An Improved GPU MapReduce Framework for Data Intensive Applications

Razvan Nitu, Elena Apostol, Valentin Cristea
Faculty of Automatic Control and Computers,
University Politehnica of Bucharest
razvan.nitu@cti.pub.ro,
{*elena.apostol, valentin.cristea*}*@cs.pub.ro*

***Abstract*—The MapReduce paradigm is one of the best solutions for implementing distributed applications which perform intensive data processing. In terms of performance regarding this type of applications, MapReduce can be improved by adding GPU capabilities. In this context, the GPU clusters for large scale computing can bring a considerable increase in the efficiency and speedup of data intensive applications.**

In this article we present a framework for executing MapReduce using GPU programming. We describe several improvements to the concept of GPU MapReduce and we compare our solution with others.

***Keywords*-GPU MapReduce; shared memory; Hadoop; OpenCL**

I. Introduction

In the last years the performance of CPU cores remained almost the same, while in the case of graphics processors, the performance increased dramatically [1]. For many applications, migrating from a CPU to a GPU environment offers a gain in cost-effectiveness and performance.

GPU clusters for large scale computing brings a considerable increase in the efficiency and speedup of resource intensive applications. Thus emerged the concept of adding GPU processing to the MapReduce paradigm. However, most projects that have explored this idea are focusing on single GPU development, neglecting the compute power of GPU clusters. There are some limitations that are to be taken into consideration when implementing applications using GPU MapReduce on clusters or Clouds. First of all, the MapReduce model does not handle explicitly GPU based solutions. Secondly, the communication between GPUs is difficult. And finally, a GPU does not have virtual memory.

Hadoop's MapReduce framework [9] is one of the best solutions for implementing distributed applications which perform intensive data processing. In terms of performance regarding this type of applications, Hadoop's MapReduce can be improved by adding GPU capabilities. To our knowledge, there is no official Hadoop MapReduce implementation that rely on GPU processing.

In this paper we present our solution of a GPU MapReduce Framework (GMRF). GMRF provides a complete solution for executing the MapReduce tasks in the GPU environment. If the tasks are highly parallelizable, this framework adds a speedup up to ten times than CPU processing.

The main objective of this research is creating a framework for applications, that will allow fast processing of large amounts of data, by efficiently using the high parallelism of GPUs in a distributed environment. For that purpose, we designed and implemented a GPU MapReduce framework and we added a component that enables the integration of the applications developed using this framework in Hadoop, in order to take advantage of its processing and distributed model. We also present a model for efficiently mapping in the GPUs' memory the intermediate results obtained in the "map" phase.

Our GPU MapReduce framework can be used both independently or in a distributed system by integrating it in Hadoop. It is generic and it allows the users to implement applications that take advantage of the GPU processing power. The user only implements the specific functions of the MapReduce paradigm, the framework being responsible for all operations specific for GPU programming. The applications supported by our framework can be of any nature, and the overhead introduced by this generalization must not affect the overall desired performance.

Our framework can use and offers support for the major data formats (matrix, vector, text, hash). New data formats can be easily added. Also, the general structure of the system is modular, in order to allow flexible addition of new features.

Another characteristic of our framework is that, although generic, it has several optimizations that results in better execution time than other similar projects.

The rest of this paper is organized as follows. Section II presents related research approaches for GPU MapReduce framework. In section III we describe in detail the proposed architecture. This section also depicts the way the modules introduced by our solution integrate with the existing ones in Hadoop. Section IV contains the implementation details and some design features for the proposed solution. Section V shows various experiments and their results. It also provides a comparison between our solution and related works, distinguishing the situations in which our solution proves to be more efficient that the others. Finally, section VI concludes this paper, and offers directions for future work.

978-1-4799-6569-4/14/$31.00 ©2014 IEEE

II. Related work

There are few implementations of GPU MapReduce, and to our knowledge only two of them use the compute power of a GPU cluster.

