Hadoop Mapreduce OpenCL Plugin

Vivek Viswanathan
Dept of Computer Science
Illinois Institute of Technology
Chennai, Tamil Nadu, India
+91-9500011466
vviswan2@hawk.iit.edu

ABSTRACT

Modern systems generates huge amounts of information right from areas like finance, telematics, healthcare, IOT devices to name a few, the modern day computing frameworks like Mapreduce needs an ever increasing amount of computing power to sort, arrange and generate insights from the data.

This project is an attempt to harness the power of heterogeneous computing, more specifically take benefit of parallelism offered by modern day GPU's and accelerate data processing of Mappers and Reducers in the Mapreduce framework. In this regard, AMD's opensource APARAPI library was used as the foundation to develop a simple to use heterogeneous framework. The framework uses both the CPU and GPU to perform processing by translating sections of Mapreduce Java bytecodes to OpenCL and executing the same in parallel in AMD GPU's.

Typical Mapreduce tasks were then run in this framework with various types of workloads and while no perceptible improvement in speedups were noticed the framework shows excellent promise in computing complicated parallel data in upcoming high performance computing workloads.

KEYWORDS

Hadoop,Mapreduce,OpenCL,CUDA,APARAPI,rootbear, Apache, Storm, Spark, JNI.

1. INTRODUCTION

While the size and complexity of data processing keeps including from financial analytics, healthcare data and records, data from IOT devices the task of processing them normally falls to the Hadoop platform using Mapreduce framework. The speed of computing using Mapreduce is limited by the amount of processor cores available in the compute/data nodes and with raising thermals/die space limitations makes increasing the compute power a costly proposition. To mitigate the scenario this project aims to investigate the feasibility to harness the growing and inherent parallel processing power of GPU's by developing a framework for current Big Data data systems namely Mapreduce. The solution developed involves seamless integration of Mapreduce with OpenCL GPU computing framework running on AMD video cards in Linux environment with Hadoop environment.

The framework was then tested with IO intensive, Compute intensive and mixed workloads to measure the relative of performance of the system when compared with native implementation.

1.1 Background Information:

In the current world of big data sciences where hardware is becoming extremely affordable with distributed systems especially the systems based on Hadoop storing all the data generated be complex and varied and systems, processing all the generated data still requires heavy amount of CPU processing with multiple worker nodes and high power consumption. In such a scenario this project attempts to bridge the performance of Hadoop clusters by leveraging the native parallelism of GPU's to increase the speed up of the cluster as a whole by developing a plugin for Hadoop and GPUGPU computing world.

OpenCL is a popular computing framework for HPC (High Performance Computing) which has the ability to run on AMD, Intel and Nvidia (till Femi architecture) platforms without the limitations of other for Vendor lock-in for Nvidia GPU's like CUDA, platform lock-in like C++ AMP along with inability to run on CPU's coupled with startup overheads which makes small tasks acceleration time consuming and difficult and finally OpenACC framework's lack of maturity and support by major players makes the OpenCL framework an viable and interesting framework for the project. OpenCL is primarly programmed using the OpenCL 'C' wrappers.

Mapreduce is the primary programming framework/model of Hadoop clusters and developed using the Java programming language.

1.2 Motivation:

There is an explosion of wearable devices and sensors in the market with tiny pedometers to advanced ECG sensors and body area networks which could be worn around the human body which generates humongous amounts of data. Example of such devices are American Megatrends (AMI),Inc B.O.L.T. and Vitals Fit product lines. AMI"s healthcare group product is among the top 10 finalist team in Tricorder X-Prize competition run by the X-Prize foundation and the product named VitalsFit generates a

heavy amounts of data right from acting as activity tracker to ECG, Blood Oxygen, Glucose, Urine analysis and host of other sensor parameters from a single unit.

Conventional way of storage of this type of data would be to use a Blade Server with SAN boxes and arrays of JBOD"s running any server OS with a RDBMS. Since the amount of data generate and transmitted by these sensors is huge conventions RDBMS running SAN based blade solutions would prove to be prohibitively expensive to store and then to perform any analytics on the data which runs to the tune to hundreds of Terabytes of different type of structures, semi-structured and un-structured data which would keep growing.

