HadoopCL: MapReduce on Distributed Heterogeneous Platforms Through Seamless Integration of Hadoop and OpenCL

Max Grossman
Department of Computer Science
Rice University
jmg3@rice.edu

Mauricio Breternitz
AMD
mauricio.breternitz@amd.com

Vivek Sarkar
Department of Computer Science
Rice University
vsarkar@rice.edu

***Abstract*—As the scale of high performance computing systems grows, three main challenges arise: the programmability, reliability, and energy efficiency of those systems. Accomplishing all three without sacrificing performance requires a rethinking of legacy distributed programming models and homogeneous clusters. In this work, we integrate Hadoop MapReduce with OpenCL to enable the use of heterogeneous processors in a distributed system. We do this by exploiting the implicit data-parallelism of mappers and reducers in a MapReduce system. Combining Hadoop and OpenCL provides 1) an easy-to-learn and flexible application programming interface in a high level and popular programming language, 2) the reliability guarantees and distributed filesystem of Hadoop, and 3) the low power consumption and performance acceleration of heterogeneous processors. This paper presents HadoopCL: an extension to Hadoop which supports execution of user-written Java kernels on heterogeneous devices, optimizes communication through asynchronous transfers and dedicated I/O threads, automatically generates OpenCL kernels from Java bytecode using the open source tool APARAPI, and achieves nearly 3x overall speedup and better than 55x speedup of the computational sections for example MapReduce applications, relative to Hadoop.**

I. Introduction

As hardware becomes more affordable, the scale of distributed systems and the range of applications used on them is rapidly increasing. Large clusters of machines are being used for data-intensive and high performance applications in science, finance, and medicine. The scale of these systems is not the only thing increasing. Heterogeneous systems which combine both a CPU and GPU [1] to gain advantages in performance and energy efficiency for suitable classes of applications have increased the complexity of large scale systems. As the use of these heterogeneous, distributed systems broadens they have come into the use domain of software developers whose skills and experience vary widely. This increases the burden on programmers to achieve efficient execution for their workloads, hence the need for expressive and efficient programming environments.

A. Challenges to High Performance in Large Scale and Heterogeneous Systems

One challenge with ever-growing systems is the dramatic increase in complexity. Programmers not only need to distribute their workloads across cores and be aware of intra-node issues such as cache misses and thread-level parallelism, but also be conscious of system level issues during inter-node parallelization. Without awareness of these factors and their effect on each other, performance suffers. The added layer of complexity from multi-node execution makes conceptualizing a system much more difficult for developers. This problem is exacerbated when designing applications that may need to run on multiple different systems with different processor architectures and interconnect topologies.

Closely related to complexity is programmability. Programmability measures the ability of a programmer to express their logic without solving problems unrelated to the actual task being performed (such as network bandwidth utilization or load balancing). In many existing distributed programming models (such as MPI), programmers are forced to perform explicit transfers between processes and use an additional thread-level programming model, such as OpenMP. Programmers also need to take into account the distinct processing requirements of their workloads and adapt them to the architectures in heterogeneous systems. This approach not only adds difficulties for the programmer, but also produces code that must be re-tweaked for every new platform it is used on. As systems become larger and more complex, even the tried-and-true programming models will cease to be sufficient for efficient execution, in terms of both programmability and performance. New programming models which use higher abstractions to hide complexity without restricting freedom of expression are necessary.

Another challenge in large scale systems which has received interest lately is reliability. As systems scale up, the time between hardware failures will decrease. Therefore, it is necessary to detect these failures and adapt around them at runtime without wasting execution time or unnecessarily terminating jobs. Reliability was the main driver for constructing our system, HadoopCL, on top of Hadoop. Hadoop has a robust job management system which uses replication and error detection to schedule around failures.

Finally, energy efficiency is a major concern with large scale systems. The combined cost of powering and cooling the hardware will make increasing beyond existing cluster sizes infeasible. Power-efficient processors consume less energy, produce less heat, and as a result make running larger systems feasible. Using power-efficient processors often implies adding heterogeneity to a system, increasing its complexity and decreasing programmability. This work addresses energy efficiency by offloading computation to GPUs, though we lacked the infrastructure to directly measure energy efficiency.

Complexity, programmability, reliability, and energy consumption all represent significant obstacles to achieving high performance and data throughput in distributed systems. This work presents modifications to the existing Hadoop distributed system which improve on performance and energy consumption by executing the computation of a Hadoop MapReduce job on heterogeneous processors without sacrificing the existing benefits of Hadoop. The contributions of HadoopCL include:

1) Extension of Hadoop Mapper and Reducer classes to support execution of user-written Java kernels on heterogeneous devices, with a focus on minimizing required modifications for legacy code.
2) The use of dedicated communication threads and asynchronous communication to maximize utilization of available bandwidth and limit blocking on communication.
3) Automatic translation of Java bytecode to OpenCL kernels using APARAPI [2], and extensions to APARAPI's existing features.
4) Evaluation of HadoopCL's performance in two multi-node clusters containing multi-core CPUs, GPUs, and APUs (CPU & GPU on a die with physically shared memory).

B. Hadoop

One of the industry leaders in addressing some of the issues in large scale systems as described in Section I-A is Hadoop [3], built based on Google's MapReduce [4] model and distributed file system. Hadoop is a Java-based system which provides reliability guarantees, reduces the software complexity perceived by the programmer, and provides an easily programmable interface in a popular programming language.