MARS (MapReduce Framework on Graphics Processors) [2] is the first large scale GPU system, but still has limited scalability as it uses only one GPU. MARS uses a large number of GPU threads for Map and Reduce tasks, assigning each thread a small number of pairs $(key, value)$ in order to efficiently make use of the strong parallelism of GPU environments. To avoid potential conflicts between concurrent writings during the stage of saving intermediate or final results, MARS uses a lock-free scheme that ensures the fairness of the parallel execution with a small synchronization overhead. Paper [2] presents a series of tests conducted on MARS: matrix multiplication (MM), grep (SM), page view rank (PVR), page view count (PVC), inverted index (II) and similarity score (SS). These tests show that MARS is $1.5x$ - $1.6x$ faster than a CPU MapReduce implementation (Phoenix [6] for example). The best results are obtained for computational intensive applications, applied to large data sets (e.g. SS or MM).

MapCG [3] is a MapReduce solution that provides source code level portability between CPU and GPU. For the execution of MapReduce tasks, it allows the use of several CPUs and GPUs simultaneously. However, this doesn't improve the performance, but it actually reduces it. MapCG uses a lightweight memory allocator. For each execution of a MapReduce job, new blocks must be allocated to store data generated in "map" and "reduce" phases. These blocks are in large numbers, but are very small and they are used only during the execution of the current MapReduce job. The lightweight memory allocator used in MapCG was implemented based on these observations, and can allocate CPU or GPU memory. An improvement of MapCG, compared with MARS, is replacing the sorting of intermediate results with the usage of a Hashtable. The authors used in testing the same applications as MARS, plus they added two new ones: K-means and N-body. The results show that MapCG can efficiently execute code on both CPU and GPU with a speedup of $1.6x$ - $2.5x$, compared with MARS or Phoenix.

Article [4] presents a solution for a GPU MapReduce using Hadoop and OpenCL (MRCL). This implementation doesn't have any other mechanisms for improving performance. For each Mapper node there are two CPU threads: the first one initializes the context for the OpenCL platform, and the second one transforms the input data in a format that can be processed by the GPU. When these two threads finish the execution, the kernel function starts. The map results are written in parallel in a GPU cache buffer and then they are transferred to the local disk. The same steps are executed in the Reduce stage. According to the authors, the performance achieved by a Hadoop cluster that uses MRCL is $1.6x$ - $2.0x$ higher than in a classic Hadoop cluster.

In article [5] GPMR (Multi-GPU MapReduce) is described as a stand-alone MapReduce library. This library was designed to be easy to use and provide full access to the GPU. Each stage of the MapReduce pipeline is programmable (on Mars some stages were built-in), but it also provides standard implementations for Partitioner, Scheduler and Sorter. Each GPU is controlled by a separate process and each process is running a MapReduce pipeline. GPMR allows customization of the MapReduce pipeline using communication-reducing stages. There are three basic steps: Map, Sort and Reduce. Map can be divided into several sub-steps, whilst Sort and Reduce are indivisible. GPMR does not handle fault tolerance, and it doesn't work with a distributed file system (such as HDFS). The authors conclude that their solution is scalable.

GPMR has several improvements over Mars and MapCG. First of all, it has better scalability (GPMR can use multiple GPUs, while Mars uses a single GPU and MapCG can use more but with performance degradation). Secondly, in GPMR the scheduling is performed by the user, as is in MARS, the library and not the user is scheduling threads and blocks.

The tests conducted on GPMR consist of the same applications as in MARS and MapCG: MM, K-means, WO (word occurrence), and so on. For the comparison with MARS, the tests on GPMR were made using a single GPU and four GPUs in the same node. The second set of tests were use to measure the scalability and were made using up to 64 GPUs. The experiments show that GPMR, compared with MARS, has a speedup between $2.6x$ and $37.3x$ when using a node with a single GPU, and between $10.7x$ and $129.4x$ when using a node with 4 GPUs.

III. GMRF Architecture

Figure 1 depicts the GMRF's architecture. GMRF is deployed in a distributed environment that consists in one Master node and many Slave nodes. On each Slave node that has enabled GPU power there is a GMRF module.

GMRF consists of the following components: *GPU MapReduce* module, *GPU Controller*, *Combiner* module, and *GPU Info* module.

GPU MapReduce module is an implementation of the MapReduce paradigm on GPU. It can function independently or as a module in the Hadoop MapReduce framework. It has a general structure and allows the users to implement applications of any kind that can benefit from the parallel processing power of the GPU.

GPU Controller is the component that connects the GPU MapReduce module to the component that is in charge of controlling tasks on current worker node (in Hadoop: Task Tracker). This component interprets the configuration policies and generates a series of commands required for the task execution performed by the GPU MapReduce module.

