A Customizable MapReduce Framework for Complex Data-Intensive Workflows on GPUs

Zhi Qiao*, Shuwen Liang*, Hai Jiang† and Song Fu‡
*‡Department of Computer Science and Engineering, University of North Texas
†Department of Computer Science, Arkansas State University
*Email: {ZhiQiao, ShuwenLiang}@my.unt.edu
†Email: hjiang@astate.edu
‡Email: song.fu@unt.edu

***Abstract*—The MapReduce programming model has been widely used in big data and cloud applications. Criticism on its inflexibility when being applied to complicated scientific applications recently emerges. Several techniques have been proposed to enhance its flexibility. However, some of them exert special requirements on applications, while others fail to support the increasingly popular coprocessors, such as Graphics Processing Unit (GPU). In this paper, we propose *MR-Graph*, a customizable and unified framework for GPU-based MapReduce, which aims to improve the flexibility and performance of MapReduce. *MR-Graph* addresses the limitations and restrictions of the traditional MapReduce execution paradigm. The three execution modes integrated in *MR-Graph* facilitates users to write their applications in a more flexible fashion by defining a Map and Reduce function call graph. *MR-Graph* efficiently explores the memory hierarchy in GPUs to reduce the data transfer overhead between execution stages and accommodate big data applications. We have implemented a prototype of *MR-Graph* and experimental results show the effectiveness of using *MR-Graph* for flexible and scalable GPU-based MapReduce computing.**

***Keywords*—** MapReduce, GPU, Customizable, Data Intensive, Iterative, Recursive

I. Introduction

The performance of CPUs has stagnated while both the programmability and availability of Graphics Processing Units (GPUs) have been improved dramatically. Modern GPU are equipped with thousands of processing units and is particularly suitable for data-intensive computations and massively parallel applications. However, GPU's programmability is still a challenge. The complicated and highly parallele architecture makes GPU difficult to extract massively parallel computing components from applications and schedule them efficiently on a huge pool of processing resources.

The MapReduce programming model provides an easier way for parallel data processing in the cloud. With MapReduce, users need only write Map and Reduce functions to solve problems in parallel. The underlying programming details, such as how to handle communication among data nodes, are transparent to users. Data affinity across the network and fault tolerance among multiple nodes can be achieved automatically, which allows programmers to focus more on the target problem itself and the algorithm design.

Because of its massive parallelism, GPU has been explored for MapReduce execution and processing large datasets. In today's scientific analysis applications, not only the data volume grows fast, but also the data type and data dependency become much more complicated [1]. However, most existing GPU-based MapReduce implementations use the traditional MapReduce programming model, which processes data in a single-step fashion with one round of Map followed by another round of Reduce operations. This makes applications with complex data dependency difficult and even impossible to be implemented and executed by MapReduce.

In this paper, we present *MR-Graph*, a customizable GPU-based MapReduce framework, which is designed for complex data intensive scientific analysis applications. *MR-Graph* provides multiple configurable options or modes, allowing users to customize their MapReduce workflow and data locality. The new execution paradigm enables *MR-Graph* to handle complicated data dependency effectively. *MR-Graph* is designed to have a two-layer structure, in which the lower level is the *execution unit* for user-defined computations on the corresponding data items, while the upper layer is a *graph-based job control unit* to coordinate tasks and communication between each task.

MapReduce applications are classified by *MR-Graph* into *iterative*, *recursive* and *hybrid* modes based on the characteristics of an application's computations. Programmers can thus organize the computations of their applications in the proper mode and provide the configuration information to the job control unit. Then, user-defined map and reduce functions can be run on the execution unit. We have implemented a prototype of *MR-Graph* based on Thrust, a light-weight CUDA C++ library, with optimization for guaranteed performance. Experimental results show the flexibility and effectiveness of *MR-Graph* running MapReduce applications on GPUs.

The remainder of this paper is organized as follows. Section II presents the background and related workson MapReduce and GPU computing, as well as the challenges for complex applications. The three MapReduce computation modes are described in Section III. Section IV presents the design of the proposed *MR-Graph* framework for flexible MapReduce execution on GPUs and the implementation issues. Experimental results are shown in Section V. Section VI concludes

978-1-4673-8590-9/15/$31.00 ©2015 IEEE

this paper with remarks on the future work.

II. Background and Related Work

A. MapReduce Programming Model

MapReduce is a functional programming model that simplifies data parallelism based on two primitives, `Map` and `Reduce`. Map function performs the procedure similar to filtering, and Reduce function performs the summarization operation. MapReduce was originally introduced by Google [2] for their large-scale data processing in distributed computing environments, and it has been widely adopted. It has attracted huge attention due to its simple design and significant performance gains. MapReduce has been applied to various applications in business and scientific computing domains. Several commercial and research implementations have been developed.