Thus the storage and processing of this data falls to Hadoop based clusters since conventional systems could not handle such type of loads. Hadoop based cluster solution is proposed with HDFS file system, HBase NoSql database engine for datastore, Apache Storm for real time analytics of the data and Apache Ambari manager for management of the cluster.

Apache Hadoop is a software framework for processing of huge volumes of data across cluster of machine with a single programming model. Hadoop include HDFS or the Hadoop Distributed file system which is a distributed file system with inbuilt fault tolerance, replication and high availability (from 2.0.0).

In such a system the amount of mathematical processing required to computer each user's health-parameters, relating the trends pulled from the body with their physical activities and sleep patterns, designing mapping solution between their EMR (Electronic Medical Records) and current health status requires running of complicated computation intensive algorithms.

While the current Big Data systems could handle such loads it requires heavy amount of computation nodes in-terms of data nodes and usage of alternate frameworks like Spark, Storm with huge amounts of RAM to perform In-Memory Processing to achieve with less latency and high throughput.

The above approach is a financially costly proposition and also draws high power consumption owning to addition nodes added. Hence the reasoning to integrate the popular Mapreduce framework with GPU's to offload the processing to investigate the feasibility of such a system while measuring improvements to such a system.

2. PROPOSED SOLUTION

In Mapreduce the Map phase is fully parallel and the combiner phase is local to a machine and is semi-parallel and in a typical computation anywhere from 60% to 80% of the total time would be spent on these two phases of Hadoop execution. The objective was to create a plugin for which would be inhereited by Mappers and Reducers to accelerate the parallelizable sections of the code while maintaining the below points,

a. Seamless wrapper for Mapper class with little or no re-write of the existing code except for inheritance and class import to support running for java kernels.

b. Reducing of IO and communication layers between Mapreduce and OpenCL to achieve minimum latency and maximum use of bandwidth between the CPU and the GPU.

OpenCL till current version of 2.0 does not support the usage of managed code like JVM or .Net framework and supports only C99 standard for programming. This places a restriction on integrating with Hadoop Mapreduce since vast majority of Mapreduce programs are written in Java and in some cases Python with very less C99 programming support.

For creating the wrapper to Mapreduce two approaches are possible. The first is to develop Java Native Language Interface (JNI) wrappers for the sections which the programmer wishes to accelerate and develop the program in OpenCL in C99 and finally push the data back to the managed code sections.

Another approach is to support runtime translation of Java bytecodes to OpenCL by leveraging the support provided by translation libraries like 'rootbear' for CUDA support or Aparapi for OpenCL. These libraries provide support for running managed code by invoking and building equivalent OpenCL or CUDA compiles are developing JNI wrappers. This framework attempts to achieve to provide seamless integration with minimal code re-writes and hence uses the Native translation library framework.

Hence the project objectives as below in addition to points (a and b) are to,

c. Create a transparent framework to end user without requirement on OpenCL knowledge and JNI experience.

d. Benchmark the new framework on IO Intensive, Compute Intensive and Mixed workloads.

For reducing complexity the Hadoop system was limited to a single node cluster though the system could be expanded to multi-node setups.

2.1 Hardware and Software configuration of the development and test setup:

The framework was developed and tested on a system with the below Hardware configuration,

i. OS: Ubuntu 14.04 LTS

ii. CPU: 6-core Core™ i7-5820K @ 3.5Ghz

iii. RAM: Crucial DDR4-2133Mhz 8GB*2

iv. GPU: SAPPHIRE AMD R9 290X 4GB GDDR5 with core clock @ 1.0 GHZ. Cross-Fire disabled.

The complete list of software used for evaluation of the system included,

i. Hadoop version: 2.6.0

ii. Aprapi version: GitHub version 1.0.

ii. AMD OpenCL Drivers V 2.9.

iv. AMD OpenCL™ Accelerated Parallel Processing (APP) V3.0.

v. Eclipse Mapreduce plugin for version 2.6.0

vi. Eclipse Kepler.

vii. Joda time library to Datatime conversion

For development and testing Ubuntu 14.04 was the base Operating System and used for creating a single node Hadoop cluster.

Apache Hadoop version 2.6.0 source was downloaded from hadoop's git repository and built and installed in Ubuntu OS to create a single node cluster with One Name Node, One Data node and other supporting wrappers.

