scheduled, its warps become active on the SM. Warp is the smallest scheduling unit on the GPU.

In both cluster and cloud environments, GPUs are often shared by many concurrent GPU kernels (most likely submitted by multiple users). To enable sharing GPUs remotely, a number of software frameworks such as rCUDA [13] have been developed. Recently, new-generation GPUs like NVIDIA Fermi GPUs support concurrent kernel executions. Taking advantage of this new capability, a number of multi-kernel optimization techniques [14], [15] have been developed to improve the utilization of GPUs. Ravi et al. [16] proposed kernel consolidation to enable space sharing (different kernels run on different SMs) and time sharing (multiple kernels reside on the same SM) on GPUs. In contrast, G2 utilizes slicing the kernels into multiple *slices* to create more opportunities for time sharing. GPU virtualization has also been investigated [15] in the previous studies. GPU accelerations have also been considered in large-scale data-intensive applications including MapReduce [17] and data mining [18]. While those applications are generally applicable to graph processing, the optimization techniques of G2 are specially redesigned for large-scale graph processing in the cluster/cloud environment.

B. General Graph Processing Engines

Large graphs have arisen in a wide range of data-intensive applications like social networks and Web. In social networks, nodes often represent users and edges may often represent relationships between users (friendship). Nowadays, there are plenty of large social networks. For example, the social network of Facebook consists of a billion nodes and more than a hundred billion edges in 2012 [19]. LinkedIn contains almost 218 million nodes in the first quarter 2013 [20]. Moreover, social networks are evolving

in an unprecedented rate. For example, it has been reported that the size of the Facebook network has increased from roughly one million users in 2004 to one billion users in 2012 [19].

Distributed and parallel algorithms have been a classical way to improve the performance of graph processing. On multi-core CPUs, parallel libraries such as MTGL [21], [22] have been developed for parallel graph algorithms. To facilitate developing distributed graph algorithms in the cluster/grid settings, software libraries such as Parallel BGL [23] have been developed.

Emerging platforms including cloud computing and GPUs are becoming popular for graph applications. Various specific parallel graph algorithms have been developed and optimized in the cloud [2] and on the GPU [9], [24], [25]. Recently, vertex-oriented general graph processing engines are developed in the cloud [1], [26], [3], [27] or on the GPU [12]. However, none of them has leveraged the power of cloud computing and GPUs as a whole. G2 is the first attempt to bridge this gap.

The vertex-oriented programming model is based on the key observation from previous studies [28], [29], [4] that many common graph algorithms can be formulated using a form of the bulk synchronous parallel (BSP) model. Local computations are performed on individual vertices. Vertices are able to exchange data with each other. The same computation and communication procedures are executed iteratively with barrier synchronization at the end of each iteration. This common algorithmic pattern is also adopted by distributed graph processing frameworks such as Pregel [1] and distributed GraphLab [26]. For example, Pregel applies a user-defined function *Compute()* on each vertex in parallel in each iteration of the BSP execution. The communications between vertices are performed with message passing interfaces. Hama [30] and Giraph [27] are two open-source projects targeting at large graph processing. Distributed GraphLab is different from other engines in that it supports asynchronous executions whereas others are synchronous. Synchronous executions are simple and suitable for the GPU architecture, since asynchronous executions require complicated concurrency control. For those reasons, G2 chooses to support synchronous executions.

On the GPU, Medusa [12] shares the same design goal as Pregel in providing a programming framework to ease development of graph algorithms, and in hiding the complexity of the underlying runtime from developers. G2 adopts the fine-grained API from Medusa, extends the support to multiple graph partitions and supports two levels of parallelism for graph processing.

III. System Overview

In this section, we present the rationales on the G2 design, followed by a system overview. We present the details on the key optimization techniques for G2 in Section IV.

A. Design Rationales

A general large-scale graph processing engine should have the following three key features:

Performance: Since we mainly consider supporting batch graph processing tasks, throughput is the key optimization metric for performance. Also, a high throughput leads to high performance and productivity as well as low total ownership cost. On the other hand, we attempt to reduce the response time of each job, if our optimization technique can significantly reduce the response time without significant throughput degration.

Scalability: Due to the increasing data volume, the effectiveness of how a system handle such big and complex graphs is important.