Hadoop is a well-known open-source MapReduce implementation licensed by Apache. Dispite the fact that it has been widely used, the resilience centric design significantly impact its overall performance. Other popular Big Data processing platform such as Apache Spark [3] and Apache Flink [4] add the support to in-memory processing, steam processing and iterative algorithms thanks to the rich extension provided by research community. Recent works even shows the usage of Spark in GPU environment [5]. However, Spark lack of support to dataset larger than RAM before version 1.5, whereas Flink does not provide integrated storage system. In addition, many extensions that enrich the functionality of both are require to run on top of Hadoop, which introduce the extra system overhead and redundency. Mars [6] and GPMR [7] are two MapReduce frameworks used on GPU platforms, which only support traditional MapReduce model. Twister [8] is a CPU-based MapReduce runtime focusing on iterative MapReduce. Carlos et al. [9] present a flexible MapReduce workflow for data analysis. However, it is only applicable to the AWARD (Autonomic Workflow Activities Reconfigurable and Dynamic) framework. Other implementations, such as Pilot-MapReduce [10], limit their applications to specific abstractions, e.g., Pilot.

B. GPU Architecture and CUDA

GPU has been widely used as a general-purpose high-throughput computing device. A GPU contains a large number of Single-Instruction-Multiple-Data (SIMD) multiprocessors, that can run thousands of threads concurrently. Compared with CPU, which provides a low latency and low throughput and performs well for sequential and low-parallelism jobs, GPU provides a high throughput and high latency and excels in massive data parallel applications. Since MapReduce applications typically involve high data parallelism, GPU better suits their needs.

Compute Unified Device Architecture (CUDA) is a general-purpose parallel computing and programming environment [11], which facilitates the application programmers to rein in the fiery GPU computability. It leverages the parallel computing engine in GPUs to solve complex computational problems in a more efficient way than on a CPU. CUDA allows application developers to use C as a high-level programming language with many useful tools/libraries, such as Thrust [12], a C++ template library that allows user to implement high-performance parallel code with minimal programming efforts. In this paper, we implement the proposed *MR-Graph* framework by using Thrust.

C. Challenges for MapReduce in the Cloud

The traditional MapReduce model works well for applications with large datasets, although it supports only a single-round process, that is one round of Map followed by one round of Reduce. However, the size and complexity of applications' data increase tremendously in the cloud. The single-round process significantly limits the application of MapReduce in the cloud computing environment, which is elaborated from the following three aspects.

1) Communication Overhead: MapReduce adopts the divide-and-conquer technique to break large data into smaller pieces, which are reordered and reassembled to produce the final outputs. An important side effect of this simple yet powerful model is the heavy communication overhead. Traditionally, MapReduce launches many worker nodes to handle Map and Reduce tasks. Each worker node takes a copy of the data to process, and then sends the results back to the master node. When thousands of workers transfer data simultaneously, traffic congestion will inevitably happen, especially when data blocks are large. Moreover, in the GPU architecture, data need to be copied further to the GPU memory before being processed. This extra hardware layer makes the communication overhead even higher. Several implementations minimize this overhead by moving the computation, instead of data, to the worker nodes that have the corresponding data stored locally. But these solutions raise another problem, that is data stored on different nodes have to be independent. If the MapReduce tasks need to exchange data between nodes, the communication overhead will again become a major problem.

2) Acyclic Workflow: MapReduce was originally designed for acyclic dataflow where data are processed in the batch mode. First, the MapReduce runtime gathers and prepares all required data. Then a large number of workers are launched to execute Map and Reduce functions in two sequential phases. Once all tasks are completed, the final results are generated. However, MapReduce does not work well for iterative algorithms, which are widely used in big data processing, such as data mining and machine learning. A typical iterative algorithm, such as K-Means, requires revisiting the same dataset multiple times to produce gradually developed or frequently updated data. Application programmers can write scripts to repeatedly launch MapReduce jobs with updated data. However, this needs to copy large sets of data back and forth, causing considerable communicate overhead. Since MapReduce is designed to produce stateless programs, any attempt to reuse the data from a previous round without data movement requires the modification of the runtime system.

3) Ordinary Data: Due to the high communication overhead of data movement from the Map tasks to the Reduce tasks, MapReduce is usually applied to big datasets to mitigate the cost. However, the simplicity of the programming model and the stateless implementation make MapReduce not easy to be applied to handle applications with complex datasets, such as data clustering, machine learning and computer vision. The current MapReduce model requires application programmers to convert the raw data to a MapReduce-ready format. This preprocessing can take several stages in order to eliminate data dependency and noises and to extract useful information for execution. Once all the raw data are precisely organized and fine-tuned into big yet simple datasets, the MapReduce application can start execution. In this process, again, the involved datasets are copied back and forth between disks and memory multiply times, making the execution inefficient.

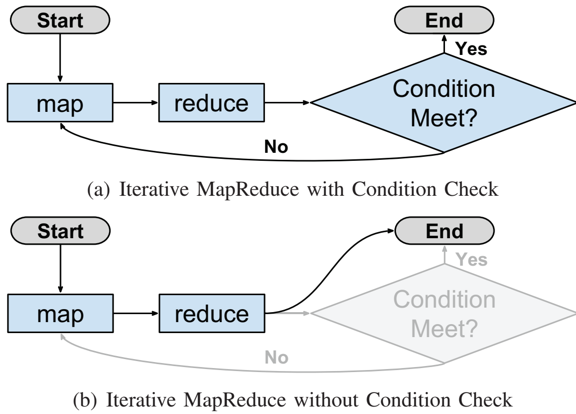
III. MapReduce Modes

To address the limitations of the MapReduce programming model, we have done a comprehensive analysis of the programming patterns of many complex scientific applications. To adapt MapReduce to different applications, we categorize the operation patterns into three modes, which are *Iterative*, *Recursive* and *Hybrid* modes.

