GPU-Accelerated Cloud Computing for Data-Intensive Applications

Baoxue Zhao, Jianlong Zhong, Bingsheng He, Qiong Luo, Wenbin Fang, and Naga K. Govindaraju

Abstract Recently, many large-scale data-intensive applications have emerged from the Internet and science domains. They pose significant challenges on the performance, scalability and programmability of existing data management systems. The challenges are even greater when these data management systems run on emerging parallel and distributed hardware and software platforms. In this chapter, we study the use of the GPU (Graphics Processing Units) in MapReduce and general graph processing in the Cloud for these data-intensive applications. In particular, we report our experiences in developing system prototypes, and discuss the open problems in the interplay between data-intensive applications and system platforms.

1 Introduction

In recent years, *Big Data* has become a buzz word in both industry and academia, due to the emergence of many large-scale data-intensive applications. These data-intensive applications not only have very large data volume, but may also have complex data structures and high update rates. All these factors pose significant challenges on the performance, scalability and programmability of existing data management systems. We elaborate more details about these challenges in the following.

B. Zhao • Q. Luo (✉)
Department of Computer Science and Engineering, HKUST, Hong Kong
e-mail: bzhaoad@cse.ust.hk; luo@cse.ust.hk

J. Zhong • B. He
School of Computer Engineering, Nanyang Technological University, Nanyang, Singapore
e-mail: jzhong2@ntu.edu.sg; bshe@ntu.edu.sg

W. Fang
San Francisco, CA, USA
e-mail: wenbin@cs.wisc.edu

N.K. Govindaraju
Microsoft Redmond, Redmond, WA, USA
e-mail: nagag@microsoft.com

X. Li, J. Qiu (eds.), *Cloud Computing for Data-Intensive Applications*,
DOI 10.1007/978-1-4939-1905-5_5

Performance Many processing tasks are driven by data updates, which require on-line response. Examples include traffic control and video surveillance. The performance issue is crucial for these on-line applications over large amounts of data.

Scalability Also due to the increasing data volume, it is essential for systems to scale with data growth.

Programmability To meet performance and scalability requirement of big data, data management systems are run on parallel and/or distributed platforms. Programming on such systems is much more challenging than sequential programming.

Many data processing algorithms and systems have been developed, including MapReduce [20] and general graph processing. MapReduce was originally proposed by Google for the ease of development of web document processing on a large number of machines. This framework provides two primitive operations (1) a map function to process input key/value pairs and to generate intermediate key/value pairs, and (2) a reduce function to merge all intermediate pairs associated with the same key. With a MapReduce framework, developers can implement their application logic using the two primitives. The MapReduce runtime will automatically distribute and execute the task on a number of computers. MapReduce is mainly for flat-structured data, and can be inefficient for unstructured data [54]. Thus, general graph processing systems are proposed for processing unstructured data such as social networks and graphs. Representative systems include Google's Pregel [54] and Microsoft's Trinity [63].

Both MapReduce and general graph processing platforms have been developed and run on various kinds of distributed and parallel systems. This chapter focuses on two emerging platforms: GPUs (Graphics Processing Units) and the Cloud, as the representatives for many-core and distributed computing platforms, respectively. Specifically, we review the related work for MapReduce and general graph processing on GPUs and Cloud platforms. Moreover, we report our experiences in developing system prototypes: Mars [32] and MarsHadoop [25] for GPU-based MapReduce; Medusa [74,76] and Surfer [16] for GPU-based and Cloud-based graph processing, respectively. Finally, we discuss the open problems in the interplay between data processing systems and system platforms.

Organization The remainder of this chapter is organized as follows. Section 2 reviews the background on GPUs and Cloud Computing, and related work on MapReduce and general graph processing frameworks on GPUs and in the Cloud. Section 3 introduces the MapReduce work on GPU Clusters. Section 4 presents the parallel graph processing techniques on GPUs and the Cloud. We conclude our chapter and discuss the open problems in Sect. 5.

2 Background and Related Work

2.1 Cloud Computing

With the rapid growth of the Internet and other application areas such as finance, biology and astronomy, great amounts of data are produced continuously and need to be processed with high time constraints. To handle data-intensive problems, the concept of *Cloud Computing* has been proposed. A few Internet corporations implemented their own Cloud Computing platforms to provide elastic computing services for their customers. Such as Google Compute Engine, Amazon EC2 and Microsoft Azure. The Cloud Computing services are provided at different levels: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). Customers who use services at a higher level do not need to know the lower level details of the Cloud, e.g., network topology or resource allocation. In view of these features, Cloud Computing poses new challenges for computing tasks compared with traditional computation platforms such as HPC clusters. In Sect. 4.2 we will discuss the challenges in detail and introduce the solutions for graph processing in the Cloud.

2.2 The GPU

In recent years, GPUs (Graphics Processing Units) have shifted from pure graphics processors to general-purpose parallel processors. A modern GPU can run a massive number of lightweight threads simultaneously to process data in the SIMD (Single Instruction, Multiple Data) style and can provide an order of magnitude speedup over traditional CPU with higher cost-efficiency. GPUs are widely used in desktops, servers and clusters as parallel computing platforms to accelerate various kinds of applications. However, writing a correct and efficient GPU program is challenging in general, and even more difficult for higher level applications. First, the GPU is a many-core processor with massive thread parallelism. To fully exploit the GPU parallelism, developers need to write parallel programs that scale to hundreds of cores. Moreover, compared with CPU threads, the GPU threads are lightweight, and the tasks in the parallel algorithms should be fine-grained. Second, the GPU has a memory hierarchy that is different from the CPU's, and is exposed to the programmer for explicit use. Since applications usually involve irregular accesses to the memory, careful designs of data layouts and memory accesses are key factors to the efficiency of GPU acceleration. Finally, since the GPU is designed as a co-processor, developers have to explicitly perform memory management on the GPU, and deal with GPU specific programming details such as kernel configuration and invocation. All these factors make the GPU programming a difficult task. In this chapter, we will introduce the GPU-accelerated MapReduce system (Sect. 3) and GPU-accelerated graph processing library (Sect. 4.1), both of which ease the effort of developing higher level applications using GPUs.

2.3 MapReduce

MapReduce [20] is a programming model proposed by Google, and has been widely used for large scale data processing. Many researchers and engineers have tried to implement their own MapReduce systems since the publication of the MapReduce paper. Hadoop [3] is the most successful and widely-used open source implementation of MapReduce on the distributed computing platform. There are also some implementations on other platforms, such as multi-core shared-memory systems and GPUs.

Phoenix [62,67,72] is a MapReduce library that runs on shared memory systems leveraging the power of multi-core CPUs. Our previous work Mars [25, 32] is among the first GPU-accelerated MapReduce frameworks. Mars provides a set of higher level programming interfaces, and hides the vendor-specific lower-level details such as thread configurations. The design goal of Mars is to ease the GPU programming effort for MapReduce applications. Since Mars, there has been a number of improvements [11,14,36,40] as well as extended GPU-based MapReduce implementations to support multi-GPUs on a single-machine [17,18,41], integrated architectures [15] and clusters [8,64,65,68].

MapCG [36] is a single-machine MapReduce framework that provides source code portability between CPU and GPU. The users only need to write one version of the code, and the MapCG runtime is responsible for generating both CPU and GPU specific codes and executes them on the corresponding platforms: multi-core CPUs and GPUs. Different from Mars, MapCG does not need the counting step to compute the required memory space on GPUs. Instead, it allocates a large block of GPU memory space, and uses the *atomicAdd* operation to manage and record the memory space required by each thread. Another improvement of MapCG is that it avoids the sorting of intermediate key/value pairs by using a memory efficient hash table. Ji et al. [40] proposed to use the GPU shared memory to buffer the input and output data of the *Map* and *Reduce* stages. Chen et al. [14] proposed to use shared memory in a more efficient way by carrying out reductions in shared memory. Grex [11] is a recent GPU-based MapReduce framework. It provides some new features over Mars and MapCG, including parallel input data splitting, even data distribution to *Map*/*Reduce* tasks to avoid partitioning skew, and a new memory management scheme. Experiments show that Grex achieves 12.4× and 4.1× speedup over Mars and MapCG, respectively.

Chen et al. [15] designed a MapReduce framework for integrated architectures, i.e. AMD Fusion chip as a representative in their implementation. In their framework, the workload can be partitioned across the CPU cores and GPU cores by two different schemes: the *map-dividing scheme* and the *pipelining scheme*. In the first scheme, each of the *Map*/*Reduce* stage is executed by both CPU cores and GPU cores simultaneously, whereas in the second scheme, each stage is executed by only one type of cores. To leverage the memory overhead, a *continuous reduction* strategy based on *reduction object* is used in the framework. The strategy is very similar to the optimization work in [14]. Their experiment results show 1.2–2.1× speedup over the best multi-core or discrete GPU-based implementation.

