Hadoop+: Modeling and Evaluating the Heterogeneity for MapReduce Applications in Heterogeneous Clusters

Wenting He [*†], Huimin Cui [*], Binbin Lu [*], Jiacheng Zhao [*†], Shengmei Li [*],
Gong Ruan [*†], Jingling Xue [¶], Xiaobing Feng [*], Wensen Yang [+], Youliang Yan [+]

[*]SKL Computer Architecture, Institute of Computing Technology, CAS, China
[†]University of Chinese Academy of Sciences, China
[¶]School of Computer Science and Engineering, University of New South Wales, Australia
[+]Shannon Laboratory, Huawei Technologies Co., Ltd., Shenzhen, China
{hewenting, cuihm, lubinbin, zhaojiacheng, lishengmei, ruangong, fxb}@ict.ac.cn
jingling@cse.unsw.edu.au
{yangwensen, yanyouliang}@huawei.com

ABSTRACT

Despite the widespread adoption of heterogeneous clusters in modern data centers, modeling heterogeneity is still a big challenge, especially for large-scale MapReduce applications. In a CPU/GPU hybrid heterogeneous cluster, allocating more computing resources to a MapReduce application does not always mean better performance, since simultaneously running CPU and GPU tasks will contend for shared resources.

This paper proposes a heterogeneity model to predict the shared resource contention between the simultaneously running tasks of a MapReduce application when heterogeneous computing resources (e.g. CPUs and GPUs) are allocated. To support the approach, we present a heterogeneous MapReduce framework, Hadoop+, which enables CPUs and GPUs to process big data coordinately, and leverages the heterogeneity model to assist users in selecting the computing resources for different purposes.

Our experimental results show three benefits. First, Hadoop+ exploits GPU capability, and achieves 1.4x to 16.1x speedups over Hadoop for 5 real applications when running individually. Second, the heterogeneity model can be used to allocate GPUs among multiple simultaneously running MapReduce applications, bringing up to 36.9% (17.6% in average) speedup when multiple applications are running simultaneously. Third, the model is verified to be able to select the optimal or most cost-effective resource consumption.

1. INTRODUCTION

MapReduce is emerging as an important programming model in data centers for large-scale data-parallel applications, due to the ease of development and deployment in large-scale clusters. It has been widely used in a number of domains, including non-compute-intensive domains, such as log analysis, and compute-intensive domains, such as deep learning. To efficiently support these diverse applications, heterogeneous clusters are being widely adopted in data centers, to gain advantages in performance and energy consumption. In such heterogeneous clusters, for a MapReduce application, both of its performance and cost would vary with the consumed resources. Therefore, two questions arise. First, how to select computing resources for a MapReduce application under different user purposes, e.g., performance or cost-efficiency? Second, how to allocate computing resources across multiple simultaneously running MapReduce applications? To answer the above two questions, a heterogeneity model is required to model the behaviors of MapReduce applications in heterogeneous clusters.

MapReduce was originally proposed by Google in 2004 [14], which automatically divides large input data into multiple splits for parallel processing in distributed environments. Since then, a number of researchers have made great efforts on implementing MapReduce on modern architectures. As the *multi-core CPU* has become mainstream, Phoenix is proposed to automatically distribute map and reduce tasks on multiple cores in a single shared-memory machine [30]. Based on Phoenix, Ostrich partitions a MapReduce job into a number of small sub-jobs, and iteratively processes one at a time to use resources efficiently [9]. As *GPUs* are emerging, Mars [20] and MapCG [21] are proposed, which utilize a large number of GPU threads for map and reduce tasks and assign each thread a small number of (key, value) pairs to process. As *heterogeneous CPU/GPU clusters* are emerging, HAPI [29] and HadoopCL [19] are proposed to integrate OpenCL codes into MapReduce to enable the use of accelerators in heterogeneous clusters.

Despite the efforts on MapReduce framework on a variety of architectures, modeling heterogeneity is still a big challenge for MapReduce applications. Consider a representative GPGPU platform, a six-core Intel Xeon E5-2620 configured with two NVIDIA Tesla C2050 GPUs. The memory controller and I/O resource are shared by the CPU cores and GPUs (via corresponding host threads on the CPU).

ICS'15, June 08 – 11, 2015, Newport Beach, CA, USA.