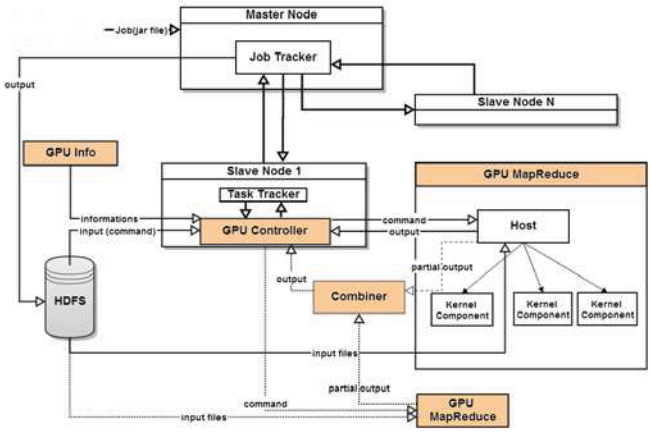

Figure 1. **GMRF Architecture**

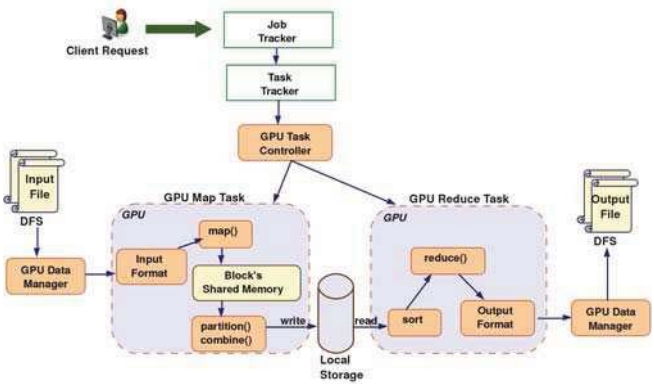

Figure 2. **GPU MapReduce framework**

Also, it takes the results at the end of the execution processes and sends them to the TaskTracker. Besides these basic functionalities, it also has a number of advanced features related to data compression and devices' memory management.

GMRF uses an instance of the GPU Controller for each Worker node scheduled by Hadoop. Besides this component that is used within each node, there is also a global component (Reducer) that is executed upon the completion of all Map tasks. The Reducer receives for processing the resulting intermediate data.

Some information obtained by the GPU Controller during execution, may be sent to TaskTracker, and then to the JobTracker. The JobTracker can pass it to the scheduling module in order to improve the scheduling algorithm.

The **Combiner** component is used when the GPU Controller decides to execute a task in several steps. Its role is to group the data in a single output.

The **GPU Info** module is used by the GPU Controller to obtain information about the available devices on the platform.

IV. Implementation Details

Figure 2 depicts the steps required for performing a GPU MapReduce task, starting from the client request up to the final step of writing the results in the distributed file system. We integrated our module in the Hadoop framework. Currently, in Hadoop there are only tasks that are executed using CPU power.

The GPU tasks are implemented using the OpenCL library [10]. Since Hadoop is written in Java, we used the JOCL (OpenCL Java language binding)[11] solution to integrate these two languages.

As we described in the introduction of this section, the real bottleneck in GPU cluster computing is the communication. To resolve this we propose a model for an efficient usage of the GPU block's shared memory and the device memory. The intermediate data resulted from the map task is stored in the GPU block's shared memory. The data is prioritized using a bucket implementation in order to reduce the number of keys, thus using more efficiently the very limited shared memory. The structuring of the data is done by the combine function. This function is called after each resulting (key,value) pair.

GPU Controller

GPU Controller is implemented in Java. It makes the connection between the GPU MapReduce module and Hadoop. When the user executes a job, the master component that controls the cluster, the JobTracker, schedules several slave nodes (by default the workload distribution is done automatically in Hadoop, but their number can be controlled by setting the corresponding parameters). The TaskTracker from each slave node starts the GPU Controller, and handles the management of input and output data.

The system contains two classes that modify the Hadoop default policy for the management of input and output files:

- *CommandInputFormat* - controls the management of the input files. We use this class to pass to the GPU Controller the file containing the control policies (decompression is done, if applicable).
- *PlainTextOutputFormat* - controls the management of the output files. It exports the output generated by the GPU Controller and possibly can do compression, if the user wants it.