While previous implementations use a single GPU, MGMR [18, 41] is a single-machine MapReduce system supporting multiple GPUs. With the support of host memory, MGMR can handle large-scale data exceeding GPU memory capacity. In addition, MGMR uses the GPUDirect technology to accelerate inter-GPU data transmission. The Pipelined MGMR (PMGPR) [17] is an upgraded version of MGMR. PMGMR takes advantage of the CUDA stream feature on Fermi and Kepler GPUs to achieve the overlap of computation and memory copy. For Fermi GPUs, PMGMR uses a runtime scheduler to resolve the dependency among different CUDA streams to achieve the highest concurrency, whereas for Kepler GPUs, PMGMR exploits Kepler's Hyper-Q feature to automatically reach the best performance. PMGMR achieves 2.5× speedup over MGMR.

To support MapReduce on GPU clusters, MarsHadoop [25] makes a simple extension by integrating single-node Mars into Hadoop using the Hadoop streaming technology [5]. Most of the work on MapReduce on GPU clusters integrates GPU workers into Hadoop [8,27,29,64,68,73], and there is also an implementation using MPI (Message Passing Interface) [65].

MITHRA [27] is a Hadoop-based MapReduce framework for Monte-Carlo simulations. It leverages GPUs for the *Map/Reduce* stage computation. Like MarsHadoop, MITHRA uses Hadoop streaming technology to invoke the GPU kernels written in CUDA. Pamar [68] integrates GPUs into the Hadoop framework using JCUDA API. The main focus of Pamar is to provide a framework that can utilize different types of resources (CPU and GPU) transparently and automatically. Surena [8], on the other hand, uses Java Native Interface (JNI) to invoke CUDA code from the Java-based Hadoop framework. Shirahata et al. [64] proposed a hybrid map task scheduling techniques for MapReduce on GPU clusters by dynamic profiling of the map task running on CPUs and GPUs, and demonstrated a speedup of nearly two times over the original scheduler of Hadoop. Lit [73] and HadoopCL [29] improves previous Hadoop-based works by automatically generating kernel codes for GPU devices, so that they hide the GPU programming complexity from users. Specifically, Lit uses an annotation based approach to generate CUDA kernel code from Java code, whereas HadoopCL generates OpenCL kernel code from Java code using an open source tool called Aparapi [2]. In these Hadoop-based extensions, various features of Hadoop such as reliability, scalability and simplified input/output management through HDFS are inherited while the *Map* and *Reduce* stages are parallelized on the GPUs.

GPMR [65] is a MapReduce library on GPU clusters. It is not based on Hadoop; instead it implements the MapReduce model using MPI. GPMR supports multiple GPUs on each node. Compared with the Hadoop-based GPU MapReduce implementation, GPMR is more flexible and more efficient since it exposes more GPU programming details to the application level. Therefore, users can apply application-specific optimizations. Moreover, GPMR also parallelizes the sort and partitioning modules of MapReduce. The scalability of GPMR to the number of GPUs is limited, as the speedup of most applications decreases dramatically when there are more than 16 GPUs [71].

In addition to the MapReduce frameworks supporting NVIDIA GPUs, there are some efforts using other types of CPUs and GPUs. Among them, StreamMR [23] is a MapReduce framework for AMD GPUs, and CellMR [61] is a MapReduce framework supporting asymmetric Cell-Based Clusters. Our enhanced Mars implementation [25] also supports AMD GPUs and co-processing of different types of processors (Multi-core CPU, NVIDIA GPU and AMD GPU).

2.4 General Graph Processing

Graphs are common data structures in various applications such as social networks, computational chemistry and web link analysis. Graph processing algorithms have been the fundamental tool in such fields. Developers usually apply a series of operations on the graph edges and vertices to obtain the final result. Example operations are breadth first search (BFS), PageRank [58], shortest paths and customized variants (for example, developers may apply different application logics on top of BFS). The efficiency of graph processing is essential for the performance of the entire system. On the other hand, writing every graph processing algorithm from scratch is inefficient and involves repetitive work, since different algorithms may share the same operation patterns, optimization techniques and common software components. A programming framework supporting high programmability for various graph processing applications and providing high efficiency can greatly improve productivity.

Recently, we have witnessed many research efforts in offering parallel graph processing frameworks on multi-core/many-core processors and cloud computing platforms. Those frameworks embrace architecture-aware optimization techniques as well as novel data structure designs to improve the parallel graph processing on the target platform. The most popular paradigm so far is vertex-oriented programming. The introduction of vertex-oriented programming is based on the observations in previous studies [21, 51, 52] that many common graph algorithms can be formulated using a form of the bulk synchronous parallel (BSP) model (we call it *GBSP*). In GBSP, local computations are performed on individual vertices, and vertices are able to exchange data with each other. These computation and communication procedures are executed iteratively with barrier synchronization at the end of each iteration. This common algorithmic pattern is adopted by common parallel graph processing frameworks such as Pregel [54], GraphLab [53] and Medusa [76]. For example, Pregel applies a user-defined function *Compute()* on each vertex in parallel in each iteration of the GBSP execution. The communications between vertices are performed with message passing interfaces.

3 MapReduce on GPU Clusters

In this section, we introduce the MapReduce implementation on GPU clusters. We first give an overview of the previous work: Mars single-machine and MarsHadoop. We also give an alternative of multi-machine implementation of Mars, called Mars-MR-MPI. Finally we study the performance of different implementations.

3.1 Mars Overview

Mars [25, 32] is a GPU-based MapReduce implementation. It hides the programming details of vendor-specific GPU devices, and provides a set of user-friendly higher level programming interfaces. The design of Mars is guided by three goals:

1. *Programmability*. Ease of programming releases programmers' efforts on GPU programming details and makes them more focused on higher-level algorithm design and implementation.
2. *Flexibility*. The design of Mars should be applicable to various kinds of devices including multi-core CPUs, NVIDIA GPUs and AMD GPUs. Moreover, it should be easy for users to customize their workflows.
3. *Performance*. The overall performance of GPU-based MapReduce should be comparable to or better than that of the state-of-the-art CPU-based counterparts.

Mars provides a set of APIs, which are listed in Table 1. These APIs are of two types: user-implemented APIs, which the users should implement, and the system-provided APIs, which is a part of the Mars library implementation and can be called by the users directly. The APIs of Mars are similar to those of other MapReduce frameworks, except that Mars uses two steps for both the *Map* and *Reduce* stages. Firstly, the count functions (MAP_COUNT or REDUCE_COUNT) are invoked to compute the sizes of the key/value pairs and then the MAP or REDUCE is invoked to emit the actual key/value pairs.

Figure 1 shows the workflow of Mars. The Mars workflow contains three stages for a MapReduce job—*Map*, *Group* and *Reduce*. Before the *Map* stage, the input data on the disk is transformed into input records using the CPU and those records are copied from the main memory into the GPU device memory. Both the *Map* and *Reduce* stages use a two-step lock-free scheme, which avoids costly atomic operations and dynamic memory allocation on GPU devices. Take the *Map* stage as an example. In the first step *MapCount* invokes the user-defined MAP_COUNT function to compute the sizes of the intermediate key/value pairs for each thread. Then a prefix sum operation is performed to compute the writing locations for each thread as well as the total sizes of the output. Finally Mars allocates device memory space for the output. In the second step, the user-defined MAP function is invoked on each thread to map the input records to intermediate records and output them to the device memory according to the pre-computed writing locations. The lock-free scheme of the *Reduce* stage is similar to that of the *Map* stage. The *Group* stage sorts

Table 1 Mars APIs [25,32]

Function name	Description	Type
MAP_COUNT	Calculates the output buffer size of MAP	User
MAP	The map function	User
REDUCE_COUNT	Calculates the output buffer size of REDUCE	User
REDUCE	The reduce function	User
EMIT_INTERMEDIATE_COUNT	Emits the key size and the value size in MAP_COUNT	System
EMIT_INTERMEDIATE	Emits the key and the value in MAP	System
EMIT_COUNT	Emits the key size and the value size in REDUCE_COUNT	System
EMIT	Emits the key and the value in REDUCE	System

the intermediate key/value pairs according to the key field, so that the intermediate key/value pairs with the same key are stored consecutively as a group.