The MapReduce model contains two high-level computational stages: map and reduce. In the map stage, input (key,value) pairs are passed to a mapper which generates 0 or more output (key,value) pairs. These (key,value) pairs output from the mapper act as input to the reducer stage. In the reduce stage, outputs of the map stage which share the same key are passed to the same reducer. The reducer then generates 0 or more (key,value) pairs based on this input as the final output of the job. Both the map and reduce computation are therefore implicitly data parallel across (key,value) pairs and have no data-dependencies across inputs, trivially mapping to many- or multi-core hardware.

While Hadoop is popular for its simplicity, it also demonstrates inefficiencies which decrease its usefulness for certain problems due to sub-par computational performance and network utilization. Mapper and reducer tasks run inside of potentially short-lived Java virtual machines. Creating, managing, and executing inside these JVMs incurs processor and memory overhead and may reduce the effectiveness of JIT compilation. However, using separate JVMs provides a significant reliability advantage by isolating the Hadoop system from mapper and reducer failures. Using JVMs also provides the programmer with a powerful programming language and useful libraries to work with. Therefore, it is important to optimize the performance of these JVMs. Additionally, Hadoop includes a distributed file system, HDFS. Our own evaluations show that HDFS I/O operations consume a significant percentage of total execution time in most applications, but HDFS provides data reliability via replication as well as a number of other desirable features. As a result, optimizing communication to and from HDFS is important for improved performance.

C. OpenCL & APARAPI

OpenCL [5] is an industry standard, SIMD, heterogeneous programming platform, tackling many of the same problems as Hadoop but with a different approach. While OpenCL is not a distributed programming model, it addresses many of the programmability issues of working in heterogeneous systems by providing a standard API for different architectures. This decreases the complexity visible to the programmer and makes code easily portable between different processor architectures (CPUs, GPUs, APUs, etc).

OpenCL uses a hierarchical, batched threading model. Threads are grouped into thread groups, which share some local memory and are able to synchronize with each other. Generally, multiple thread groups are launched in a single kernel invocation by an OpenCL programmer. All threads in a single thread invocation execute the same kernel, just as the same function is applied to all inputs in the map and reduce stages of a Hadoop job. OpenCL also has a separate address space from the host program, and requires explicit writes and reads into this conceptually (and possibly physically) separate memory.

APARAPI [2] is an open source tool developed at AMD which provides JIT compilation of Java bytecode to OpenCL kernels and execution of those kernels on the heterogeneous devices available in an OpenCL platform. It only supports a subset of the bytecode specification, most notably not supporting any objects but *this* and only allowing primitive types or single-dimensional arrays of primitives to be used in the kernel. Otherwise, compilation and execution of OpenCL kernels is efficient and straightforward. APARAPI handles kernel translation, OpenCL memory allocation, data transfers, and kernel invocation completely transparent to the Java programmer. In short, APARAPI provides a quick path from executing inside a JVM to executing in native threads on heterogeneous devices.

II. Approach

This section will cover the techniques and algorithms used in executing Hadoop Mappers and Reducers on heterogeneous hardware.

A. Heterogeneous Mapper and Reducer

As described in Section I-B, the computation in Hadoop jobs is encapsulated in user-implemented mappers and reducers. Mappers transform an input (key,value) pair to zero or more output (key,value) pairs. Reducers take as input the outputs of the mappers as (key,value-list) pairs, where the value list is composed of all values associated with the same key in the output of the mappers. Reducers then generate an output of 0 or more (key,value) pairs. Mappers and reducers are therefore inherently data parallel, and exhibit high degrees of parallelism for data intensive applications.

The main contribution of this work is heterogeneous mappers and reducers which, when extended, automatically:

1) Execute user-written map and reduce computation natively on all available devices in a platform.
2) Use multiple input and output buffers, dedicated communication threads, and asynchronous kernel execution to maximize utilization of disk and inter-device bandwidth.

Figure 1 is a high-level system diagram of the HadoopCL system contained in a single node. The TaskRunner manages the spawning of child JVMs within a node, which execute the mapper and reducer computation isolated from the Hadoop sytem. TaskRunner is mostly unmodified, but is responsible for tracking device usage and assigning a device to each JVM as it is spawned based on a device management algorithm.

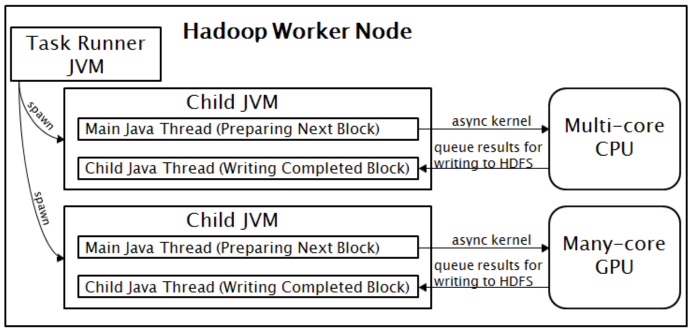

Hadoop Worker Node
Task Runner JVM
spawn
spawn
Child JVM
Main Java Thread (Preparing Next Block)
Child Java Thread (Writing Completed Block)
async kernel
queue results for writing to HDFS
Multi-core CPU
Child JVM
Main Java Thread (Preparing Next Block)
Child Java Thread (Writing Completed Block)
async kernel
queue results for writing to HDFS
Many-core GPU

Fig. 1. System diagram for HadoopCL.

Each Child JVM is assigned a single HDFS chunk of keys and values for processing. These Child JVMs can execute either map or reduce computation, and much of the HadoopCL runtime implementation is shared between heterogeneous mappers and reducers. Each Child JVM contains two Java threads with different responsibilities. The main Java thread is responsible for

1) Reading and buffering input keys and values from the input HDFS chunk.
2) Enqueueing the asynchronous OpenCL commands to 1) copy buffered data to the assigned device, 2) launch an asynchronous mapper or reducer kernel to operate on that input, and 3) retrieve the output of the kernel from the device when the buffer is full.
3) Feeding the generated outputs to a dedicated communication thread.