DOI: http://dx.doi.org/10.1145/2751205.2751236.

When CPU and GPU tasks are running simultaneously, the resource contention would bring unexpected performance interference. We take KNN as an example to demonstrate the problem. When only one GPU task is running, the data processing speed of 60MB/s can be attained, however, when we launch a CPU task to work with the GPU task, the overall data processing speed (CPU+GPU) would decrease to 53MB/s. The reason is that resource contention slows the GPU task down. It demonstrates that more resources do not always bring performance gain (More details in Section 2). **Therefore it is challenging to model the performance against different resource consumptions.**

In this paper, we propose an approach to model the heterogeneity for MapReduce applications in heterogeneous clusters. To this end, we introduce a heterogeneous MapReduce framework, Hadoop+, which enables user-provided CUDA/OpenCL functions to be integrated as plug-ins into Hadoop. Moreover, Hadoop+ enables users to tune and control the simultaneously running GPU and CPU tasks flexibly in order to maximize data processing speed or select a most cost-effective configuration.

This paper makes the following contributions:

- We present a Hadoop+ framework, which enables user-provided CUDA/OpenCL Map and(or) Reduce functions to be embedded into Hadoop as plug-ins, thereby enabling CPUs and GPUs to process big data coordinately. Our experimental results show that Hadoop+ exploits the capability of GPUs, and achieves speedups ranging from 1.4x to 16.1x over Hadoop for 5 real applications when running individually.

- We create a heterogeneity model to predict the shared resource contention among simultaneously running heterogeneous tasks (CPU and GPU, and further predict the performance gain when allocating a computing resource to an application.). Using the model to allocate GPUs among applications achieves up to 36.9% (17.6% in average) performance improvement when multiple MapReduce applications are running simultaneously.

- Our approach can assist users to select an optimal or most cost-effective resource allocation strategy for an application, with no need of running the application under all possible resource allocation strategies.

The rest of the paper is organized as follows. Section 2 introduces the background and our motivation. Section 3 presents our Hadoop+ framework. Section 4 discusses our methodology for modeling heterogeneity. Section 5 describes our experimental validation. Section 6 discusses the related work. Section 7 concludes.

2. BACKGROUND AND MOTIVATION

2.1 MapReduce

MapReduce [14] is a data-parallel programming model for processing big data in data centers. Programmers are only required to specify a *Map* function which takes a (key, value) pair as input and generates a list of intermediate (key, value) pairs, and a *Reduce* function which takes all values associated with the same key and produces a list of (key, value) pairs as output. The Apache Hadoop [5] is a Java-based open-source implementation of MapReduce programming model, built based on a distributed file system (HDFS).

Algorithm 1 The KNN program in Hadoop

```
function MAP($train, TEST$)
    $Compute\_All\_Distances(train, TEST)$
end function
function REDUCE($test, DistanceOf(test, TRAIN)$)
    $Select\_TopK(Distance(test, TRAIN))$
end function
```

Algorithm 1 shows the pseudo-code of K-nearest neighbors (KNN) algorithm [11] written in Hadoop. KNN is a widely used algorithm in classification. Given a set of training examples and a number of testing samples, it computes the K nearest neighbors in the training set for each testing sample.

The programmer only needs to implement the two functions of MapReduce primitive *Map* and *Reduce*. In particular, *Map* takes one train sample (denoted as $train$) and the test set (denoted as $TEST$) as input, calculates the distance of $train$ to every item in the test set $TEST$, and *Reduce* gets a list of distances attaching to the same test sample, selects the topK nearest neighbors for each test sample.

The MapReduce framework automatically divides the input data into multiple splits, which will be processed in parallel. Each split is processed by a map task, also called a mapper. The outputs of the map tasks are grouped by key, and the outputs with the same key are shuffled to one reduce task (also called a reducer) for processing.

2.2 Motivation Example

2.2.1 Experimental Methodology

To examine the behavior of a MapReduce application in heterogeneous clusters, we implement the above KNN in the Hadoop+ framework, and more details of the framework will be discussed in Section 3. Here, we only highlight some features for discussing the motivation example:

- Hadoop+ will launch a user-provided map/reduce task written with CUDA/OpenCL on a GPU, when a GPU resource is available.