A. Iterative Mode

Iterative algorithms, such as those in machine learning and data clustering applications, requires the MapReduce dataflow to access the intermediate data multiple times to generate gradually developed or frequently updated data. These data are dynamically produced and evolved through iterations in the execution.

The traditional MapReduce programming model processes the input dataset in a single-round fashion, as discussed in Section II-C2. These data, are usually referred to as static data, having little data dependency. Therefore, an *iterative mode* is required for MapReduce to process the dynamic data in those iterative applications.



(a) Iterative MapReduce with Condition Check

(b) Iterative MapReduce without Condition Check

Fig. 1. Iterative MapReduce Mode

Figure 1(a) shows the flowchart of our generalized iterative MapReduce. During the execution, *map* initially reads data from the input files. Then the produced data is sorted and summarized by *reduce*. After that, a condition is checked to determine whether to continue the execution for the next iteration or not. In the former case, iterative MapReduce loads intermediate data as a new input, performs map and reduce operations until the condition defined by the user is satisfied.

As an example, K-Means Clustering (KMC) algorithm can be implemented with our iterative MapReduce mode. KMC aims to partition n observations into k clusters where all the observations in a cluster are close to the nearest mean. In real implementation, it usually takes a set of points in space and determines the clusters that can best approximate the space. It also uses a fix-sized, randomly generated set of cluster centers to start with. Since it may take a large number of rounds to reach a theoretically close result, in practice, the number of iteration is usually predefined and fixed for a quick convergence. If the condition check is disabled, the iterative mode will be reduced to the traditional MapReduce operation model with a single-round processing, as shown in Figure 1(b).

B. Recursive Mode

For many Big Data applications, the raw data are either too complex or contain a large amount of noise that cannot be processed by MapReduce directly. Data pre-processing and post-processing become a labor-intensive and time-consuming task in addition to the MapReduce tasks. For instance, an extremely large dataset usually requires researchers to adopt a multi-stage approach, which partitions data into reasonable sizes, and then processes each part separately. Similarly, for large intermediate data, partial reduction of the sub-stages is usually used to minimize the communication overhead.

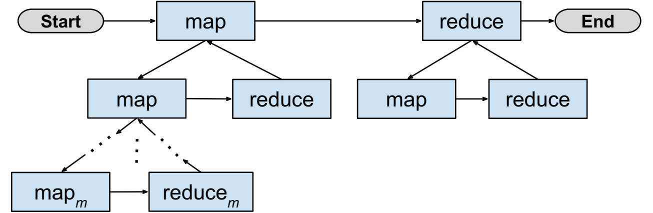


Fig. 2. Recursive MapReduce Mode

Recursive MapReduce mode binds multiple *map()* and *reduce()* functions together for data pre-processing, post-processing and refinement. Different from the *iterative mode* where the MapReduce workflow evolves horizontally, the *recursive mode* evolves vertically for a fine-grained data processing, as shown in Figure 2. Traditional MapReduce model cannot easily handle such situations as it does not allow multiple consecutive *map()* or *reduce()* functions. The original jobs have to be divided into multiple stages, each with a corresponding function. If the intermediate data between them are saved on the disk storage as in original MapReduce, the I/O overhead is significant. Flexible MapReduce frameworks help address this by allowing many *map()* or *reduce()* functions to be executed consecutively and keeping the intermediate data in memory for fast reuse.

C. Hybrid Mode

Not all MapReduce execution patterns fall into either iterative or recurve MapReduce modes. A combination of both may better suit them. This *hybrid mode* is more flexible and can be applied to complicated cases. It is created by inserting the condition check and a loop to each Map-Reduce pair in a recursive manner, shown in Figure 3. In the hybrid mode, functions in the Map-Reduce pair may vary in different iterations as in the iterative mode. This enables flexible function repetition at different levels so that the workflow can be expanded horizontally. Meanwhile, each function can be replaced by a Map-Reduce pair so that the workflow can also be expanded vertically.

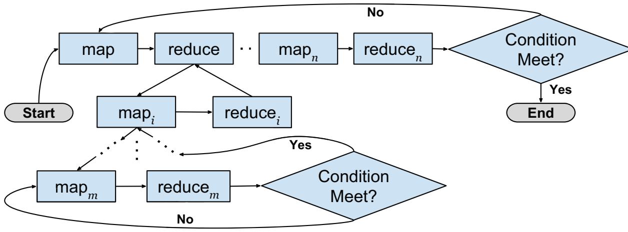


Fig. 3. Hybrid MapReduce Mode

IV. Design of MR-Graph System

Our proposed *MR-Graph* system aims to enhance the flexibility of MapReduce to process complex applications and datasets. First, programmers should be able to follow MapReduce-type programming paradigm and develop more complex applications easily. Multiple Map-Reduce pairs and flexible Input/Output buffers can be specified. Second, memory hierarchy will be exploited to handle big data applications. Intermediate results may not need to be sent back to secondary storages as in traditional MapReduce. Third, GPU is leveraged as a new computing platform. The extraordinary computing power and shared memory architecture of GPUs can improve the overall performance.

