GPU-In-Hadoop: Enabling MapReduce Across Distributed Heterogeneous Platforms

Jie Zhu[1], Juanjuan Li[1], Erikson Hardesty[1], Hai Jiang[1], Kuan-Ching Li[2]
[1]Dep. of Computer Science, Arkansas State University, USA
{jie.zhu, juanjuan.li, erikson.hardesty}@smail.astate.edu, hjiang@astate.edu
[2]Dept. of Computer Science and Information Engr., Providence University, Taiwan
kuancli@pu.edu.tw

***Abstract*—As the size of high performance applications increases, four major challenges including heterogeneity, programmability, failure resilience, and energy efficiency have arisen in the underlying distributed systems. To tackle with all of them without sacrificing performance, traditional approaches in resource utilization, task scheduling and programming paradigm should be reconsidered. As Hadoop has handled data-intensive applications well in Clouds, GPU has demonstrated its acceleration effectiveness for computation-intensive ones. This paper intends to integrate Hadoop with CUDA to exploit both CPU and GPU resources. Hadoop will schedule MapReduce's Map and Reduce functions across multiple nodes, whereas CUDA code helps accelerate them further on local GPUs. All available heterogeneous computational power will be utilized. MapReduce in Hadoop will ease the programming task by hiding communication details. Hadoop Distributed File System will help achieve data-level fault resilience. GPU's energy efficiency characteristics help reduce the power consumption of the whole system. To achieve Hadoop and GPU integration, four approaches including Jcuda, JNI, Hadoop Streaming, and Hadoop Pipes, have been accomplished. Experimental results have demonstrated their effectiveness.**

***Index Terms*—MapReduce, CUDA, Hadoop**

I. Introduction

The term "Big Data" has been used to describe data sets which are so large that traditional means of data storage, management, search, analytics, security, and other processing have become a challenge. Big Data is characterized by the large quantity of digital information that can come from many sources and in variant data formats. The sheer quantity of data makes it difficult to process or analyze. These difficulties have led to shifts in programming paradigms in order to efficiently handle big data both in terms of performance and programmability.

In Clouds, MapReduce is a commonly used programming paradigm originally proposed by Google [15]. A map reduce framework for programming graphics processors. It reduces the programming complexity so that developers can easily exploit the parallelism in complicated applications over computer clusters or clouds composed of inexpensive commodity machines that alone would not have enough computational power to handle Big Data applications. Hadoop has been popular for its MapReduce and Hadoop Distributed File System (HDFS) for programmability and fault resilience, respectively.

As MapReduce tackles data-intensive applications effectively, GPU computing might be a good candidate for computation-intensive ones. GPU's energy efficiency characteristics also help the power consumption of the whole system. Both CPU and GPU resources will be exploited efficiently. However, how to integrate MapReduce and GPU computing smoothly remains as a technical issue. Since Hadoop's MapReduce is written in Java and most GPU programs are in CUDA, Java and CUDA integration is the target. This paper makes the following contributions:

- Four approaches of GPU and Hadoop integration, JCUDA, JNI, Hadoop Streaming, and Hadoop Pipes, have been implemented.
- Detailed analysis and comparison of these four approaches have been accomplished.
- Experimental results have demonstrated their effectiveness.

The remainder of this paper is organized as follows: Section II briefly introduces Hadoop and GPU. Section III provides the implementation details of the four approaches. Experiments and performance analysis are given in Section IV. Section V includes the related work. Finally, the conclusion and future work are given in Section VI.

II. Techniques for Distributed Heterogeneous Computing

A. Hadoop and MapReduce

Hadoop meets the challenges of Big Data by simplifying the programming for data-intensive applications. It provides a scalable and reliable mechanism for processing large amounts of data over a cluster of commodity hardware as well as providing a cost-effective way for storing the data. Hadoop also provides new and improved analysis techniques that enable sophisticated analytical processing of multi-structured data.