Since the three stages are loosely coupled modules, the Mars framework can fit three kinds of user-customized workflows, according to whether the *Group* and *Reduce* stages are required:

- MAP_ONLY. Only the *Map* stage is required and executed.
- MAP_GROUP. Only the *Map* and *Group* stages are executed.
- MAP_GROUP_REDUCE. All three stages—*Map*, *Group*, and *Reduce* are executed.

The GPU-based Mars single-machine implementation was evaluated in comparison with the CPU-based MapReduce framework Phoenix [62] as well as the native implementation without MapReduce using a set of representative MapReduce applications. The results show that the GPU-based Mars was up to 22 times faster than Phoenix, and Mars applications had a code size reduction of up to 7 times compared with the native implementation. In summary, Mars greatly simplifies the MapReduce programming on CUDA-based GPU and achieves great efficiency.

3.2 MarsHadoop

While the single-GPU Mars makes a showcase of implementing MapReduce using GPU, it cannot handle very large data set due to the limited memory capacity of a single GPU. In many data-intensive applications, the data scale exceeds far beyond the memory capacity of a single computer, let alone the GPU memory, which usually has a much smaller capacity than main memory.

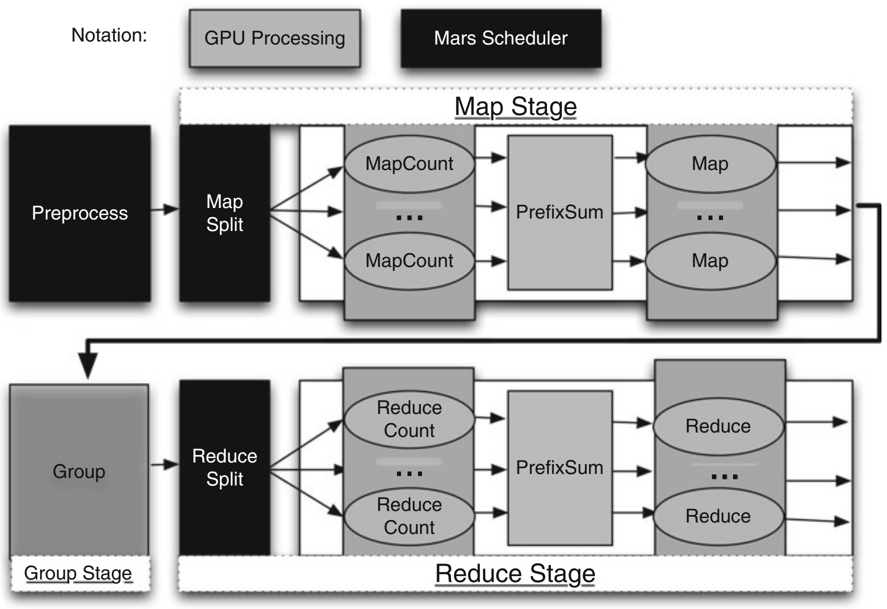

Fig. 1 The workflow of Mars on the GPU [25, 32]

We briefly introduce the previous multi-machine implementation of Mars—MarsHadoop [25]. MarsHadoop is implemented by integrating Mars into the widely used Hadoop MapReduce framework. This integration provides an easy way of exploiting computation power of multiple GPUs for MapReduce, while it also inherits the scalability and fault-tolerance features of Hadoop, as well as the distributed file system support.

Figure 2 shows the workflow of MarsHadoop. A Mapper/Reducer in MarsHadoop is executed on either CPU or GPU, depending on the underlying processors. MarsHadoop Mappers and Reducers are integrated using Hadoop streaming technology [5], which enables the developers to use their customized Mapper/Reducer implementation with any programming language in Hadoop. The Mapper/Reducer executable reads data from *stdin* and emit record to the *stdout*, while the actual task execution is the same as on single machine. The preliminary experiments using Matrix Multiplication on three nodes (one master node and two slave nodes) showed that MarsHadoop was up to 2.8 times faster than the Hadoop without GPU.

3.3 Mars-MR-MPI

In the following we present an alternative to MarsHadoop, which we call Mars-MR-MPI.

MapReduce-MPI (MR-MPI) [59, 60] is an open source, lightweight implementation of the MapReduce model using MPI and C/C++. It is designed for

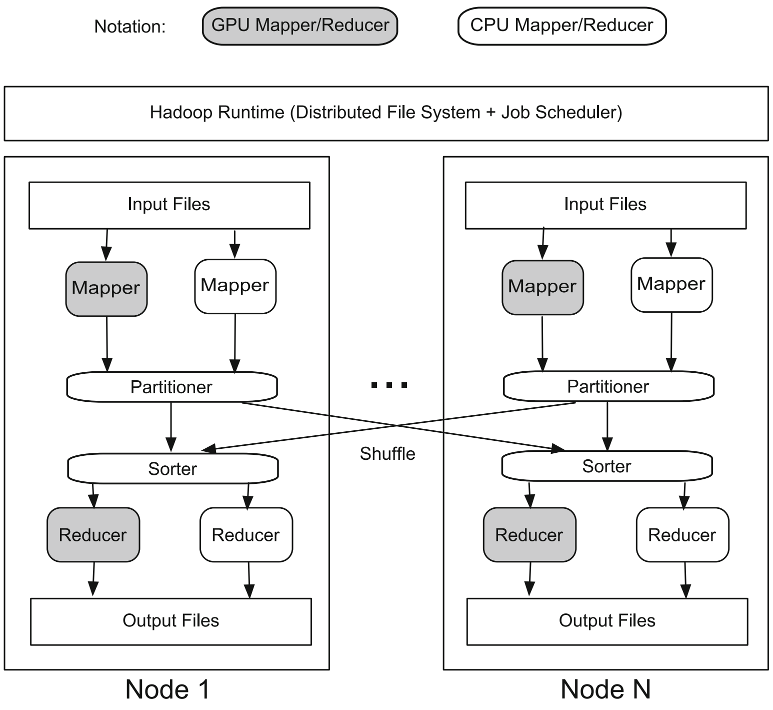

Fig. 2 MarsHadoop workflow [25]. The Map and Reduce tasks are executed on GPUs or CPUs

distributed-memory clusters. MR-MPI has well organized APIs and complete document support. It has been successfully used for SOM [66] and shows good scalability on hundreds of cores. It can process large data sets far exceeding the accumulated memory size of the cluster by using on-disk virtual pages. Moreover, since MR-MPI is implemented using C/C++, the GPU code can be invoked seamlessly with no or little extra runtime cost. We choose MR-MPI as an alternative for extending Mars to multiple machines in view of these features.

The Mars-MR-MPI is implemented as follows.

1. **Initial Data Loading.** Since the MR-MPI framework has no distributed file system support, to avoid the overhead of remote reading, we partition the input data into chunks of equal size and distribute them evenly to the local file system of each node. This approach is comparable with Hadoop since most Hadoop map tasks will read HDFS blocks stored in local node.
2. **The *Map* Stage.** Instead of assigning only one block of data (i.e. 64 MB, which is the default block size of Hadoop) to each map task, we use persistent map tasks for Mars-MR-MPI. In other words, we launch only a small number of map tasks for a job, with each task processing many blocks of input data iteratively. The processing of each data block is the same as the *Map* stage of single-machine Mars, followed by an additional step of organizing the intermediate key/value

pairs generated from this block into an MR-MPI *KeyValue* object and add this object to the MR-MPI runtime framework. The GPU device for each map task is chosen according to the MPI rank of the current process. Using this approach, the number of GPU context switches is reduced with more work done in each context.

3. **The *Reduce* Stage.** In MR-MPI, each time the reduce callback function [6] is called, it will process only one *Key-MultiValues* tuple, which contains a unique key and a list of values. A parallel reduce for one tuple is inefficient if the tuple contains only a small number of values. Therefore, we allocate a global fixed-size memory buffer and pass it to the reduce callback function to accumulate the tuples. Once the size of the accumulated tuples exceeds a threshold ratio of the buffer size, we will perform a parallel reduction on the GPU for all tuples and empty the buffer for accumulating new tuples. In addition to the global buffer, a boolean flag is passed to the callback function, to indicate whether this is the first call on the GPU device. If so, we initialize the GPU device context.

Both the MarsHadoop and the Mars-MR-MPI are based on existing MapReduce frameworks, namely Hadoop and MR-MPI. Compared with Hadoop, MR-MPI provides an interface that is more flexible for integrating the GPU into MapReduce, and the persistent tasks can be used to reduce overhead and better utilize device resources.

3.4 Experiments

We study the performance of the two alternatives of MapReduce on a GPU cluster experimentally.