- Hadoop+ will also launch a traditional map/reduce task on one or more CPU cores, in the same way with Hadoop. The Hadoop *Mapper* or *MultithreadedMapper* can be used to issue single-threaded or multi-threaded map tasks.

The platform we use is an Intel six-core Xeon E5-2620 chip, configured with two NVIDIA Tesla C2050 GPUs. Other hardware parameters and the parameters for KNN are listed in Section 5. In this paper, we use the metric of *data processing speed* to represent the performance of an application, which is calculated with the equation:

$$dps = V/T \qquad (1)$$

where V represents the total data volume processed on the cluster and T represents the time for processing the data.

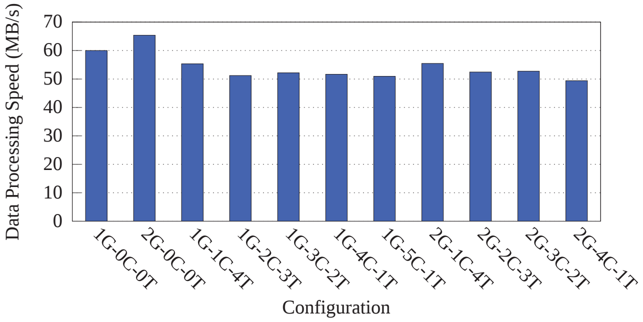

Figure 1: KNN's data processing speed.

2.2.2 *Key Observations*

We evaluate 11 computing resource configurations, as shown by the horizontal axis in Figure 1. Each configuration is denoted as gG-cC-tT, where g represents for the number of simultaneously running GPU map tasks, c for the number of simultaneously running CPU tasks, and t for the number of threads inside each CPU map task. Figure 1 shows the performance, with the vertical axis representing the data processing speed. We make two significant observations from the results:

- *Using two GPUs only brings very slight performance gain over one GPU.* When only one GPU is used, the data processing speed is 60MB/s (the first bar in Figure 1). However, when another GPU is exploited simultaneously, the data processing speed is only slightly increased to 65MB/s (the second bar in Figure 1).
- *Coordinating CPUs together with one GPU leads to worse performance than that of only using one GPU.* When only one GPU is used, the data processing speed is 60MB/s (the first bar). However, When one or more CPU tasks are running simultaneously with one GPU task (bars 3-7), the overall performance would decrease unexpectedly, varying from 58MB/s to 51MB/s. A similar observation can be found for the configurations containing two GPU tasks.

2.2.3 *Analysis*

First we demonstrate the different behaviors of a CPU task and a GPU task in Hadoop+, as shown in Figure 2. As the red line shows, the I/O traffic of a CPU task keeps almost unchanged during the task execution. The reason is that Hadoop+ leverages the execution mechanism in Hadoop for CPU tasks, which iteratively reads only a small piece of data and processes them quickly, thus the I/O traffic keeps low. However, the behavior of the GPU task is different, as shown by the blue line. To obtain high GPU occupancy, the GPU task reads a chunk of data, transfers it to the GPU, and launches the GPU kernel to process it, thus it exhibits obvious phase behavior. In particular, the I/O traffic is high when the GPU task is reading data from HDFS (via its host thread), and low when the GPU task is executing the kernel.

To analyze the reason for the observations in Section 2.2.2, we take one GPU task, denoted x, and examine its performance under the 11 configurations. We find that the key reason is shared I/O resource contention among CPU and GPU tasks. To demonstrate this, we comment out the computation in x and run it under the 11 configurations. In this way, each GPU task reads a split from HDFS without any following computations. Figure 3 shows the data reading speed of x. When only x is running, the data reading speed can reach 72MB/s (the first bar 1G-0C-0T), while it drops to 36MB/s when another GPU task is running simultaneously (the second bar 2G-0C-0T). Furthermore, when 4 single-threaded CPU tasks and another GPU task are running together with x, its data reading speed would decrease to only 14MB/s (the last bar 2G-4C-1T).

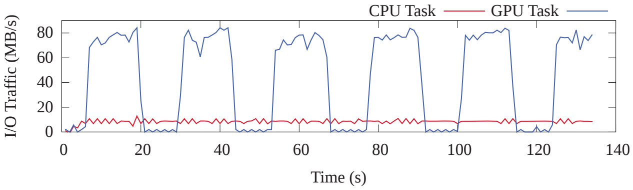

Figure 2: Behaviors of CPU/GPU tasks.