The Hadoop MapReduce framework is based on two primitives, Map and Reduce, from functional programming. The general form is as follows:

- Map: (k1, v1) $\rightarrow$ list (k2, v2)
- Reduce: (k2, list (v2)) $\rightarrow$ list (k3, v3)

The Map function takes an input key/value pair (k1, v1) and outputs a list of intermediate key/value pairs (k2, v2). The Reduce function takes all values associated with the same key and produces a list of key/value pairs. User-defined projects

978-1-4799-4860-4/14/$31.00 copyright 2014 IEEE
ICIS 2014, June 4-6, 2014, Taiyuan, China

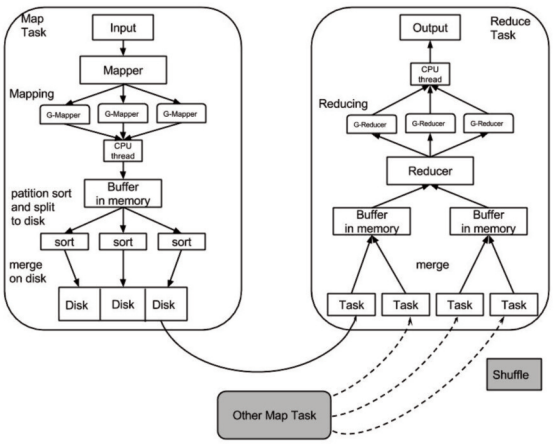

Figure 1: MapReduce in Hadoop

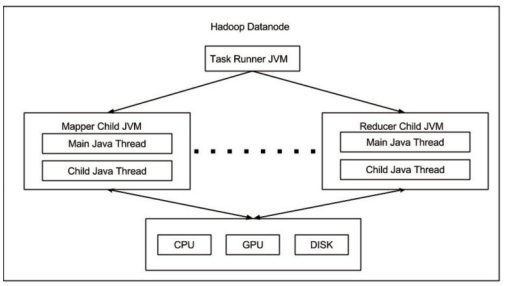

Figure 2: Hadoop MapReduce on one Data Node

implement the application logic inside the Map and the Reduce functions. The MapReduce runtime manager handles the parallel execution of these two functions.

As shown in Fig. 1, the first phase of a MapReduce program is called mapping. A list of data elements is provided, one at a time, to a function called the Mapper which transforms each element individually to an intermediary data element. Reducing allows users to aggregate values together. A reducer function receives an iterator of input values from an input list, and then it combines these values together and returns a single output value. Both the map and reduce computations are therefore implicitly data parallel across (key, value) pairs and have no data dependencies across input and output, trivially mapping to many or multi-core hardware.

However, in Hadoop MapReduce, mapper and reducer tasks run inside of potentially short-lived Java virtual machines. Task creation, management, and execution incur process and memory overheads as well as reduce the effectiveness of JIT compilation. However, using separate short-lived JVMs contributes to a significant reliability advantage by separating the Hadoop system from mapper and reducer. As a result, it is important to optimize the performance of these JVMs.

In each data node of Hadoop, the TaskRunner generates and manages many child JVMs, which execute Map and Reduce functions independently as shown in Fig. 2. TaskRunner monitors device usage and assigns each JVM to a device.

Each child JVM is assigned one chunk of keys and values for either map or reduce computation. It contains two

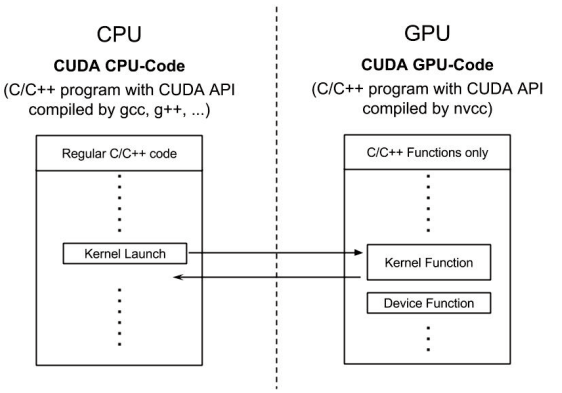

Figure 3: CUDA Program Layout

JAVA threads: main and communication threads. The main thread reads input keys and values from Hadoop Distributed File System (HDFS), and then, it executes Map or Reduce function and saves the generated results in a queue where the communication thread fetches the data and writes it back to HDFS.

B. GPU and CUDA

In recent years, GPUs have become a powerful coprocessor for general purpose computing because of the high computational power and rapidly improving programmability. General-purpose computation on GPUs (GPGPUs) has emerged in various High Performance Computing (HPC) domains, such as bioinformatics, medical imaging, embedded system design, machine learning, data mining, and more.

Unlike Hadoop MapReduce which targets at data-intensive applications for high throughput results, GPU intends to speed up computation-intensive ones for high performance achievement. In Hadoop, both computations and data are spread across multiple machines where CPU's work might be done be GPU instead for performance gains. Therefore, Map and Reduce functions should be passed down to GPU for execution. However, how to activate GPU computing is the new challenge.