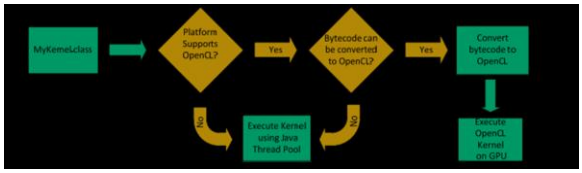
AMD OpenCL drivers version 2.9 which support OpenCL 2.0 functionality and AMD OpenCL Accelerated Parallel Processing (APP) SDK version 3.0 was installed with provides path to the GPU via OpenCL drivers for computation.

To ease programming development Eclipse Kepler was used in place of Marven to provide support for source level running and debugging and for faster development time.

For integrating Eclipse with Hadoop, Eclipse Mapreduce plugin for Hadoop was built and integrated with Eclipse IDE.

Finally Aparapi library v1.0 which was the first release after the codebase moved to gitbase from google code was downloaded and the paths setup in Eclipse and Hadoop to allow Hadoop to access GPU space.

2.2 APARAPI Library:



*picture courtesy AMD APARAPI presentation

APARAPI or A Parallel API is a library for executing Heterogeneous parallel work codes in Java initially Developed by AMD.

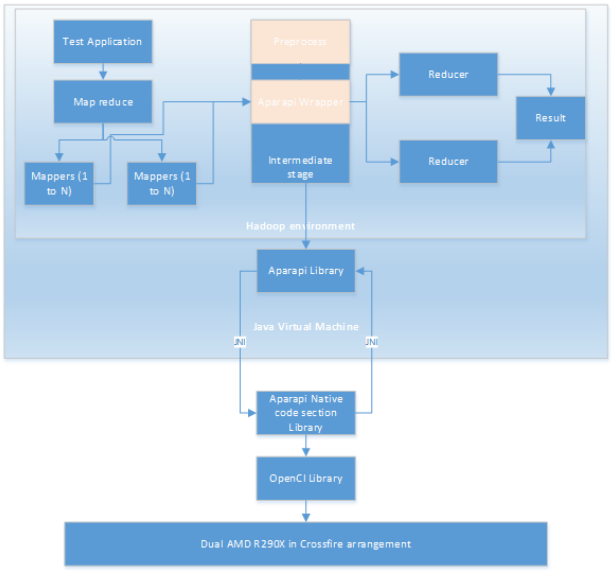
This is achieved by converting JavaByte codes to a series of expression trees to a series of if-else or if-then-else statements and then converting the same to OpenCL language using OpenCL compilers.

The programmer extends the kernel class and Java Bytecode is compiled using conventional method.

The Library then attempts to execute the code in the native GPU and if execution in GPU fails the library falls back to executing using Java Thread Pools (JTP).

While Aparapi does not support Mapreduce by itself the library was used as part of larger framework to support Mapreduce running on Accelerators by importing Aparapi to Hadoop ecosystem.

2.3 System Design:



The architecture of the framework is given in the above figure where mappers and reducers are inherited to create custom mappers and reducers. For example our class would extend the mapper class and reducer class. In addition apaprapi.kernel base class is extended and run () method is overloaded with the custom algorithm which the programmer wants to use in the project.

While doing the same the below limitations needs to be handed which are,

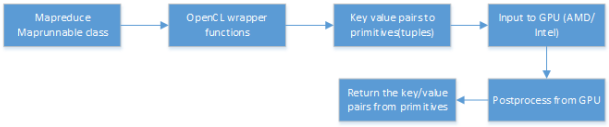
a. There is no support for Objects. All code needs to be converted to parallelizable code before the framework is called to execute the code.

b. Due to lack of object support only primitive types restricted to short, int, float and double are supported.

c. 'char' data type is not supported and the programmer needs to convert the String/Character

array to 'short' or 'int' before passing for calculation.

d. Since OpenCL kernels require memory allocation before the kernel is called, any buffers needs to be pre-allocated before kernel execution is called.



The above Figure shows the flow of the framework with the Maprunnable interface connected to OpenCl wrappers which are in-turn converted to tuples and fed into the GPU via OpenCl plugins.