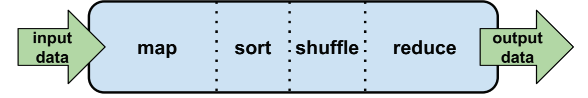
A. Programming with MR-Graph

Programming with *MR-Graph* is similar to that with the traditional MapReduce. Other than providing different user-defined Map and Reduce functions, programmers need to specify the relationship between Map-Reduce pairs as well as the buffers for input and output data.

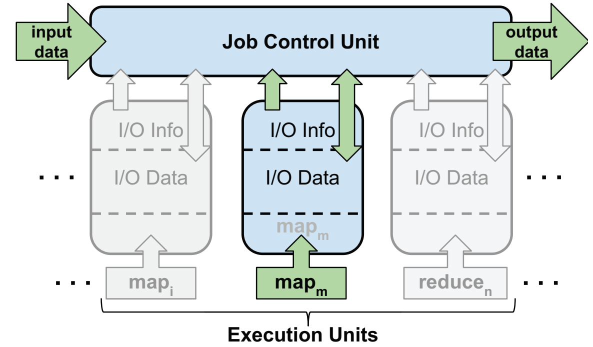
MR-Graph supports a function call graph developed based on the aforementioned iterative, recursive and hybrid MapReduce modes. In a function call graph, nodes are Map and Reduce functions with user-provided information to perform the application logic. Two nodes are connected if one node's output is used as the input to the other node.

As in the iterative mode, if some Map-Reduce pairs need to be repeated for certain number of times, they are placed in a loop with one special *Condition Node* at the end. A Condition Node contains the condition specified by a users, and according to the condition check result it determines to go to the next iteration or terminate the workflow. Map-Reduce pairs keep updating their intermediate data until the condition is satisfied [13].

B. MR-Graph Runtime System



(a) Traditional MapReduce Model



(b) MR-Graph with Execution Unit

Fig. 4. MapReduce execution models

The traditional MapReduce runtime system adopts a Map-Shuffle-Reduce pipeline as shown in Figure 4(a). Users define the Map and Reduce functions, whereas Shuffle is provided by the MapReduce runtime system and transparent to users. The main purpose of Shuffle is to handle communication between computers without the involvement from programmers. The traditional MapReduce emphasizes the right execution order. If multiple Map-Reduce pairs are required, shell scripts are used for job connection and saving the intermediate data on secondary storages. The movement of a large amount of intermediate data can compromise the system performance.

In *MR-Graph*, the order of function execution is enforced by the function call graph. *MR-Graph*'s runtime system consists of two layers as shown in Figure 4(b) to emphasize data and computation. The upper layer, *Job Control Unit (JCU)*, loads the function call graph, maintains the execution order, and loads and redistributes data. The lower layer, *Execution Unit (EU)*, runs user-defined Map and Reduce functions as well as system-default sort algorithms on GPU platforms. This two-layer strucure enables efficient computation concatenation and effective data sharing.

C. Job Control Unit

The Job Control Unit (JCU) loads and parses user files to build up the function call graph using linked list as its data structure. Then, depth-first-traversal is adopted to visit all nodes in a sequential order. Nodes for Map and Reduce functions specify where to load the input data, where to find the user-defined function code, and where to send the output data. All of these will be provided to the Execution Unit (EU)

for calculation. After each Map function, one system-default Sort function will be activated automatically. This sorting algorithm is executed by the Execution Unit. The traditional MapReduce combines the Sort function as a part of Shuffle to reduce the communication volume. In the *MR-Graph* system, sorting is activated implicitly by JCU on EU. Condition Nodes allow JCU to verify if the contained data have reached some specified values, which is used to determine the continuity of a loop and termination of the entire job.

In addition to job control, JCU is responsible for data management. Data locality is important for MapReduce's performance [14]. In the existing implementations, such as Hadoop and Sphere inherited from the Google MapReduce model, most of the I/O operations target file systems and disk storages. They mainly focus on the performance of single-round data processing, instead of complex application logic and datasets.