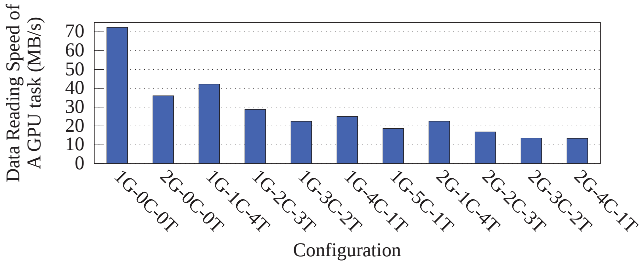

Figure 3: KNN's data reading speed.

2.2.4 *Summary - The Challenge*

The observations and our analyses demonstrate that it is a challenge to model the heterogeneity for MapReduce applications running in heterogeneous clusters. In particular, the challenge can be summarized into the following questions:

- What factors would affect the performance gain when allocating a computing resource to an application?
- How will the performance contribution of one computing resource, e.g., GPU, vary with applications?
- How to select a resource configuration for an application for different purposes, e.g., to obtain best performance, or to be most cost-effective?

3. HADOOP+ FRAMEWORK

Figure 4 gives an overview of our Hadoop+ framework. Besides the *Map* and *Reduce* primitives in Hadoop, Hadoop+ provides another two primitives, *PMap* and *PReduce*, to programmers. The difference is that the *PMap* and *PReduce* in Hadoop+ enable programmers to write explicit parallel CUDA/OpenCL functions running on GPUs as plug-ins, as shown by the box of "User-Provided PMap/PReduce Function" in Figure 4. Meanwhile, users can also use the *Map* and *Reduce* functions in Hadoop. In Hadoop+, users can provide *Map*, *PMap* or both, and *Reduce*, *PReduce* or both.

To support explicit parallel Map functions, Hadoop+ provides different input parameters for *Map* and *PMap*. In particular, the input of *Map* is $(key, value)$, while the input

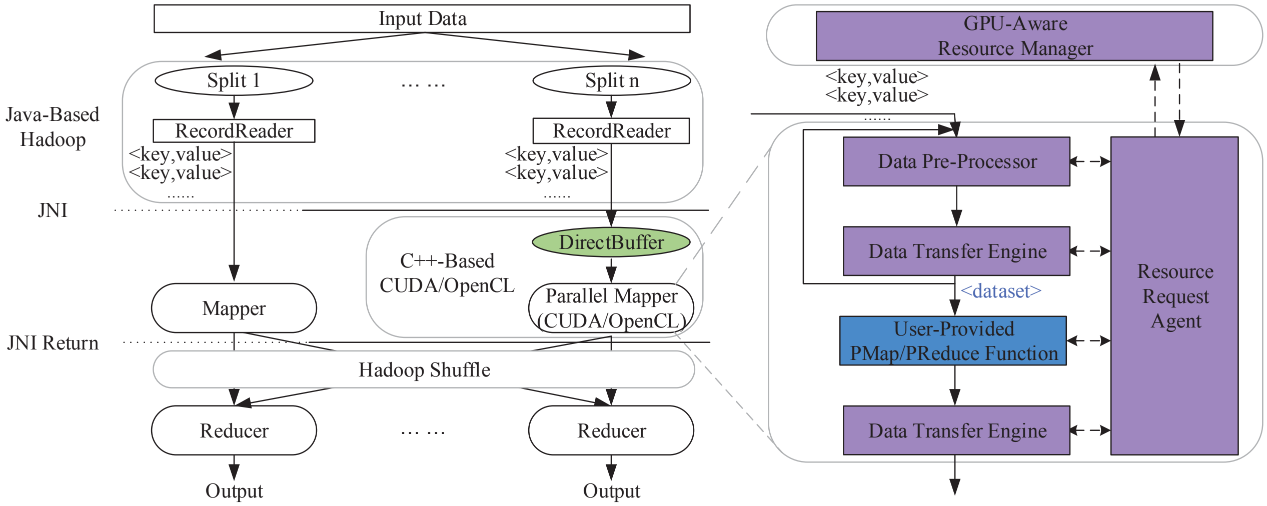

Figure 4: Overview of Hadoop+ framework. (Only parallel mapper is shown for clarity.)