Programmability: Parallel and distributed programming is considered as a more challenging task than sequential programming, and even more challenging for graph tasks. Programmability measures the convenience of the programming interfaces that the graph processing systems expose to users.

To develop a large-scale graph processing system with high performance, scalability and programmability, we have the following design rationales.

Firstly, encouraged by the success of GPU accelerations in graph processing [9], [10], [11], [12], we argue that the GPU should be considered as a viable architectural support for high performance of large-scale graph processing. However, using the GPU alone does not guarantee high performance. Moreover, compared with CPUs, GPUs are still expensive devices. Thus, we need throughput-oriented optimizations for graph processing on CPU/GPU architectures.

Secondly, we choose in-memory processing, rather than in external storage. The DRAM price has dropped significantly, almost with double-digit yearly drops in the past decade. In-memory data processing becomes a popular paradigm for data-intensive applications. As specific for graph processing systems, we have two other arguments: 1) accesses to graph data are mostly random, and external storage has very poor random accesses; 2) compared with other data types such as texts and multimedia, graphs are relatively small, usually in the scale of TBs (e.g., assuming each edge takes eight bytes for a hundred billion edges of Facebook graph [19]).

Thirdly, even the baseline graph processing engine should store the graph into partitions, as opposed to a flat storage. A general graph processing engine should allow the flexibility of adopting a new graph partitioning algorithm. Moreover, with the GPU-enabled processing, graph partitioning should adapt to the heterogeneity of CPU/GPU architectures.

Lastly, we adopt the vertex-oriented programming model for programmability, as many general large-graph processing engines do. The vertex-oriented programming model can express many graph operations [1], [26], [3], [27].

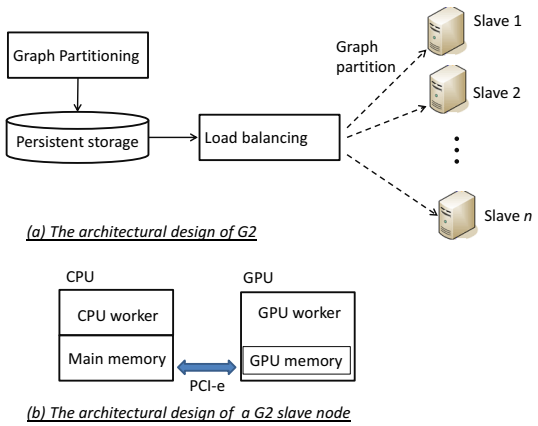

Figure 1. G2: a GPU-accelerated in-memory graph processing engine.

B. Architectural Design Overview

We develop a GPU-accelerated large-scale graph processing engine named *G2*. G2 supports a GPU-friendly fine-grained API for exploiting the massive thread parallelism of GPUs (Section IV-A).

G2 shares many similar designs as other synchronous engines. For example, its execution is based on BSP, and messages are the main mechanism for inter-vertex information exchange. Figure 1(a) shows the architectural design of G2. Taking a graph as input, G2 first performs graph partitioning according to the specified graph partitioning algorithm $\mathbb{A}$. Suppose $\mathbb{A}$ can divide the graph into k partitions at one time. We may use $\mathbb{A}$ recursively so that one graph partition can reach a fine granularity (Section IV-B). This paper uses the graph partitioning algorithm in Surfer [3], which is an enhanced version of metis [31]. With the message passing and in-memory processing requirements, we find that MPI (Message Passing Interfaces) is a good start for building a system prototype of G2 on top of Medusa [12].

Graph partitions are stored in persistent storage (such as Amazon EC2) for future usage. Prior to executing graph processing tasks, G2 starts n slave nodes and loads the graph partitions into the main memory of each slave nodes. Load balancing is considered in this loading process.

Figure 1(b) shows the architectural design of a slave node. It is a heterogeneous platform, with one CPU worker and one GPU worker to handle graph processing tasks on the CPU and on the GPU, respectively. The CPU worker and the GPU worker are independent with each other, and execute multi-threaded Pthread and CUDA programs, respectively. For GPU processing, graph partitions are stored in the main memory initially, and is copied to the GPU memory when necessary. G2 embraces a runtime system to management the data transfer and concurrent kernel executions on the GPU (Section IV-C). Each of the CPU and the GPU workers are implemented as a MPI process.