Nvidia CUDA is a C/C++ extension with CUDA Runtime or Driver APIs. Therefore, corresponding runtime libraries should be linked during the compilation with NVCC compiler which splits CUDA programs into two parts: CPU-Code and GPU-Code. The CPU-Code is regular C/C++ code with CUDA runtime library whereas GPU-Code includes a kernel function and several device functions, as shown in Fig. 3. Nvidia NVCC compiler invokes regular C/C++ compilers such as GCC and g++ to translates CUDA CPU-Code into normal CPU-executable files. In the meantime, NVCC also converts GPU-Code into virtual GPU Assembly language code (PTX) or GPU-executable files (CUBIN).

The PTX file is a human-readable (but hardly understandable) file containing a specific form of assembler source code. CUBIN file is a CUDA binary file that can directly be loaded and executed by a specific GPU. However CUBIN file is specific for GPU Compute Capability which indicates GPU version, and CUBIN files that have been created for

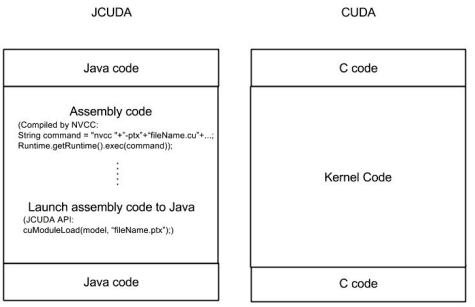

Figure 4: Layout Comparison of JCuda and CUDA

one Compute Capability cannot be loaded on a GPU with a different compute capability. Thus, in this paper the usage of PTX files is preferred, since they are compiled at runtime for GPU of the target machine.

As Nvidia GPU families are dominating the market, Hadoop and CUDA integration becomes an attractive issue. Some existing systems such as Mars [14] use Hadoop Streaming to launch GPU kernels. The interfaces between GPU and Hadoop become a bottleneck. Further investigation is required.

III. GPU and Hadoop Integration Approaches

Since Hadoop is implemented in Java, programmers only need to write MapReduce map and reduce functions in Java. Hadoop schedules these functions' executions on different machines as shown in Fig. 2. Utilizing GPU in Hadoop means that these functions will be executed by GPU, not CPU-based JVM anymore. Java-based Map and Reduce functions should play as proxies and be able to call the CUDA CPU-code which in turn calls CUDA GPU-Code. In Fig. 2, the main thread will copy the buffered data to GPU, launch a mapper or reducer kernel, and copy the results back from GPU into the buffer. The actual computations are done on GPU instead. For the main thread, to enable the execution on GPU, there exist four approaches: JCuda, JNI, Hadoop Streaming, and Hadoop Pipes.

A. JCuda Approach

JCuda enables Java programs to call CUDA kernels directly. Since the program is written in Java, it cannot rely on NVCC compiler to separate the program into CPU-Code and GPU-Code. Programmers have to write CPU-Code in Java with JCuda Driver API to replace the original CUDA Runtime or Driver API, as shown in Fig. 4. GPU-Code including kernel and device functions is still written in C/C++ as before, and should be converted into PTX or CUBIN file by NVCC compiler. There are two ways to compile GPU-Code:

1) Just-In-Time (JIT) Compilation: Java programs call a Java API, Runtime.getRuntime() to let NVCC compile GPU-Code into PTX or CUBIN.
2) Off-line Compilation: GPU-Code should be compiled in advance.

Finally, JCuda programs can load kernels from PTX and CUBIN files through JCuda Driver API, cuModuleLoad, for the execution on GPU.

```
buffersize = MAX
initialize buffer[buffersize]
while stdin(key,value) !=0 do
    if size==max then
        launch kernel to GPU device
        release buffer
    end
    insert (key,value) to buffer
end
```

Figure 5: Pseudocode of Map Function in JCuda Approach

1) Compilation and Execution: Since Map and Reduce functions in Hadoop are written in Java, JCuda is a natural consideration. Now these functions can be re-written with JCuda Driver API. However, since Hadoop disables Java's command line interface, NVCC cannot be activated as in the original JCuda. Therefore, Just-In-Time compilation fails in Hadoop. GPU-Code has to be compiled in advance.

