HadoopCL2: Motivating the Design of a Distributed, Heterogeneous Programming System With Machine-Learning Applications

Max Grossman, Mauricio Breternitz, and Vivek Sarkar

Abstract—Machine learning (ML) algorithms have garnered increased interest as they demonstrate improved ability to extract meaningful trends from large, diverse, and noisy data sets. While research is advancing the state-of-the-art in ML algorithms, it is difficult to drastically improve the real-world performance of these algorithms. Porting new and existing algorithms from single-node systems to multi-node clusters, or from architecturally homogeneous systems to heterogeneous systems, is a promising optimization technique. However, performing optimized ports is challenging for domain experts who may lack experience in distributed and heterogeneous software development. This work explores how challenges in ML application development on heterogeneous, distributed systems shaped the development of the HadoopCL2 (HCL2) programming system. ML applications guide this work because they exhibit features that make application development difficult: large & diverse datasets, complex algorithms, and the need for domain-specific knowledge. The goal of this work is a general, MapReduce programming system that outperforms existing programming systems. This work evaluates the performance and portability of HCL2 against five ML applications from the Mahout ML framework on two hardware platforms. HCL2 demonstrates speedups of greater than 20x relative to Mahout for three computationally heavy algorithms and maintains minor performance improvements for two I/O bound algorithms.

Index Terms—MapReduce, heterogeneous, distributed, programming model, GPU, auto-scheduling

1 Introduction

As heterogeneous systems become more generally accessible, they continue to be applied to a wider range of computational problems. Recently, machine learning (ML) has been added as a domain whose performance can be improved with the use of heterogeneous systems. However, ML algorithms are not always as amenable to heterogeneous execution as the scientific and graphical applications that GPUs are commonly used for, and so it can be challenging for ML domain experts to build performant implementations of novel algorithms on heterogeneous hardware.

This work guides the development of a novel distributed and heterogeneous programming system through the porting of two machine learning applications. Through profile-driven and iterative optimization of the programming system's runtime, both application performance and programming model flexibility are incrementally improved. The goal of this work is not to create a ML-specific tool, but rather to create a high-performance, MapReduce programming system whose features and capabilities have been guided by the requirements of ML algorithms. This section will introduce the current state-of-the-art in ML programming systems. We will consider the well-known distributed programming model, Hadoop MapReduce [1], and an industry-standard ML library built on it, Mahout [2].

1.1 Hadoop

Hadoop is a distributed MapReduce[3] programming system. It improves on other distributed frameworks in many areas, including programmability and flexibility.

Hadoop MapReduce programming model. Hadoop's programmability is derived from its high-level MapReduce programming model and its simple, object-oriented API. From a programmer's perspective, the MapReduce programming model divides computation into two stages: map and reduce. The map stage applies a function to each of many input key-value pairs (kv-pairs), and outputs zero or more kv-pairs per input. Then, the reduce stage's kernel is applied to all map output values paired with the same key, reducing that collection of inputs to zero or more output kv-pairs. In addition to map and reduce, Hadoop also supports a combine stage that acts as an intermediate reduce and executes spatially near each map instance, reducing data movement and memory utilization as a result. The application-specific logic for each of these stages is implemented as single-threaded logic in Java classes. The workload for a Hadoop job can then be transparently mapped to multi-processor, shared-memory machines by taking advantage of the parallelism inherent in the MapReduce model.

Hadoop is flexible in terms of the applications that can execute on it and the data types it supports. This flexibility can primarily be attributed to its use of the object-oriented JVM. The ability to represent, serialize, and strongly type-check user objects is useful when building complex applications.

- M. Grossman and V. Sarkar are with the Rice University, Department of Computer Science, 6100 Main St., Houston, TX. E-mail: {max.grossman, vsarkar}@rice.edu.
- M. Breternitz is with the AMD Research, 7171 Southwest Parkway, Austin, TX. E-mail: mauricio.breternitz@amd.com.

Manuscript received 18 July 2014; revised 9 Mar. 2015; accepted 11 Mar. 2015. Date of publication 19 Mar. 2015; date of current version 12 Feb. 2016. Recommended for acceptance by A. Dubey.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2015.2414943

1045-9219 © 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: b-on: Universidade de Lisboa Reitoria. Downloaded on May 13,2020 at 15:32:37 UTC from IEEE Xplore. Restrictions apply.

1.2 Mahout

Mahout is an open-source collection of ML applications implemented in sequential Java and Hadoop MapReduce. Mahout includes clustering, classification, and recommendation frameworks which operate on general-purpose sparse vectors and hence support application to a variety of fields. Mahout builds on the flexibility and programmability of Hadoop when applied specifically to ML applications by supporting plug-and-play of new algorithms in any of these frameworks.

Mahout facilitates the porting of complex applications to distributed systems by domain experts. Because Mahout is based on the Hadoop MapReduce programming system, it allows domain experts to implement their applications as simple, single-threaded Java code but execute them on massively parallel clusters. By motivating this work with applications from Mahout and using the Wikipedia data set in our evaluation, we ensure that the programming system that results is flexible enough to support real-world applications and real-world datasets.

2 Motivation

While the benefits of Hadoop and Mahout are clear, their limitations are more nuanced and generally focused in one area: performance. To be clear, raw performance is not the objective of Hadoop and Mahout; they were designed as highly-programmable frameworks for software development. However, as they become more popular and are applied to domains with tighter QoS requirements, the performance deficits in Hadoop become more of a problem for users. This section goes into detail on the aspects of Hadoop and Mahout which cause suboptimal performance.

2.1 Performance Loss in Hadoop and Mahout

Running in the JVM introduces overheads from JVM initialization, garbage collection, JIT compilation, class loading, I/O abstractions, and many layers of object abstraction. These problems are exacerbated by Hadoop's extensive use of separate, short-lived JVMs for process isolation.

Serialization and deserialization of arbitrary data types is also a large source of overhead in Hadoop. Key-value pairs are frequently serialized, even when passing them around within the same JVM. While this work focuses on improving computational performance, it is important to also consider I/O optimizations or risk becoming I/O bound.

Hadoop and Mahout largely ignore the problem of memory management by leaning on JVM garbage collection to do well enough while statically dividing system memory between processes. However, it is often the case that either 1) over-utilization leads to garbage collection and/or swapping, or 2) underutilization leaves performance on the table. In both cases, naive memory management leads to performance loss.

2.2 Balancing Programmability and Performance

If the performance of Hadoop and Mahout is to be improved, it should not come at the cost of the programmability of the MapReduce programming model. To prevent this, we studied the requirements of two Mahout applications and used the insights gained to guide this work. First, we use the KMeans clustering job as a positive test case to verify that for embarassingly parallel and computationally heavy applications we achieve a significant performance improvement. Second, we use the Pairwise Similarity job (from the Mahout recommender pipeline) to ensure there are not performance regressions for applications which are more I/O bound and demonstrate more complex control flow.

In studying these applications, we identified several features that are required for a new programming system to remain relevant to real-world applications:

1) Efficient execution of computationally diverse kernels and support for arbitrary control flow
2) A high level and familiar programming model and language
3) Complex data types (sparse vectors, composites, etc.)
4) Accurate resource management techniques which prevent under- or over-utilization
5) Dynamic memory allocation
6) Support for large, out-of-core data sets
7) Globally accessible readable and writable data structures
8) Easy-to-use tools that make correctness and performance errors easier to diagnose and fix

Section 3 will go into further detail on how each of these features is supported in HCL2.

This work extends the Hadoop MapReduce programming system to support execution on multiple architectures in a single job. The end goal of this extension is to improve the computational performance of MapReduce applications. This work also modifies Hadoop's I/O subsystems to prevent accelerated applications from becoming immediately I/O-bound. Conducting the development of this programming system in parallel with the port of KMeans and Pairwise ensures that any changes made for the sake of performance do not inhibit the generality of the resulting programming model or its applicability to real-world MapReduce applications.

This paper makes contributions in the following areas:

1) Task management techniques for multi-process, heterogeneous systems including automatic scheduling of computational tasks across a distributed, heterogeneous system.
2) Memory management techniques for multi-process, heterogeneous systems including dynamic, concurrent memory allocation and garbage collection in OpenCL kernels.
3) Compilation techniques for transformation and optimization of heterogeneous kernels.
4) Co-development of the profiling and debugging tools of a programming framework with the framework itself.

Section 3 covers the API exposed to programmers and techniques used at runtime to efficiently schedule a distributed Hadoop MapReduce job on heterogeneous processors. Section 4 evaluates the effectiveness of this work on five machine-learning applications from the Mahout framework. Section 5 discusses related work. Finally, Section 6 draws conclusions from this work and summarizes its contributions.

Authorized licensed use limited to: b-on: Universidade de Lisboa Reitoria. Downloaded on May 13,2020 at 15:32:37 UTC from IEEE Xplore. Restrictions apply.