IV. Design and Implementation of G2

This section presents three key optimization techniques for G2, including two-level parallelism with fine-grained API support, load balancing, and runtime system.

A. Two-Level Parallelism

Basically, G2 consists of two-level parallelism: multiple-node processing at the coarse-grained level, and thread parallelism within a CPU/GPU at the fine-grained level. Existing vertex-oriented engines such as Pregel mainly expose the coarse-grained parallelism. They cannot fully utilize the massive thread parallelism of the GPU. Thus, G2 supports fine-grained APIs that are suitable for BSP execution model on the CPU/GPU. The basic idea is that users define the APIs with the granularity of vertices/edges/messages, and the G2 system automatically executes those APIs on graph partitions.

As a start, we adopt the graph processing APIs of Medusa [12]. Medusa is designed for graph processing on the GPUs within a machine. It supports the APIs namely EMV APIs, which enhance the current single vertex-based API design to support efficient and fine-grained graph processing on the GPU. For completeness, we briefly introduce the Medusa APIs, and more details can be found in the paper [12].

In order to write a graph processing program in G2, developers simply implement their codes very similar to those in Medusa. A program basically consists of two kinds of APIs, namely EMV APIs and system-provided APIs.

First, Medusa provides six device code APIs for developers to write GPU graph processing algorithms. Each API is either for processing vertices ($VERTEX$), edges ($ELIST$, $EDGE$) or messages ($MESSAGE$, $MLIST$), as shown in Table I. Using these APIs, programmers can define their computation on vertices, edges and messages. The vertex and edge APIs can also send messages to neighboring vertices. The idea of providing these APIs is mainly for efficiency. It decouples the single vertex API into separate APIs which target individual vertices, edges or messages. Each CPU/GPU thread executes one instance of the user-defined API. The thread configuration such as the number of threads is tuned to maximize CPU/GPU utilization. The fine-grained data parallelism exposed by the EMV model can better exploit the massive parallelism of the GPU.

Second, Medusa hides the GPU-specific programming details with a small set of system provided APIs, as shown in Table II. Particularly, Medusa provides $EMV < type >:: Run()$ to invoke the device code API, which automatically sets up the thread block configurations and calls the corresponding user-defined function. Medusa allows developers to define an *iteration* which executes a sequence of $EMV < type >:: Run()$ calls in one host function (invoked by $Medusa :: Run()$). The iteration is performed iteratively until predefined conditions are

Table I
USER-DEFINED APIS IN THE EMV MODEL

API Type	*Parameters*	*Variant*	*Description*
$ELIST$	Vertex v, Edge-list el	Collective	Apply to edge-list el of each vertex v
$EDGE$	Edge e	Individual	Apply to each edge e
$MLIST$	Vertex v, Message-list ml	Collective	Apply to message-list ml of each vertex v
$MESSAGE$	Message m	Individual	Apply to each message m
$VERTEX$	Vertex v	Individual	Apply to each vertex v
$Combiner$	Associative operation o	Collective	Apply an associative operation to all edge-lists or message-lists

Table II
SYSTEM PROVIDED APIS AND PARAMETERS IN MEDUSA

API/Parameter	*Description*
$AddEdge$ (void* e), $AddVertex$(void* v)	Add an edge or a vertex into the graph
$InitMessageBuffer$(void* m)	Initiate the message buffer
$maxIteration$	The maximum iterations that Medusa executes ($2^{31}-1$ by default)
$halt$	A flag indicating whether Medusa stops the iteration
$Medusa$ :: Run(Func f)	Execute f iteratively according to the iteration control
$EMV<type>$:: Run(Func f')	Execute EMV API f' with $type$ on the GPU

satisfied. Medusa offers a set of configuration parameters and utility functions for iteration control.

In summary, the fine-grained API of Medusa is suitable for G2, which offers simple yet effective CPU/GPU co-processing to exploit both processors.

B. Load balancing

There are a few considerations for load balancing of G2.

Firstly, slave nodes in G2 may have different processing capabilities and memory capacities. As shown in Figure 1, G2 runs on heterogeneous architectures. The heterogeneity can come from 1) users can acquire virtual machines of different types: some with GPUs and others without; 2) the GPUs can belong to different generations and thus have different hardware processing power. Thus, we need to assign the graph partitions according to the processing capability of the slave node. Basically, a node with higher processing power should be assigned with more graph partitions. The other constraint is in-memory processing. All the assigned graph partitions should fit into the main memory of the slave node. Also, a graph partition should be smaller than the main memory and the GPU memory of each slave node.