Hadoop works in distributed computing environments. The locations of GPU-Code vary from machine to machine. It is better to embed the compiled GPU-Code in JCuda programs. Also, Its execution in Hadoop can be implemented in two ways:

1) GPU-Code can be loaded to HDFS , and then pre-launched with a specific configuration in Hadoop's Datanodes. This configuration process becomes time-consuming as more nodes are involved in Hadoop systems. Therefore, it only works with small applications and fewer nodes.
2) GPU-Code in PTX or CUBIN format can be embedded into a Java string. Since CUBIN varies based on GPU types, PTX is the better option for portability. Although programmers have to make this happen manually, it avoids the hassle of updating configuration files on all Datanodes. It helps achieve portability and scalability in Hadoop. We have adopted this approach.

With JCuda, Hadoop can offload mapper and reducer work to GPUs gradually as shown in Fig. 5.

2) Setting JCuda Library in Hadoop: There are several ways for Hadoop to JCuda runtime library on Datanodes. One approach is to compress the native library '.so' file into a JAR file, submit it to a subdirectory in "\hadoop\lib" , and then include the subdirectory in the library path environment variable. A MapReduce job will be able to find and unpack the JAR file in the distributed environments.

Another approach is to specify library JAR file location using '-libjars' option in the 'hadoop jar…' command line. The JAR will be placed in distributed cache and will be made available to all of the Hadoop tasks. The advantage of the distributed cache is that the library JAR might still be there even for the next program run (the size of distributed cache can be adjusted by a configuration variable with default value of 10GB). Hadoop keeps track of the changes to the distributed cache file by examining their modification timestamps.

The major drawback to use JCuda for Hadoop and GPU integration is the software dependency. Since all CUDA Runtime and Driver APIs should be replaced by JCuda Driver APIs, JCuda versions will depend on CUDA versions. The

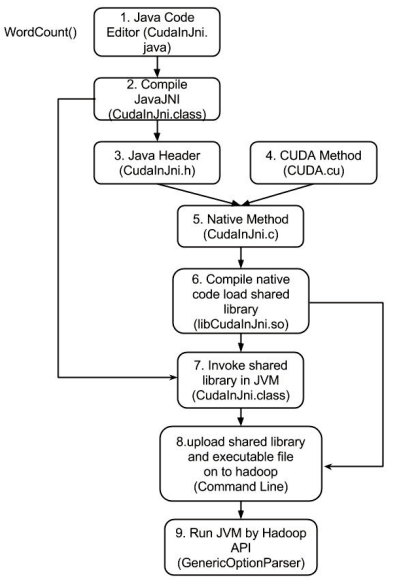

Figure 6: Flow Chart of CUDA and Hadoop integration in JNI

latest JCUDA is based on CUDA 5.5 and it can support all properties before version 5.5. However, once Nvidia releases a new CUDA version, new JCuda one will suffer few months delay.

B. JNI Approach

JNI (Java Native Interface) is a programming framework that allows Java Code running in a JVM to call native applications. This enables users to include native methods in the Java Program to handle situations in which an application cannot be written entirely in Java. Including platform-sensitive languages in the standard library allows all Java applications to access this functionality in a safe manner.

Word Count application is used to demonstrate the process of JNI as shown in Fig. 6. Java applications invoke C functions, which include CUDA kernel calls. The process consists of the following steps:

1) Create a class (JavaJni.java) that declares the native method. For example, a program includes a class named CudaInJni that contains a native method called WordCount, via keyword "native" to indicate that this portion is implemented in another language.
2) Use Javac to compile the JavaJni source file and achieve the class file, JavaJni.class.
3) Use Javah –jni to generate a C header file (JavaJni.h) containing the function prototype for the native method implementation.
4) Create a CUDA source file with CUDA APIs or Thrust library.
5) Write the C implementation of the native method, including header: jni.h and cuda.cu.
6) Compile C implementation into a native library, called libHadoopJni.so, via local C compilers and linkers. This library is system dependent. For example, in Linux the library file is "libCudaInJni.so". However, in Mac, the extension name ".so'' is converted into ".dylib'' whereas in Windows, it is ".dll".
7) Run the JavaJni program through Java runtime interpreter. Both the class file (JavaJNI.class) and the native library (libHadoopJni.so) are loaded. Method WordCount() is contained in the native library to process loading, which is a static initializer that invokes System.loadLibrary () to load native library CudaInJni during the runtime class loading.
8) Upload shared library and executable file to Hadoop.
9) Execute JNI project through GenericOptionParser.

```
Hadoop Streaming:

    $hadoop jar local/path/to/hadoop-streaming.jar \
    -mapper mapperProgram  -reducer reducerProgram \
    -file mapperProgram  -file reducerProgram \
    -input hdfs/input/streaming  -output hdfs/output/streaming
```