Pseudocode for the algorithm executed by the main thread can be found in Algorithm 1 below. The child communication thread is responsible for receiving output data from devices and writing it back into the HDFS system. The child thread uses a concurrent queue containing buffers of output data. This queue is fed by the main thread and read by the child thread. While only two buffers are shown in use in Algorithm 1, in practice more than two buffers are used. This leads to potential accumulation of buffers in the input queue to the I/O thread.

```
initialize buffers;
while have remaining input (key,value) pairs do
    if buffers[preparing] is full then
        wait for processing of buffers[executing];
        launch kernel on buffers[preparing];
        hand off buffers[executing] to child I/O thread;
        swap preparing and executing buffers;
    end
    add current (key,value) to buffers[preparing];
end
```

Algorithm 1: Algorithm executed by main thread in each Child JVM.

Using asynchronous kernel launches, asynchronous copies to device memory, and a dedicated I/O thread maximizes the overlap of computation and communication, keeps the network and device bandwidth utilized, and prevents the main thread from blocking on most operations.

B. Heterogeneous Execution of JIT Compiled OpenCL Kernels

HadoopCL relies on APARAPI [2], an independently developed tool, to translate the bytecode compiled from user-written Java to OpenCL kernels. OpenCL kernel code is generated for both the user-written map and reduce functions, as well as for HadoopCL glue code which passes keys and values into the user-written functions. The HadoopCL glue code is written in such a way as to change its own memory access patterns and loop iteration chunking for optimal performance on the executing architecture. It currently supports optimizations for GPUs and multi-core CPUs.

In order to fully support this work, APARAPI was extended to support asynchronous kernel execution. This extension was achieved by modifying the APARAPI C++ runtime to store references to OpenCL events which are satisfied by completion of processing a certain buffer. A unique integer ID is associated with each event. These unique IDs are returned to the APARAPI Java runtime through JNI, where they are stored in a FIFO queue. In order to wait on the kernel which was launched furthest in the past and is therefore the most likedly to have already completed the next ID in the queue is removed from the front, passed to the APARAPI C++ runtime, and used to find the correct OpenCL event to wait on in order to ensure that execution has completed. This enables concurrent execution of kernels on all devices available in a platform.

C. Programming Framework

One of the main benefits of Hadoop is its programmability. It derives this programmability from the high-level abstractions of MapReduce and the use of Java as a high-level language. While the primary goal of HadoopCL is improved performance, it is important to consider the impact these modifications have on the programming model's flexibility for developers.

The use of APARAPI places some limitations on the set of the Java language constructs that can be used within mappers and reducers. APARAPI cannot translate arbitrary programs. It does not support references to arbitrary objects and only supports variables with primitive or primitive array types. There is support for translation of some of the more useful Java librariy calls, such as those in java.util.Math. These limitations mean that many Java mappers or reducers will need to go through some modifications to be usable in the HadoopCL framework. However, the resulting code will still be written in (a subset of) standard Java.

For copying arrays of primitives to and from the device, APARAPI uses a number of type-specific methods. For instance, `Kernel.put(int[] arr)` and `Kernel.get(int[] arr)` are used to copy integer arrays. This conflicts with Hadoop's extensive use of generics, and means that the types of input keys and values must be known at compile time for mappers and reducers that are to be executed as OpenCL kernels. As a result the generic Hadoop Mapper and Reducer classes are replaced by a number of auto-generated, type-specific HadoopCL Mapper and Reducer classes which can be automatically replaced by OpenCL. For instance, extending `Mapper<IntWritable, FloatWritable,IntWritable,FloatWritable>` in Hadoop would be translated to extending `IntFloatIntFloatHadoopCLMapperKernel` in HadoopCL. Because these type-specific OpenCL mapper and reducer classes are formulaic, auto-generating them for arbitrary combinations of key and value types is trivial (while keeping in mind APARAPI only supports a subset of Java types). However, HadoopCL includes support for commonly used object types, such as pairs or triples of primitives, and generates the glue code for converting these objects to primitive arrays which APARAPI is capable of translating to OpenCL.

OpenCL does not support dynamic memory allocation. This implies that HadoopCL must preallocate all memory for mapper/reducer output keys and values before launching kernels. The developer must provide the framework with a limit on the number of output (key,value) pairs that will be generated from any input pairs passed to mappers and reducers. This value is then used to calculate the required size of preallocated memory. Future work would expand on this approach to make it more flexible for applications which may generate unbalanced numbers of output (key,value) pairs from different inputs to prevent over-allocation. In our experience, this requirement does not hinder application development.

As a case study to concretize the differences described above, let us compare the Hadoop and HadoopCL implementations of the mapper computation from a Pi benchmark which approximates the value of pi using randomly generated numbers. The original implementation is below:

```
class PiJavaMapper extends
    Mapper<DoubleWritable, DoubleWritable,
           IntWritable, IntWritable> {

  public void map(DoubleWritable key,
      DoubleWritable value,
  Context context)
      throws IOException,
  InterruptedException {
    double x = key.get() - 0.5;
    double y = value.get() - 0.5;

    if(x * x + y * y > 0.25) {
      context.write(new IntWritable(0),
        new IntWritable(1));
    } else {
      context.write(new IntWritable(1),
        new IntWritable(1));
    }
  }
}
```

The HadoopCL-compatible Java implementation is as follows:

```
class PiCLMapper extends
    DoubleDoubleIntIntHadoopCLMapperKernel {

  protected void map(double key,
      double val) {
    double x = key - 0.5;
    double y = val - 0.5;

    if(x * x + y * y > 0.25) {
      write(0, 1);
    } else {
      write(1, 1);
    }
  }

  public int getOutputPairsPerInput() {
    return 1;
  }
}
```

As described in the text above, the main differences are:

1) Extension of `DoubleDoubleIntIntHadoopCLMapperKernel` instead of `Mapper<DoubleWritable,DoubleWritable,IntWritable,IntWritable>`
2) A different map signature, `map(double, double)` instead of `map(DoubleWritable, DoubleWritable, Context)`
3) A replacement `write` method which writes to the preallocated output array, replacing `context.write()`.