The experiments are conducted on a 20-node GPU cluster on Amazon EC2. Each node is a Hardware Virtual Machine(HVM)-based instance equipped with 8 Intel Xeon E5-2670 Processors, 15 GB memory, 60 GB local storage and a single NVIDIA Kepler GK104 GPU card (with 1536 CUDA cores and 4 GB memory). The Operating System is Ubuntu 12.04 with CUDA SDK v5.0 installed.

We first study the performance of MarsHadoop using a classic data-intensive MapReduce workload—Inverted Index, which extracts the URL links from a set of HTML files. Each map task takes a block of data as input and emit $\langle url, filename \rangle$ for each URL extracted from the input, whereas each reduce task simply outputs the list of file names for each unique URL. For MarsHadoop streaming job, we parallelize the map tasks on the GPU, whereas the reduce tasks are sequentially executed as they only perform outputting (this kind of reduce task is called IdentityReducer in Hadoop). We implemented the sequential Inverted Index using Hadoop Java API and ran it as Hadoop sequential job as the baseline. We set the number of map slots on each node to 4 in Hadoop configuration. For the sequential job, each map uses one CPU core whereas for the MarsHadoop streaming job, the four map slots on each node share the single GPU to make full use of the GPU computation resources.

We use two Wikipedia web page data sets of different sizes from the Purdue MapReduce Benchmarks Suite [9], one 50 GB and the other 150 GB. Figure 3 shows the execution time of the map tasks in MarsHadoop streaming jobs and Hadoop sequential jobs. For both data sets, the MarsHadoop streaming jobs are more efficient than Hadoop sequential jobs.

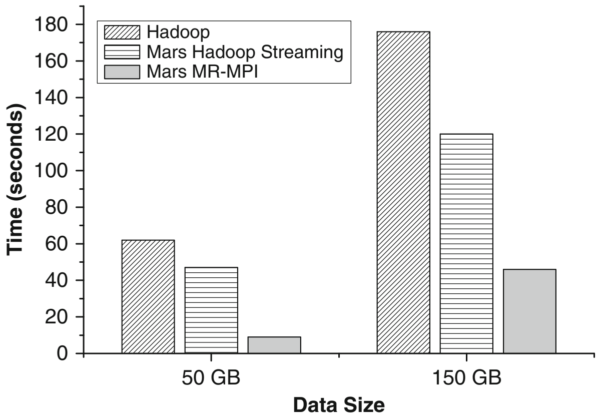

Fig. 3 The execution time of Inverted Index map tasks in MarsHadoop streaming jobs, Hadoop sequential jobs and Mars-MR-MPI jobs using 50 and 150 GB data sets

Next we study the performance of Mars-MR-MPI, in comparison with MarsHadoop. Figure 3 shows the execution time of map tasks in Mars-MR-MPI and MarsHadoop for Inverted Index respectively. We observed that for both 50 GB and 150 GB data sets, the map time of Mars-MR-MPI is 3–5× faster than MarsHadoop streaming. Compared with Mars-MR-MPI, the inefficiency of MarsHadoop streaming is due to the following two factors:

1. Inter-process data communication overhead with Hadoop streaming, since the input data are passed into the Mars map executable by *stdin* and the output passed out by *stdout*.
2. The latencies from Hadoop internal, such as the latency of a TaskTracker requesting new tasks from JobTracker. The GPUs will be idle during the latency intervals, leading to under-utilization.

As the Mars-MR-MPI is more efficient, we further study its scalability and performance bottlenecks. We configure Mars-MR-MPI to process 2.5 GB data on each node. Figure 4 shows the time consumption of each MR-MPI stage with the cluster size varied from 2 to 20 nodes. We observe that when the number of nodes increases, for the stages involving only local operations, including Map stage, Sort/Hash stage and Reduce stage, the time nearly keeps stable. The reason is that the local operations are performed independently on each node and the workload on each node is even. However, for the Network I/O stage which involves large amount of inter-node communication, the time increases with the cluster size.

We further give the time breakdown of Mars-MR-MPI running Inverted Index with two data sets on the entire cluster (Fig. 5). The result shows that the Network I/O takes 30 and 36 % of the total time on 50 and 150 GB data sets, respectively. We conclude that the network I/O is the performance bottleneck when the cluster size or data size becomes larger.

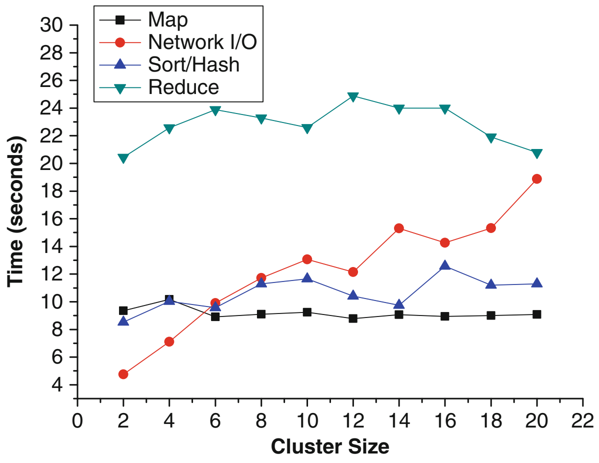

Fig. 4 The time consumption of each MR-MPI stage for Mars-MR-MPI with cluster size varied, running Inverted Index on 50 GB data set

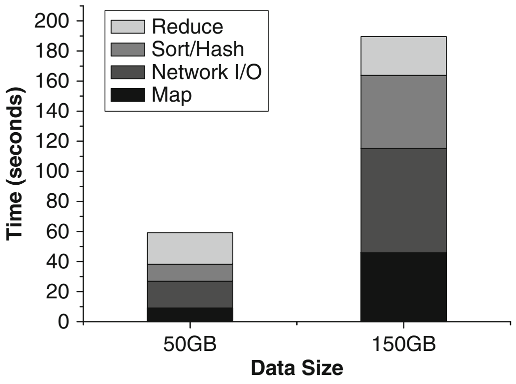

Fig. 5 The time breakdown of Mars-MR-MPI running Inverted Index on the entire cluster

Both MarsHadoop and Mars-MR-MPI are built on top of existing MapReduce frameworks and do not parallelize the internals of the frameworks, such as sort or hash partition. The performance of this type of extension is closely dependent on the performance of the frameworks themselves. We now study the performance of the GPMR, a recent stand-alone MapReduce library based on the GPU. We select two examples enclosed with the GPMR source code: K-means Clustering and Integer Counting. The former example is map-computation-bound, whereas the latter is communication-bound.

To test the GPMR scalability on various number of nodes, we launch one MPI process for each GPU. Due to the limitation of the GPU-based radix sort used in GPMR, we let each process consume 32 million randomly generated integers for Integer Counting, and 32 million randomly generated two-dimensional floating point data for K-means Clustering. The results are shown in Fig. 6. In the K-mean Clustering results in Fig. 6a, we notice that (1) The Map time keeps nearly stable for various number of nodes and it dominates the total running time; (2) The Network I/O, Sort and Reduce stages consume very little time. The reason is that the K-means Clustering needs a lot of computation in the map stage and produces very small size intermediate data. In contrast, in Fig. 6b, since Integer Counting produces a large amount of intermediate result, both the Map and Network I/O stages dominate the total running time. It also requires a moderate amount of Sort and Reduce time to handle the intermediate data, but the time of these operations does not change much with the variation of number of nodes, as these are local operations. We conclude that the GPMR framework scales well to cluster size for compute-bound applications, but suffers from the network I/O bottleneck for communication-bound ones.

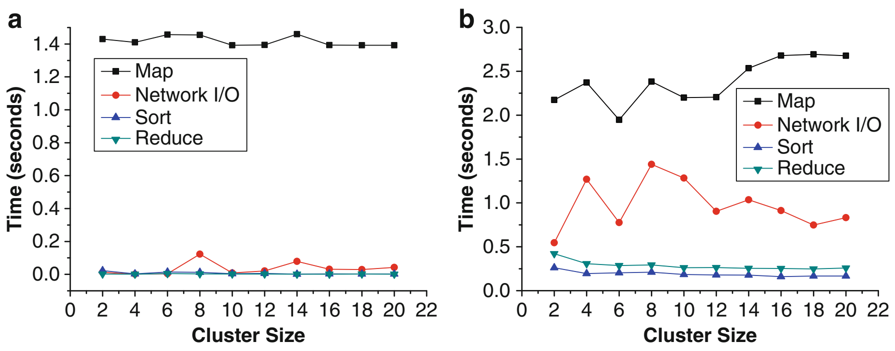

Fig. 6 The time consumption of each MapReduce stage in GPMR with number of nodes varied. (**a**) K-means Clustering with each GPU processing 32 million randomly generated two-dimensional floating points. (**b**) Integer Counting with each GPU processing 32 million randomly generated integers

Finally, we study the GPMR scalability to data size. Due to the limitation of the GPU-based radix sort, GPMR cannot handle more than 32 million integers on each process unless it uses CPU-based sort algorithms. The scalability test is only conducted for K-means Clustering. We use 20 nodes with one process on each node and let each process consume various numbers of points ranging from 32 million to 160 million. The result is shown in Fig. 7. The map time grows linearly to the data size, whereas the Network I/O, Sort and Reduce consumes very little time.