Figure 7: Hadoop Streaming: Command Line Integration

C. Hadoop Streaming Approach

Hadoop Streaming is a generic API that allows Map and Reduce functions to be written in virtually any language. Both of them receive input from ''stdin'' and emit output (key/value pairs) to ''stdout''.

In a Streaming implementation, input and output are always represented textually. The input (key/value) pairs are written to stdin for a mapper or reducer, with a tab character separating a key from its value. Both functions write their output to stdout in the same format: key and value separated by a tab, and pairs separated by a carriage return. The inputs to the reducer are sorted so that while each line contains only a single key/value pair, all the values for the same key are adjacent to one another. Map and Reduce functions are implemented in two independent files. Hadoop Streaming uses the command line interface to redirect the data flow as shown in Fig. 7.

The main purpose of streaming is to reduce development complexity since programmers do not have to follow Hadoop APIs. People have used different kinds of high-level languages such as C, python, or Perl to work with Hadoop. However, efficiency is the one of design goals and the runtime overhead is more than usual.

Compared with other approaches, Hadoop Streaming does not support shuffle stage to optimize the output of mapper. Hadoop will automatically wait on all map tasks until they finish and transfer the results to reducers in the following ways:

1. All key/value pairs are stored before being presented to the reducer function.

2. All key/value pairs sharing the same key are sent to the same reducer.

These two points are vital, because as a reducer accepts many key/value pairs if the reducer encounters a key that is different from the last key which were just processed. Hadoop knows that the previous key will never appear again. In some cases, if keys from input files are all the same, Hadoop will

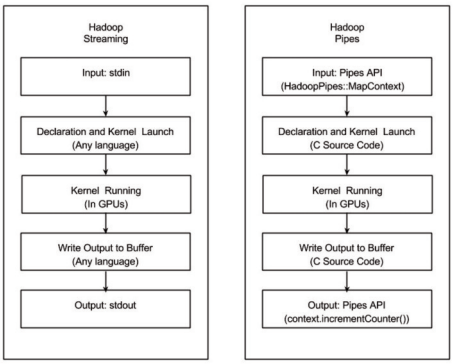

Figure 8: Comparison of Hadoop Streaming and Pipes

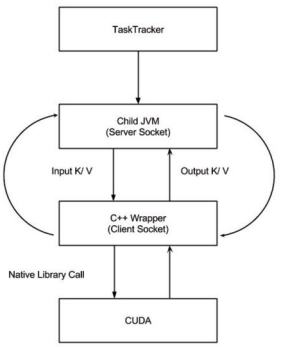

Figure 9: Flow chart of GPU-Hadoop Pipes

only use one reducer and gain no parallelization. Users should come up with a more unique key if this happens.

D. Hadoop Pipes Approach

Hadoop Pipes is a programing interface that enables users to utilize C/C++ source code for mapper and reducer implementation. The difference between Hadoop Streaming and Hadoop Pipes is shown in Fig. 8. Hadoop Pipes still have to define one instance of a mapper and reducer. Unlike the classes of the same name in Hadoop itself, both Map and Reduce functions take a single argument. The one for Map function references an object of type MapContext, whereas the one for Reduce references an object of type ReduceContext. As well as the map and reduce are implemented by C/C++ code, rather than any other code like Hadoop Streaming.

Unlike Hadoop streaming, Hadoop Pipes connect JVM and C code through a Socket rather than a JNI, as shown in Fig. 8 and 9. Keys and values in the Hadoop Pipes are in byte buffers, represented as Standard Template Library strings. This makes the interface simpler although it leaves a slightly greater burden on the application developer, who has to convert all C types to corresponding Hadoop designed types.

The command line interface of Hadoop Pipes is shown in Fig. 10. Both Map and Reduce functions are assembled into one C/C++ program.