4) The addition of `getOutputPairsPerInput()`, which the application developer uses to indicate to HadoopCL the maximum number of outputs that will be emitted per input. This is used for output array preallocation.

III. Related Work

There have been several previous investigations of using GPUs to accelerate MapReduce frameworks.

HAPI[6] is a simpler implementation of heterogeneous MapReduce which also uses APARAPI 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 preprocess(), gpu(), and postprocess() methods that must be implemented by application developers. Preprocess() translates input objects to HAPI specific objects which it is able to convert for use by APARAPI. Gpu() is the compute intensive part of map computation which is executed on the device. Postprocess() translates the output objects to Hadoop objects and outputs them to the Hadoop system. While the basic approach is similar, HAPI requires much more non-application code to be written by the developer (in a 50 line example presented in this paper, the actual computational kernel takes up only 3 lines) and separates out the application code into three separate methods. HadoopCL uses much more auto-generated code to provide more flexibility to and require less glue code from the application developer. While 80x speedup is achieved from an NBody benchmark, evaluation is only done on a single node which ignores network communication overheads, a significant part of real-world Hadoop jobs. There is no current support for multi-devices in HAPI, and the implementation leaves most of the CPU idle, using a single core to manage a single GPU. HadoopCL automatically utilizes all OpenCL devices in a platform, taking into account device type by optimizing memory accesses and loop chunking on the device.

[7] presents a single-node MapReduce Framework. While not built on Hadoop, it executes user-written mapper and reducer CUDA kernels on a GPU and so is able to avoid the problem of automated kernel generation. It also uses parallel reduction on the GPU to improve reducer performance, an optimization which could be added to HadoopCL in future work. The main difference between this work and HadoopCL are that HadoopCL 1) is multi-node and deals with communication challenges as a result, 2) cleanly integrates into the existing Hadoop framework, 3) uses APARAPI to translate Java kernels rather than requiring complete rewrites of existing kernels. In general, HadoopCL is applicable to a larger set of data-intensive applications due to its distributed nature, and enables more rapid porting of existing Hadoop applications.

GPMR [8] presents a high performance, standalone distributed mapreduce implementation in C++ and CUDA which uses the GPUs to execute mappers, reducers, and a number of intermediate stages added to the MapReduce pipeline. These intermediate stages take advantage of the GPU's parallelism to minimize inter-node communication. Similar to our work, much of the work presented in this paper is on optimizing or hiding communication. Several substages are added to the MapReduce pipeline to minimize communication costs, in a way that wouldn't be possible in HadoopCL because of its construction on top of Hadoop. They support data sets that do not fit into a single GPU's memory, a feature that was a design consideration in HadoopCL from the start. This work tries to expose many features of CUDA (such as local synchronization) to the application developer, which HadoopCL does not permit. GPMR also supports multiple GPUs per node with dedicated processes for each GPU, similar to HadoopCL. However, no application computation is executed on the CPU which may lead to underutilized hardware, something HadoopCL does support.

	Map In	Map Out/Reduce In	Reduce Out
Pi	(double,double)	(int,int)	(int,int)
Blackscholes	(int,float)	(int,float)	(int,float)
KMeans	(double,double)	(int,pair)	(double,double)
Sort	(long,long)	(int,long)	(int,long)

TABLE I
INPUT AND OUTPUT TYPES FOR EACH BENCHMARK

Mars [9] is another standalone MapReduce implementation, with added-in support for execution on GPUs. This work predates and is similar to GPMR, but only supports execution on a single GPU, does not consider inputs that do not fit into a single GPU's memory, and places more restrictions on the feature set in CUDA that is exposed to application developers. As with the framework presented in [7] and GPMR [8], Mars application code is written in low level CUDA or C++, rather than Java.

Chen et al [10] [11] did work on advanced optimizations to a MapReduce implementation executing on a heterogeneous architecture including modifications to the MapReduce scheduling algorithm, improvements in usage of GPU scratchpad memory, intermediate/immediate reduction, and runtime tuning. The evaluations in these papers show good load balancing and speedups of up to 28.68× relative to sequential execution. While the implementation and performance evaluations in this paper were not done on top of Hadoop, many of the ideas could be used to improve HadoopCL performance in future work.

IV. Experimental Setup

To evaluate HadoopCL's performance, we use 4 benchmarks: Pi, KMeans, Sort, and Blackscholes. Though small in size, these benchmarks are representative of high-performance data-intensive computing. This section will describe the function and implementation of each benchmark as well as the methodology used in gathering results.

A. Pi

The Pi implementation used in this work is based on the example provided with Hadoop. Pi is a common Hadoop benchmarking workload which estimates the value of pi by randomly placing 2D points and classifying them as inside or outside the unit circle.

The map computation takes as input a pair of Doubles, representing the 2D coordinates of a point. It calculates this point as either inside or outside the unit circle, and outputs two integers. The first integer indicates whether the point is inside the circle (0 or 1). The second integer is always 1, indicating that this is the data for a single 2D point.