of $PMap$ is *dataset*, i.e., a list of $(key, value)$. Meanwhile, the input parameters for *Reduce* and *PReduce* are identical, which are the outputs with the same key from all map tasks. The *PMap* and *PReduce* functions are defined as follows:

$PMap : (k1, v1) * - > (k2, v2) * .$

$PReduce : (k2, v2*) - > (k2, v3) * .$

We continue to use the example of KNN to illustrate Hadoop+. Algorithm 2 shows the pseudo code, with the CUDA kernel omitted. Comparing with Algorithm 1, the function takes a *train_dataset* rather than an individual *train* as the input, and writes explicitly parallel CUDA code in $PMap$.

Algorithm 2 The KNN program in Hadoop+

```
function PMap($train\_dataset, TEST$)
    for all $train$ in $train\_dataset$ do
        $Compute\_Distances\_CUDA(train, TEST)$
        $Insert\_Distance\_CUDA(train, TEST)$ $to$ $TopK(TEST)$
    end for
end function
function Reduce($test, DistanceOf(test, TRAIN)$)
    $Select\_TopK(Distance(test, TRAIN))$
end function
```

3.1 Overall Workflow

As Figure 4 shows, Hadoop+ also divides the input data into multiple splits, which will be processed in parallel, with each split being processed by a mapper. If a mapper does not include *PMap*, the split is processed in the same way as Hadoop, which processes the $(key, value)$ pairs sequentially. Otherwise, if a mapper includes *PMap*, each split is parsed into a "dataset" before processing, which combines a list of $(key, value)$ pairs, together with the meta data describing the format of the dataset. Programmers take the dataset as the input of the $PMap$ function, and Hadoop+ leverages JNI (Jave Native Interface) to invoke the native functions.

Hadoop+ extends the resource manager in Hadoop for managing the non-preemptible GPU resources, as shown by the "GPU-Aware Resource Manager" in Figure 4. At runtime, each map task has a "Resource Request Agent", which is responsible for requesting computing resources from the "GPU-Aware Resource Manager". For clarity, we denote the map task which successfully obtains a GPU as a *GPU mapper*. If a mapper does not successfully obtain a GPU, it would be a CPU mapper and be processed in the same way with Hadoop, so we discuss only the GPU mapper in this section.

As in Hadoop, the input split is first read from HDFS (Hadoop Distributed File System) and parsed into $(key, value)$ pairs by the RecordReader. To enable the $(key, value)$ pairs to be accessible and further processed by native functions, we need to put them into a buffer that can be accessed by both the Java and native codes. Therefore, we take advantage of the $DirectBuffer$ mechanism in Java, which resides outside of the garbage-collected heap. In particular, the Java code allocates a DirectBuffer and sequentially puts $(key, value)$ pairs into the buffer, and the native code reads the buffer for processing. To assist the native code to correctly parse the buffer, Hadoop+ automatically generates the meta data from the input file format description, which is provided by users in an XML format. In particular, the meta data serve to instruct the native code how to parse the input file, as the RecordReader for Hadoop.

With the data in the $DirectBuffer$, the GPU mapper processes the data with four steps. First, the data are preprocessed with some built-in or user-defined functions by the "Data Pre-Processor", e.g., text parsing, byte order conversion, etc. Second, the "Data Transfer Engine" accumulates a chunk of data and transfers the data to the target device, according to the obtained computing resource via "Resource Request Agent". Third, the "User-Provided PMap Function" would be invoked, and finally the output $(key, value)$ pairs are transferred back to Hadoop+ for shuffling. Thus the mapper task terminates.

Afterwards, the output of the mappers with the same *key* would be shuffled to one reducer, and the same workflow in Figure 4 applies for parallel reducers. If no *PReduce* is provided, the reducer would be processed in the same way with Hadoop.

3.2 Resource Management

In Hadoop+, there are two components responsible for resource management. The "GPU-Aware Resource Manager" is global and maintains all the resources in the cluster, serving all map and reduce tasks. The resource manager is located on a preset node denoted as RM in Hadoop scheduler, YARN [3]. And there is a "Resource Request Agent" for an individual map or reduce task, to communicate with the resource manager.