IV. Experimental Results

Four approaches of integrating Hadoop with GPU are compared for performance and complexity analysis. The experiment is conducted on a 64-bit server with an Intel Genuine 8 processor i7 CPU (3.07GHz). The equipped GPUs are NVidia Corporation GF100 (GeForce GTX 480) and GF106GL (Quadro 2000). The server is running with the GNU/Linux operating system of kernel version 2.6.18. The Hadoop system and the testing applications are implemented with CUDA 5.0 and associated NVCC compiler.

```
Hadoop Pipes:
    1. copy the binaries file to HDFS:

        $hadoop fs -put wordcount.out hdfs/path/pipes/

    2. run the word count:

        $hadoop pipes -program local/path/to/wordcount \
        -input hdfs/input/pipes -output hdfs/output/pipes
```

Figure 10: Hadoop Pipes Command Line Interface

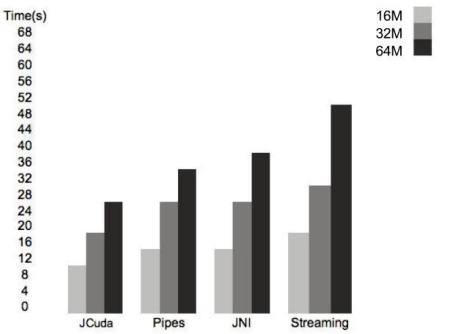

Figure 11: Performance Comparison over Word Count Application

Word Count (WC) is selected for testing and it counts all occurrences of each unique word. The input is a collection of text, with words taken from a predetermined corpus. The typical CPU implementation reads a chunk of the input data for each mapper, scans one word as provided by the specified text input format and emits [W, 1] for every found word W. With the hash implementation, the real inputs for the mapper actually become [hash (W), 1].

In the WC application, Hadoop exhibits stable performance for different data sizes as shown in Fig. 11. It is clear that JCuda approach outperforms the other three whereas Hadoop Streaming delivers the worst performance. This also indicates the tightness between Hadoop's Map/Reduce function and CUDA kernel functions. In JCuda approach, Java code calls binary GPU code directly; but Hadoop Streaming goes through much slower file system stdin/stdout. As data size increases, the increase of the execution time is lower than expected. This implies Hadoop prefers larger processing data. On the other hand, these four approaches do not exhibit dramatic performance difference. This means that Hadoop task scheduling and file operations are still the major bottlenecks.

The overall comparisons of these four integration approaches are listed in Table 1. Each column illustrates a comparison result about one aspect. "High Performance" indicates the overall performance of Hadoop on GPU clusters. "Devel-

Connecting approaches	High Performance	Development Complexity	Program Translation	Testing Difficulty
Jcuda	XXXXX	XX	X	XXX
JNI	XXX	XXX	XXX	X
Pipes	XXXX	XXXX	XXXX	XXXX
Streaming	XX	XXXXX	XXXXX	XXXXX

Table I: Overall Comparison of Four Approaches

opment Complexity" demonstrates the difficulty in deployment with a particular approach. "Program Translation" refers to the effort in translating a current CPU-Hadoop project to a GPU one. "Testing Difficulty" indicates the degree of difficulty of setting up the testing.

The number of X's in the table cells reflects the advantage points of each approach. Five X's implies the best case and one X implies the worst case. Overall, JCuda will yield the highest performance, but it is also the most expensive for development and translation. JNI is quite difficult to be tested. Hadoop Pipes shows no obvious shortcomings; it is relatively more worth implementing. However, it can only be achieved in C language which limits its scope of application. Hadoop Streaming offers the best development complexity, program translation, and testing difficulty. However, it is quite hard to improve its performance. If a project seeks easy development and testing without sacrificing performance too much, Hadoop Streaming might be a good candidate.

V. Related Work

There are several MapReduce studies that consider GPU-based computing environments. Bingsheng He, et al. [4] proposed the first framework for distributed MapReduce on CUDA named Mars [14] in 2008. Their outstanding work mainly adopted Hadoop Streaming for GPU utilization. Stuart et al. [5] have proposed a MapReduce-based volume rendering application for a multi-GPU computer cluster, whereas this study focuses on interface connection and considers running applications in Multi-GPU cluster environments.

There are also several studies related to using GPUs to accelerate MapReduce in cluster computing environments. Okur, et al [6] provide a framework called HAPI that is a simpler implementation of heterogeneous MapReduce using APAR API to transfer the computationally intensive parts of a Hadoop job to the GPU but only applies to mappers. HAPI provides a heterogeneous mapper class, which includes preprocessor, GPU execution, and postprocessor methods that must be implemented by application developers. HAPI implementation is only done on a single node so that communication overheads are ignored. There is no support for multi-devices, and most time CPU is idle since one core is used to manage a single GPU. In this work, both JCUDA and JNI can utilize all GPUs and CPUs.