2.4 System Implementation:

Some of the challenges while developing this system includes very poor synchronization between Mapreduce objects and Aparapi, lack of String handling support, memory constraints where the mapper or reducer data is bigger than GPU memory and Cluster support.

With the limitations the code flow and development of programs to make use of the GPU framework is discussed below with the below example of an addition program,

In a typical Mapreduce application to add a series of numbers to itself the mappers would perform the addition while the reducer would be the aggregator. The code for mapper would be like,

```
public class AdditionCPU extends Mapper<LongWritable, Text, Text, FloatWritable> {

    public void map(LongWritable key, Text value, Context contex) throws IOException, InterruptedException {

……..

        while (tokenizer.hasMoreTokens()) {

            String token = tokenizer.nextToken();

            Float F = Float.parseFloat(token);

            values.add(F);

            temp = F+F;

            addition.add(temp);

        }

……..

            contex.write(word, val);

……..
```

Example of CPU Mapper code for Adding a Number by itself

The above example is a generic Mapreduce code for adding a number where the data comes from the input string array which is then converted to float and the two numbers added.

The same program when developed with the current plugin would entail changes like below,

```
public class AdditionGPU extends Mapper<LongWritable, Text, Text, FloatWritable> {

    public void map(LongWritable key, Text value, Context contex)
            throws IOException, InterruptedException {
        final ArrayList<Float> tempValues = new ArrayList<Float>();
……..
        while (tokenizer.hasMoreTokens()) {
            String token = tokenizer.nextToken();
            try{
            Float f = Float.parseFloat(token);
            tempValues.add(f);
            }
            catch(Exception e){
                System.out.println(e);
            }
        }

        size = tempValues.size();
        final float[] values = new float[size];
        final float[] addition = new float[size];
       Float temp;
        for(int i = 0; i<tempValues.size(); i++){
            temp = tempValues.get(i);
            values[i]=temp.floatValue();;
        }

        Kernel kernel = new Kernel() {
           @Override public void run() {
                int globalID = getGlobalId();
                   addition[globalID] = values[globalID] *2;
            }
        };
     Range range = Range.create(size);
     kernel.setExecutionMode(Kernel.EXECUTION_MODE.GPU);
     kernel.execute(range);
     kernel.dispose();
……..
            contex.write(word, val);
……..
```

Example of GPU Mapper code for Adding a Number by itself

The above code example shows the changes required for executing in the GPU. First the values which are converted to float are stored in a temporary ArrayList whose size is calculated. A series of *final* arrays are created onto which the values are then copied from the ArrayList.

A new kernel is then created and the run method overloaded with the program which in this scenario was the number multiplied by 2.

The range and execution mode needs to be specified and the maximum size of buffer should not exceed the video memory which in the current system was 4GB and the kernel is executed.

Once the execution is finished the kernel is disposed by called the dispose function.

Internally during execute phase Aparapi converts the Bytecode inside the Kernel method to OpenCL and run the same in the GPU. If the execution mode specified is the CPU then the program is executed in CPU using Java Thread Pools (JTP).

A similar approach needs to be carried out in sections where parallel programming is required and other sections of the code remains a constant.

For example the below would be the implementation of reducer which for the addition program does not require any changes.

```
public void reduce(Text key, Iterable<FloatWritable> values, Context context)  throws IOException, InterruptedException
    {
……
        context.write(key, new FloatWritable(result));
    }
```

Example of reducer which is not changed since no parallel computation is carried out.

The below would the example of the AdditionDriver which is written like any conventional Mapreduce driver.

```
public static void main(String[] args) throws Exception {

    String input = "Addition10kb.txt";
    String output = "out";
        Configuration conf = new Configuration();
……
        job.setMapperClass(AdditionGPU.class);
        job.setReducerClass(AdditionReduceGPU.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(FloatWritable.class);
……

        if (!job.waitForCompletion(true))
            return;
……
        System.out.println("Time taken for running="+ ((double)elapsedTime/1000) +" Seconds");

}
```

Example of Mareduce driver calling the new GPU calling convention modified Mapper and Reducer.

As seen from example below, in the developed framework only sections which require parallel programming is modified and other sections could be re-used from existing code repositories.