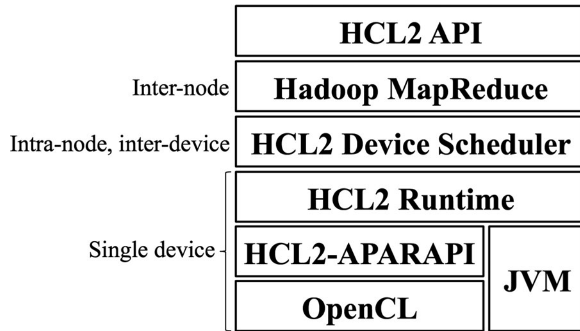

Fig. 1. The HCL2 software stack.

3 Methods

Past work at Rice University [4] developed a prototype distributed, heterogeneous programming system which accelerated Hadoop mappers and reducers using OpenCL. This work, called HadoopCL, translated JVM bytecode to OpenCL kernels at runtime and used the Java Native Interface (JNI) to transfer data between the JVM and OpenCL address spaces and execute OpenCL kernels. HadoopCL was a first step towards accelerating MapReduce computation but remained too simple a framework to efficiently tackle the real-world, irregular, and dynamic ML applications in Mahout, or the data sets they run on. In particular, HadoopCL failed to meet criteria 3, 4, 5, 7, and 8 listed in Section 2.2:

1) HadoopCL only supported primitive key and value types for inputs and outputs.
2) HadoopCL required that the programmer statically specify the OpenCL devices to run all mappers and reducers on.
3) HadoopCL had no support for dynamic memory allocation, the number of output kv-pairs per input kv-pair was statically defined so that sufficient memory could be pre-allocated for all possible outputs.
4) HadoopCL only supported read-only globally shared data structures.
5) HadoopCL offered no native tools for profiling performance and debugging errors in user code.

The work presented in this paper benefits from the experiences gained implementing HadoopCL. However, the design, architecture, and implementation of this work all started from a clean slate. Assumptions made during the implementation of HadoopCL meant that a major code refactoring would have been required for the existing HadoopCL code base to support all of the requirements listed in Section 2.2. Rather than handicap ourselves from the start, we chose to discard the existing technical debt and use the lessons learned to build a more flexible and featureful system.

For the remainder of this paper, the past work will be referred to as HadoopCL (HCL) and this work as HadoopCL2 (HCL2).

HCL2 supports a more general class of applications than HCL while still executing Hadoop mappers and reducers in native threads on heterogeneous hardware. This section presents HCL2's API and software stack (depicted in Fig. 1) using an illustrative example, starting at the top layer and working down.

3.1 An Illustrative Example

Calculating π through random sampling is a common example used to illustrate MapReduce programming models. By taking random samples inside the unit square on the xy-axis, π can be estimated using the following equation:

$$\pi = 4 * m/(m + n),$$

where m is the number of samples that satisfy $x^2 + y^2 \leq 1$ and n is the number of samples that do not.

This example can be expressed as a MapReduce computation. The map stage takes the x and y coordinates of a single sample as input. It determines whether that point is inside the unit circle. If the point is inside the unit circle, it outputs a kv-pair of `(true, 1)`. Otherwise, it outputs `(false, 1)`. The reduce stage will only execute twice: once for the `true` key and once for the `false` key. The reduce stage sums the values for each key, outputting the input key paired with the accumulated sum of its input values. Therefore, the final output will be two kv-pairs: `(true, m)` and `(false, n)`. From these values, π can be computed.

For the remainder of our methods discussion, the Pi example will be used to concretely illustrate how a MapReduce application is built in HCL2 and scheduled on a distributed, heterogeneous system.

3.2 HCL2 API

We start with a description of the HCL2 API presented to the user. Later sections will discuss the lower layers of the HCL2 software stack that are responsible for scheduling programs defined with this API.

The guiding principle of the HCL2 API was to retain as much similarity to Hadoop MapReduce as possible. As a result, HCL2 applications are developed entirely in the Java programming language and compiled into JARs, like Hadoop applications. Similar to Hadoop, the map, combine, and reduce stages are each defined by Java classes which extend type-specific Mapper, Combiner, and Reducer superclasses. Listed below are example HCL2 mapper and reducer implementations of the Pi example introduced in Section 3.1:

```
public class PiMapper extends
  IntPairBooleanLongMapper {
 void map(int pid, double valx, double valy) {
  double distance_squared =
   Math.pow(valx, 2) + Math.pow(valy, 2);
  if (distance_squared <= 1) {
   write(true, 1);
  } else {
   write(false, 1);
  }
 }
}
public class PiReducer extends
  BooleanLongBooleanLongReducer {
 void reduce(boolean inside,
   HadoopCLLongValueIterator values) {
  long count = 0;
  do {
   count += values.get();
  } while (values.next());
  write(inside, count);
 }
}
```

Authorized licensed use limited to: b-on: Universidade de Lisboa Reitoria. Downloaded on May 13,2020 at 15:32:37 UTC from IEEE Xplore. Restrictions apply.

The types in each mapper and reducer superclass name (e.g., `IntPairBooleanLongMapper`) indicate the input and output key and value types for that computation. These superclasses are auto-generated for a range of primitive, composite, and sparse vector data types. For example, `PiMapper` takes a kv-pair of (`int`, `Pair`) as input and outputs a kv-pair of (`boolean`, `long`). The `Pair` type is an HCL2-supported composite type containing two double-precision floating-point values.

HCL2 supports globally shared read-write sparse vectors within a Hadoop job. Many Mahout applications exhibit a pattern of 1) initialize global data on task setup, 2) read and modify global data at each kv-pair, and 3) if global data was modified then write those modifications to the Hadoop Distributed Filesystem (HDFS) on task cleanup. Therefore, to support the target ML applications used to guide this work, HCL2 exposes a simple API for interacting with global sparse vectors.

The API for initializing global sparse vectors is straightforward. During job initialization, a sparse vector Java object is passed to HCL2 with flags indicating if it is writable and a unique ID to identify that global vector.

These globals are made accessible to the application code through the API below. Each global sparse vector is keyed by its unique integer ID. The dimensions, values, and length for that sparse vector can be fetched using that ID.

```
int[] getGlobalIndices(int GID);
double[] getGlobalVals(int GID);
int getGlobalLength(int GID);
```

Utility functions are also provided for quick lookup, and for manipulation of elements in the global vectors using supported mathematical operations (e.g., increment).

These user-initialized global sparse vectors are stored in HDFS files so that they can be accessed from inside the Hadoop job. Section 3.5 provides more details on how these global data structures are managed at runtime and made accessible to user computation inside a job.

3.3 Hadoop MapReduce

Once application-specific mappers and reducers have been implemented in Java and compiled to JARs, a Hadoop job can be created from that Hadoop application and scheduled on a distributed system using Hadoop MapReduce. The work described by this paper does not alter the inter-node MapReduce scheduler provided by Hadoop, but it is briefly described here for continuity.

A Hadoop job is an instance of a Hadoop application containing a map, a reduce, and an optional combine stage that is executed on user-specified input. The map, reduce, and combine stages of a Hadoop job are split into many parallel tasks. Each of these Hadoop tasks iterates over the input data assigned to it and applies the user-provided map or reduce function to each input kv-pair.

Hadoop TaskTrackers in each node pull tasks from a centralized Hadoop JobManager. The JobManager manages the tasks that make up each job, tracking which tasks are eligible for execution. Each TaskTracker manages a constant number of task slots in a node. A single slot generally maps to a single CPU core. The TaskTracker greedily pulls work from the JobManager as slots become available. Each task is executed in a Child JVM running as a separate process. In this way, Hadoop jobs that have been split into tasks can be scheduled across a distributed, homogeneous system and use all available CPU cores.

For our Pi example, the input is a dataset of randomly generated two-dimensional points. This dataset would be partitioned, and a single partition assigned to each mapper task. Each partition would be processed by a different task running inside a Child process. Once all mappers had completed, the reduce stage tasks would be scheduled. Because Pi uses a boolean key for the reduce stage, there are only two reduce keys. Hadoop MapReduce would likely create one task for each, though this depends on Hadoop tunables. These tasks would again be executed by Child JVM processes. The final output would be persisted in HDFS.

3.4 HCL2 Device Scheduler

As described in Section 3.3, the Hadoop TaskTracker schedules tasks into task slots at the intra-node level. In general, each task slot maps to a single CPU core. In HCL2, the TaskTracker must also assign each task a device to use.

HCL2 supports three types of HCL2 "devices": native OpenCL threads on GPUs, native OpenCL threads on CPUs, and execution in the JVM. HCL2 is sufficiently flexible to support additional OpenCL architectures as they become available.