Chen, et al [8][9] worked on advanced optimizations to MapReduce implementation over a heterogeneous architecture. They have modified MapReduce scheduling algorithm, improved the usage of GPU scratchpad memory, accomplished intermediate/immediate reduction, and performed runtime tuning. The evaluation results have demonstrated good load balancing effect and higher speedup over sequential execution. Although their work was not done on top of Hadoop, many of their ideas are quite useful for Hadoop.

VI. Conclusions and Future Work

The strengths of Hadoop and GPU naturally complement each other. Hadoop provides a robust and proven distributed system with a MapReduce execution model and distributed file system. However, its computational performance is lacking. CUDA enables execution of Hadoop computation in GPU native threads on heterogeneous high-performance, low-power architectures. The four different approaches provide seamless integration of these two prevalent programming models to provide a high performance distributed system with usability on par with Hadoop. These approaches are analyzed and compared thoroughly. The experimental results have indicated their effectiveness in porting common Hadoop applications to distributed heterogeneous environments.

The future work includes reducing JCuda runtime overhead in integrating with Hadoop and adopting GPU-based MapReduce schemes for high performance two-level MapReduce systems.

References

[1] NVIDIA CORPORATION, CUDA Programming Guide, http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[2] NVIDIA CORPORATION, CUDA Reference Manual, http://developer.nvidia.com/cuda

[3] The Hadoop Distributed File System: Architecture and Design http://hadoop.apache.org/docs/r0.18.0/hdfs_design.pdf

[4] B. He, W. Fang, Q. Luo, N. K.Govindaraju, and T. Wang, "Mars: A mapreduce framework on graphics processors," Parallel Architectures and Compilation Techniques, pp. 260–269, 2008.

[5] J. A. Stuart, C.-K. Chen, K.-L. Ma, and J. D. Owens, "Multi-GPU volume rendering using mapreduce," in Proceedings of 1st International Workshop on MapReduce and its Applications, June 2010.

[6] S. Okur, C. Radoi, and Y. Lin, "Hadoop+aparapi: Making heterogeneous mapreduce programming easier," https://netfiles.uiuc.edu/okur2/www/docs/hadoop+aparapi.pdf

[7] B. Catanzaro, N. Sundaram, and K. Keutzer, "A map reduce framework for programming graphics processors," in Proceedings of Workshop on Software Tools for Multi-Core Systems, 2008.

[8] Linchuan Chen, Xin Huo, and Gagan Agrawal, "Accelerating MapReduce on a Coupled CPU-GPU Architecture," in Proceedings of The International Conference for High Performance Computing, Networking, Storage and Analysis, 2012.

[9] Linchuan Chen and Gagan Agrawal, "Optimizing MapReduce for GPUs with Effective Shared Memory Usage," in Proceedings of HPDC, 2012.

[10] Jeffrey Dean and Sanjay Ghemawat, "MapReduce: simplified data processing on large clusters," Communications of the ACM, (1): 107, January.

[11] Richard M. Yoo, Anthony Romano, and Christos Kozyrakis, "Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system," in Proceedings of IEEE International Symposium on Workload Characterization (IISWC), pages 198–207. IEEE, October 2009.

[12] Y.Yan, M. Grossman, and V. Sarkar, "JCUDA: A Programmer Friendly Interface for Accelerating Java Programs with CUDA," in Proceedings of Euro-Par Conference Series. August 2009.

[13] M. Grossman, M. Pretermits, V. Sarkar, "HadoopCL: MapReduce on Distributed Heterogeneous Platforms Through Seamless Integration of Hadoop and OpenCL," in Proceedings of HDPIC 2013.

[14] Bingsheng He, Wenbin Fang, Naga K. Govindaraju, Qiong Luo, and Tuyong Wang, "Mars: a MapReduce Framework on Graphics Processors," in Proceedings of 17th International Conference on Parallel Architectures and Compilation Techniques, pages 260–269, 2008.

[15] J. Dean and S. Ghemawat, "Mapreduce: Simplified Data Processing on Large Clusters," Proc. Sixth Conf. Symp. Opearting Systems Design and Implementation (OSDI), 2004.