Although the parallel processing power of a GPU is very high, it has important limitations regarding the memory. In that regard, when there are large amounts of data that have to be processed, the data should be split between multiple slave nodes, because the data processed by each GPU must be only a few times bigger than its global memory in order to have an efficient processing. This distribution is necessary because when the data size allocated for a GPU processing exceeds its global memory, the GPU controller splits the task in several subtasks that are run in several steps on the same

GPU, and if the number of steps is very high, the solution loses its effectiveness.

The GPU Controller component receives as input a set of policies that determine the execution parameters of the entire system. Based on these policies, and on on some internal information, the GPU Controller generates the command for the execution of the GPU MapReduce module. The format of the command depends on the type of processing desired by the user. The GPU Controller must identify this type, in order to build the correct command.

The GPU Controller has the following functionalities:

- It interprets the control policies, and it defines the commands used by the GPU MapReduce module in the tasks' execution.
- The file that contains the control policies may be compressed. In this case the GPU Controller is able to analyze control policies without performing special processing, as the file will be decompressed by the TaskTracker before sending it to the GPU Controller.
- It divides a task in several subtasks. If the input data allocated by the JobTracker to be processed on a slave node exceeds the GPU memory, it will not be processed in a single execution step, but rather the GPU Controller will divide the input, and will control the multi-step execution. The intermediate results are written in HDFS, and after all the steps are completed, the data will be processed by our Combiner module. The execution of the Combiner is controlled by the GPU Controller.
- It dynamically determines the size of the device's memory that can be allocated. The GPU Controller uses GPU Info in order to obtain information regarding the available devices.
- It deals with the decompression of the input data. If the input files are compressed, the GPU Controller decompresses them and it saves them as temporary files in HDFS. The temporary files are deleted after the GPU finishes processing them.

A user can choose whether the output will be compressed by setting a corresponding parameter when the job is submitted in Hadoop. If this parameter is set, after the data processing is finished the GPU Controller will send the data to the TaskTracker, which will apply compression before writing them in HDFS. The advantage of using compression is that it saves space in HDFS.

The user can also choose the number of reduce tasks. For applications that implements matrix operations the reduce operation is not necessarily, but for the other data types at least one reducer must be executed.

GPU MapReduce Module

This module implements the MapReduce paradigm, and allows the user to easily implement applications that benefit from the highly parallel processing capability of GPUs.

If that GPU's memory's capacity is enough to store all the input files, the entire processing operation is run in a single step, otherwise the GPU Controller will group the files, and the processing will be performed in several stages.

Currently this module allows the processing of three input data types:

- **Matrix Data Type**: For this type the input files contain representations of matrices. The framework provides all types of matrix operations using MapReduce paradigm. A user has to define three simple functions: *map*, *reduce* and *initMap*. Also, it has to specify the method or formula for calculating the size of the resulting matrix. The Host component (from GPU programming) is required to know the size of this matrix in order to allocate memory for intermediate and final data.
- **Vector Data Type**: for this data type the file format consist of a sequence of real numbers. The MapReduce module offers the possibility to specify how to divide the vector in the map and reduce steps It also offers a series of defined vector operations using the MapReduce paradigm.
- **Text Data Type**: as a third option, a user may send any number of file and it can specify two ways of processing them: using $< key, value >$ pairs or using "plain text". In both cases, the user must implement the *map* and *reduce* functions.

The users can set the number of reducers, but regarding the number of mappers the situation changes depending on the type of the input data. For the Matrix Data Type the number of mappers equals the number of lines of the resulting matrix. For the Vector Data Type the number of mappers is a parameter set by the user, same as the number of reducers. Minimum one mapper for each file must be set in the case of Text Data Type. For big files, we deduced based on several experiments that the most efficient is to have one mapper for each 8192 lines, but not more than 32 mappers per file. (These experiments were conducted on several types of graphical cards on G5k [7].)

GPU MapReduce consists of two main components: the Host component executed on the CPU, and a number of components called Kernel functions that are executed in parallel on the GPU.

As depicted in figure 3, the Host component has a serial execution, as for the Kernel functions are executed in parallel (one execution instance on each work-item).