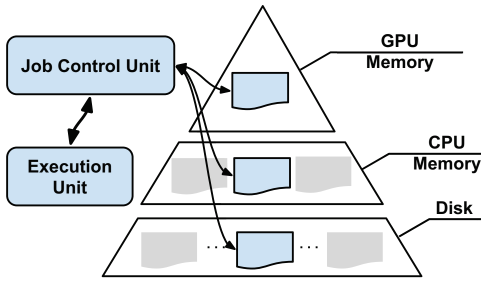


Fig. 5. MR-Graph Data Locality

In contrast, the proposed *MR-Graph* system explores the flexible memory hierarchy in GPUs to reduce the access latency and increase the data bandwidth. More specifically, the intermediate data will be stored in GPU's global memory to minimize the communication cost. However, GPU has a limited amount of memory, which makes it impractical to hold all intermediate data. Thus, the main memory and disk storages are used as backups. As shown in Figure 5, for a task node, users can configure its output to the main memory or even to a disk file if the data is too large. In addition, JCU can save the data in the GPU's memory transparently [15]. In a GPU cluster, each compute nodes is usually equipped with several gigabytes of GPU memory on each GPU device, tens of gigabytes of main memory, and several terabytes of disk storage. JCU intends to keep data closer to GPU and send them to slower and larger storage only when the data size is beyond the capacity of the GPU memory. Overall, MR-Graphs three-layer memory hierarchy is capable of handling most MapReduce jobs with multiple terabytes of data (including the intermediate data).

The MR-Graph's data I/O behavior is shown in Figure 6. For each Execution Unit, JCU loads the input data to keys and values containers that reside in GPU's global memory or the main memory based on the users' settings. Each Execution Unit gets the input data in one of the two ways: 1) reads new data from the input files, or 2)loads intermediate data from previous tasks. In the first case, JCU launches an input file stream and reads data from the specified file(s). To read a file, the Execution Unit activates a main memory container for the input data, and then copies data to the GPU container. To better utilize the system resource, the container in the main memory will be released after the current task is finished. Thrust performs garbage collection automatically when the vector containers are no longer used.

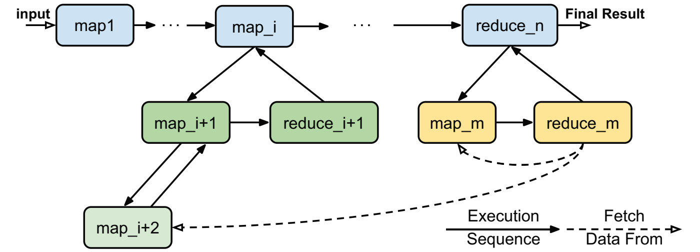


Fig. 7. Data Retrieval from Multiple Sources

In the second case, a function's input comes from a previous task. In some special cases, tasks may require the intermediate data from the task that is executed several steps earlier. For example, in Figure 7, $reduce_m$ fetches data from both map_m and map_{i+2}. JCU locates the input nodes by following the backward pointers. These nodes contain pointers to the output buffers which could be in the main memory, GPU memory, or even disks. If necessary, JCU loads data into GPU memory so that the next nodes can start execution immediately.

D. Execution Unit

The *MR-Graph* system exploits GPUs for massively parallel data processing. The Execution Unit sets up the GPU kernel and launches the executions. Kernels can run user-defined Map and Reduce functions as well as MR-Graph system's sorting functions. They are executed on thousands of GPU cores with data provided by JCU. EU is a unified unit as all Map, Reduce and Sort functions use it. Job scheduling is based on the data size.

Since traditional MapReduce and MR-Graph use computer cluster and GPU as the computing platforms respectively, their communication mechanisms are different. Communication in GPU with shared memory is simpler than that with the distributed memory structure in computer clusters. Therefore, the communication mechanism of MR-Graph focuses on exploring the memory hierarchy for high-throughput data processing rather than on reducing the communication overhead. The execution mechanism on GPU is more computationally powerful and energy efficient than on computer clusters.

The Execution Unit in *MR-Graph* has the following unique features.

Unified Interface: In MR-Graph, data is transferred/saved in a key-value pair format. A list of values acts as the index for each key. EU takes in and processes key-value pairs, then generates different key-value pairs as output. Since GPU is not good at processing complex data structure, in GPU MapReduce, key-value pairs are usually stored separately in

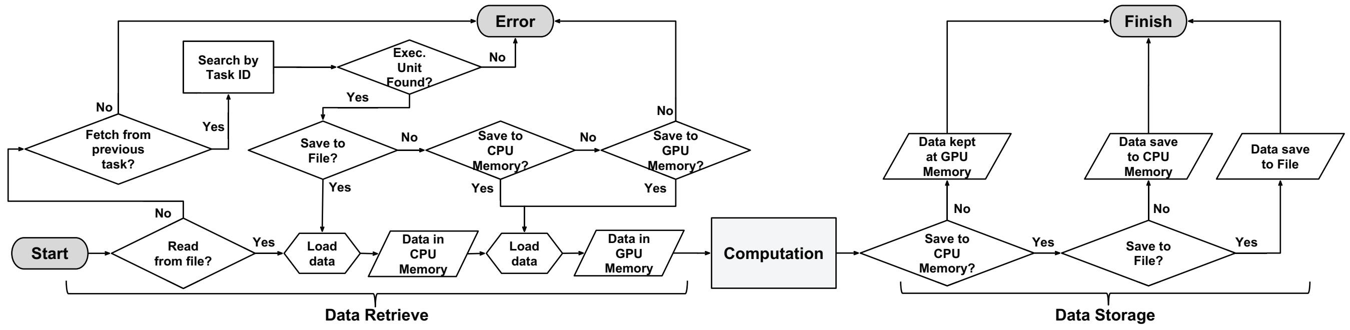


Fig. 6. Input/Output Data Management in JCU

two containers. In MR-Graph, four buffers are allocated in GPU global memory and reserved for input keys and values as well as the output keys and values. Each function node in MR-Graph has access to these four buffers. To unify the interface prepared by JCU, Map, Reduce and Sort functions use the same parameter list: the input/output keys and values. The Execution Unit will only provide the basic data processing functionality, such as reading keys and values from the input buffers, executing user-defined functions or default sort functions, and saving results output buffers.