While each task only uses a single device, multiple tasks may be assigned to the same device. Running multiple tasks simultaneously on the same device keeps device utilization high even when some tasks are blocked on I/O, at the cost of potentially increased overhead from context switching and resource contention. Note that there is no direct relationship between Hadoop slots and HCL2 devices; the slot a task is placed in does not affect the device it is assigned.

Deciding which device to assign to a given task is a complex problem. While the performance tradeoffs between OpenCL CPU and GPU devices are well understood [5], [6], the tradeoffs between OpenCL and JVM execution are more interesting. Using the JVM eliminates the need to perform transfers into a separate address space (as is necessary in OpenCL). Depending on the characteristics of the data in an application, using the JVM may also offer memory and I/O benefits. OpenCL's batched execution model requires buffering of many input data points. This increases the working set size of HCL2 tasks and may produce bursty I/O. However, batching reads in HCL2 also enables optimizations in the runtime that can hide I/O overhead.

The simplest approach to device selection that HCL2 supports is to use manual, hard-coded programmer hints to select different devices for different tasks.

HCL2 also has an internal auto-scheduling framework which, when enabled, constructs a relationship between the computational load on a HCL2 device and a task's expected execution time. By learning from past executions of tasks of the same type, persisting this information across jobs, and using low overhead techniques to construct this relationship, the auto-scheduling framework can match or beat the performance of manual, programmer-defined scheduling. The following sections will discuss the techniques used by the auto-scheduler in more detail.

Authorized licensed use limited to: b-on: Universidade de Lisboa Reitoria. Downloaded on May 13,2020 at 15:32:37 UTC from IEEE Xplore. Restrictions apply.

Measuring and storing past performance. HCL2 stores per-device historical performance information for every task type. This historical information is used to characterize task performance on each device in a platform. A task type corresponds to a single mapper, combiner, or reducer class.

For every possible `(task-type, device-type)` tuple, the HCL2 auto-scheduler stores a list of past performance data points. Each element in this list includes two things: the computational bandwidth achieved by an instance of this task type in kv-pairs/ms, and an estimate of the average load on all devices in the system during the execution of that task.

Computational bandwidth is measured differently for OpenCL and JVM devices. For both, it is straightforward to measure the number of kv-pairs processed in each task by incrementing a counter for each kv-pair read from the input of a task. To calculate the time taken to process those kv-pairs on OpenCL CPU and GPU devices, a millisecond-granularity timestamp is taken at the start and end of every kernel launch.

Because JVM execution is not batched and each kv-pair is processed individually, placing timing statements around every call to a map or reduce function would significantly add to the overhead of the HCL2 runtime when auto-scheduling on JVM devices. Instead, we can only time the overall task. While this technique induces less overhead than the technique for OpenCL devices due to fewer timing statements, it also strictly underestimates the computational bandwidth of the JVM as other operations, such as I/O, are included in the elapsed time measured.

The estimated average load during a task's execution is measured in units of tasks running per device, and is calculated as the mean of the device load at the start of the task and at the end of the task.

The computational bandwidth of a task is calculated by the task itself and communicated to the TaskTracker when the task completes successfully. The TaskTracker writes these metrics to a local file immediately so that they can be reloaded on startup if the TaskTracker is shut down. The TaskTracker also passes these metrics to the auto-scheduling framework so that task characterizations can be constructed or revised.

Task characterization. In the HCL2 auto-scheduler, task characterizations are created for each task type. Task characterizations use the historical data described in Section 3.4 to predict the performance of instances of that task type on each device in a platform, and provide a confidence measure for that performance prediction. Each HCL2 task characterization constructs an internal function:

$$f(D, L) \rightarrow R$$

from device type D and current device load L to expected execution rate R where R is measured in kv-pairs per millisecond.

f has a different shape (e.g., linear, exponential, etc.) for different device types. The function for each device type was chosen experimentally using performance data from manually scheduled runs. We plotted the performance of different devices running KMeans against device load and looked for trends in the data. Based on the trends we observed (discussed below), we chose a function shape to fit to the data for each device.

For JVM and OpenCL CPU devices, we found there was a clearly linear relationship between device load and task processing rate. Therefore, we use linear regression to construct a linear function from device load to task processing rate. Related works [7], [8] have also used linear relationships to predict CPU performance. Generating f has a computational complexity of $O(C^2N)$, where C is the number of features and N is the number of data points. For OpenCL CPU devices, we only consider the load on that device so C is equal to one. For JVM devices, we consider the load on all devices in the system so C may be greater than one.

As an example, the function constructed for OpenCL CPU devices running the Pairwise mapper was $f(OpenCL\ CPU, L) = 26.67 - 1.17L$. This relationship indicates that adding more load to an OpenCL CPU device causes the expected execution rate for all tasks on that device to drop.

For OpenCL GPU devices, there was no clear relationship between device load and task performance. Rather, we observed two clusters of performance: a small cluster of slow executions caused by initialization overheads, and a larger cluster of higher-performing executions. We chose to use K-nearest neighbors to estimate task performance on GPUs. This approach predicts task performance using the mean of the K performance measurements most similar to the current one in terms of device load. K-nearest neighbors naturally disregards outliers.

The computational complexity of predicting the performance of a given task on a given GPU using K-nearest neighbors is $O(N)$ where N is the number of past performance data points. For GPUs, only the load on the GPU in question is considered as an input when predicting performance.

Before predicting task performance, a task characterization must first report if it has sufficient historical performance data on which to base a prediction. In our implementation, a task characterization is "confident" it can make an accurate performance prediction if there are any similar past executions in its historical performance data. A past execution is similar if it is for the same device and executed at a similar device load. We use an n-dimensional Euclidean distance measure to determine similarity between device loads, where n is the number of devices in a platform. Any device loads within a constant, experimentally-chosen radius of the current device load are considered similar.

Types of scheduling decisions. There are two types of scheduling decisions in HCL2: speculative and performant.

Speculative scheduling decisions are made to fill in gaps in a task characterization's knowledge of a particular device's performance. A speculative scheduling decision is made when a task characterization indicates it has no confidence in its performance predictions for a device at the current device load. By scheduling the current task on the no-confidence device, a performance data point is added to that task characterization, allowing it to better predict performance for future task executions. Having a well-defined model of each task's performance on each device is important in making accurate and well-performing scheduling decisions.

Authorized licensed use limited to: b-on: Universidade de Lisboa Reitoria. Downloaded on May 13,2020 at 15:32:37 UTC from IEEE Xplore. Restrictions apply.

While speculative scheduling may lead to suboptimal task placement, any short-term performance gains that are lost are outweighed by the long-term benefits of well-characterized performance. As a result, there is generally a period of suboptimal scheduling and performance during the early executions of a new task type. Section 4 will characterize this further.

Performant scheduling decisions are made to achieve maximum performance for a task, given the available HCL2 devices in a platform and the task characterization constructed from past executions on those devices.

Auto-scheduler core. The core of the HCL2 auto-scheduler resides in the TaskTracker and is responsible for:

1) Keeping track of the current device load in a node, measured in tasks executing per device.
2) Selecting a device for each task based on the current load in the node and the known task characterizations (described in Section 3.4).
3) Communicating the selected device to the task.

The current load for all devices is stored in an integer array, with one entry for each device. The auto-scheduler increments the load for a device when a new task is assigned to it. The auto-scheduler also maintains a mapping from executing tasks to the device each is running on. When a task signals the TaskTracker that it completed successfully, the TaskTracker signals the auto-scheduler to remove that task from its accounting. The auto-scheduler uses the task-to-device mapping to decrement the appropriate device load.

In our implementation, the device for a given task is selected by first querying the "confidence" level of its task characterization for each device in the current platform. If the task characterization has no confidence for one or more devices at the current device loads, this task is speculatively scheduled on a randomly selected no-confidence device. Otherwise, a performance prediction for each device is made. The task is assigned the device with the highest predicted performance.

Once a task has been assigned a device, a device ID is passed to the Child process running that task as a Java environment variable. This environment variable is read by the HCL2 Runtime (discussed in Section 3.5) and work is only scheduled on the selected device.

Static scheduler. In addition to the manual programmer-controlled scheduler and the auto-scheduler, HCL2 supports a third Device Scheduler: the static scheduler. The static scheduler uses the task performance profiles generated by the auto-scheduler to make scheduling decisions, but does not update those performance profiles. The static scheduler avoids the computational overheads incurred when performance profiles are updated with new performance data.

3.5 HCL2 Runtime

Once a device has been assigned to a task by the Device Scheduler, the Hadoop TaskTracker launches a separate Hadoop Child process. That Child process is responsible for processing the input assigned to the task using the device assigned to it. Hadoop uses a separate process for each task to improve component isolation and system robustness. Within the Hadoop Child process, the HCL2 Runtime is responsible for scheduling execution of user-defined computation on the device assigned to this task, as well as handling any necessary communication or management work.