The Host component implements a set of operations, such as: reading the input data, data grouping, allocating GPU memory, copy the data on the GPU, execution control, getting data from GPU, generating the output, and additional operations required for the supported data types. Since these operations must be performed serially, the Host is executed on the CPU.

Regarding the Kernel functions, the operations performed by them depend on the type of data used in processing.

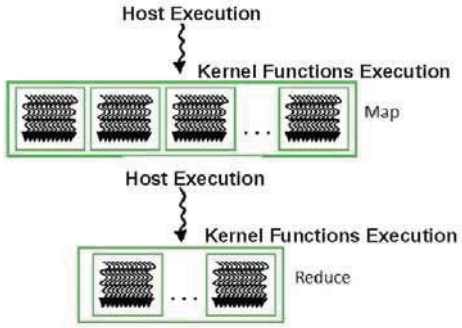

Figure 3. Execution Model for the GPU MapReduce Module

The main kernel functions are implementing the MapReduce paradigm, namely the map and reduce functions. These functions have a generic structure and allow the user to control the scope of the application by implementing specific functions. The particular functions implemented by the user are called by the generic functions.

Besides the Map and Reduce kernel functions, there are other auxiliary kernel functions: e.g. *initMap* function used in the processing of text. This function analyzes the input text, and determines the number of intermediate results. This information is used by the Host component to allocate the space for storing output from the map stage. This step introduces an 'overhead' but it is necessary because the GPU memory can not be dynamically allocate. In addition, in some cases, a hashmap must be used for the work-items to processes the data.

The *initMap* function is also used for processing matrices, in order to determine the size of the resulting matrix. In this case only one instance of *initMap* is executed. It is still executed on the GPU, as we want the loading and compiling of the function to be performed at run-time - and not to have to re-build the entire framework when this function is changed.

The GPU MapReduce Module is implemented using the OpenCL language. All the operations specific to the OpenCL programing in terms of memory management and kernel functions execution are implemented in the framework, the user is only responsible for the implementation of the functions specific to the MapReduce paradigm.

Combiner

This module is used if the GPU Controller decides that it's necessary for the input data to be processed in several steps.

As we stated earlier, if the input files assigned to a slave node exceed a certain percentage of GPU memory, and the input data is text, the GPU Controller splits the input files into several groups, and the execution will be done by the GPU MapReduce module in several steps. The output of each step will be written into a temporary file, and the Combiner will be used for the aggregation of the results.

In the case of the processing based on $< key, value >$ pairs, a hashtable is used for a rapid aggregation. Afterward the pairs are exported from the hashtable, and processing of the output continues (as depicted in Figure 2).

If the "plain text" processing is used, then all the files are read and their content is exported line by line. The output will be managed by the GPU Controller.

The operations executed by the Combiner are serial, and because of this, it is executed on the CPU, as in this case the GPU implementation is totally ineffective.

This module is implemented in the *C* language, and currently supports the combination of subtasks that carry out text-type processing.

GPU Info

This module provides information to the GPU Controller regarding the number of available devices on the desired platform and the amount of memory that can be allocated on each device.

For some tasks, the size of the input files is very large, and thus the GPU Controller must divide that task into several subtasks, which will be executed independently. For the GPU Controller to know the threshold size for the data input according to the platform and the device to be used, it uses the GPU Info module. The GPU Info returns all the available information for a corresponding platform - identified by name-, or a error code if there is no platform with that name.

From the total amount of available memory for a given device, the GPU Controller allows the input data to use maximum 70%, the rest being allocated for the data used in processing.

The role of this module is quite important because, in its absence, the user should know the amount of memory that can be allocated on each device, and should communicate these values to the GPU Controller.

This module is also used for the process of task scheduling, as the information returned by it are forwarded to the TaskTracker and JobTracker in *Keepalive* messages. This information can be used by the Task Scheduler for a more efficient scheduling.

GPU Info is implemented in the *OpenCL* language, but uses only the host component, in which a set of specific functions are calls to determine the available platforms and devices and their properties.

V. Experimental results

In this section we present the evaluation for our GPU MapReduce solution (GMRF). We test different features of the proposed solution, and for some scenarios we analyze the performance in comparison with other solutions.

MARS [2] is one of the reference GPU MapReduce implementations. As its source code is available, we use some of the MARS applications provided by the developers to make a comparison with our own solution.