The resource manager maintains a pool for available GPUs, to ensure that one GPU would not be allocated to multiple tasks simultaneously. Each machine in the cluster updates its own available GPUs to the resource manager per heartbeat, which allows the resource manager to know the status of all machines. The resource manager processes GPU resource requests from all tasks in an FIFO order by default.

For each task, the resource request agent automatically detects user-provided native functions. If CUDA or OpenCL functions are detected, it would request a GPU resource from the resource manager. When the resource manager receives the request, it seeks for a GPU on that machine. Once the request is responded as success, the resource request agent would get the allocated GPU, and launch CUDA or OpenCL kernels to the corresponding GPU.

Meanwhile, the resource manager maintains a model to determine the resource configuration, i.e., the number of simultaneously running CPU and GPU tasks, together with the number of threads for each CPU task. The model is obtained using the approach in Section 4. It can assist users to determine the resource configuration for an application for different purposes, e.g., to obtain the best performance, or the most cost-effective. Furthermore, it can be used to allocate GPUs across multiple simultaneously running applications.

4. MODELING THE HETEROGENEITY

In this section, we continue to take KNN as our example to answer the questions raised in Section 2.2.

4.1 Problem Formulation

The problem of modeling the resource heterogeneity per node can be formulated as:

On a GPGPU platform with n GPUs, with the host CPU having p cores, given a resource configuration, i.e., the number of GPU tasks (g), the number of CPU tasks (c), together with the number of threads per CPU task (t), how to determine the data processing speed under the configuration for any given MapReduce application?

We will continue to use the metric of data processing speed introduced in Section 2. In particular, for the GPGPU platform in the problem formulation, the data processing speed of the platform is:

$$dps = \frac{c}{tc} + \frac{g}{tg} \tag{2}$$

where tc is the time for processing a split using a CPU mapper, and tg for a GPU mapper. $\frac{1}{tc}$ and $\frac{1}{tg}$ are the data processing speeds of one CPU and GPU task respectively. Therefore, the data processing speed of the platform is the accumulated processing speed of the simultaneously running c CPU tasks and g GPU tasks.

If a user aims to obtain the best-performing configuration, we can use the model to maximize the dps. Alternatively, users can also define other objectives, e.g., cost-efficiency, which will be discussed in Section 5.

4.2 Characterizing A Task

In Equation 2, tc and tg would vary with the resource configurations due to resource contention. It is impractical to profile the processing time under all configurations. Therefore, we model dps using the behavior of a single-threaded CPU and a GPU task running individually on the platform, to avoid profile space explosion. We call the two tasks as the **CPU_base** and **GPU_base** of a MapReduce application on the platform, denoted as β_{CPU} and β_{GPU} respectively.

Table 1: List of parameters for modeling the *dps*.

Platform	p	number of CPU hardware threads
	n	number of GPUs
App.	tdc_0	time of reading a split into CPU memory for CPU_base
	tpc_0	time of processing a split on CPU for CPU_base
	tdg_0	time of reading a split into CPU memory for GPU_base
	tpg_0	time of processing a split on GPU for GPU_base
	$tdcg_0$	time of transferring a split from CPU into GPU memory
Conf.	g	number of concurrent GPU tasks
	c	number of concurrent CPU tasks
	t	number of threads per CPU task

First, we characterize β_{CPU} and β_{GPU} into two feature vectors, one for CPU and the other for GPU.

$$\begin{cases} fv_{cpu}(task) = (tdc_0, tpc_0) \\ fv_{gpu}(task) = (tdg_0, tdcg_0, tpg_0) \end{cases} \tag{3}$$

where the meanings of tdc_0, tpc_0, tdg_0, tpg_0 and $tdcg_0$ are listed in Table 1. We introduce a 0 in the subscript to emphasize that these features are collected *only* for the base tasks. Therefore, the feature vectors are determined only by the application and the platform, and do not vary with configurations.

Note that the data reading time for β_{CPU} and β_{GPU} may differ, due to the different data processing patterns on CPUs and GPUs, as shown in Figure 2.