Output Data Size Predication: Before activating Execution Unit, Job Control Unit will loads data into input key and value buffers. But the size of output data still needs to be calculated at runtime. Since GPU does not support dynamic memory allocation in device memory during GPU computing, most GPU MapReduce implementations usually require users to pre-calculate the size of output data size. So memory allocation takes place before GPU kernel launches. With CUDA Thrust library, MR-Graph can solve this problem smoothly. During kernal launch, JCU reserves a piece of global memory for each declared buffer. During the runtime, EU will reallocate this reserved address space to the user defined size by invoking Thrust *resize* function. JCU and EU work together to handle data. The whole precoess is completely transparent to users.

Sort Stage: In traditional MapReduce, sorting intermediate data is usually a part of shuffle stage. After data sorted in order, shuffle stage will deliver some values by keys to the cooresponding reduce workers. In MR-Graph, since communication is handled by Job Control Unit, shuffle stage is omitted. Same as other parts of MR-Graph, sort stage is also implemented by Thrust. The inputs of sort function are the key and value buffers located at GPU global memory. The technique *sort by key*, is used to sort two device vector arrays, key and value, by their corresponding keys. Since each value can be viewed as key's index, this technique is actually binding the key and its index together for a temporary key-value pair. Therefore sort by key is in fact sorting key-value pairs by their keys. Once it is done, both keys and values will be sorted accordingly.

Processing characters in GPU: CUDA only support standard C/C++ data type such as *int*, *double*, and *char*, but not *character string*. Therefore, the size and the index of the character string need to be managed. In MR-Graph, char

CharArr	E	x	a	m	p	l	e	F	o	r	M	R	-	G	r	a	p	h
CharIdx	0							7			10							17
CharLen	7							3			7							1

Fig. 8. Character String Processing in GPU

arrays are represented in three different device vector arrays, as shown in Figure 8 : In order to sort a character string, MR-Graph GPU runtime will do the following:

1) Concatenate all the strings together into a single char vector, CharArr.
2) In int vector CharIdx, mark the starting index for each string.
3) Save the length of each string in an integer type vector, CharLen.

After the above steps, char string is ready for GPU runtime to process. But in order to construct the comparison operator, runtime will further combine the start index and the length into a Thrust tuple. Since reordering integers in GPU is much faster than moving blocks of char string around, to reduce the communication overhead, MR-Graph does not sort the actual char string arrays, but the index and length tuples instead. This is a straightforward algorithm for this framework. The data rearrangement can be accomplished later using the reordered the index vector.

V. Performance Evaluation

Testing Environment

All experiments are conducted on a workstation containing four Intel Xeon E5-2620 CPU (24-Core in total operate at 2.00Ghz) with 32GB RAM and two Nvidia Tesla K-20Xm GPUs (0.73 GHz, 5,760MB global memory). It is running the GNU/Linux operation system with kernel version 2.6.32. Testing applications are implemented with C++11 and CUDA 6.5 and compiled with g++ and NVCC compiler in CUDA Toolkit 6.5.

K-20Xm is based on Kepler GK110 architecture. A simplified architecture is shown in Figure 9. Each GK110 consists of 15 streaming-multiprocessors unit (SMX) and can execute thousands of light-weighted threads concurrently. By using