As depicted in the section dedicated to presenting the implementation details, currently GMRF supports four types of data processing: matrix, vector, plain text and key/value pairs. We will test only two types of data processing: matrix and key/value pairs, as MARS provides support only for these types of data processing. The two applications that we will use for testing have a wide degree of applicability in engineering and also a high potential for parallelization: word count (WC) and matrix multiply (MM).

Word Count Tests

In Figure 4 we present a comparison between the two frameworks. For these tests we measured the processing time for a WC application using various input sizes. On average, our solution obtained a time which is approximately 1.5 - 2.2 better than the MARS's time.

The compared MARS's application is implementing a case-insensitive analysis, and also it removes words shorter than six characters, which makes the number of words found to be approximately 30% lower than the traditional Word Count implementation (without the two conditions). This implies that MARS' application has a smaller processing time than in the case of implementing the traditional WC. In the case of our framework, we implemented both types of WC - case-sensitive and case-insensitive. For both WC applications we obtained a better performance than the MARS's case-insensitive WC (in Figure 4 we compared with our case-insensitive WC, but the differences - regarding processing time - when compared with the case-sensitive WC are insignificant).

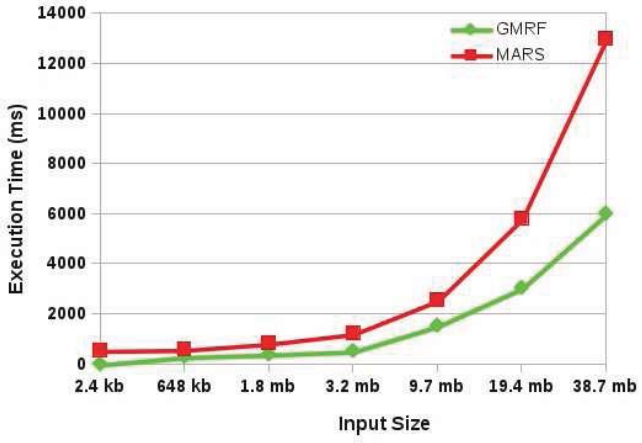

Figure 4. Word Count: MARS vs GMRF(our solution)

Comparison between execution times for each MapReduce steps: The main enhancements brought by our proposed solution in the case of $<$ key, value $>$ pairs processing are the use a hashtable for grouping intermediate data before the Reduce phase and the execution of this step on CPU, and not on GPU, due to its serial processing nature.

In Figure 5 we observe the time required for each execution phase for a WC application implemented on MARS. MARS uses for data grouping Bitonic sort, which has a complexity of $O(n * \log^2(n))$. This complexity is reflected in the increase of execution time for this stage.

Figure 6 shows the time required for each step in the case of the same WC application implemented on our proposed framework. In our case, a hashtable is used for grouping the data, and the execution is done on the CPU. The complexity of the grouping phase, using a hashtable, is $O(n)$ for best case scenario and $O(n^2)$ for worst case scenario. Because we use a large hashtable and a function that generates fewer hash collisions (*dbj2*[8]), the time increases approximately linearly with the input size, which results in obtaining time values that are much smaller than the ones of MARS, for both this stage and for the entire processing.