The reduce computation counts the number of values associated with each key (in Pi there are only two reduce input keys, 0 and 1) and outputs an accumulated sum. Once the Hadoop job has completed, it is trivial to check the results against the true value of pi.

400,000,000 2D points were used as input to all Pi jobs in the results.

B. KMeans

KMeans is an iterative clustering algorithm and also a common Hadoop benchmarking workload. In our tests, we applied KMeans to 2D points whose coordinates were double floating point values.

The map stage of kmeans takes a pair of Doubles, representing a 2D point. The map function iterates through the current known centroids and finds the closest centroid to the input point. Finally, it outputs an integer indicating the closest cluster/centroid, and a pair containing the coordinates of the point.

The reduce stage of kmeans consumes all of the points which have been classified as belonging to a certain cluster, and recalculates the centroid of the cluster by finding the point which is closest to the average of all points' positions. The new centroid is then output as a pair of doubles.

The implementation of KMeans required adding the ability to retrieve global data from arbitrary devices. This was accomplished by allowing the creation of specially named system properties whose values are automatically interpreted as floating point and made accessible from OpenCL kernels.

In our tests, a single iteration of the KMeans algorithm is tested on 20,000,000 points and 10,000 clusters.

C. Blackscholes

Blackscholes is a common data-parallel financial application, but is not generally executed in Hadoop because it has essentially no required reduction.

The map stage takes as input a unique integer ID and floating point value. The Blackscholes financial algorithm is executed on the input value. Then two pairs are output: the integer ID paired with a put value, and the integer ID paired with the call value.

The reduce stage is essentially a no-op, simply passing the outputs of the map stage as the final output of the job.

In the Blackscholes tests, 401920000 input pairs were used.

D. Sort

Our sort benchmark is a distributed radix sort of 64-bit long integers, which maps well to MapReduce.

The map stage takes as input a pair of longs, both of which are values to be sorted. It outputs a radix for each paired with the input value. In our implementation we use the top 32 bits of each 64-bit long as the radix.

The reduce stage then performs a local sort on all values which share a radix and outputs the sorted results.

In these distributed sort tests 400,000,000 values are sorted.

E. Methodology

Results were gathered on two heterogeneous clusters, the DAVINCI cluster at Rice University and an AMD APU cluster at AMD-Austin.

Each node in DAVINCI contains two six-core Intel X5660 CPUs running at 2.80GHz and two discrete NVIDIA Tesla M2050 GPUs, connected by PCIe. Each GPU has access to 2.5GB of global memory. There is 48GB of system memory accessible from the CPUs. All tests were run using two nodes in the cluster: 1 NameNode and 1 DataNode.

Each node in the AMD APU cluster contains a single AMD A10-5800K APU (3.8Ghz) with Radeon(tm) HD Graphics on an ASUS FM2-A85XA-G65 motherboard with 16GB of RAM. The interconnect for these nodes is 1Gb Ethernet. Tests in the AMD APU cluster were run using 2 nodes (like DAVINCI) and 10 nodes.

Each test on the DAVINCI cluster at Rice University was run 5 times. Each test on the AMD APU cluster was run 3 times. The minimum time for each system (Hadoop and HadoopCL) is reported in our results. OpenCL v1.2 and Java Hotspot v1.6.0_25 were used. For OpenCL, 196 threads per thread group were used on the multi-core CPU and 256 threads per thread group on the GPU. For tests with Hadoop, an HDFS chunk size of 64MB was used. For tests with HadoopCL, an HDFS chunk size of 256MB was used. The larger chunk size for HadoopCL reflects the fact that individual Child JVMs in Hadoop are generally only responsible for keeping a single CPU core occupied, while HadoopCL Child JVMs feed an entire device. These values were arrived at after evaluating several different configurations for Hadoop and HadoopCL.

As results were collected, we were able to quickly draw the conclusion that I/O was a major bottleneck for some of these applications. The I/O bottleneck became more apparent with OpenCL-accelerated computation. As a result, tests on DAVINCI were also run in Hadoop and HadoopCL with compression enabled for the initial inputs in HDFS and the intermediate outputs of the mapper stage. These tests were run with all of the same parameters and configurations as uncompressed tests. While many different compression codecs can be used in Hadoop, exhaustive tests with the Sort benchmark on the BZip2, GZip, Default, Snappy, and LZO codecs indicated that the best combination was to use the Default(ZLib) compression codec for the initial inputs, and LZO compression for the mapper outputs. Though further investigation may reveal different optimal combinations for the other benchmarks, time constraints meant that all compression tests were run with these codecs.

Benchmark	Speedup w/o compression	Speedup w/ compression
Pi	1.50x	2.41x
Blackscholes	2.47x	2.91x
KMeans	2.25x	2.74x
Sort	1.54x	1.49x

TABLE II
OVERALL SPEEDUP OF THE HADOOPCL IMPLEMENTATION OF THE BENCHMARKS ON THE DAVINCI CLUSTER WITH AND WITHOUT COMPRESSION, RELATIVE TO HADOOP WITHOUT COMPRESSION.

Benchmark	Speedup on 2 nodes	Speedup on 10 nodes
Pi	1.53x	3.79x
Blackscholes	1.25x	1.38x
KMeans	2.07x	5.05x
Sort	1.07x	0.90x

TABLE III
OVERALL SPEEDUP OF THE HADOOPCL IMPLEMENTATION OF THE BENCHMARKS ON THE AMD APU CLUSTER USING 2 AND 10 NODES, RELATIVE TO HADOOP ON 2 NODES.