During initialization of the Child process, the HCL2 Runtime loads the global sparse vectors described in Section 3.2 from HDFS. If this Child is assigned the JVM device, then no further action is necessary as the globals are now in the JVM's address space. If this child is assigned an OpenCL device, OpenCL buffers are pre-allocated and initialized with these global values before processing begins.

When using the JVM device, the HCL2 Runtime mirrors the workflow of a normal Hadoop Child process. It iterates single-threaded over the input kv-pairs, calls the user-defined map or reduce function on each, and outputs kv-pairs one at a time. The work described in this paper does not significantly change this process.

When running on an OpenCL device, the HCL2 runtime chunks input and output data points into data buffers. The HCL2 runtime follows the following steps to process these data buffers on an OpenCL device:

1) The bytecode loaded for this task's `map()` or `reduce()` function is translated to an OpenCL kernel. This step will be discussed in more detail in Section 3.6.
2) A dedicated input I/O thread, called the Input Aggregator, buffers many kv-pairs from the input stream for this task in to a data buffer `D`.
3) Once it is full, `D` is passed to the Buffer Executor, a separate thread which allocates memory on the OpenCL device assigned to this task, transfers the inputs contained in `D` to the allocated OpenCL memory buffers, and launches the OpenCL kernel created in step 1.
4) The Buffer Executor detects the completion of processing for `D`, transfers its outputs back to the JVM, and passes them to a dedicated I/O thread, the Output Writer, to be written out.
5) If there are inputs left to process, control loops back to step 1. Otherwise, this task terminates.

Note that while these steps are described sequentially, most of the actual processing of a data buffer `D` is asynchronous and does not require a component (e.g., Input Aggregator) to block on `D` completing unless the storage or compute resources required for forward progress by a component have been exhausted. Examples of resources that may cause blocking are OpenCL memory buffers, pre-allocated JVM buffers, or the OpenCL device.

HCL2 runtime memory management. The HCL2 Runtime explicitly pre-allocates and manages both OpenCL and JVM memory buffers. This memory management was implemented to limit dynamic allocations on the JVM's heap and on OpenCL devices so as to prevent operating system and JVM out-of-memory errors, as well as limit overhead from excessive JVM garbage collection. HCL2 has the added complexity of two entities competing for heap allocations within a single process: the JVM memory manager and the OpenCL runtime.

There are three types of buffers used in the HCL2 runtime, each used for a different stage of processing. Each

Authorized licensed use limited to: b-on: Universidade de Lisboa Reitoria. Downloaded on May 13,2020 at 15:32:37 UTC from IEEE Xplore. Restrictions apply.

buffer type has a fixed number of buffer instances that can be instantiated at any time. Each buffer instance is either 1) owned by the component of the HCL2 runtime which is currently operating on it, 2) stored temporarily in a queue of pending work, or 3) stored in a pool of free, pre-allocated buffer instances which are not in active use. The three buffer types are described in detail below:

1) *Input Buffer*: An Input Buffer is used to store input data in the JVM. These buffers are allocated by the Input Aggregator, filled with input data from the current task's input stream, and passed to the Buffer Executor once full. Input Buffers are released by the Buffer Executor after their contents have been transferred from the JVM to an OpenCL device. Input Buffers encapsulate primitive Java arrays which store Java objects in a format that OpenCL can process.
2) *Output Buffer*: An Output Buffer is used in the JVM to store data output by an OpenCL kernel. Like Input Buffers, Output Buffers store Java objects as primitive arrays. Output Buffers are allocated by the Buffer Executor and transferred to directly from the OpenCL device. Once the OpenCL outputs have been pulled from the OpenCL address space into an Output Buffer, the contents of these buffers are written to the next stage in the MapReduce pipeline and released by the Buffer Executor.
3) *Kernel Buffer*: A Kernel Buffer is a JVM object that consumes minimal JVM memory but is associated with a set of OpenCL buffers in the OpenCL address space. Each Kernel Buffer acts as a handle that HCL2 runtime components must acquire to access the associated OpenCL buffers. The Buffer Executor allocates a Kernel Buffer from a pre-allocated pool and transfers the contents of an Input Buffer to the OpenCL buffers associated with that Kernel Buffer. That same Kernel Buffer is held by the Buffer Executor for the lifetime of the OpenCL kernel processing its contents.

These data buffers are also used to circumvent Hadoop serialization and deserialization overheads. Since objects are already stored as primitive arrays in Output Buffers, we output these primitive values directly with no serialization overhead.

3.5.1 Scheduling Pi on the HCL2 Runtime

When scheduling Pi on the HCL2 Runtime in a single Hadoop Child process, each data buffer `D` will be specialized to match the input and output types of the Pi mappers and reducers. For example, consider the Pi mapper, which takes as input an integer key and a pair of `doubles` as its value. When stored in an Input Buffer, these inputs would be serialized into:

```
int[] inputKeys;
double[] inputVals1;
double[] inputVals2;
```

Other than the types stored in data buffers, the HCL2 Runtime workflow described above is identical for all tasks.

3.6 HCL2-APARAPI

The HCL2 Runtime described in Section 3.5 is responsible for scheduling OpenCL computation within a single Child process. However, Section 3.2 explained that HCL2 exposes a Java API, and that HCL2 applications are compiled to JVM bytecode and stored in a JAR. To support executing the map, reduce, and combine computation defined by this JVM bytecode in both the JVM and on OpenCL devices, HCL2 must automatically generate OpenCL kernels from JVM bytecode.

HCL2 uses a modified version of the open-source APARAPI framework [9] to translate each mapper, combiner, and reducer's definition from JVM bytecode into OpenCL kernels. APARAPI is an open source, general-purpose framework that enables transparent execution of Java programs on OpenCL devices through an API similar to Java's `Runnable`. APARAPI includes a runtime translator from JVM bytecode to OpenCL kernels and handles all OpenCL memory allocation, data transfer, and kernel execution for the programmer. HCL2 uses a modified version of APARAPI's bytecode translator, but adds its own HCL2-specific support for performing all other OpenCL management operations. HCL2 extends APARAPI to support dynamic memory allocation in OpenCL, which is discussed briefly in Section 3.6.

In HCL2, APARAPI's bytecode translator is applied to the bytecode loaded for a mapper, combiner, or reducer class during initialization of an HCL2 task, generating a `String` object containing OpenCL kernel code. This `String` is passed from the JVM to native code through JNI and compiled into an OpenCL program—the generated OpenCL program can then be executed.

An example of the auto-generated code for the Pi example from Section 3.1 is included below:

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable

  void Pi$PiMapper__map(This *this, int pid,
    double valx, double valy) {
   double distance_squared = pow(valx, 2.0) +
    pow(valy, 2.0);
   if (distance_squared <= 1) {
    __write(this, 1, 1);
   } else {
    __write(this, 0, 1);
   }
  }
```

Concurrent dynamic memory allocation in OpenCL. HCL2-APARAPI supports dynamic memory allocation inside OpenCL kernels. HCL2-APARAPI's dynamic memory allocator uses a shared heap of OpenCL memory split between free and allocated partitions by an atomically incremented pointer. This pointer always points to the lowest unallocated memory and is initialized to the base address of the shared heap. A thread requests memory by incrementing the pointer to a higher memory address. If the address incremented to is below the limit of the heap, then that allocation is successful and the thread is guaranteed exclusive access to that range of memory.

HCL2's dynamic memory allocator includes a thread-local garbage collector that can reclaim temporary allocations which are not written as final output. These reclaimed

Authorized licensed use limited to: b-on: Universidade de Lisboa Reitoria. Downloaded on May 13,2020 at 15:32:37 UTC from IEEE Xplore. Restrictions apply.

memory segments can then be re-used to satisfy future allocations from the same thread.

Every allocation has a header storing:

1) The size of the allocation, in bytes.
2) The address of the previous allocation performed by the current thread. Collectively, these pointers form a linked list of past allocations performed by each thread.
3) A flag that is initialized to `false` but is set to `true` when this allocation is written as output from a mapper/reducer.

Once processing of a kv-pair completes, all temporary allocations which were not written as final output can be identified and reclaimed into a thread-local list of free memory ranges by traversing the past allocations list and searching for allocations which were not marked as final output. Future memory allocations by that thread then check this free list before attempting an allocation on the shared heap.

This technique permits allocations to be performed without an expensive atomic increment on the heap's free memory pointer and improves memory efficiency by increasing the percentage of OpenCL memory used to store non-temporary values.

If memory allocation by a thread fails then that fault is handled by aborting processing of the current input kv-pair and marking it incomplete. The completeness of each input kv-pair is checked by the JVM on kernel completion. Any outputs written for an incomplete kv-pair before the fault are not stored as final output in an HCL2 Output Buffer. The incomplete input element is then marked for retry. Because the OpenCL buffers associated with a Kernel Buffer are isolated in the OpenCL address space and the Buffer Executor does not release a Kernel Buffer until processing of all elements completes, all input kv-pairs that require a retry are guaranteed to still be on the device. An OpenCL kernel can immediately be relaunched to retry kv-pairs which faulted.

Dynamic memory allocation is currently exposed to the programmer as method calls (e.g. `int[] allocInt(int)`). In future work, HCL2's programmability could be improved by replacing the JVM `NEW` bytecode instruction during bytecode-to-kernel translation.

3.7 Native Profiling and Debugging Tools

Both distributed and heterogeneous development make profiling and debugging more difficult. Combining the two into a single programming system magnifies the opacity of system state. This section will briefly cover work supporting native profiling and debugging tools in HCL2.

HCL2 profiler. At the core of the HCL2 Profiler (HCL2P) is a runtime logging component that logs timestamps of important events in both JVM and native execution. For example, timestamps are written at the start of input aggregation, at each OpenCL kernel launch, and for each event in the OpenCL profiling API. By correlating JVM and OpenCL timestamps within a node, fine-grain profiling is achieved. HCL2 does not currently support correlating these events across multiple nodes.

Following job completion, the timestamp logs can be fetched, post-processed, and visualized using a standalone tool. This produces a visual timeline of input aggregation, output dumping, kernel processing, thread blocking, and other important HCL2-specific states.

3.7.1 Data Buffer Debugger

The HCL2 Debugger (HCL2D) uses a similar workflow of runtime logging and post-processing. HCL2D uses runtime checkpointing of kernel inputs to enable offline debugging of the correctness and performance of HCL2 OpenCL kernels.

On each kernel launch, a snapshot is taken of OpenCL buffer contents and sizes, kernel source code, and any other state necessary to fully recreate the same kernel launch. This snapshot is written to a dump file on disk, ensuring that the saved state persists even if the process writing it crashes.

The buffers which must be saved are determined based on directional information (`IN`, `OUT`, `INOUT`) passed down from higher HCL2 layers. Only the contents of `IN` and `INOUT` buffers are written to the dump file. All buffers have their dimensionality saved.

Upon successful completion of the associated kernel launch, the dump file is deleted from disk to prevent out-of-space errors in storage-constrained systems.

Once Hadoop job execution completes or fails, any dump files remaining on disk must be associated with kernel launches that either failed or were in-progress at job termination. A utility was implemented to parse the generated dump files and perform an identical re-execution based on their contents. This offline execution can be inspected using other debugging tools, from simple prints to `gdb`.

3.8 Summary

In this section we summarized the HCL2 software stack and toolset, including:

1) The Java API exposed to programmers for writing HCL2 mappers, reducers, and combiners.
2) The Hadoop MapReduce framework used to distribute tasks to multiple nodes.
3) The Device Scheduler used within each node to assign a device to each task, and the auto-scheduler it uses to make these scheduling decisions automatically.
4) The HCL2 Runtime used within each task to schedule work on a single OpenCL device.
5) The modifications made to APARAPI to support dynamic memory allocation.
6) HCL2's native debugging and profiling tools.

In the next section, we will evaluate these different components for performance and usability.

4 Experimental Evaluation

While HCL2 was implemented based on analysis of KMeans and Pairwise Similarity, it is important to validate that the resulting system also performs well on other applications and across a range of platforms.

HCL2 performance was evaluated across five benchmarks from the Mahout framework: KMeans, Pairwise Similarity, Fuzzy KMeans, Dirichlet Clustering, and the Naive

Authorized licensed use limited to: b-on: Universidade de Lisboa Reitoria. Downloaded on May 13,2020 at 15:32:37 UTC from IEEE Xplore. Restrictions apply.

TABLE 1
Overall Execution Time and Speedup of Manually Scheduled HCL2 Jobs Relative to Mahout Using one NameNode and One DataNode

Benchmark	Platform A		
	Mahout	HCL2	Speedup
KMeans	1374211.9	182185.4	7.54x
Fuzzy KMeans	1532909.0	129085.0	11.88x
Dirichlet	818331.7	178464.3	4.59x
Pairwise	186010.7	110931.8	1.68x
Bayes	137143.3	125929.3	1.09x
	Platform B		
KMeans	3579745.4	565304.3	6.33x
Fuzzy KMeans	4208923.5	197390.0	21.32x
Dirichlet	1388293.1	146072.3	9.50x
Pairwise	353348.2	187261.4	1.89x
Bayes	215068.4	144089.5	1.49x

Bayes Trainer. Tests were performed on two different hardware platforms. In Platform A, each node contains a 12-core 2.80 GHz Intel X5660 CPU, two discrete NVIDIA M2050 GPUs each with ~ 2.5 GB of global memory, and 48 GB of system RAM. All nodes in Platform A are connected by Infiniband, with a peak bandwidth of 40 Gb/s. For Platform B, each node contains a single AMD A10-7850K APU which includes four CPU cores running at 3.7 GHz, a Radeon GPU with ~ 2 GB of global memory, and ~ 14.5 GB of system memory. All nodes in Platform B are connected by 1 Gb Ethernet. Both platforms use Java 1.7.0.

The following sections will go into detail on the performance of manually and automatically scheduled HCL2 relative to the original Mahout implementations of each benchmark. Code snippets from the HCL2 implementation of an example application, Fuzzy KMeans, will be used to subjectively evaluate HCL2 programmability. All performance evaluation is done using a subset of the Wikipedia dataset.

4.1 Performance Evaluation

Table 1 lists the overall execution times achieved by both platforms on all benchmarks for both Mahout and HCL2. These experiments are performed with one Hadoop NameNode and one Hadoop DataNode. Speedups of HCL2 relative to Mahout on Platforms A/B range from 1.09x/1.49x for the Naive Bayes Trainer up to 11.88x/21.32x for Fuzzy KMeans. The device selection for manually scheduled HCL2 was decided upon by the authors after extensive testing of a variety of configurations, and is listed in Table 2. Note that Dirichlet does not have a combine or reduce stage.

TABLE 2
Manually Selected Devices for the Map, Combine, and Reduce Stages of Each Benchmark

Benchmark	Map	Combine	Reduce
KMeans	OpenCL-GPU	OpenCL-CPU	JVM
Fuzzy KMeans	OpenCL-GPU	OpenCL-CPU	JVM
Dirichlet	OpenCL-GPU	N/A	N/A
Pairwise	JVM	OpenCL-CPU	OpenCL-CPU
Bayes	JVM	OpenCL-CPU	OpenCL-CPU

TABLE 3
Percent Execution Time Spent by Platform A in Read I/O, Kernel Execution, and Write I/O While Executing Fuzzy KMeans and Bayes

	Map Stage			Reduce Stage		
	Read	Exec	Write	Read	Exec	Write
Fuzzy	5%	94%	1%	23%	65%	12%
Bayes	97%	2%	1%	22%	73%	5%

The intrinsic profiling features of HCL2 can be used to take a more in-depth look at the performance characteristics of the Fuzzy KMeans and Bayes benchmarks as a way of understanding where HCL2 performs well and poorly. Parsing of the profiling logs described in Section 3.7 produced the statistics in Table 3 for Fuzzy KMeans and Bayes running on Platform A. Because HCL2 focuses on accelerating computation, it makes sense that the performance improvement from using HCL2 would be larger for compute-bound applications like Fuzzy KMeans than for more I/O-bound applications like Bayes.

4.2 Scalability

As Hadoop and Mahout are used in extremely parallel distributed systems containing hundreds or thousands of nodes, it is important to consider the scalability of HCL2. To do so, the scalability of each benchmark is evaluated on both platforms.

Fig. 2 shows the strong scaling of mean execution time on Platforms A and B. The number of nodes tested on both platforms was limited by resource availability. None of the benchmarks achieve perfect scalability on either platform or either programming system to four or eight data nodes. This is due to I/O overheads from HDFS, increased workload imbalance as the workload is spread thinner across the system, and limited parallelism of the reduce stage. Both HCL2 and Mahout demonstrate similar scaling across applications and nodes as both are still data-parallel MapReduce programming systems, and no new bottlenecks are introduced by HCL2 that would limit scaling. We also continue to see speedup from HCL2 relative to Mahout at higher node counts.

4.3 Comparison with HadoopCL