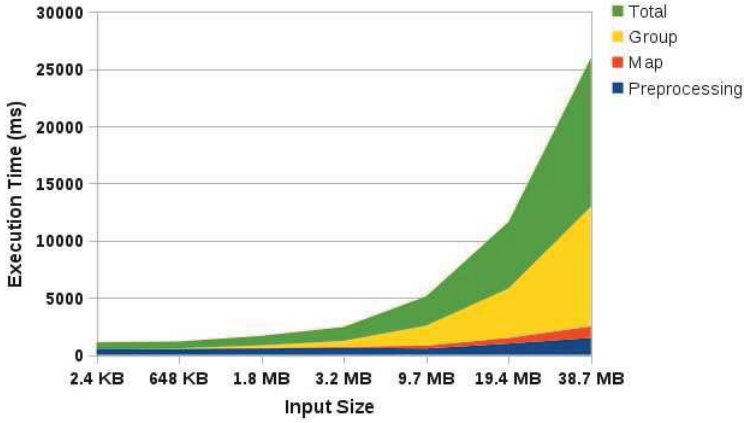

Figure 5. Execution Time for MapReduce Steps - MARS

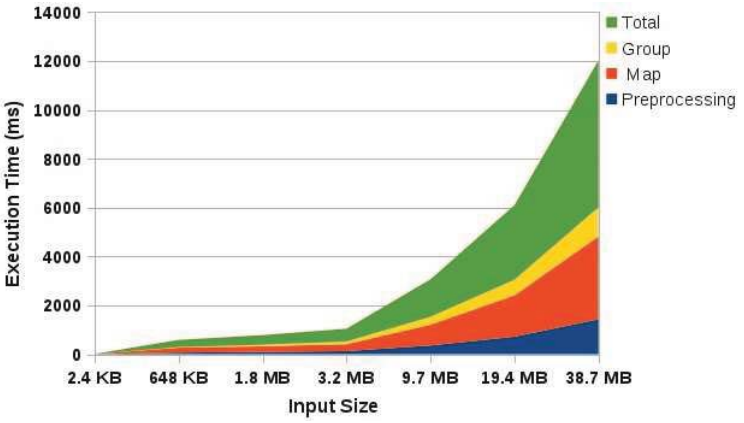

Figure 6. Execution Time for MapReduce Steps - Proposed Solution

Matrices Multiply Tests

In the case of matrices multiplication, our proposed framework manages to get times much better than Mars only for small matrices, and almost the same time values for big matrices. One possible cause is that our framework is generic, and thus, to preserve this generality, some optimizations can not be applied. Some of these optimizations are related to placing several matrix fragments in fast OpenCL memory areas (private or local memory).

To improve a multiply operation, each work-item must copy in its local memory an area of a column from the corresponding matrix. If this optimization is applied by the generic map function, the user would not be able to implement operations such as addition of matrices (because it needs lines and not columns) or filters.

Evaluating the performance in a Cluster

In this subsection we evaluate the effects of using GPU acceleration at the node level in a Hadoop cluster.

To evaluate whether replacing the traditional MapReduce processing with our proposed GPU based brings major improvement, we will compare a WC application implemented on the standard Hadoop with one that is implemented using Hadoop and GMRF component with GPU acceleration.

We will perform a set of tests to verify if the high processing power of GPU MapReduce module is sufficient to compensate the overhead introduced by the GPU Controller with operations such as analyzing the control policies and preparing the output data. First we performed these tests on a configuration with a single node (local), with the purpose of obtaining the results without being altered by the distribution of the execution.

Figure 7. Hadoop Classic vs Hadoop + GPU - Local Execution

As shown in Figure 7, by using GPU acceleration for each Hadoop node, the total execution time decreases between 5% - 25%.

Even better results are obtained when using GPU acceleration across multiple nodes, as Hadoop naturally distributes the execution, and thus the use in parallel of the GPUs can greatly improve performance.

In general, the processing performance for a node starts to degrade when the input data exceeds by a large factor the GPUs' memory size. Thus distributing the processing across multiple nodes, that benefit from GPU acceleration, is the best solution.

The next tests show the behavior of the GMRF component deployed on a real cluster. The tests are conducted on the Grid5000[7] cluster - site Adonis. For testing we use three nodes, each node having available two *Nvidia Tesla S1070* GPUs.