Secondly, as we may apply the graph partitioning multiple times in a recursive manner, graph partitions form a *hierarchy*. The number of cross-partition edges is a straight measure for network traffic, which varies across different partition pairs. As the previous study [3], the subpartitions generated from the same graph partition tend to have a larger cross-partition edges than those generated from different graph partitions at the same level. Given a data graph, we need to determine the number of graph partitions and assign graph partitions carefully according to the hierarchy.

We solve the problem as the steps illustrated in Algorithm 1. We briefly present some remarks on those steps.

In Line 2, we choose the largest of the three values for fine granularity and better quality of load balancing.

Algorithm 1 Graph partitioning and load balancing

1: Check whether the graph can fit into the memory of all the slave nodes. If false, raise an error to users.
2: Calculate $P = max(n^2, \frac{||G||}{m}, \frac{||G||}{d})$ (n is the number of slave nodes, m and d are the smallest main memory capacity and GPU memory respectively in the slave nodes, $||G||$ denotes the size of the graph G);
3: Divide the graph into P partitions with the specified graph partitioning algorithm;
4: Estimate the processing capability of the slave nodes;
5: Calculate the number of partitions for each slave node, according to their processing capability and memory capacities;
6: Assign graph partitions to each slave node;

In Line 4, we estimate the *processing capability* for each slave node. This is a tricky problem, since slave nodes can have different processing capabilities on different graph processing tasks. For example, the performance improvement of the GPU over the CPU varies significantly on graph processing tasks, which can be over an order of magnitude differences according to the previous study [12]. As a start, we assemble a series of common graph processing tasks (e.g., those in Section V) as the benchmark, and run them against each kind of slave nodes. Suppose the total execution time of finishing the benchmark to be t, we calculate its processing capability to be $\frac{1}{t}$. A higher processing capability means a shorter time for processing the same amount of graph data.

In Line 5, we use an iterative approach in calculating the number of graph partitions for each slave node. At the beginning of each iteration, we *pretend* to assign all the graph partitions to the slave nodes, and each slave node gets its share proportionally to its processing capability. If all slave nodes can hold the assigned graph partitions

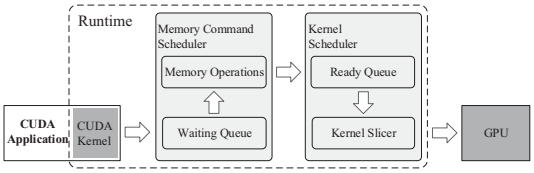

Figure 2. Runtime systems for G2: automatic memory management and kernel execution.

in the main memory, we stop the iteration. Otherwise, we take away some graph partitions from those "overflowed" machines till the remaining partitions can fit into main memory. All those graph partitions are taken as the input to the next iteration and the slave nodes with available main memory are the candidates to assign more graph partitions. In Line 6, we assign the graph partitions to each slave node according to the result of Line 5. We assign graph partitions with the priority that assigning the graph partitions generated with a lower ancestor in the partition hierarchy.

Finally, it is possible to have other partitioning methods (e.g., [32]). It is our future work to evaluate their impacts on G2.

C. Runtime System

The runtime system of G2 addresses two issues for GPU-based graph processing. Firstly, GPUs are used as a co-processor. A graph partition needs to be transferred from the main memory to the GPU memory before the kernel execution and the results may be copied back after the kernel execution. Secondly, GPU kernels can have very different resource usage. For example, some graph operations (like PageRank) are more computation-intensive and other operations (like BFS) are more memory-intensive. They can be combined to run on the GPU for further improvement.

The G2 runtime system embraces a two-level scheduler to handle the memory transfers and kernel execution. Such a design is inspired by the classic multi-level process scheduler design in operating systems [33]. Particularly, in operating systems, the memory and the CPU are two major resources and they are managed by two levels of schedulers: one for selecting the processes to allocate in the main memory (after memory allocations, processes are considered to be *ready*), and one for selecting the ready process to execute on the CPU. Similarly, a kernel also has states, depending on whether the input data are available on the GPU. Thus, we develop two levels of scheduling: memory command scheduler and kernel scheduler. As a start, we adopt our previous work [34] that performs kernel slicing and scheduling for higher resource utilization. Due to the space limitation, we refer the readers to our paper [34] for more details. We briefly present the design for completeness.