The complete system was developed in Java using Eclipse IDE with Hadoop Mapreduce plugin. The total lines of program including both CPU and GPU sections is around 5000 lines of code written in Java.

3. EVALUATION

3.1 Methodology:

The framework was evaluated against three specific type of workloads which included IO Intensive, Memory Intensive and Mixed workloads.

All the programs were run in pure CPU mode which did not have any Aparapi classes and framework, GPU mode which used Aparapi and finally in JTP mode which used Aparapi but the execution mode was set to CPU which forced the running in JTP with one thread per core.

In CPU mode considering that this was a single node cluster, two mappers and one reducer was used and in GPU and JTP modes one mapper and one reducer was used since the parallelism was handed in the GPU section.

IO intensive workloads included Squares and Addition program, memory intensive workloads included Kmeans plus Blackscholes and finally for mixed workload the program to find pi for a number.

Test cases like wordcount and sort was not document as a part of this project since the use cases of GPU computing would involve computation intensive workloads. The reason is that the process of pushing the data from Arraylists to *final* Arrays and passing the same to OpenCL and executing in the GPU via PCIe interconnect would have induced heavy latency that GPU was not be able to compensate for the time lost. Since this framework supports execution of both conventional and GPU code only corresponding mathematical workloads could be accelerated.

In all the above modes three trials were run and results averaged and plotted.

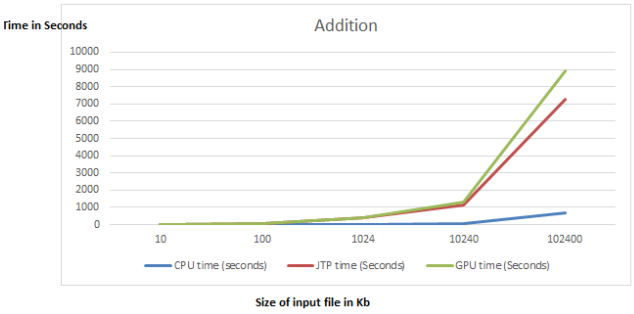
3.2 Test cases:

The below charts shows the performance of the testcases on the CPU, JTP and GPU systems.

3.2.1 Addition:

Typically most of Mapreduce programs falls with trying to count the number of occurrences, aggregating and filtering the data. Hence this program runs a simple addition code on set of input data vectors from 10 Kb to 100 Mb on the system.

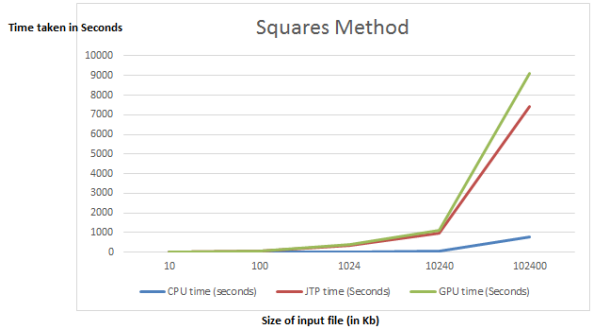
This is an IO intensive operation the maximum impact of offloading to GPU and addition arrays creation would be seen this benchmark.



As expected there is a 15X to 20X drop in performance due to offloading to GPU when compared to native execution.

3.2.2 Squares:

Mapreduce applications have a typically high requirement for multiplication and squares calculation like least squares method.

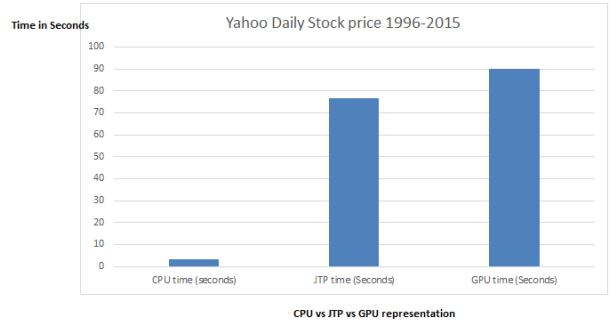


This was an IO intensive workload and GPU bottlenecks are again clearly visible here with the same margin of around 15X lesser performance when compared to CPU execution.

3.2.3 Moving Average:

Mapreduce is heavily used in the Financial industry and one of most common applications is to calculate moving average of multiple stock prices and reduce the fluctuation and volatility.