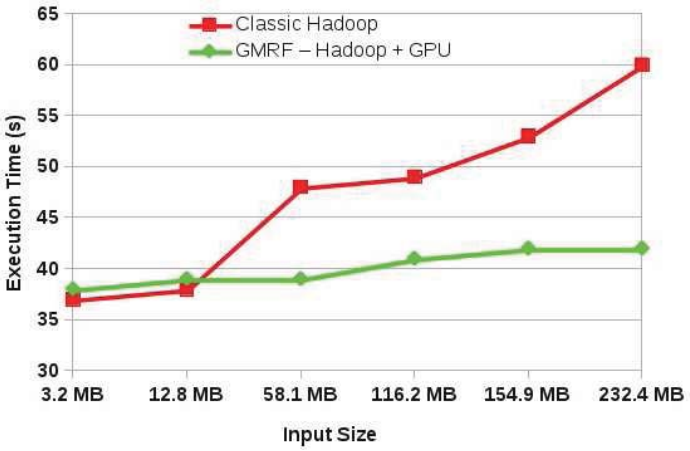

Figure 8. Hadoop Classic vs Hadoop + GPU - Distributed Execution on Grid5000

Figure 8 presents a comparison between the results obtained with *(Hadoop + GMRF)* and the ones obtained with standard Hadoop ,when executing in a distributed environment. It can be noted that for small input data the time gained by fast GPU processing on each node is not enough to compensate the overhead introduced by the operations performed by the GPU Controller. But, for large and medium input data, the proposed solution proves its usefulness, and manages to get much better times than the standard Hadoop. By analyzing the trend, we observe that the difference in performance increases in favor of our proposed solution with the increase of the size of the input data. We also observe, that in the case of *(Hadoop + GMRF)*, the total execution time is not very much affected by increasing the size of the input data, mainly because of the high parallel processing power of the GPUs.

VI. Conclusions

In this paper we present a new GPU MapReduce model, optimized to use efficiently the GPU shared memory. We have designed and implemented a hybrid CPU-GPU framework for heterogeneous environments. The framework has

the following features: is simple to use, is generic, has memory optimization techniques, and is platform independent. It is simple, as all the functionalities regarding GPU programming are already implemented. The users just have to define the functions specific to the MapReduce paradigm, without having advanced knowledge about GPU programming. It is platform independent as it is using OpenCL programming. Regarding its generality, the framework supports any application that uses any of the four main data types: matrix, vector, plain text and (key,value) pairs. New data types can be easily added to the implementation.

We understand that GPU computing is driving performance to new levels, while Cloud computing is a marvelous innovation regarding services and computing infrastructure provisioning. In the near future, an integration of these technologies will occur, and for this reason we have developed the GPU MapReduce framework described in this work.

Further optimizations can be brought to this solution. Such an example would be improving the performance of the framework for the matrix data type without losing generality. Another feature is adding support for this data type at the Combiner level.

ACKNOWLEDGMENTS

The research presented in this paper is supported by projects: "*SideSTEP - Scheduling Methods for Dynamic Distributed Systems: a self-* approach*", ID: PN-II-CT-RO-FR-2012-1-0084; *CyberWater* grant of the Romanian National Authority for Scientific Research, CNDI-UEFISCDI, project number 47/2012; *clueFarm*: Information system based on cloud services accessible through mobile devices, to increase product quality and business development farms - PN-II-PT-PCCA-2013-4-0870.

REFERENCES

[1] Shuai Che,Boyer, M., Jiayuan Meng, Tarjan, D., Sheaffer, J.W., Sang-Ha Lee, Skadron, K., *Rodinia: A benchmark suite for heterogeneous computing*, Workload Characterization, IISWC 2009, IEEE , October 2009.

[2] Bingsheng He, Luo Qiong, Govindaraju Naga ,Wang Tuyong, *Mars: A MapReduce Framework on Graphics Processors*, in PACT '08, pp. 260-269, 2008.

[3] Chuntao Hong,Dehao Chen, Weimin Zheng, Wenguang Chen, Weimin Zheng, *MapCG : Writing Parallel Program Portable between CPU and GPU*, in PACT '10, pp. 217-226, 2010.

[4] Xin Miao, Li Hao, *An Implementation of GPU Accelerated MapReduce: Using Hadoop with OpenCL for Data- and Compute-Intensive Jobs*, in IJCSS, pp. 6-11, 2012.

[5] Stuart Jeff, Owens John, *Multi-GPU MapReduce on GPU Clusters*, IEEE 2011.

[6] Yoo R.M., Romano A., Kozyrakis C. *Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system*, in Workload Characterization, 2009. IISWC 2009, pp. 198-207, Oct 2009.

[7] *Grid5000 Home Page (2014).* Retrieved June, 2014 from http://www.grid5000.fr

[8] *Hash Functions (2014)*. Retrieved June, 2014 from http://www.cse.yorku.ca/oz/hash.html

[9] *The Apache Hadoop web page*. Retrieved June, 2014 from http://hadoop.apache.org/

[10] Olav Aanes Fagerlund, *Multi-core programming with OpenCL: performance and portability*. Master of Science in Computer Science. June 2010

[11] *Java bindings for OpenCL*. Retrieved June, 2014 from http://www.jocl.org/