Figure 2 shows an overview of the G2 runtime system. The runtime system maintains two queue structures: *waiting queue* and *ready queue*. The waiting queue stores kernels in waiting state, and the ready queue stores the kernels in ready state. Initially, kernels are submitted and are temporarily buffered in the waiting queue. The memory command scheduler automatically schedules execution of GPU memory commands. If a kernel has all its input data available on the GPU (i.e, its depending memory commands and dependent executions are all finished), the kernel is migrated from the waiting queue to the ready queue. The kernel scheduler performs co-scheduling on ready kernels according to how much their resource usage is complementary with each other. The purpose of kernel slicing is to divide a (ready) kernel into multiple slices so that the finer granularity of each slice as a kernel can create more opportunities for time sharing. Then, the slices from the two kernels are co-scheduled and executed on the GPU until either kernel is finished.

The memory command scheduler is responsible for handling memory commands from applications, preparing the input data for the kernel as well as copying the output data to the main memory if appropriate. The memory command scheduler maintains the memory commands for the same kernel into one queue. When all the memory commands of the kernel are available in the queue, the memory command scheduler schedules those commands. If the commands from more than one kernel are ready for execution, we adopt the FCFS (First Come First Serve) algorithm for simplicity.

When implementing memory command scheduler, we make the following considerations. First, we need to change the synchronous function calls for memory commands to asynchronous ones. As a result, those function calls simply puts the commands into the queue of the corresponding kernel, and return immediately without really executing the memory commands. Second, we perform the memory commands of the same kernel in a batch so that we can avoid the device memory allocation deadlock. Third, we prefer to allocate page-locked host memory to exploit the GPU's capability of overlapping kernel execution with PCI-e data transfer. If the page-locked memory reaches the system limit, unlocked memory is allocated.

V. Preliminary Results

This section presents our preliminary results on our research prototype. More extensive experiments will be included in our technical report [35].

A. Methodology

We have conducted our experiments on Amazon EC2. We use eight GPU instances. Each instance has 16 vCPU, 22.5GB DRAM and 2 NVIDIA Tesla M2050 GPUs. The network is 10Gbps, shared with other instances in Amazon EC2. We develop a system prototype named G2 on top of MPICH2 v1.4.1. We use G2 to implement a set of

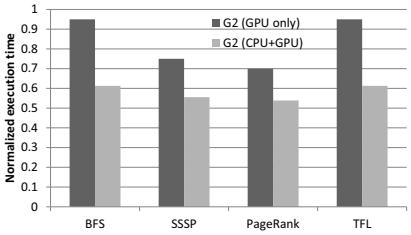

Figure 3. Overall performance of G2.

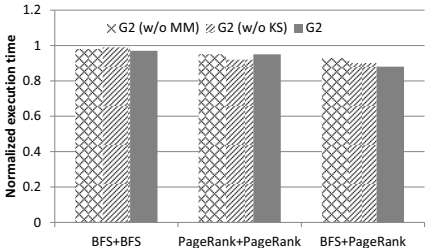

Figure 4. The impact of G2 runtime system.

common graph processing operations. The graph processing operations include PageRank, breadth first search (BFS), single source shortest paths (SSSP) and two-hop friends list (TFL). TFL finds the list of two-hop friends for analyzing social influence spread, community detection and so on. The ratio of the selected vertices is 10% in our experiments. For simplicity, PageRank runs for 10 iterations. Implementing those operations with G2 is straightforward and thus we omit the details for space interests.

As a start, we mainly use synthetic graphs for a full control on evaluating the efficiency and scalability of G2. We generate synthetic graphs simulating small world phenomenon. We first generate multiple small graphs with small-world characteristics using an existing generator [36], and next randomly change a ratio (p_r) of edges to connect these small graphs into a large graph. The default value of p_r is 10%. We varied the sizes of the synthetic graphs to evaluate the scalability. As G2 adopts in-memory processing, we use one half of the main memory to store the graph and the remaining memory for intermediate results. The local cluster supports a graph with 36 GB at most (with 144 million vertices and 8 billion edges), and the virtual cluster supports the graph up to 96 GB (with 384 million vertices and 21 billion edges). By default, we use the largest graph setting for assessing G2. *All reported results are normalized to the results of the baseline engine with CPU-only approach.*

B. Preliminary Results

We first present the preliminary results of running a single application, followed by co-running multiple applications. As a start, we present the results for co-running two applications.