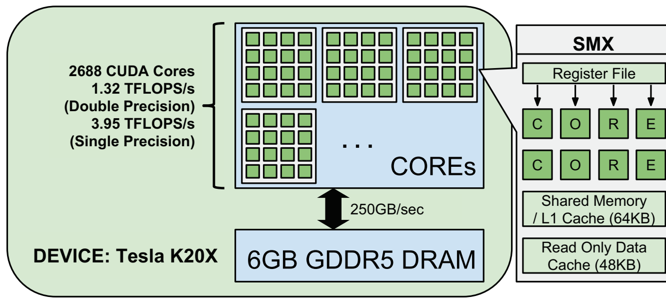


Fig. 9. Simplified View of K-20 GPU

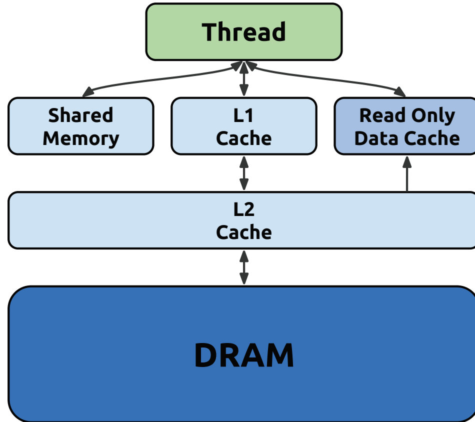


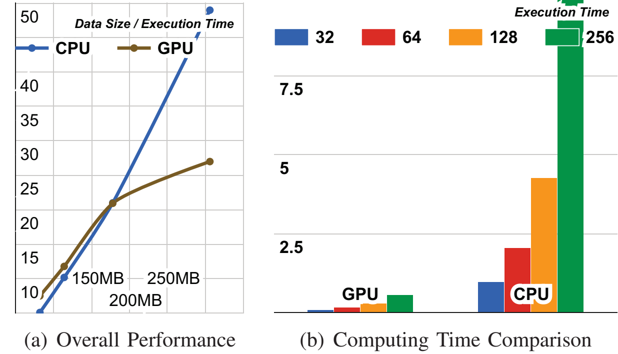
Fig. 10. Kepler Memory Hierarchy

the CUDA programming model, all these threads are mapped onto GPU cores. This allows up to 512 threads to be grouped into thread blocks that are then assigned to SMX, scheduled to work concurrently in groups of 32 threads, called warps. Figure 10 shows Kepler architecture's memory hierarchy. Each thread is assigned some registers whereas each warp has one program counter. Hence, all threads within the same blocks can access the common shared memory, which eases the synchronization of threads in same blocks and also facilitate extensive reuse of on-chip data, greatly reducing off-chip traffic. Extremely fast context switch among threads in a warp can help tolerate memory access latency.

In a computer system with multiple GPUs, Nvidia GPUDirect is the technique used to handle inter-GPU communication within a single system, so network adapters and storage devices can directly read and write data in the GPU device memory, eliminating unnecessary copies in system memory (on the CPU side) to achieve significant performance improvement in data transfer. High-speed DMA engines enable this inter-GPU communication within similar systems.

Experimental Results

The first benchmark program is a Word Count application with data size varying from 32MB to 256MB. Both CPU MapReduce and GPU MapReduce (MR-Graph) are implemented and compared. As shown in Figure 11(a), due to the GPU communication overhead, CPU MapReduce outperforms GPU one (MR-Graph) when test case is relatively small. However, as dataset increases in MR-Graph, the computing power gained from GPU starts to surpass the communication overhead. Figure 11(b) shows the pure computation time comparison between them. As dataset gets larger, MR-Graph finishes job exponentially faster then CPU MapReduce.



(a) Overall Performance (b) Computing Time Comparison

Fig. 11. Performance Profiling

The second experiment is about an application which intends to solve a real world problem. IT department in an organization such as Enterprise or University often needs to analyze their web logs for information related to their website. It is convenient to use MapReduce as the programming model to process pile of web logs. A list of the most popular websites or peak visiting time often means a lot for future developement. The first action is to find a list of the most frequently accessed URLs. In addition, popular webpages usually need to be categorized by their departments, and finally the peak visiting time for each webpage should be reported. Traditional MapReduce has to split this job into three tasks: first, URLs from the web logs are sorted out by number of visit; then based on the results, top k web sites for each departments are categorized(k is the threshold value); finally the peak visiting time for every webpage in the list will be reported. However, processing data in three separate MapReduce tasks not only requires to write an inefficient scripting language program on shell, but also spends more time on data transfer between file systems and memory. Intermediate data are definitely not reused.

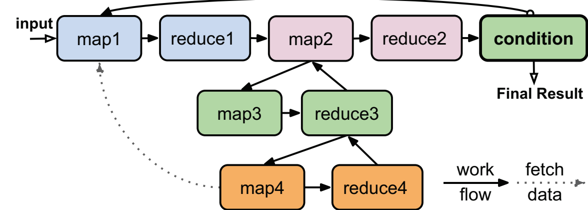
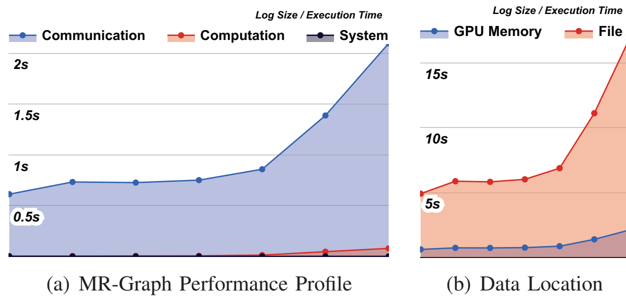


Fig. 12. MR-Graph for Intricate Web-log Analysis

MR-Graph handles this whole job in a natural and efficient way. As shown in Figure 12, first MapReduce stage will generate the top k frequently visited pages, then second stage will categorize the list; the third MapReduce stage will calculate

the peak visiting time for each webpage in the categorized list; and the final stage will gather the results and format them for the final result. Since all the intermediate data can be kept in GPU memory, no data migration is needed during the execution.