4.3 Modeling Data Processing Speed

For the problem in Section 4.1, given a configuration k, we model the $dps(k)$. In particular, a configuration k includes: the number of GPU tasks running simultaneously (g), the number of CPU tasks running simultaneously (c), and the number of threads per CPU task (t). For clarity, we list the parameters in our model in Table 1.

In this paper we focus only on data-local tasks, i.e., for a task, its data are located on where the task is executed. Modeling non-local tasks would introduce network cost and is beyond the scope of this paper.

For a given task configuration k, the execution time of one task can be computed using:

$$T_k = \begin{cases} tdc(k) + tpc(k), \ for\ CPU\ task \\ tdg(k) + tpg(k) + tdcg(k), \ for\ GPU\ task \end{cases} \tag{4}$$

where tdc, tpc, tdg, tpg and $tdcg$ are the same as in Table 1, except that these variables represent the corresponding time under a given configuration k. We observed that the time of transferring data from CPU to GPU memory is much less than the time spent on reading data from disk, i.e., 0.2 seconds vs 14.1 seconds for 1GB data. Therefore, we omit the item of $tdcg$ in Equation 4.

In particular, a map/reduce task can be divided into two phases: data reading and computing. For GPU tasks, its computation time $tpg(k)$ does not change with configurations. For multi-threaded CPU tasks, we assume the performance would be perfectly scaled up with the number of threads. The assumption is reasonable since MapReduce applications are data parallel and do not include synchronization inside a task, and the performance interference across multiple threads caused by shared cache and bandwidth contentions can be predicted using the approach in [39], which is ignored in this paper. So the data processing time under a given configuration k can be computed using Equation 5, and the data reading time will be discussed in Section 4.4.

$$\begin{cases} tpc(k) = tpc_0/t \\ tpg(k) = tpg_0 \end{cases} \tag{5}$$

4.4 Modeling Data Reading Time

In Hadoop+, the CPU and GPU tasks would contend for the shared I/O resource, and we use map tasks for discussion. As discussed in Section 2, the simultaneously running tasks would cause shared I/O resource contention and increase the data reading time for each task. Therefore for the configuration parameters in Table 1, the number of simultaneously running CPU and GPU tasks (c and g) would affect the data reading time.

We empirically observe that the data reading time is affected by two factors: the total I/O traffic, and the number of applications issuing I/O requests. Our methodology is based on the observation as follows. First, we construct a HDFS reader benchmark *base* which continuously reads data from a HDFS file and includes no computation. We launch *base* to run individually and record the data reading time $d0$. Second, we create another HDFS reader benchmark *hreader*, which reads some bytes from a HDFS file periodically. By adjusting the reading interval, we can vary its I/O traffic. Then we co-run the *hreader* with *base*, and record the data reading time of *base*. Third, we launch 2-5 *hreader* and vary their I/O traffic, co-run with *base*, and record the data reading time of *base*. Finally, we plot these points into Figures 5 and 6.

Figure 5 shows *base*'s data reading time varying with co-runners. In particular, the vertical axis shows the data reading time normalized to $d0$. Each color represents the data for one fixed number of co-runners. It gives two observations: 1) For a given number of co-runners, *base*'s data reading time increases linearly with I/O traffic, until an upper bound is reached. 2) The upper bound is determined by the number of co-runners, which can be represented as a linear function, as shown in Figure 6. In particular, when n tasks are reading a file from HDFS simultaneously, the maximal data reading time is n times of base task, since only $1/n$ of the I/O resource can be obtained for each task.

Thus, the data reading time can be represented as

$$\begin{aligned} tdc(k) = min(a * sumio + b, fmax(g + c * t)) * tdc0 \\ tdg(k) = min(a * sumio + b, fmax(g + c * t)) * tdg0 \end{aligned} \tag{6}$$

where

$$fmax(n) = n, \quad sumio = \frac{VS}{tdc_0 + tpc_0} * (c * t) + \frac{VS}{tdg_0} * g \tag{7}$$

In Equation 7, VS is the data volume of one split, therefore $\frac{VS}{tdg_0}$ is the I/O traffic for a GPU task. For a CPU task, due to the pattern of alternating data reading and computing, the exhibited average I/O traffic of each working thread would be $\frac{VS}{tdc_0+tpc_0}$. As a result, $(\frac{VS}{tdc_0+tpc_0} * (c*t) + \frac{VS}{tdg_0} * g)$ is the aggregated I/O traffic for all simultaneously running tasks. The *min* function in Equation 6 means that the maximal data reading time would not exceed the maximum shown in Figure 6.