Single-Application Evaluations. Figure 3 shows the overall performance comparison for running the four operations on the virtual cluster of Amazon EC2. For comparison, we implement two variants of G2: one with GPU only and the other one with both the CPU and the GPU. Overall, G2 with GPU only achieves an improvement by 40%. Combining the CPU and the GPU, G2 achieves an even more significant performance improvement by 50%.

Multi-Application Evaluations. As a preliminary study on multiple applications, we run three combinations on two applications: 1) BFS+BFS: each application consists of 10 times of BFS; 2) PageRank+PageRank: each application consists of a PageRank of 10 iterations; 3) BFS+PageRank: one application runs PageRank of 10 iterations and the other application runs BFS of 10 times. To assess the performance impact of individual techniques, we disable individual techniques: "G2(w/o MM)" and "G2(w/o KS)" mean the runtime system without memory management and without kernel scheduling, respectively. The results are shown in Figure 4. The runtime system can further improve the performance by 12% when automatic memory management and kernel scheduling are both enabled on BFS+PageRank. BFS+BFS and PageRank+PageRank have relatively small performance improvement, since the resource usage is not complementary for those two cases.

VI. Lessons and Open Problems

G2 is simply a starting point for GPU-accelerated large-scale in-memory graph processing. This study will open up many research opportunities towards hardware-accelerated large-scale graph processing.

Architectural Design. In G2, in-memory processing and GPU are the two key techniques for the performance. However, as the increasing popularity of graph-centric applications (such as the fast growing social graphs and web graph), it is yet to confirm whether those two techniques are the most favorable system design in terms of performance, energy consumption and total ownership cost, among others. Thus, in addition to main-memory based solution, one may investigate other emerging storage such as solid state drive (SSD), which also exhibits much faster random I/O speed than hard disks. As for the GPU, one may explore other accelerators like Intel Xeon Phi. More research efforts are required on efficient data structures and algorithms for graphs on such a hybrid system.

Application needs. Web and social network have been two main driving applications for graph processing. Their application needs evolve, from offline processing to online processing. Many application needs such as data consistency and transaction management received relatively less research attention. They have already been difficult problems for flat data like distributed relational databases, and will be more challenging for GPU accelerations and graph operations.

Computation Model. "One size does not fit all." Application needs drive the computation model. Current systems

are mainly based on vertex oriented execution model. It is an open problem to extend those models with advanced database techniques such as indexes and different needs in applications and hardware architectures.

Other optimization criteria. In cloud computing, energy consumption and monetary cost [37] are also important optimization criteria. While GPU accelerations can significantly improve the performance, we still need to carefully evaluate them in terms of energy efficiency and monetary cost efficiency. New algorithms and architectural designs may be invented for those new optimization goals.

VII. Conclusions And Future Work

Large graph processing is an important big data application. This paper proposes to accelerate large-scale graph processing with GPUs in the cloud environment. Specifically, we develop *G2* as a GPU-accelerated in-memory graph processing engine. Based on vertex-oriented programming model, we develop a series of GPU-specific optimizations including the fine-grained API design, load-balanced graph partitioning, GPU-optimized runtime systems for GPU memory management and concurrent kernel executions. Putting them all together into G2 leads to significant performance improvement over its counterparts. Our preliminary results demonstrate the performance improvement is by 50% on a virtual cluster in Amazon EC2.

VIII. Acknowledgement

The authors would like to thank the anonymous reviewers for their valuable comments. This work is partly supported by a startup grant (M4080102) from Nanyang Technological University of Singapore and a Singapore MoE AcRF Tier 2 grant (MOE2012-T2-2-067).

References

[1] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *SIGMOD*, 2010.

[2] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "PEGASUS: A peta-scale graph mining system – implementation and observations," in *ICDM*, 2009.

[3] R. Chen, M. Yang, X. Weng, B. Choi, B. He, and X. Li, "Improving large graph processing on partitioned graphs in the cloud," in *ACM SoCC*, 2012, pp. 3:1–3:13.