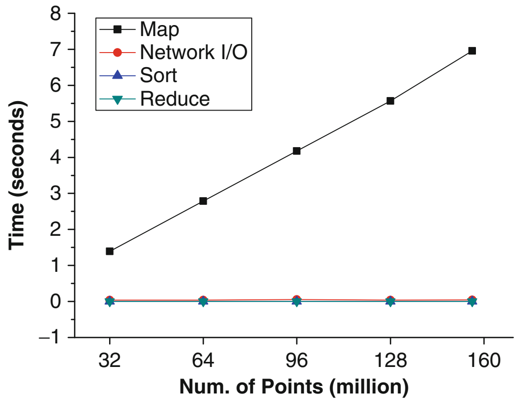

Fig. 7 The time consumption of each MapReduce stage in GPMR with data size varied, running K-means Clustering on 20 nodes

4 Graph Processing on GPUs and the Cloud

In this section, we introduce two representative vertex-oriented programming frameworks on GPUs and in the Cloud—Medusa and Surfer. Particularly, Medusa is a parallel graph processing library running on multi-GPUs on a single machine. It uses a novel graph programming model called "Edge-Message-Vertex" and simplifies the graph processing using GPUs. Surfer uses a network performance aware graph partitioning framework to improve the performance of large graph processing on the Cloud.

4.1 Parallel Graph Processing on GPUs

Recent years have witnessed the increasing adoption of GPGPU (General-Purpose computation on Graphics Processing Units) in many applications [57], such as databases [33, 34] and data mining [26]. The GPU has been used as an accelerator for various graph processing applications [31, 35, 48, 70]. While existing GPU-based solutions have demonstrated significant performance improvement over CPU-based implementations, they are limited to specific graph operations. Developers usually need to implement and optimize GPU programs from scratch for different graph processing tasks.

Medusa is the state-of-the-art vertex-oriented graph processing framework on multiple GPUs in the same machine [76]. Medusa is designed to ease the pain of leveraging the GPU in common graph computation tasks. Extending the single vertex API of Pregel, Medusa develops a novel graph programming model called "Edge-Message-Vertex" (EMV) for fine-grained processing on vertices and edges. EMV is specifically tailored for parallel graph processing on the GPU. Medusa provides a set of APIs for developers to implement their applications. The APIs are based on the EMV programming model for fine-grained parallelism.

Medusa embraces an efficient message passing based runtime. It automatically executes user-defined APIs in parallel on all the processor cores within the GPU and on multiple GPUs, and hides the complexity of GPU programming from developers. Thus, developers can write the same APIs, which automatically run on multiple GPUs.

Memory efficiency is often an important factor for the overall performance of graph applications [31, 35, 48, 70]. Medusa has a series of memory optimizations to improve the locality of graph accesses. A novel graph layout is developed to exploit the *coalesced* memory feature of the GPU. A graph aware message passing mechanism is specifically designed for message passing in Medusa. Additionally, Medusa has two multi-GPU-specific optimization techniques, including the cost model guided replication for reducing data transfer across the GPUs and overlapping between computation and data transfer.

Medusa has been evaluated on the efficiency and programmability. Medusa simplifies programming GPU graph processing algorithms in terms of a significant reduction in the number of source code lines. Medusa achieves comparable or better performance than the manually tuned GPU graph operations.

The experiments were conducted on a workstation equipped with four NVIDIA Tesla C2050 GPUs, two Intel Xeon E5645 CPUs (totally 12 CPU cores at 2.4 GHz) and 24 GB RAM. The workloads include a set of common graph processing operations for manipulating and visualizing a graph on top of Medusa. The graph processing operations include PageRank, breadth first search (BFS), maximal bipartite matching (MBM), and single source shortest paths (SSSP). In order to assess the queue-based design in Medusa, we have implemented two versions of BFS: BFS-N and BFS-Q for the implementations without and with the usage of queue-based APIs, respectively. Thus, BFS-Q is work optimal whereas BFS-N is not. Similarly, two versions of SSSP are implemented: SSSP-N and SSSP-Q without and with the usage of queue-based APIs, respectively.

The experimental dataset includes two categories of sparse graphs: real-world and synthetic graphs. Table 2 shows their basic characteristics. We use the GTgraph graph generator [4] to generate power-law and random graphs. To evaluate MBM, we generate a synthetic bipartite graph (denoted as BIP), where vertex sets of two sides have one half of the vertices and the edges are randomly generated. The real world graphs are publicly available [1, 7].

MTGL [13] is used as the baseline for graph processing on multi-core CPUs. The BFS and PageRank implementations are offered by MTGL. We implement the Bellman-Ford SSSP algorithm and a randomized maximal matching algorithm [10] using MTGL. The best result was obtained when the number of threads was 12 on the experiment machine. MTGL running on 12 cores is on average 3.4 times faster than that running on one core. Due to the memory intensive nature of graph algorithms, the scalability of MTGL is limited by the memory bandwidth.

Figure 8 shows the speedup for Medusa over MTGL running on 12 cores. The *speedup* is defined as the ratio between the elapsed time of the CPU-based execution and that of Medusa-based execution. PageRank is executed with 100 iterations. Medusa is significantly faster than MTGL on most comparisons and delivers a

Table 2 Characteristics of graphs used in the experiments

Graph	Vertices (10^6)	Edges (10^6)	Max d	Avg d	σ
RMAT	1.0	16.0	1,742	16	32.9
Random (Rand)	1.0	16.0	38	16	4.0
BIP	4.0	16.0	40	4	5.1
WikiTalk (Wiki)	2.4	5.0	100,022	2.1	99.9
RoadNet-CA (Road)	2.0	5.5	12	2.8	1.0
kkt_power (KKT)	2.1	13.0	95	6.3	7.5
coPapersCiteseer (Cite)	0.4	32.1	1,188	73.9	101.3
hugebubbles-00020 (Huge)	21.2	63.6	3	3.0	0.03

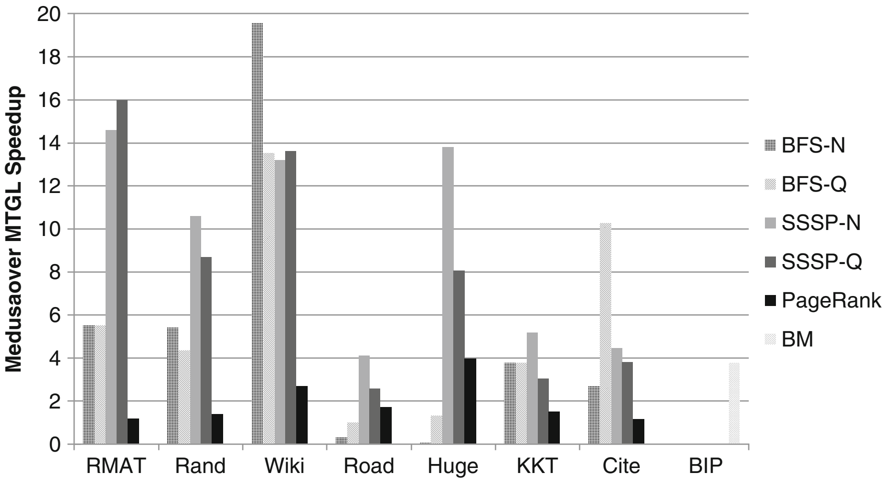

Fig. 8 Performance speedup of Medusa running on the GPU over MTGL [13] running on 12 cores

performance speedup of 1.0–19.6 with an average of 5.5. On some graphs such as Road, BFS-N is notably slower than MTGL-based BFS, because the work-inefficient issue of BFS-N is exaggerated on the graphs with large diameter.

The work-efficient BFS and SSSP algorithms (BFS-Q and SSSP-Q) achieve better performance on the graphs with large diameters, and can degrade the performance in some cases (e.g., Rand, Wiki and KKT) due to the computation and memory overhead in maintaining the queue structure. This is consistent with the previous studies [37].