Here yahoo's daily closing stock data was used from November 1996 till May 2015 and moving average performed over a 60 day interval. Since it is only one dataset the calculation time is represented in a bar chart.



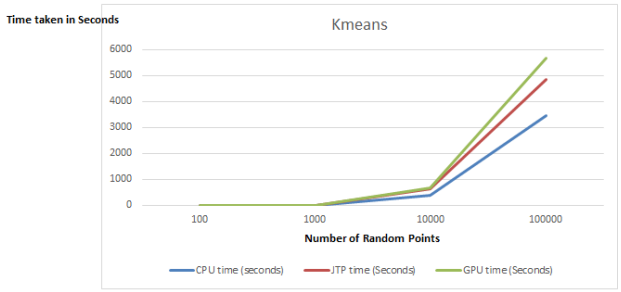
In this benchmark GPU had even higher Gap to CPU with a 40X drop in performance during calculation.

This goes to show in typical, simple IO related operations or arithmetic even if the code is parallelizable the IO overheads does not allow the GPU system to be any faster than CPU.

3.2.4 Kmeans:

Kmeans is a popular cluster analytics tool which is used in Datamining which is also used in unsupervised learning and classification. Due to its popularity in datamining kmeans is a popular benchmark to run in Hadoop clusters.

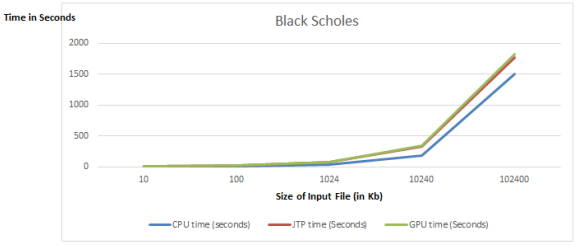
Kmeans is a memory intensive workload and was ideal case of higher end arithemetic operation which would be run on Hadoop cluster. For this test sample points of 100, 1000, 10,000 and 100,000 random 1-D points were given as input from a text file like used for IO intensive test-cases sample set.



GPU performance is good for this testcase but the CPU performance is still higher by around 20%. Another interesting observation is that JTP (Java Thread Pool) is still faster than GPU pointing to the fact that pushing data from Java to OpenCL and passing it via PCIe is causing bottlenecks.

3.2.5 Blackscholes:

Blackscholes is popular financial tool benchmark with full mapper stage but no calculation in the reducer stage. This benchmark should also be able to make maximum use of GPU computation power. Similar to addition test case, a sample of 10Kb to 100MB file was used for simulation.



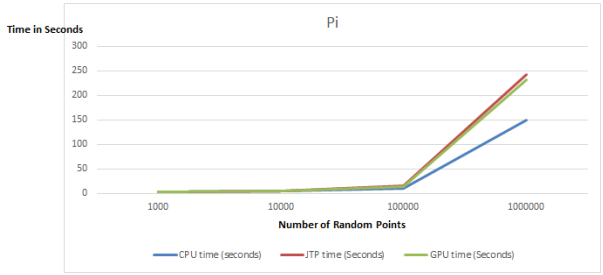
Performance in Blackscholes is very similar to the one achieved in Kmeans though the performance of GPU and JTP is similar while GPU having the additional overhead of passing to the GPU to execute showing the increase in performance of GPU's.

Still native CPU execution still holds a healthy 20% lead over GPU section.

3.2.6 Pi:

A common hadoop bechmarking program used to calculate the value of Pi by creating random 2D input and calculating whether the points are inside or outside a circle to calculate the value of Pi.

For Pi benchmark, the number of tasks was kept as 10 and number of points generated varied from 1000 to100,000.



Native CPU in benchmark also has a health lead of 25% though the GPU processing is faster than executing in JTP.

3.3 Analysis of results:

The results Preliminary analysis points to the fact the results and speedup are not as expected when compared to competing analysis made on the subject.

From doing the timing calculation it is clear the cost of pushing data to GPU and retrieving it back via the use of additional buffers and across native and managed code is not compensated by the increase in parallel performance offered by the GPU. This is particularly visible in IO intensive workloads where the performance of CPU was on average 20X faster than GPU computation.