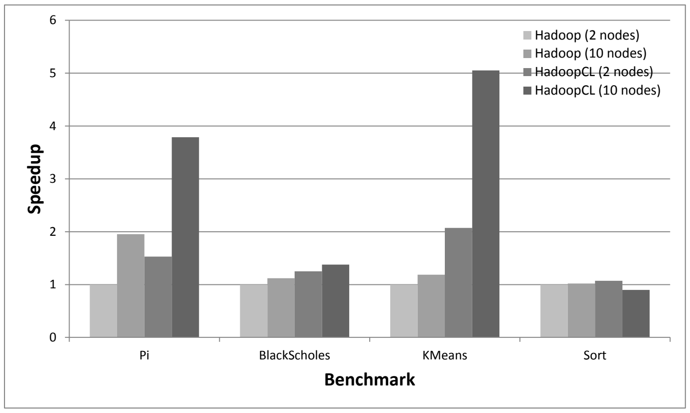


Fig. 3. Overall speedup of all benchmarks on the AMD APU cluster without compression, normalized to Hadoop without compression.

V. RESULTS

In this section we present performance results of the benchmarks described in Section IV and perform more detailed analysis to explain the reasons for performance improvement or loss.

A. Overall Performance

The overall performance of all applications tested on the DAVINCI cluster running in Hadoop and HadoopCL with and without compression is shown in Figure 2. Table II shows the actual speedups achieved on DAVINCI using HadoopCL with and without compression. These measurements include overhead from all network and inter-device I/O.

On the AMD APU cluster, tests were only run without compression but were run on both 2 and 10 nodes. The results are shown in Figure 3, and actual speedup values are in Table III.

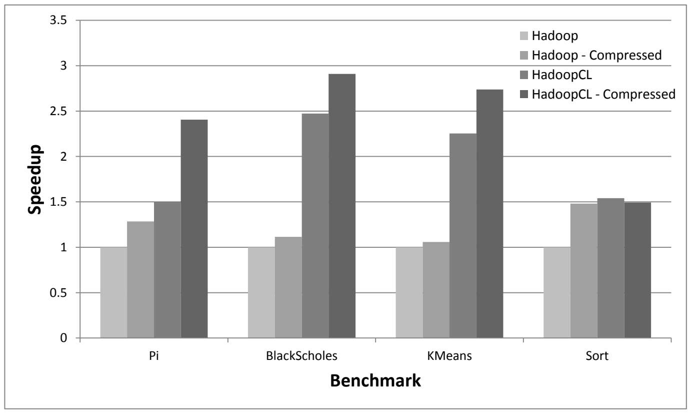


Fig. 2. Overall speedup of all benchmarks on DAVINCI with and without compression, normalized to Hadoop without compression.

We studied more detailed metrics on DAVINCI for the applications with consistently best and worst performance relative to Hadoop. All of the below tests were gathered without compression enabled.

HadoopCL demonstrated the worst relative performance on the distributed sort benchmark. We observed that removing all computation from the Sort benchmark but producing the same communication patterns led to near-identical performance in HadoopCL, and only a drop in execution time of 10-20% on Hadoop. In addition we can observe that adding compression to Hadoop Sort caused execution time to nearly match HadoopCL performance. From this, we are able to conclude that 1) the Sort benchmark is I/O bound and the benefits of improved computation are therefore negligible, and 2) as expected the overheads in HadoopCL have increased relative to Hadoop though the I/O optimization strategies seem to be effective and the cause of any Sort speedup. The increased overhead in HadoopCL is most likely caused by the added communication between discrete devices and overhead incurred from APARAPI's translation mechanism.

KMeans's performance was also investigated. Kmeans demonstrated a speedup of 2.25x without compression and 2.74x with compression on DAVINCI. The command line tools 'ps' and 'nvidia-smi' were used to collect statistics on CPU and GPU utilization in the child node for a larger input size. The collected data can be seen in Figure 4. Note that the time scales start at 4000 seconds. Prior to that, no activity was seen in the node. We infer this to mean that time was entirely spent performing I/O. Figure 4(a) shows that Hadoop is able to keep all 12 cores utilized, but we can see that it takes much longer for it to complete the work due to inefficiencies in virtual machine execution. The HadoopCL results show that the CPU and GPUs are fully utilized for much briefer periods. Executing in native threads on both devices clearly leads to the input data being processed in a much shorter time span. We can also observe 2 trailing threads in Figure 4(b), most likely dedicated I/O threads completing writes to HDFS.

B. Mapper Performance

Statistics on mapper performance in Hadoop and HadoopCL were also collected, including information on how long the

Benchmark	Improvement in proessing bandwidth
Pi	1.67x
Blackscholes	5.19x
KMeans	55.41x
Sort	1.59x

TABLE IV
PROCESSING BANDWIDTH IMPROVEMENT OF HADOOPCL MAPPERS AS A FACTOR OF HADOOP MAPPER PERFORMANCE IN MB/S.

Benchmark	Percent Time Blocked on I/O
Pi Mapper	23.28%
Pi Reducer	0.00%
Blackscholes Mapper	19.19%
Blackscholes Reducer	58.12%
KMeans Mapper	8.06%
KMeans Reducer	23.68%
Sort Mapper	59.41%
Sort Reducer	42.51%

TABLE V
MEAN PERCENT OF TOTAL EXECUTION TIME MAPPERS AND REDUCERS SPENT BLOCKED WAITING FOR BUFFERS TO BE WRITTEN TO HADOOP.

main thread spent blocked on different operations and how long the processing of an entire HDFS chunk took. Figure 5 shows a histrogram of the processing bandwidth in MB/s Hadoop and HadoopCL mappers were able to achieve while executing the Blackscholes benchmark. Please note the difference in scale on the vertical axis. Table IV shows the improvement in average processing bandwidth for all benchmarks of HadoopCL mappers as a factor of Hadoop mappers, in MB/s. This shows the significant computational performance improvement from executing in native threads on heterogeneous devices. With improved I/O performance, much higher speedups of the overall application would be achievable.