(a) MR-Graph Performance Profile (b) Data Location

Fig. 13. Performance Measurement on Web Log Analysis

Our test case size varies from 100 to 1 million of records. During the experiment, MR-Graph shows that its well maintained construction has negligeable impact on the overall performance. Figure 13(a) shows MR-Graph's performance profiling. GPU computation only takes one-tenth of data communication time for web log analysis. After measuring the breakdown of the overhead caused by all major components in the system, MR-Graph shows excellent stablility during the experiment. The overhead caused by MR-Graph takes at most 1/10th of the computation time or 1/100th of the overall execution time, and remains stable as datasize increase.

Compared with the traditional MapReduce approach, MR-Graph effectively utilizes the memory hierarchy to keep all reusable data closer GPU global memory. By avoiding redundent communication, MR-Graph greatly improves the overall performance. For a comparison, Figure 13(b) shows that when each MapReduce stage always sends data back to file systems, the execution time will be at least 4 times longer for the same job.

VI. Conclusion and Future Work

This paper introduces a customizable and unified GPU MapReduce framework, MR-Graph which enables users to deploy their applications based on three major MapReduce modes. Map-Reduce pair can be duplicated horizontally and extended vertically so that the restriction in the original MapReduce is removed. With a three-tier memory hierarchy, intermediate data are placed close to GPU and do not need to be transferred back and forth for better data reusability. MR-Graph is also ready for Big Data processing since oversized data can be moved out of GPU memory to CPU memory or even hard disks. MR-Graph's runtime system uses Job Control Unit to handle task execution order and data input/output. Unified Execution Unit helps accelerate user-defined Map and Reduce functions as well as system-default Sort functions. Experimental results have demonstrated that MR-Graph framework works for real cases and has effectively controlled its construction overhead while maintaining the necessary configurability.

MR-Graph intends to explore the customizability of the MapReduce model and open as many configurable options to users as possible. However the tradeoff is the development agility. The future work includes reducing users' configuration efforts, improving MR-Graph's performance through multi-GPU [1] [16] [17], benchmarking computation-intensive applications [18] and effectively using shared memory [15]. MR-Graph can be a promising candidate for solving complex applications through MapReduce paradigm in GPU Clouds.

References

[1] Y. Chen, Z. Qiao, S. Davis, H. Jiang, and K.-C. Li, "Pipelined multi-gpu mapreduce for big-data processing," in *Computer and Information Science*. Springer, 2013, pp. 231–246.

[2] J. Dean and S. Ghemawa, "Mapreduce: Simplied data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107 – 113, 2008.

[3] *Apache Spark*. [Online]. Available: http://spark.apache.org/

[4] *Apache Flink*. [Online]. Available: https://flink.apache.org/

[5] "Tutorial: Spark-gpu cluster dev in a notebook," 2014. [Online]. Available: http://iamtrask.github.io/2014/11/22/spark-gpu/

[6] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a mapreduce framework on graphics processors," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008, pp. 260–269.

[7] J. A. Stuart and J. D. Owens, "Multi-gpu mapreduce on gpu clusters," in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, 2011, pp. 1068–1079.

[8] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative mapreduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, 2010, pp. 810–818.

[9] C. Goncalves, L. Assuncao, and J. C. Cunha, "Data analytics in the cloud with flexible mapreduce workflows." in *CloudCom*. Citeseer, 2012, pp. 427–434.

[10] P. K. Mantha, A. Luckow, and S. Jha, "Pilot-mapreduce: an extensible and flexible mapreduce implementation for distributed data," in *Proceedings of third international workshop on MapReduce and its Applications Date*, 2012, pp. 17–24.

[11] *NVIDIA CUDA Programming Guide 6.5*. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/

[12] *Thrust*. [Online]. Available: https://developer.nvidia.com/Thrust

[13] R. Tudoran, A. Costan, and G. Antoniu, "Mapiterativereduce: a framework for reduction-intensive data processing on azure clouds," in *Proceedings of third international workshop on MapReduce and its Applications Date*. ACM, 2012, pp. 9–16.

[14] J. Xie, S. Yin, X. Ruan, Z. Ding, Y. Tian, J. Majors, A. Manzanares, and X. Qin, "Improving mapreduce performance through data placement in heterogeneous hadoop clusters," in *IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. IEEE, 2010, pp. 1–9.

[15] L. Chen and G. Agrawal, "Optimizing mapreduce for gpus with effective shared memory usage," in *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, 2012, pp. 199–210.

[16] Y. Chen, Z. Qiao, H. Jiang, K.-C. Li, and W. W. Ro, "Mgmr: Multi-gpu based mapreduce," in *Proceedings of the 8th International Conference on Grid and Pervasive Computing*, 2013.

[17] H. Jiang, Y. Chen, Z. Qiao, K.-C. Li, W. Ro, and J.-L. Gaudiot, "Accelerating mapreduce framework on multi-gpu systems," *Cluster Computing*, vol. 17, no. 2, pp. 293–301, 2014.

[18] W. Jiang and G. Agrawal, "Mate-cg: A map reduce-like framework for accelerating data-intensive computations on heterogeneous clusters," in *IEEE 26th International Conference on Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 2012, pp. 644–655.