Another possible reason would the cost of creating buffers to push to the GPU space. It is to be noted in the performance graphs that the performance penalty between pure CPU execution (JTP) and pushing the data to GPU was similar. This shows that while GPU is able to compute the data inspite of passing data through PCIe interconnect the cost of creating final buffers to pre-allocate memory and conversion from objects to native data types and pushing the same to OpenCL space takes a heavy performance penalty. Also calling the OpenCL compiler run-time could be another possible factor in reduced performance.

Still GPU computation was able to come close to CPU benchmark's in computationally intensive workloads like Black Scholes and Kmeans hence showing a wide difference in actual performance depending on workloads.

Another difference in lack of performance could be due to the usage of AMD video cards which are primarily designed for Desktop Gaming compared to dedicated GPU accelerator's like Nvidia Tesla and XeonPhi.

4. RELATED WORK

There have been quite a number of studies on suitability of integrating Hadoop with GPU primarily with CUDA, most of the work deals with interfacing in C99 level.

MARS (Map Reduce Framework for Graphics Processors) proposed by Bingsheng He et al proposed a framework where CUDA code is written in backend and the corresponding JNI wrappers would be called from Mapreduce and algorithms implemented in CUDA.

GPMR proposed by Stuart et al proposed and implemented a dedicated framework based on C++ which interfaces Mapreduce and C99.

JavaCL is another Mapreduce OpenCL framework proposed which provides JNI wrappers to be embedded into Java/Mapreduce codebase and the algorithms written in OpenCL.

Finally MapCG is another framework similar to MARS designed for CUDA uses a GPU based hash table to remove sorts before reduce on key-value pairs.

All the above frameworks are based on Java to CUDA or OpenCL integration by usage of two different programming languages by developing and passing the buffers in Java via JNI and then developing the program in CUDA/OpenCL separately while the current framework attempts at seamless integration by run time compiling and linking.

In the paper namely "A Research of MapReduce with GPU Acceleration" in which the authors have implemented a custom wrapper around Mapreduce to accelerate the same into GPU for computation but actual implementation details are not given and source code is not available in the Opensource domain for comparison.

A system implementation very similar to the proposed design is presented in the paper "HadoopCL" by Max Grossman et al where the implementation uses the similar approach of using Aparapi wrapper and creates Mapreduce to GPU calls using which the authors have achieved impressive speedup of 3.5X over non accelerated tasks but the source code is not available/provided in the Opensource domain and key implementations details are not elaborated in the paper to either implement or to evaluate and benchmark against the proposed solution.

While the proposed framework uses the similar approach of HadoopCL which is using Aparapi , the proposed framework differs from 'HadoopCL' in flowing methods which are,

a) Only the required parallel sections are accelerated and not the full program to allow the programmer to decide on which section he/she wants to accelerate.

b) While 'HadoopCL' is designed to run Mapreduce in a single node system while the proposed solution could run

typical Hadoop clusters in all the Data nodes if GPU is available or would fall back to CPU mode if unavailable.

There are multiple commercial implementation of Mapreduce-GPU integration an example is from SteamComputing.eu where there are solutions for GPU acceleration for both Mapreduce and Storm which could be beneficial to speedup both real-time data using Strom and for data mining using Mapreduce but being proprietary closed source implementation it is not feasible to get implementation or speedup details for comparison and benchmarking.

The final implementation of Hadoop-OpenCL computation is from a commercial entity named ParallelX where the implementation includes a Mapreduce to OpenCL running on Amazon AWS cloud and claimed to have an extremely high speedup 5X to 5000X on Mapreduce tasks. Similar to the above example the solution being commercial closed source implementation, it is not possible to get further details about the implementation or the workloads where the suggested speedups where achieved.

5. CONCLUSION

A framework was developed integrating Mapreduce with OpenCL and successfully tested in Hadoop Mapreduce framework proving the feasibility of such a system. The system was then tested with mixture of IO intensive, memory intensive and mixed workloads performance measured running in both CPU mode and GPU mode.

While the overall performance offered by AMD GPU's was not better than pure CPU calculations, some of the performance deficit could be attributed to executing lot of double precision operations on Desktop CPU card optimized for single precision operations. Also the video card was used for rendering Display and other related video related operations while performing operations which could cause bottlenecks in the PCIe lane.