To quantify the cost of writing to the Hadoop system from HadoopCL, we recorded the time the main Java thread spent blocked on a buffer being written to the Hadoop system for HadoopCL mappers and reducers. The main thread is required to block at this point to ensure all information has been written to the Hadoop system from that buffer before the buffer is re-used for storing input. Note that this does not account for most of the I/O overhead in the overall application, as those operations are performed external to the mapper and reducer. Rather, these measurements indicate an avenue of future improvement in HadoopCL itself. Table V shows the average time each benchmark's mappers and reducers spent blocked. There is a clear relationship between the time spent blocked on I/O and a benchmark's overall performance. While KMeans mappers only spent 8.06% of their time blocked, Sort mappers spent 59.41%. If we recall from Figure 2, KMeans performed the best relative to Hadoop, while Sort was the only benchmark that demonstrated no improvement. While the focus of this work was executing MapReduce computation on heterogeneous hardware, it is clear that there must be future work in the I/O techniques used in order for the full performance benefit of heterogeneous hardware to be realized.

VI. CONCLUSION

As distributed systems become larger, more pervasive, and more heterogeneous, the experience and knowledge required to efficiently execute complex and critical applications to them has risen higher than what most application developers and domain experts are capable of. This change in high performance computing has led to a higher demand for high level distributed and heterogeneous programming models which hide hardware complexity from the user and allow tuning experts to manipulate platform configurations in order to optimize performance, energy efficiency, and reliability.

The strengths of Hadoop and OpenCL naturally complement each other. Hadoop provides a robust and proven distributed system with a MapReduce execution model and distributed file system. However, our tests show that its computational performance is lacking. OpenCL enables execution of Hadoop computation in native threads on heterogeneous high-performance, low-power architectures. APARAPI allows the seamless integration of these two prevalent programming models to provide a high performance distributed system with usability on par with Hadoop. Our experimental results support this claim on common Hadoop workloads with minimal code change required.

There are several avenues of future work available in the area of heterogeneous MapReduce. Our analysis of the benchmarks shows that while heterogeneous processors provide a significant performance benefit, overhead from I/O limits the possible gains. Not only does I/O place a high lower bound on execution time, but it also causes underutilization of available hardware as the input data bandwidth is smaller than the processing bandwidth available. Perhaps a deeper dive into the Hadoop implementation or a modified HadoopCL communication algorithm would help to minimize this overhead. Current work at AMD investigating using GPUs to efficiently compress HDFS blocks prior to network communication may help to fix this problem. This work is also related to a future collaboration between AMD and Oracle via OpenJDK, Project Sumatra. Sumatra aims to allow the Java Virtual Machines JIT (Just In Time Compiler) to generate GPU ISA code directly, this will result in optimized GPU execution and allow the JVM to target only code which seems suited to GPU offload. While our work with APARAPI within Hadoop applies only to data-parallel mapper and reducer computation, Sumatra's scope will include a number of parallel operations available in the language as well tight coupling with the JVM for optimal performance. Additional future work specific to HadoopCL would include extensions to the APARAPI runtime and translator which may decrease code changes required for porting to HadoopCL and improve performance. More test configurations of Hadoop and HadoopCL (such as multi-node or on different systems) may produce interesting results, though we expect HadoopCL to maintain a performance advantage so long as the interconnect is sufficient to prevent network communication from dominating execution time. Detailed analysis of the benefits of shared memory systems (such as the AMD APUs used in our evaluation) would also be useful.

ACKNOWLEDGMENT

The authors would like to thank Gary Frost at AMD for his work on APARAPI [2]. APARAPI was not developed as part of this work, but was invaluable in implementing it.

The authors would like to thank Yunming Zhang at Rice University for his help in collecting performance statistics in the evaluation of HadoopCL.

The authors would like to thank Sabarishyam Srivasaraju at AMD for his help in evaluating HadoopCL on the AMD APU cluster.

This work was supported in part by the Data Analysis and Visualization Cyberinfrastructure funded by NSF under grant OCI-0959097.

REFERENCES

[1] AMD, "Amd accelerated processing units." [Online]. Available: fusion.amd.com

[2] G. Frost, "Aparapi in amd developer website." [Online]. Available: http://developer.amd.com/tools/heterogeneous-computing/aparapi/

[3] O. O'Malley, "Terabyte sort on apache hadoop," 2008.

[4] S. G. Jeffrey Dean, "Mapreduce: Simplified data processing on large clusters."

[5] K. Group, "Opencl." [Online]. Available: http://www.khronos.org/opencl/

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

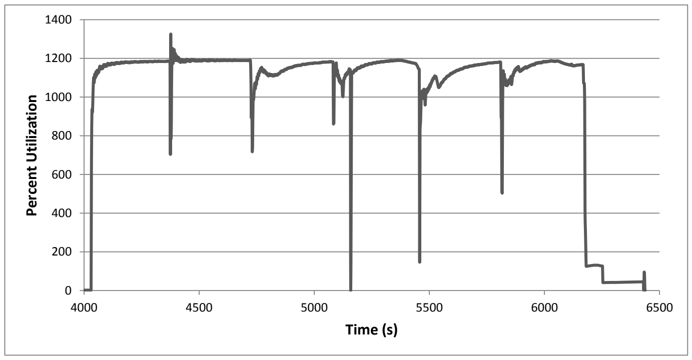
[7] B. Catanzaro, N. Sundaram, and K. K., "A map reduce framework for programming graphics processors," in *Workshop on Software Tools for MultiCore Systems*, 2008.