[4] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein, "Graphlab: A new framework for parallel machine learning," in *UAI*, 2010.

[5] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *SC*, 2008.

[6] R. Nath, S. Tomov, T. T. Dong, and J. Dongarra, "Optimizing symmetric dense matrix-vector multiplication on GPUs," in *SC*, 2011.

[7] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: A MapReduce framework on graphics processors," in *PACT*, 2008.

[8] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander, "Relational joins on graphics processors," in *SIGMOD*, 2008.

[9] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in *HiPC*, 2007.

[10] V. Vineet and P. J. Narayanan, "CUDA cuts: Fast graph cuts on the GPU," in *CVPR Workshops*, June 2008.

[11] J. Zhong and B. He, "Parallel graph processing on graphics processors made easy," *PVLDB (demo)*, vol. 6, pp. 1–4, 2013.

[12] ——, "Medusa: Simplified graph processing on GPUs," *IEEE TPDS*, vol. 99, no. PrePrints, p. 1, 2013.

[13] J. Duato, A. Pena, F. Silla, R. Mayo, and E. Quintana-Orti, "rCUDA: Reducing the number of GPU-based accelerators in high performance clusters," in *HPCS*, 2010.

[14] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, "PTask: operating system abstractions to manage GPUs as compute devices," in *SOSP*, 2011.

[15] V. Gupta and et al, "Gvim: GPU-accelerated virtual machines," in *HPCVirt*, 2009.

[16] V. T. Ravi and et al, "Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework," in *HPDC*, 2011.

[17] W. Fang, B. He, Q. Luo, and N. K. Govindaraju, "Mars: Accelerating mapreduce with graphics processors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 4, pp. 608–620, Apr. 2011.

[18] R. Wu, B. Zhang, and M. Hsu, "Gpu-accelerated large scale analytics," Tech. Rep. HPL-2009-38, 2009.

[19] Socialbakers, "Facebook pages statistics," http://www.socialbakers.com/facebook-pages/, 2011.

[20] LinkedIn, "Numbers of LinkedIn members as of 1st quarter 2013," http://www.statista.com/statistics/198224/quarterly-member-numbers-of-linkedin/, 2013.

[21] J. Berry, B. Hendrickson, S. Kahan, and P. Konecny, "Software and algorithms for graph queries on multithreaded architectures," in *IPDPS*, March 2007.

[22] D. Bader and K. Madduri, "SNAP, small-world network analysis and partitioning: An open-source parallel graph framework for the exploration of large-scale networks," in *IPDPS*, 2008, pp. 1–12.

[23] D. Gregor and A. Lumsdaine, "The parallel bgl: A generic library for distributed graph computations," in *POOSC*, 2005.

[24] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in *PPoPP*, 2011.

[25] L. Luo, M. Wong, and W.-m. Hwu, "An effective GPU implementation of breadth-first search," in *DAC*, 2010.

[26] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed GraphLab: A framework for machine learning and data mining in the cloud," *PVLDB*, 2012.

[27] Apache Giraph, *http://giraph.apache.org/*.

[28] M. Delorimier and et al, "GraphStep: A system architecture for sparse-graph algorithms," in *FCCM*, 2006.

[29] J. Lin and M. Schatz, "Design patterns for efficient graph algorithms in MapReduce," in *MLG*, 2010.

[30] "Apache hama project," http://incubator.apache.org/hama/.

[31] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[32] A. Gharaibeh, L. B. Costa, E. Santos-Neto, and M. Ripeanu, "On graphs, gpus, and blind dating: A workload to processor matchmaking quest," in *IPDPS*, 2013.

[33] W. Stallings, *Operating Systems: Internals and Design Principles, 6/E*. Pearson Education India, 2009.

[34] J. Zhong and B. He, "Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling," *IEEE TPDS*, vol. 99, no. PrePrints, p. 1, 2013.

[35] ——, "Gpu-accelerated large-scale graph processing in the cloud," Tech. Rep. NTU-PDCC, 2013. [Online]. Available: http://pdcc.ntu.edu.sg/

[36] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining," in *SDM*, April 2004.

[37] H. Wang, Q. Jing, R. Chen, B. He, Z. Qian, and L. Zhou, "Distributed systems meet economics: Pricing in the cloud," in *HotCloud*, 2010, pp. 1–7.