4.2 Parallel Graph Processing on the Cloud

Large graph processing has become popular for various data-intensive applications on increasingly large web and social networks [43, 44]. Due to the ever increasing size of graphs, application deployments are moving from a small number of HPC servers or supercomputers [28, 46] towards the Cloud with a large number of commodity servers [44, 54]. Early studies on parallel graph processing in the Cloud are to adopt existing distributed data-intensive computing techniques in the Cloud [19,39]. Most of these studies [43,44,77] are built on top of MapReduce [19], which is suitable for processing flat data structure, not particularly for graph structured data. More recently, systems such as Pregel [54], Trinity [63] and Surfer [16] have been developed specifically for large graph processing. These systems support a vertex-oriented execution model and allow users to develop custom logics on vertices. The Medusa system [76] has been extended to support the GPU-enabled cloud environment [75]. In those Cloud-based graph processing systems, network performance optimizations are the key for improving the overall performance.

Most vertex-oriented graph processing systems share the same network performance issue. Take Pregel as an example. Pregel executes user-defined function *Compute*() per vertex in parallel, based on the general bulk synchronous parallel (BSP) model. By default, the vertices can be stored in different machines according to a simple hash function. However, the simple partitioning function leads to heavy network traffic in graph processing tasks. For example, if we want to compute the two-hop friend list for each account in a social network, every friend (vertex) must first send its friends to each of its neighbors, and then each vertex combines the friend lists of its neighbors. Implemented with the simple partitioning scheme, this operation results in a great amount of network traffic because of shuffling the vertices.

A traditional way of reducing data shuffling in distributed graph processing is graph partitioning [22, 45, 49]. Graph partitioning minimizes the total number of cross-partition edges among partitions in order to minimize data transfer. The commonly used distributed graph processing algorithms are multi-level algorithms [46, 47, 69]. These algorithms recursively divide the graph into multiple partitions with bisections according to different heuristics.

It is well understood that large graphs should be partitioned; however, little attention is given to how graph partitioning can be effectively integrated into the processing in the Cloud environment. There are a number of challenging issues in the integration. First, graph partitioning itself is a very costly task, generating lots of network traffic. Moreover, partitioned graph storage and vertex-oriented graph processing need a careful revisit in the context of Cloud. The Cloud network environment is significantly different from those in previous studies [46, 47, 49], e.g., Cray supercomputers or a small cluster. The network bandwidth is often the same for every machine pair in a small cluster. However, the network bandwidth of the Cloud environment is uneven among different machine pairs. Current Cloud infrastructures are often based on tree topology [12, 30, 42]. Machines are first grouped into *pods*, and then pods are connected to higher-level switches. The intra-pod bandwidth is much higher than the cross-pod bandwidth. Even worse, the topology information is usually unavailable to users due to virtualization techniques in the Cloud. In practice, such network bandwidth unevenness has been confirmed by both Cloud providers and users [12,42]. It requires careful network optimizations and tuning on graph partitioning and processing.

We briefly describe the approach adopted by Surfer [16]. Surfer uses a network performance aware graph partitioning framework to improve the network performance of large graph processing on partitioned graphs. Specifically, the graph partitions generated from the framework improve the network performance of graph processing tasks. To capture the network bandwidth unevenness, Surfer models the machines chosen for graph processing as a complete undirected graph (namely *machine graph*): each machine as a vertex, and the bandwidth between any two machines as the weight of an edge. The network performance aware framework recursively partitions the data graph, as well as the machine graph, with bisection correspondingly. That is, the bisection on the data graph is performed with the corresponding set of machines selected from the bisection on the machine graph. The recursion terminates when the data graph partition can fit into main memory. By partitioning the data graph and machine graph simultaneously, the number of cross-partition edges among data graph partitions is gracefully adapted to the aggregated amount of bandwidth among machine graph partitions. To exploit the data locality of graph partitions, Surfer develops *hierarchical combination* to exploit network bandwidth unevenness in order to improve the network performance.

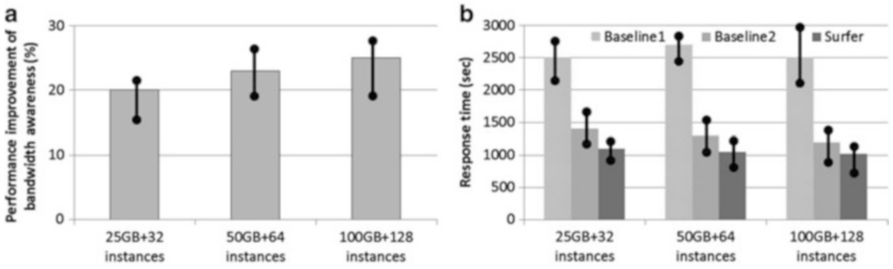

Fig. 9 Network performance aware graph partitioning (**a**) and NR (**b**) on Amazon EC2 with the number of medium instances varied

Surfer has been evaluated on a real-world social network and synthetic graphs of over 100 GB each in a 32-node cluster as well as on Amazon EC2. We briefly discuss the experimental results on network performance aware graph partitioning and graph processing on Amazon EC2.

We compare Surfer with two baselines: “Baseline 1” is the baseline Surfer with local combination, but with graph partition storage distribution generated from ParMetis, and “Baseline 2” is “Baseline 1” with the bandwidth aware graph partitioning, without hierarchical combination. Figure 9a shows the performance improvement of the network bandwidth-aware optimization on graph partitioning, and Fig. 9b compares the response time of *Network ranking (*NR*)* with different approaches. NR is to generate a ranking on the vertices in the graph using PageRank [58] or its variants. In the experiment, the number of medium instances is increased from 32 to 128 and the size of synthetic graphs is increased from 25 to 100 GB. We measure 100 times of each experiment on the same set of instances, and report the average and the range for the elapsed time of graph partitioning and processing. The variation is acceptable in Amazon EC2. Due to the network bandwidth unevenness in Amazon EC2, the network performance aware optimizations improve both graph partitioning and processing, with 20–25 % performance improvement for graph partitioning and with 49 and 18 % performance improvement for NR over Baseline 1 and 2 respectively. This demonstrates the effectiveness of the network performance aware optimizations of Surfer on the public Cloud environment.

5 Summary and Open Problems

In this chapter, we have introduced the MapReduce implementations on GPU clusters, as well as two state-of-the-art graph processing frameworks running on GPUs and the Cloud.

The MapReduce model is designed originally for big data processing on large clusters, thus scalability is a very important feature. The approaches of integrating GPU parallelism into Hadoop [3] using various technologies such as Hadoop streaming, Hadoop pipes, JCUDA and Java Native Interface can inherit the outstanding scalability and fault-tolerance features of the Hadoop framework. However they usually incur low efficiency due to inter-process communication cost and under-utilization of GPU resources. Moreover, only the map and reduce stages are parallelized whereas the time-consuming sort process remains sequential.

Similar to Hadoop-based work, the Mars-MR-MPI can also process out-of-memory data sets with the support of the MR-MPI framework. Moreover, it is more efficient than the Hadoop-based work using streaming. However, its performance is still limited by the network I/O cost. The stand-alone GPMR framework [65] exposes the GPU programming details to the users, which makes the programming more complex, but can achieve a better performance if the program is well-tuned. GPMR processes input data in chunks. However, due to the lack of a buffer mechanism during the MapReduce pipeline, it cannot handle data sets exceeding the main memory capacity. The scalability of GPMR is still limited, though it involves less overhead and is more optimized than the Hadoop-based GPU MapReduce work.

Graph are very common in data-intensive applications. Compared with other data-intensive applications such as text processing, graph applications are usually more complex and need more computation and communication. To cope with challenges in processing large graphs, several general graph frameworks have been proposed in recent years. Medusa is a representative vertex-oriented graph processing framework on GPUs. It simplifies the graph processing on GPUs and achieves comparable or better performance than the manually tuned GPU graph operations. Surfer focuses on improving the network performance of graph processing on the Cloud by employing network performance-aware graph partitioning strategies. As such, both the graph partitioning and processing efficiency can be improved in a public Cloud environment.

We conclude the chapter with a few open problems as the future research directions.