[8] J. Stuart and J. Owens, "Multi-gpu mapreduce on gpu clusters," in *Parallel & Distributed Processing Symposium*, 2011.

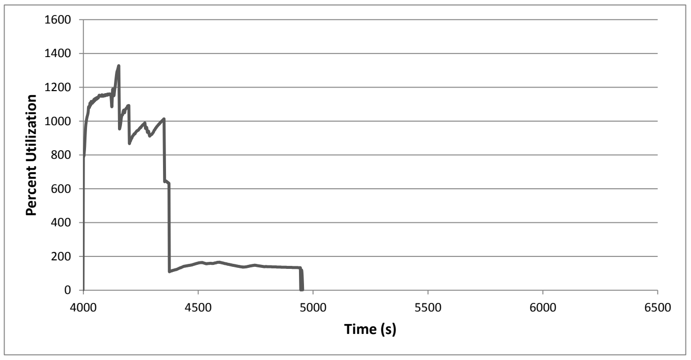
[9] W. Fang, B. He, Q. Luo, and N. Govindaraju, "Mars: Accelerating mapreduce with graphics processors," in *IEEE Transactions on Parallel and Distributed Systems Vol. 22 No. 4 pgs. 608-620*, April 2011.

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

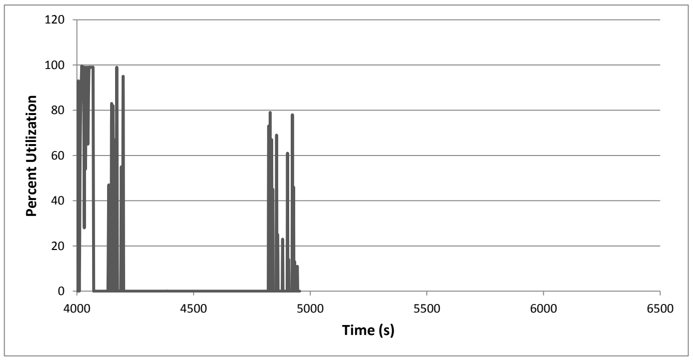
[11] Linchuan Chen, Gagan Agrawal, "Optimizing MapReduce for GPUs with Effective Shared Memory Usage." HPDC, 2012.



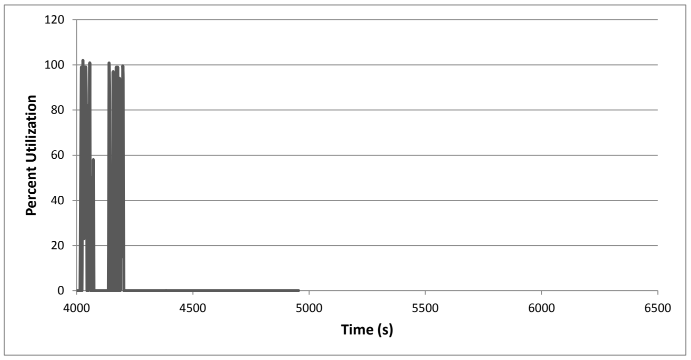
(a) Hadoop CPU



(b) HadoopCL CPU

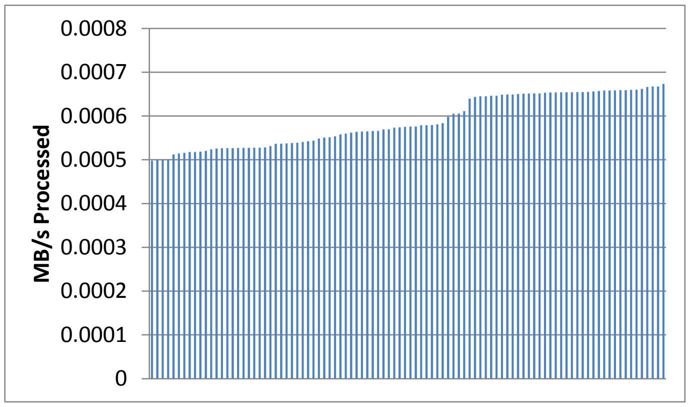


(c) HadoopCL GPU 0

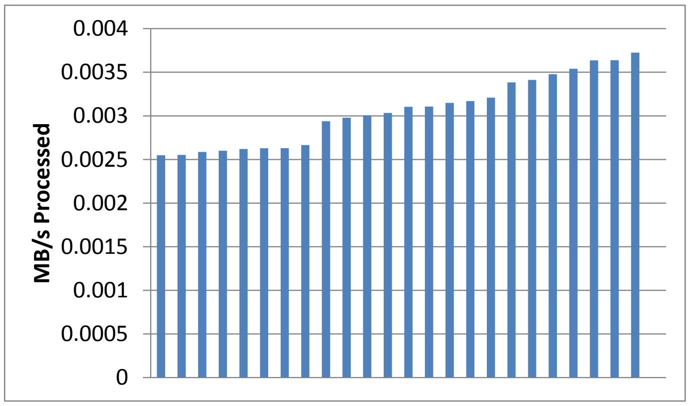


(d) HadoopCL GPU 1

Fig. 4. Processor utilization for Hadoop and HadoopCL collected using ps and nvidia-smi during a KMeans execution on 400,000,000 points. CPU utilization goes up to 1200% as it was run on a 12-core CPU. GPU utiliation only goes up to 100%. Note that the time scales on all graphs are kept the same for easier comparison.



(a) Hadoop



(b) HadoopCL

Fig. 5. Processing bandwidth of Hadoop and HadoopCL mappers in MB/s running the Blackscholes benchmark. Each bar represents a single mapper processing a single chunk. Note the difference in scale on the vertical axis.