HCL2 uses the lessons learned from HadoopCL[4] to build a more performant, flexible, and featureful programming

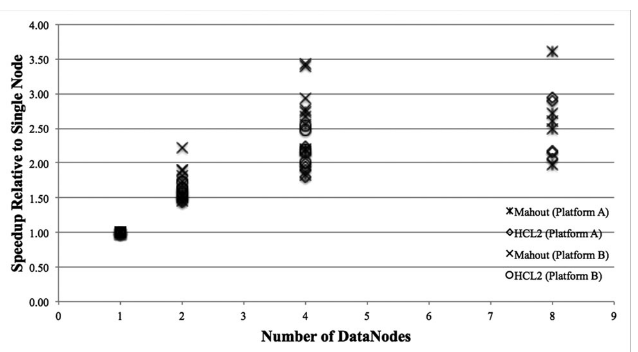

Fig. 2. Speedups of all benchmarks running on Mahout and HCL2 in Platform A and B as the number of DataNodes(DN) increase.

Authorized licensed use limited to: b-on: Universidade de Lisboa Reitoria. Downloaded on May 13,2020 at 15:32:37 UTC from IEEE Xplore. Restrictions apply.

TABLE 4
Speedup of HCL2 Relative to HadoopCL on the Four Benchmarks Used in the Original HadoopCL Paper, Taken from the Mean of 10 Runs of Each

Benchmark	HadoopCL	HCL2	Speedup
Pi	477,680 ms	299,140 ms	1.60x
Blackscholes	762,511 ms	675,547 ms	1.13x
Sort	846,935 ms	572,861 ms	1.48x
2D KMeans	302,544 ms	295,068 ms	1.03x

system. Table 4 compares HCL2 and HadoopCL performance on the benchmarks used in the original HadoopCL paper. Note that we cannot compare the performance of the five benchmarks used in this paper because they can not be implemented in HadoopCL as HadoopCL does not support sparse vector types, dynamic memory allocation, or globally shared writable data structures.

On all benchmarks we observe slight to modest performance improvements relative to HadoopCL. Considering each runtime is different in how it chunks data, schedules work, and constructs kernels it is difficult to perform a one-to-one comparison and precisely identify causes for this performance improvement. HadoopCL also lacks the native profiler support that HCL2 has. In general, we attribute it to improved compute-I/O overlap, better device & memory utilization from the HCL2 runtime, and support for using JVM execution for tasks where it is beneficial.

4.4 Compilation Overhead

HCL2 uses a modified version of APARAPI's bytecode-to-OpenCL runtime translator to convert map, combine, and reduce classes to OpenCL kernels. Table 5 quantifies the expense of that runtime compilation by measuring the amount of execution time devoted to generating and compiling kernels in each task. Note that the this time is overlapped with other initialization operations in each task. Because of this overlap it is difficult to calculate how much compilation contributes to overall execution time. An upper limit on the percent of overhead can be calculated by dividing the "Time to Translate & Compile" by the "Avg Job Time" in Table 5, which gives us a maximum of 1.7 percent of execution time spent translating and compiling kernels in Fuzzy KMeans.

4.5 Auto-Scheduling

HCL2's support for auto-scheduling removes the burden of performance tuning from the programmer while still finding a well-performing schedule based on task performance characteristics and device occupancy.

TABLE 5
Time Used on Average to Translate Bytecode to OpenCL Kernels and Compile Those Kernels to Executable Objects

Benchmark	Time to Translate & Compile	Avg Job Time
KMeans	2,132 ms	182,185 ms
Fuzzy	2,242 ms	129,085 ms
Dirichlet	1,049 ms	178,464 ms
Pairwise	2,489 ms	110,932 ms
Bayes	2,034 ms	125,929 ms

This data is taken from 10 manually scheduled runs on Platform A and is averaged across all 10 runs.

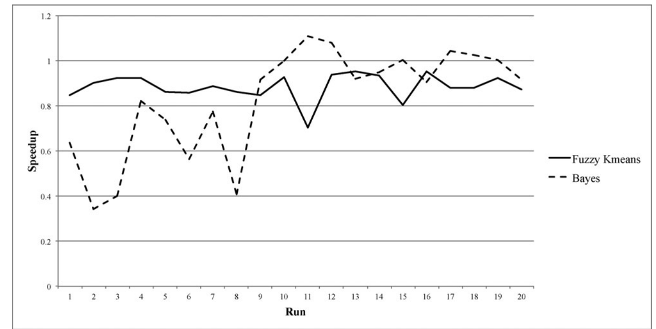

Fig. 3. Progression of execution time for auto-scheduled HCL2 Fuzzy KMeans and Bayes jobs relative to the mean execution time of manually scheduled HCL2 jobs on Platform A.

To evaluate the HCL2 auto-scheduler, 20 jobs of each benchmark were run consecutively: 10 with auto-scheduling applied only to the map stage followed by 10 with auto-scheduling applied only to the reduce stage. During mapper auto-scheduling, reduce tasks were assigned to the same device as was chosen for manual scheduling. During reducer auto-scheduling, the HCL2 static scheduler was used for mapper tasks.

Fig. 3 shows the progression of performance over all auto-scheduled runs of Fuzzy KMeans and Bayes on Platform A. These benchmarks were chosen as representations of good and poor auto-scheduling. For Fuzzy KMeans, the HCL2 auto-scheduler task placement quickly converges. There is little loss in performance relative to manually scheduled jobs for most auto-scheduled runs. Bayes requires more exploration of the task performance characteristics before the HCL2 scheduler converges. This is due to poor performance when executing the map stage of Bayes on GPUs, leading to drastic performance variation for any Bayes job which speculatively schedules map tasks on the GPU.

We observe a downwards spike in Fuzzy KMeans execution time on run eleven, caused by the switch from mapper auto-scheduling to reducer auto-scheduling. At this point, no historical performance information is available on Fuzzy KMeans reducer performance, so speculative scheduling decisions are made on suboptimal devices.

Note that these graphs show the raw execution time for individual runs as a way of illustrating the real-world performance progression one could expect from using this auto-scheduling framework. Random variations in performance can be attributed to environmental factors as other jobs in the same compute cluster use a shared resource (e.g. the network).

The only benchmark on either platform that fails to achieve parity with manual scheduling is Pairwise on Platform B, where only approximately 83 percent of peak manually-scheduled performance is achieved. This is a result of scheduling the mapper stage in OpenCL CPU threads instead of on the JVM. This mis-scheduling is caused by inaccuracies in the technique used to calculate task performance on the JVM, described in Section 3.4. This is only a factor for kernels where an OpenCL device performs similarly to the JVM. The Pairwise Mapper is an

Authorized licensed use limited to: b-on: Universidade de Lisboa Reitoria. Downloaded on May 13,2020 at 15:32:37 UTC from IEEE Xplore. Restrictions apply.

TABLE 6
Relative Performance of Auto-Scheduled and Manual Runs Based on the Number of Auto-Scheduled Runs Whose Execution Time was Within 10 Percent of the Mean of the Manually Scheduled Runs, and the Relative Speedup of the Fastest Auto-Scheduled Run Relative to the Mean of the Manually Scheduled Runs

Benchmark	Platform A		Platform B	
	# Runs	Relative Perf.	# Runs	Relative Perf.
KMeans	17/20	0.99x	19/20	1.02x
Fuzzy KMeans	8/20	0.98x	18/20	0.97x
Dirichlet	7/10	0.95x	9/10	0.98x
Pairwise	17/20	0.96x	0/20	0.83x
Bayes	11/20	1.06x	19/20	1.04x

example where the resulting scheduling actually resulted in a significant performance loss.

Table 6 shows how many auto-scheduled jobs completed within 10 percent of the execution time of the manually scheduled jobs, and how well the fastest auto-scheduled job performed relative to the fastest manually scheduled job. The main outlier (explained above) is the Pairwise benchmark on Platform B.

One item of note in Table 6 is the relative ability of the auto-scheduler on each platform to reach performance similar to manual scheduling. Platform B is measurably better at achieving performance parity than Platform A on four out of the five benchmarks (Pairwise's poor performance was explained earlier). The logs of the auto-scheduling system explain that this is a result of Platform A having twice as many GPUs as Platform B. Because the confidence of a task's performance prediction uses a distance measure based on system load, this increased dimensionality in the device load vector leads to an increased amount of time spent performing speculative execution on Platform A than Platform B, as a larger search space must be covered. Hence, more time and more jobs are spent with tasks being speculatively scheduled on suboptimal devices on Platform A.

It is also important to understand the overhead added when performing auto-scheduling. This overhead comes from three places: the timing statements used to measure the elapsed time of device computation, the computation necessary to revise a task's performance characterization based on new performance data, and the computation needed to make a scheduling decision based on task performance profiles. We can measure this overhead by directly recording the time spent in each job on revising performance characterizations and making scheduling decisions using millisecond-granularity timing statements. Note that the HCL2 Static Scheduler from Section 3.4 was added as a way of reducing auto-scheduling overhead by using stored task characterizations but not updating them.

Table 7 shows the average percentage of total execution time spent revising performance characterizations, making performant scheduling decisions, and making speculative scheduling decisions for all auto-scheduled jobs. Note that performant scheduling consumes an order of magnitude more execution time than speculative scheduling. This is a result of much more performant scheduling being done than speculative scheduling, as full task performance characterization is often achieved after at most five jobs.

TABLE 7
Overhead Added by the Auto-Scheduler

Benchmark	Speculative	Performant	Revising
KMeans	0.004%	0.07%	0.02%
Fuzzy KMeans	0.0026%	0.07%	0.04%
Dirichlet	0.0021%	0.07%	0.02%
Pairwise	0.0017%	0.03%	0.04%
Bayes	0.0027%	0.07%	0.5%

These percentages are means across all auto-scheduled runs on all platforms.

Table 8 shows how many tasks the auto-scheduler placed on each device type for the fastest run of each benchmark on Platforms A and B. In most cases, the scheduling decisions made by the auto-scheduler are identical to those made manually by expert tuning. The most common difference was execution by the auto-scheduler of tasks on the OpenCL CPU device instead of the JVM. This is likely a result of the measurement error discussed previously. For most tasks, this has little impact on performance as they both execute on the same physical architecture. The auto-scheduler used the GPU for the Bayes reduce stage on Platform A, and was able to achieve a 6 percent performance improvement relative to manual scheduling on the CPU as a result of lowered contention for CPU cycles.

4.6 Programmability Evaluation

HCL2 runtime and language features were primarily motivated by the characteristics and requirements of the KMeans and Pairwise Similarity Mahout jobs. Anecdotally, the implementation of Fuzzy KMeans, Dirichlet, and Bayes in HCL2 from their implementations in Mahout required about six hours of development time each. This time includes:

1) Gaining an understanding of the overall algorithm.
2) Understanding how Mahout's existing infrastructure is used to implement that algorithm.
3) Selecting the Mahout code which could be re-used in HCL2 and implementing any application-specific glue code for Mahout functionality which could not be re-used. In general, this includes pieces of Mahout's abstraction layers that make extensive use of object references and dynamic class loading, as these Java features are not supported by APARAPI's bytecode-to-OpenCL translator.
4) Correctness verification and performance tuning of the resulting HCL2 implementation. During this

TABLE 8
Number of Tasks Placed on Each Type of Device by the HCL2 Auto-Scheduler in Platforms A and B

Benchmark	Platform A		Platform B	
	Map	Reduce	Map	Reduce
KMeans	100 GPU	2 CPU	100 GPU	2 CPU
Fuzzy KMeans	50 GPU	2 CPU	20 GPU	2 CPU
Dirichlet	100 GPU	N/A	20 GPU	N/A
Pairwise	12 CPU	2 CPU	12 CPU	2 CPU
Bayes	100 JVM	2 GPU	50 JVM	2 JVM

Authorized licensed use limited to: b-on: Universidade de Lisboa Reitoria. Downloaded on May 13,2020 at 15:32:37 UTC from IEEE Xplore. Restrictions apply.