1. For both MapReduce and general graph processing, data communication i.e., network I/O is the performance bottleneck, which limits the scalability of MapReduce and graph processing algorithms on multi-GPUs and clusters. The experiment results of GPMR show that the speedup of most applications decreases dramatically when there are tens of GPUs. Zhong et al. [76] used up to four GPUs in one machine to accelerate a set of common graph algorithms such as BFS and PageRank. Their study shows that scalability for graph algorithms with light weight computation is poor since the inter-GPU communication cost can easily become the bottleneck, and the scalability issue can be magnified in a distributed environment. To reduce network I/O, we may consider using more advanced communication techniques provided by hardware vendors such as GPUDirect. We can also apply GPU or CPU based data compression algorithms [24] to reduce the amount of data transfer at the cost of increased computation.
2. Out-of-core support is missing in most GPU-based systems. In recent years we have witnessed significant improvement in the computation capability of the GPU, however, the capacity of the GPU memory rarely increases. Yet most data-intensive applications involve data that exceeds the aggregated main memory size. Currently single-GPU MapReduce implementations [14, 25, 32, 36, 40] can only process in-memory data. GPMR processes data in chunks, however, it cannot provide full out-of-core support when data size exceeds main memory capacity. Existing studies on GPU graph processing also mainly deal with in-memory processing of graphs [31, 37, 38, 55, 56, 76]. With several Gigabytes of GPU memory, the size of the maximum input graph is limited to millions of vertices and edges. External memory CPU algorithms have been widely adopted by many applications, including Hadoop shuffle and graph processing [50], for processing data larger than main memory. However, adopting external memory algorithms for GPU is challenging due to the larger gap between GPU memory bandwidth and disk bandwidth, as well as the overhead of PCIe data transfer.
3. For GPU-based MapReduce, variable-length data (keys and values) processing is challenging. To handle variable-length data, Mars uses a lock-free method by pre-computing the key and value length before emitting the actual key/value pair, and some other works use atomic operations. Both approaches bring extra overheads, and the performance of each approach is application dependent. Moreover, variable-length data leads to un-coalesced memory access on GPU.
4. Dynamic Graph Processing. Real world graphs, such as the social networks, are usually evolving. Also, some graph algorithms require changing the graph structure during runtime. However, currently there is little work on GPU-based dynamic graph processing. Narse et al. [56] presented implementations of five graph algorithms which morph structure of the input graph in different ways. They provide a set of basic methods for adding and deleting subgraphs and

require users to make their choices based on the application characteristics and the scale of the problem. The applicability of their methods are application dependent and requires non-trivial programming efforts to implement.

References

1. 10th DIMACS implementation challenge. http://www.cc.gatech.edu/dimacs10/index.shtml
2. AMD Aparapi. http://developer.amd.com/tools-and-sdks/opencl-zone/opencl-libraries/aparapi
3. Apache Hadoop. http://hadoop.apache.org
4. GTGraph generator. http://www.cse.psu.edu/~madduri/software/GTgraph/index.html
5. Hadoop Streaming. http://hadoop.apache.org/docs/stable/streaming.html
6. MapReduce-MPI Documentation. http://mapreduce.sandia.gov/doc/Technical.html/Manual.html
7. Stanford large network dataset collections. http://snap.stanford.edu/data/index.htm
8. Abbasi, A., Khunjush, F., Azimi, R.: A preliminary study of incorporating GPUs in the Hadoop framework. In: 2012 16th CSI International Symposium on Computer Architecture and Digital Systems (CADS'12), pp. 178–185. IEEE (2012)
9. Ahmad, F., Lee, S., Thottethodi, M., Vijaykumar, T.: PUMA: Purdue MapReduce Benchmarks Suite. http://web.ics.purdue.edu/~fahmad/papers/puma.pdf
10. Anderson, T.E., Owicki, S.S., Saxe, J.B., Thacker, C.P.: High-speed switch scheduling for local-area networks. ACM Transactions on Computer Systems (TOCS) **11**, 319–352 (1993)
11. Basaran, C., Kang, K.D.: Grex: An efficient MapReduce framework for graphics processing units. Journal of Parallel and Distributed Computing **73**(4), 522–533 (2013)
12. Benson, T., Akella, A., Maltz, D.A.: Network traffic characteristics of data centers in the wild. In: Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement, pp. 267–280. ACM (2010)
13. Berry, J., Hendrickson, B., Kahan, S., Konecny, P.: Software and Algorithms for Graph Queries on Multithreaded Architectures. In: IEEE International Parallel and Distributed Processing Symposium (IPDPS'07), pp. 1–14. IEEE (2007)
14. Chen, L., Agrawal, G.: Optimizing MapReduce for GPUs with effective shared memory usage. In: Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing, pp. 199–210. ACM (2012)
15. Chen, L., Huo, X., Agrawal, G.: Accelerating MapReduce on a coupled CPU-GPU architecture. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, p. 25. IEEE Computer Society Press (2012)
16. Chen, R., Yang, M., Weng, X., Choi, B., He, B., Li, X.: Improving Large Graph Processing on Partitioned Graphs in the Cloud. In: Proceedings of the Third ACM Symposium on Cloud Computing (SoCC'12), pp. 3:1–3:13 (2012)
17. Chen, Y., Qiao, Z., Davis, S., Jiang, H., Li, K.C.: Pipelined Multi-GPU MapReduce for Big-Data Processing. In: Computer and Information Science, pp. 231–246. Springer (2013)
18. Chen, Y., Qiao, Z., Jiang, H., Li, K.C., Ro, W.W.: MGMR: Multi-GPU based MapReduce. In: Grid and Pervasive Computing, pp. 433–442. Springer (2013)
19. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: Proceedings of the 6th Conference on Symposium on Opearting Systems Design and Implementation (OSDI'04) (2004)
20. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. Communications of the ACM **51**(1), 107–113 (2008)

21. Delorimier, M., Kapre, N., Mehta, N., Rizzo, D., Eslick, I., Rubin, R., Uribe, T.E., Knight, T.F., Dehon, A.: GraphStep: A System Architecture for Sparse-Graph Algorithms. In: 2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06), pp. 143–151. IEEE (2006)
22. Derbel, B., Mosbah, M., Zemmari, A.: Fast distributed graph partition and application. In: IPDPS (2006)
23. Elteir, M., Lin, H., Feng, W.c., Scogland, T.: StreamMR: an optimized MapReduce framework for AMD GPUs. In: 2011 IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS'11), pp. 364–371. IEEE (2011)
24. Fang, W., He, B., Luo, Q.: Database compression on graphics processors. Proceedings of the VLDB Endowment **3**(1–2), 670–680 (2010)
25. Fang, W., He, B., Luo, Q., Govindaraju, N.K.: Mars: Accelerating MapReduce with graphics processors. IEEE Transactions on Parallel and Distributed Systems (TPDS) **22**(4), 608–620 (2011)
26. Fang, W., Lu, M., Xiao, X., He, B., Luo, Q.: Frequent itemset mining on graphics processors. In: Proceedings of the 5th International Workshop on Data Management on New Hardware (DaMoN'09), pp. 34–42 (2009)
27. Farivar, R., Verma, A., Chan, E.M., Campbell, R.H.: MITHRA: Multiple data independent tasks on a heterogeneous resource architecture. In: 2009 IEEE International Conference on Cluster Computing and Workshops (CLUSTER'09), pp. 1–10. IEEE (2009)
28. Gregor, D., Lumsdaine, A.: The Parallel BGL: A generic library for distributed graph computations. In: Parallel Object-Oriented Scientific Computing (POOSC) (2005)
29. Grossman, M., Breternitz, M., Sarkar, V.: HadoopCL: MapReduce on Distributed Heterogeneous Platforms Through Seamless Integration of Hadoop and OpenCL. In: Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum, pp. 1918–1927. IEEE Computer Society (2013)
30. Guo, C., Wu, H., Tan, K., Shi, L., Zhang, Y., Lu, S.: Dcell: a scalable and fault-tolerant network structure for data centers. In: ACM SIGCOMM Computer Communication Review, vol. 38, pp. 75–86. ACM (2008)
31. Harish, P., Narayanan, P.: Accelerating large graph algorithms on the GPU using CUDA. In: High performance computing (HiPC'07), pp. 197–208. Springer (2007)
32. He, B., Fang, W., Luo, Q., Govindaraju, N.K., Wang, T.: Mars: a MapReduce framework on graphics processors. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT'08), pp. 260–269. ACM (2008)
33. He, B., Lu, M., Yang, K., Fang, R., Govindaraju, N.K., Luo, Q., Sander, P.V.: Relational query coprocessing on graphics processors. ACM Transactions on Database Systems (TODS) **34**(4), 21:1–21:39 (2009)
34. He, B., Yu, J.X.: High-throughput transaction executions on graphics processors. Proceedings of the VLDB Endowment **4**(5), 314–325 (2011)
35. He, G., Feng, H., Li, C., Chen, H.: Parallel SimRank computation on large graphs with iterative aggregation. In: Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 543–552. ACM (2010)
36. Hong, C., Chen, D., Chen, W., Zheng, W., Lin, H.: MapCG: writing parallel program portable between CPU and GPU. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT'10), pp. 217–226. ACM (2010)
37. Hong, S., Kim, S.K., Oguntebi, T., Olukotun, K.: Accelerating CUDA graph algorithms at maximum warp. In: Proceedings of the 16th ACM symposium on Principles and Practice of Parallel Programming (PPoPP'11), pp. 267–276 (2011)
38. Hong, S., Oguntebi, T., Olukotun, K.: Efficient parallel graph exploration on multi-core CPU and GPU. In: 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT'11), pp. 78–88. IEEE (2011)
39. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: distributed data-parallel programs from sequential building blocks. ACM SIGOPS Operating Systems Review **41**(3), 59–72 (2007)