From the results it is clear that GPU shows tremendous performance gains when executing computationally intensive tasks like Blackscholes and Kmeans when compared to pure IO bound operations thus demonstrating the power and potential usability of GPU's in Big Data space.

Some of the key learning included cementing existing knowledge of Mapreduce and while provide a good introduction into GPGPU computing world along with challenges and advantages of integrating GPU's in Big Data space. This project has given more insight into computation of data and the knowledge to use different computing frameworks based on data input.

To conclude while a pure GPU based solution may not be the perfect solution for Big Data, a symbiosis of GPU and CPU solution as shown by the proposed framework where the programmer would take advantage of respective solutions would be the best solution for future computing requirements.

6. FUTURE WORK

While Mapreduce-GPU integration could be investigated further to improve the performance the below would be the list of improvements which could be carried out in the framework

a. Aparapi tends to add heavy amount of latency and lack of support for Objects and multiple datatypes. Hence Aparapi could be replaced with the upcoming Project Sumatra from OpenJDK which provides support for OpenCL, CUDA, Intel Phi, PTX and HSA with full support for objects and data types. This would make the framework more flexible and portable across platforms with different GPU types.

b. While Mapreduce is useful for iterative batch processing workloads to handle the new set of IOT devices, Spark is the framework of choice. Hence the framework could be ported to Spark along with providing support for Scala programming language for more wide spread adoption.

With the above proposed work, Mapreduce-GPU and Spark-GPU would be able to able to make use of different types of GPU cards and remove the limitations of the current architecture.

7. REFERENCES

[1] Pieloth, C:- GPU-basierte Beschleunigung von MapReduce am Beispiel von OpenCL und Hadoop

[2] b. Jie Zhu, Jonesboro, Juanjuan Li ; Hardesty, E. ; Hai Jiang ; Kuan-Ching Li:- GPU-in-Hadoop: Enabling MapReduce across distributed heterogeneous platforms, In: Computer and Information Science (ICIS), 2014 IEEE/ACIS 13th International Conference.

[3] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang :-Mars: a MapReduce framework on graphics processors. In: PACT 2008.

[4] Shirahata.K, Sato, H. ; Matsuoka, S :- Hybrid Map Task Scheduling for GPU-Based Heterogeneous Cluster In: Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference Tavel, P. 2007. *Modeling and Simulation Design*. AK Peters Ltd., Natick, MA.

[5] Max Grossman, Mauricio Breternitz, Vivek Sarkar :- HadoopCL: MapReduce on Distributed Heterogeneous Platforms through Seamless Integration of Hadoop and OpenCL , In: International Workshop on High Performance Data Intensive Computing 2013

[6] f. Miao Xin, Hao Li :- An Implementation of GPU Accelerated MapReduce: Using Hadoop with OpenCL for Data- and Compute-Intensive Jobs, In:- Service Sciences (IJCSS), 2012 International Joint Conference

[7] Elteir, M. Heshan Lin ; Wu-chun Feng ; Scogland, T. :- StreamMR: An Optimized MapReduce Framework for AMD GPUs , In: Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference

[8] Koichi Shirahata, Hitoshi Sato, and Satoshi Matsuoka. "Hybrid Map Task Scheduling for GPU-based Heterogeneous Clusters" In Proceedings of the 1st International Workshop on Theory and Practice of MapReduce (MAPRED'2010).

[9] A. Stuart, John D. Owens, Multi-GPU MapReduce on GPU Clusters, in: Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium, 2011

[10] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance

[11] Computer Architecture, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[12] F. Ji and X. Ma. Using shared memory to accelerate mapreduce on graphics processing units. IPDPS,2011.

[13] A. Leung, O. Lhot´ak, and G. Lashari, "Automatic parallelization for graphics processing units," in Proceedings of the 7th International Conference on Principles and Practice of Programming in Java, ACM 2009.

[14] F. Jacob, D. Whittaker, S. Thapaliya, P. Bangalore, M. Mernik, and J. Gray, "Cudacl: A tool for cuda and opencl programmers," in HIPC 2010.

[15] ParallelX: Bridging the gap between Big Data and GPU acceleration In:-http://www.parallelx.com/