```
int[] outputIndices = allocInt(inputLen);
double[] outputVals = allocDouble(inputLen);
for (int i = 0; i < inputLen; i++) {
  outputIndices[i] = indices[i];
  outputVals[i] = vals[i];
}

int closestCluster = 0;
double minDist = distance(inputIndices,
  inputVals, inputLen, getGlobalIndices(0),
  getGlobalVals(0), globalsLength(0));

for (int i = 0; i < nGlobals(); i++) {
  double dist = distance(inputIndices,
    inputVals, inputLen, getGlobalIndices(0),
    getGlobalVals(0), globalsLength(0));
  if (dist < minDist) {
    closestCluster = i;
    minDist = dist;
  }
}

if (minDist == 0.0) minDist = MIN_DIST;
double denom = 0.0;
for (int j = 0; j < nGlobals(); j++) {
  double eachCDist = distance(inputIndices,
    inputVals, inputLen, getGlobalIndices(0),
    getGlobalVals(0), globalsLength(0));
  if (eachCDist == 0.0) eachCDist = MIN_DIST;
  denom += Math.pow(minDist / eachCDist, m);
}
double probWeight = 1.0 / denom;

write(closestCluster, probWeight,
  outputIndices, outputVals, inputLen);
```

Fig. 4. Fuzzy KMeans HCL2 mapper code snippet.

step, the HCL2 Debugger was useful for analyzing any tasks that exhibited memory faults on OpenCL devices, for generating a snapshot of task inputs to help with identifying correctness errors, and for creating test inputs for isolated performance testing of auto-generated OpenCL kernels. The HCL2 Profiler was useful in this development stage as it facilitated the analysis of the computational and I/O performance of the full HCL2 system. HCL2 Profiler timeline visualizations made identification of performance hotspots straightforward and helped with locating areas where computation-communication overlap in the HCL2 system could be improved.

To understand the code changes required to port a Mahout application to HCL2, consider the map and reduce implementations for Fuzzy KMeans in Figs. 4 and 5.

The Fuzzy KMeans mapper calculates the closest cluster to the current point (represented by a sparse vector stored in `inputIndices` and `inputVals`) by iterating over all clusters stored in global vectors, accessible via `getGlobalIndices()` and `getGlobalVals()`. Once the closest cluster has been found, the probability weight for that cluster is calculated based on the distances to all clusters. The input data point is then written, along with the cluster it has been classified into and the confidence of that classification. This code is written entirely in the high-level Java programming language with no manual memory management