40. Ji, F., Ma, X.: Using shared memory to accelerate MapReduce on graphics processing units. In: 2011 IEEE International Parallel and Distributed Processing Symposium (IPDPS'11), pp. 805–816. IEEE (2011)
41. Jiang, H., Chen, Y., Qiao, Z., Li, K.C., Ro, W., Gaudiot, J.L.: Accelerating MapReduce framework on multi-GPU systems. Cluster Computing pp. 1–9 (2013)
42. Kandula, S., Sengupta, S., Greenberg, A., Patel, P., Chaiken, R.: The nature of data center traffic: measurements & analysis. In: Proceedings of the 9th ACM SIGCOMM conference on Internet Measurement Conference (IMC'09), pp. 202–208. ACM (2009)
43. Kang, U., Tsourakakis, C., Appel, A.P., Faloutsos, C., Leskovec, J.: HADI: Fast diameter estimation and mining in massive graphs with Hadoop. Tech. Rep. CMU-ML-08-117, Carnegie Mellon University (2008)
44. Kang, U., Tsourakakis, C.E., Faloutsos, C.: Pegasus: A peta-scale graph mining system - implementation and observations. In: 2009 9th IEEE International Conference on Data Mining (ICDM'09), pp. 229–238. IEEE (2009)
45. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM Journal on Scientific Computing **20**(1), 359–392 (1998)
46. Karypis, G., Kumar, V.: A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. Journal of Parallel and Distributed Computing **48**(1), 71–95 (1998)
47. Karypis, G., Kumar, V.: Parallel multilevel k-way partitioning scheme for irregular graphs. Journal of Parallel and Distributed computing **48**(1), 96–129 (1998)
48. Katz, G.J., Kider Jr, J.T.: All-pairs shortest-paths for large graphs on the GPU. In: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics Hardware, pp. 47–55. Eurographics Association (2008)
49. Koranne, S.: A distributed algorithm for k-way graph partitioning. In: Proceedings of the 25th Conference of EUROMICRO (EUROMICRO'99), vol. 2, pp. 446–448. IEEE (1999)
50. Kyrola, A., Blelloch, G., Guestrin, C.: GraphChi: Large-scale graph computation on just a PC. In: Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12), pp. 31–46 (2012)
51. Lin, J., Schatz, M.: Design patterns for efficient graph algorithms in MapReduce. In: Proceedings of the Eighth Workshop on Mining and Learning with Graphs (MLG'10), pp. 78–85. ACM (2010)
52. Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.M.: GraphLab: A new parallel framework for machine learning. In: The 26th Conference on Uncertainty in Artificial Intelligence (UAI'10) (2010)
53. Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.M.: Distributed GraphLab: A framework for machine learning and data mining in the cloud. Proceedings of the VLDB Endowment **5**(8), 716–727 (2012)
54. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: A System for Large-Scale Graph Processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD'10), pp. 135–146. ACM (2010)
55. Merrill, D., Garland, M., Grimshaw, A.: Scalable GPU graph traversal. In: Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP'12), pp. 117–128 (2012)
56. Nasre, R., Burtscher, M., Pingali, K.: Morph algorithms on GPUs. In: Proceedings of the 18th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP'13), pp. 147–156 (2013)
57. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A.E., Purcell, T.J.: A Survey of General-Purpose Computation on Graphics Hardware. In: Computer Graphics Forum, vol. 26, pp. 80–113. Wiley Online Library (2007)
58. Page, L., Brin, S., Motwani, R., Winograd, T.: The PageRank citation ranking: Bringing order to the Web. Stanford InfoLab. Technical report (1999)
59. Plimpton, S.J., Devine, K.D.: MapReduce in MPI for large-scale graph algorithms. Parallel Computing **37**(9), 610–632 (2011)

60. Plimpton, S and Devine, K: MapReduce-MPI Library. http://mapreduce.sandia.gov
61. Rafique, M.M., Rose, B., Butt, A.R., Nikolopoulos, D.S.: CellMR: A framework for supporting MapReduce on asymmetric cell-based clusters. In: 2009 IEEE International Parallel and Distributed Processing Symposium (IPDPS'09), pp. 1–12. IEEE (2009)
62. Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., Kozyrakis, C.: Evaluating MapReduce for multi-core and multiprocessor systems. In: IEEE 13th International Symposium on High Performance Computer Architecture (HPCA'07), pp. 13–24. IEEE (2007)
63. Shao, B., Wang, H., Li, Y.: Trinity: A distributed graph engine on a memory cloud. In: Proceedings of the 2013 ACM International Conference on Management of Data (SIGMOD'13), New York, New York, USA (2013)
64. Shirahata, K., Sato, H., Matsuoka, S.: Hybrid Map task scheduling for GPU-based heterogeneous clusters. In: 2010 IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom'10), pp. 733–740. IEEE (2010)
65. Stuart, J.A., Owens, J.D.: Multi-GPU MapReduce on GPU clusters. In: 2011 IEEE International Parallel and Distributed Processing Symposium (IPDPS'11), pp. 1068–1079. IEEE (2011)
66. Sul, S.J., Tovchigrechko, A.: Parallelizing BLAST and SOM algorithms with MapReduce-MPI library. In: IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW'11), pp. 481–489. IEEE (2011)
67. Talbot, J., Yoo, R.M., Kozyrakis, C.: Phoenix++: Modular MapReduce for Shared-Memory Systems. In: Proceedings of the 2nd International Workshop on MapReduce and its Applications, pp. 9–16. ACM (2011)
68. Tan, Y.S., Lee, B.S., He, B., Campbell, R.H.: A Map-Reduce based Framework for Heterogeneous Processing Element Cluster Environments. In: IEEE/ACM 12th International Symposium on Cluster, Cloud and Grid Computing (CCGrid'12), pp. 57–64. IEEE (2012)
69. Trifunović, A., Knottenbelt, W.J.: Parallel Multilevel Algorithms for Hypergraph Partitioning. Journal of Parallel and Distributed Computing **68**, 563–581 (2008)
70. Vineet, V., Narayanan, P.J.: CUDA cuts: Fast graph cuts on the GPU. In: IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (CVPRW'08), pp. 1–8. IEEE (2008)
71. Wittek, P., Darányi, S.: Leveraging on High-Performance Computing and Cloud Technologies in Digital Libraries: A Case Study. In: IEEE Third International Conference on Cloud Computing Technology and Science (CloudCom'11), pp. 606–611. IEEE (2011)
72. Yoo, R.M., Romano, A., Kozyrakis, C.: Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system. In: IEEE International Symposium on Workload Characterization (IISWC'09), pp. 198–207. IEEE (2009)
73. Zhai, Y., Mbarushimana, E., Li, W., Zhang, J., Guo, Y.: Lit: A high performance massive data computing framework based on CPU/GPU cluster. In: IEEE International Conference on Cluster Computing (CLUSTER'13), pp. 1–8. IEEE (2013)
74. Zhong, J., He, B.: Parallel Graph Processing on Graphics Processors Made Easy. Proceedings of the VLDB Endowment **6**(12), 1270–1273 (2013)
75. Zhong, J., He, B.: Towards GPU-Accelerated Large-Scale Graph Processing in the Cloud. In: IEEE Third International Conference on Cloud Computing Technology and Science (CloudCom'13), pp. 9–16. IEEE (2013)
76. Zhong, J., He, B.: Medusa: Simplified Graph Processing on GPUs. IEEE Transactions on Parallel and Distributed Systems (TPDS) **25**(6), 1543–1552 (2014)
77. Zhou, A., Qian, W., Tao, D., Ma, Q.: DISG: A DIStributed Graph Repository for Web Infrastructure (Invited Paper). Proceedings of the 2008 Second International Symposium on Universal Communication **0**, 141–145 (2008)