```
int totalEle = 0;
for (i = 0; i < valsIter.nValues(); i++) {
  valsIter.seekTo(i);
  totalEle += valsIter.currentVectorLength();
}
int[] outputIndices = allocInt(totalEle);
double[] outputVals = allocDouble(totalEle);
int nOutput = merge(valsIter, outputIndices,
  outputVals, totalNElements);
write(cluster, outputIndices, outputVals,
  nOutput);
```

Fig. 5. Fuzzy KMeans HCL2 reducer code snippet.

and straightforward method calls for accessing global data. Porting from the original Mahout code was straightforward. Because Mahout's software architecture maximizes the amount of code shared by similar algorithms, Mahout applications are easily portable to HCL2 because HCL2's features were directly motivated by Mahout requirements.

The Fuzzy KMeans reducer aggregates the points classified in the same cluster, weighted by cluster probability, to generate a new cluster centroid. This is trivially accomplished using an HCL2 intrinsic function, `merge`. This `merge` function is equivalent to Mahout's `VectorSumReducer` class, a common reducer task in the Mahout clustering framework that sums multiple vectors into a single output vector. Porting from one to the other is straightforward. `merge` is used in KMeans, Pairwise Similarity, Fuzzy KMeans, and Naive Bayes.

5 Related Work

Previous projects on programming heterogeneous architectures through MapReduce frameworks or on accelerating Hadoop have taken different approaches or focused on a subset of the problems addressed by this work.

5.1 Accelerating MapReduce

The primary goal of HCL2 was to accelerate the performance of Hadoop MapReduce with minimal impact on its high-level, productive, and popular programming model. This section describes other projects with similar goals but different approaches.

Apache Spark [10], [11] offers a different programming model abstraction on top of Hadoop-supported data stores. Spark's programming model offers more flexibility than Hadoop MapReduce, including operations like parallel map, parallel reduce, parallel filter, parallel sample, and more. Because Spark natively caches values in-memory, it primarily outperforms Hadoop MapReduce on iterative or graph applications where the output of one task is re-used as the input of another [11], [12]. As HCL2 was an investigation into adding heterogeneity to an existing programming model, we deliberately chose to limit extensions to the programming model and try to adhere as closely to Hadoop MapReduce's existing programming abstractions. Both HCL2 and Spark operate on distributed platforms and distributed datasets, though only HCL2 natively supports heterogeneous platforms. Spark and HCL2 share the goal of improving performance relative to Hadoop MapReduce, but take different approaches: Spark introduces a more flexible, featureful, and complex programming abstraction

Authorized licensed use limited to: b-on: Universidade de Lisboa Reitoria. Downloaded on May 13,2020 at 15:32:37 UTC from IEEE Xplore. Restrictions apply.

with better in-memory caching of intermediate outputs, while HCL2 tries to integrate accelerator hardware seamlessly into an existing programming model. However, Spark and HCL2 are complementary, not competitive. An interesting direction for future research is to explore how accelerators and Spark can be used together to improve both the computational and I/O performance of distributed applications on the Hadoop platform.

Work done as part of GPMR [13] developed a custom distributed MapReduce implementation in C++ and CUDA which uses GPUs to execute mappers, reducers, and a number of intermediate stages which were added to the MapReduce pipeline to reduce communication costs. GPMR exposes features of CUDA, such as local synchronization, to the application developer, which HCL2 does not. While GPMR supports multi-GPU execution it does not execute application computation on the CPU, which may lead to underutilized hardware or suboptimal scheduling.

Like HCL2, the work in [14] builds on the existing infrastructure supplied by Hadoop. This work focuses exclusively on scheduling problems, foregoing Hadoop's Java programming model for C++ and CUDA. A simpler version of HCL2's auto-scheduler is used in which 1) only the OpenCL-CPU and OpenCL-GPU HCL2 devices are supported, 2) occupancy of those devices is not considered a factor in execution time, 3) only map tasks are considered eligible for GPU execution, and 4) no information on task performance is persisted across jobs. The evaluation of this work is also performed on outdated versions of both CUDA and Hadoop.

Surena [15] also accelerates Hadoop using GPUs. Similar to the work in [14], Surena uses a simple scheduling algorithm that pre-executes a set number of tasks on all available devices before switching to auto-scheduling. The effects of device utilization on performance are also ignored, and Surena assumes that all tasks fit within GPU memory. The evaluation of Surena focuses on relatively simple applications compared to the Mahout benchmarks used to evaluate HCL2.

HCL2 builds on the lessons learned from previous work in HadoopCL [4] to be more generally applicable and performant. While this previous work achieved efficient scheduling of very simple MapReduce applications on a single platform, building HCL2 required starting from a clean slate both in terms of design and implementation. Relative to HadoopCL, HCL2 expands to include auto-scheduling, a simpler and more feature-rich programming model, a wider range of supported data types, and as a result can be evaluated on real-world, challenging, and irregular applications.

5.2 Machine Learning Programming Frameworks

While the development of HCL2 was guided by machine learning applications, its scope is not limited to them. In fact, it can be applied to any application that can be decomposed onto the MapReduce abstraction. However, as this paper is guided and evaluated by machine learning applications it is important to consider other frameworks that can be used to accelerate machine learning applications.

Caffe [16] is a single-node framework for neural networks that supports a general data storage format, has many neural network primitives as first-class citizens of the framework, and supports both CPU and GPU execution. Caffe is similar to HCL2 in that both make plugging in new algorithms simple and easy. However, Caffe only supports plugging in new algorithms related to neural networks, while HCL2 supports arbitrary MapReduce applications. Additionally, the code written for Caffe must be written specifically for the architecture used and in low-level programming languages like C++ and CUDA, whereas in HCL2 the processor architecture used is completely transparent to the user. Finally, Caffe does not natively support distributed execution, automatic scheduling, or a variety of the advanced features of HCL2.

NVIDIA cuDNN is a CUDA library that exclusively targets deep neural networks. cuDNN provides a number of primitive operations that are commonly used in deep neural networking, implemented on the GPU to achieve greater than 10 times speedup when used as a backend to Caffe, relative to Caffe on the CPU. While cuDNN offers a programmable abstraction for deep neural networks on the GPU, it offers no extensibility or distributed execution and does not support CPU execution, all of which are supported by HCL2.

6 Conclusion

HCL2's programming model makes transitioning Hadoop applications to heterogeneous execution straightforward by retaining many of the same programming concepts and APIs. From the programmer's perspective, map, combine, and reduce tasks written in Java execute transparently on any available HCL2 device. While this work focuses on HCL2's development as motivated by machine learning applications, HCL2's applicability is not limited to machine learning.

In Section 2.2 we listed eight criteria required for a new programming system to remain relevant to real-world applications. HCL2 satisfies all eight criteria:

1) HCL2 supports heterogeneous execution of any computation that can be expressed in the OpenCL kernel language, and can fall back to the JVM for more complex kernels that require Java language or library features.
2) HCL2 is programmed using the high-level Java programming language and uses MapReduce abstractions that are simple and familiar to many programmers, particularly those who have previous experience with Hadoop.
3) HCL2 supports processing of primitive, composite, and sparse vector data types.
4) HCL2 manages JVM and OpenCL memory for the programmer. HCL2 also supports manual or automatic scheduling of computation on a heterogeneous system, using historical performance data to improve scheduling decisions over time.
5) HCL2 adds dynamic memory allocation to APARAPI, facilitating kernels whose memory requirements cannot be calculated ahead of time.
6) HCL2 aggregates inputs and batches their processing on heterogeneous devices. HCL2 does not require that all inputs fit on a single device at the same time. By integrating HCL2 with Hadoop, we also gain the benefits of HDFS as a massive data store.

Authorized licensed use limited to: b-on: Universidade de Lisboa Reitoria. Downloaded on May 13,2020 at 15:32:37 UTC from IEEE Xplore. Restrictions apply.

7) HCL2 natively supports globally shared sparse vectors that are both readable and writable from the JVM and from OpenCL devices.
8) HCL2 includes native programmer-friendly debugging and profiling tools.

Moving forward, we plan to take the lessons learned from the implementation of HCL2 and apply them to transparently accelerating Scala and Spark computation with OpenCL kernels across a wider range of hardware platforms.

HCL2 makes contributions in heterogeneous scheduling, programming models, and integrated development tools. It represents a holistic approach to distributed, heterogeneous software development designed to minimize programmer burden when optimizing execution of general applications on distributed, heterogeneous systems.

ACKNOWLEDGMENTS

The authors would like to thank Gary Frost at AMD, Leonardo Piga at AMD, and Xiangyu Li at Northeastern University for their valuable input and discussion. J. M. Grossman is the corresponding author.

REFERENCES

[1] Apache hadoop. [Online]. Available: http://hadoop.apache.org/, 2007.
[2] E. A. Chu and Cheng-Tao, "Map-reduce for machine learning on multicore," in *Proc. NIPS*, 2006, vol 6.
[3] S. Ghemawat and J. Dean, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[4] M. Grossman, M. Breternitz, and V. Sarkar, "HadoopCL: MapReduce on distributed heterogeneous platforms through seamless integration of Hadoop and OpenCL," *HPDIC*, 2013
[5] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, J. C. Phillips, "GPU computing," in *Proc. IEEE*, 2008, vol. 96, no. 5, pp. 879–899.
[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IEEE Int. Symp. Workload Characterization*, 2009, pp. 44–54.
[7] M. Curtis-Maury, C. D. Antonopoulos, F. Blagojevic, and D. S. Nikolopoulos, "Predication-based power-performance adaptation of multithreaded scientific codes," *IEEE Trans. Parallel Distrib. Syst.*, vol. 19, no. 10, pp. 1396–1410, Oct. 2008.
[8] M. Curtis-Maury, A. Shah, F. Blagojevic, D. S. Nikolopoulos, B. R. de Supinski, and M. Schulz, "Prediction models for multi-dimensional power-performance optimization on many cores," in *Proc. 17th Int. Conf. Parallel Archit. Compilation Tech.*, 2008, pp. 250–259.
[9] G. Frost. Aparapi in amd developer website. [Online]. Available: http://developer.amd.com/tools/heterogeneous-computing/aparapi/, 2011.
[10] Apache. Spark: Lightning-fast cluster computing. [Online]. Available: https://spark.apache.org/, 2012.
[11] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. 2nd USENIX Conf. Hot Topics Cloud Comput.*, 2010, pp. 10–10.
[12] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. 9th USENIX Conf. Netw. Syst. Des. Implementation*, 2012, p. 2.
[13] J. Stuart and J. Owens, "Multi-GPU mapreduce on GPU clusters," in *Proc. Parallel Distrib. Process. Symp.*, 2011, pp. 1068–1079.
[14] S. Koichi, H. Sato, and S. Matsuoka, "Hybrid map task scheduling for GPU-based heterogeneous clusters," in *Proc. IEEE 2nd Int. Conf. Cloud Comput. Technol. Sci.*, 2010, pp. 733–740.
[15] A. Amin, F. Khunjush, and R. Azimi, "A preliminary study of incorporating GPUs in the Hadoop framework," in *Proc. 16th CSI Int. Symp. Comput. Archit. Digital Syst.*, 2012, pp. 178–185.
[16] Y. Jia. (2013). Caffe: An open source convolutional architecture for fast feature embedding. [Online]. Available: http://caffe.berkeleyvision.org/

Max Grossman received the BS and MS degrees from Rice University in 2012 and 2013, respectively. He is a graduate student at Rice University in the Habanero Extreme Scale Software Research Group. His research focuses on motivating programming model and runtime design by lessons learned from real-world applications in machine learning, geophysics, and medical imaging.

Mauricio Breternitz is the research fellow for Software Data Analytics at AMD Research. His current and past work focuses on novel algorithms utilizing CPU and GPUs, on system-level and architectural-level characterization of cloud workloads, and on novel approaches to utilizing CPU and GPU in cloud workloads such as MapReduce and GraphLab.

Vivek Sarkar is a professor and the chair of computer science at Rice University, where he holds the E.D. butcher chair in engineering. He conducts research in multiple aspects of parallel software including programming languages, program analysis, compiler optimizations, and runtimes for parallel and HPC systems. He currently leads the Habanero Extreme Scale Software Research group at Rice University, and serves as an associate director of the US National Science Foundation (NSF) Expeditions Center for Domain-Specific Computing.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.

Authorized licensed use limited to: b-on: Universidade de Lisboa Reitoria. Downloaded on May 13,2020 at 15:32:37 UTC from IEEE Xplore. Restrictions apply.