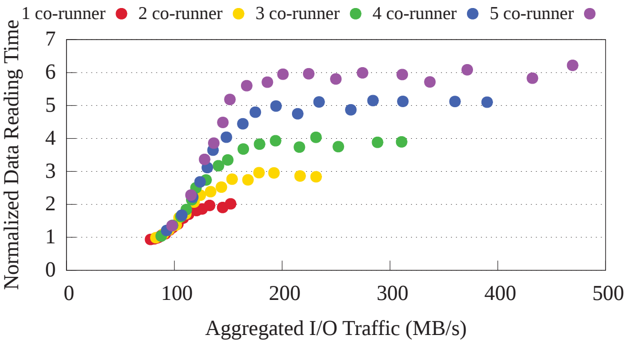

Figure 5: Data reading time with co-runners.

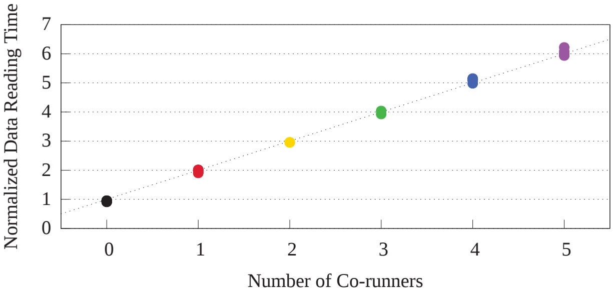

Figure 6: Maximal data reading time.

Note that the data reading model only depends on the platform and does not depend on applications. We will demonstrate this in Section 5.

4.5 Understanding the Motivation Example

Now we use KNN to answer the three questions raised in Section 2.2.4.

Q:What factors would affect the performance gain when allocating a computing resource to an application?

A:The performance is determined by the application's feature vectors, i.e., the data reading time and computation time. In other words, the performance gain is relevant to the ratio between data reading and computation, and the speedup of GPU computation over CPU computation for the application.

Q:How will the performance contribution of one computing resource, i.e., GPU, vary with applications?

A:If an application's data reading time makes up a high proportion of the processing, more computing resources would cause shared I/O resource contention, thereby resulting in little performance gain. Contrarily, if an application's computation time dominates the processing, more computing resources would bring significant performance gain.

For KNN, its feature vectors are $fv_{cpu}(KNN) = (14.1, 120.2)$, and $fv_{gpu}(KNN) = (14.1, 0.2, 2.9)$. For the GPU base task, the data reading time is 14.1 seconds while the processing time is only 2.9 seconds. Therefore KNN cannot obtain a significant speedup when more computing resources are allocated to it, as discussed in Section 2.

Q:How to select a resource configuration for an application for different purposes, e.g., to obtain the best performance, or to be the most cost-effective to satisfy some performance requirement?

A:We can use our model to predict the data processing speed under different resource configurations, as illustrated in Figure 1, and select the resource configuration under different objectives. For KNN, the configuration of 2G-0C-0T would exhibit the best performance, and the configuration of 0G-6C-1T is the most cost-effective, as discussed in Section 5.

4.6 Summary and Discussion

For an application, we run its GPU_base and CPU_base independently and obtain its feature vectors. Then we use our heterogeneity model to compute the data processing speeds under different configurations, and determine the resource configuration for different purposes. In particular, we leverage the data reading model in Equation 6 to predict the data reading time, and Equation 5 to predict the data processing time under different resource configurations.

So far our model does not consider the following issues, which will be our future work. First, the model does not include the resource contention for network communication. Second, the model does not enable multiple GPU tasks to run simultaneously on one GPU card. Third, the model does not enable multiple tasks running simultaneously on one hardware core.

5. EVALUATION

5.1 Platform and Benchmark

The heterogeneous cluster we use includes 8 nodes, with each node being an Intel 2.00GHz six-core Xeon E5-2620 chip configured with two NVIDIA Tesla C2050 GPUs, and each GPU has a 3GB global memory and 14 SMs, each containing 32 Streaming Processors(SPs). Each SM has 32768 registers and a 48KB shared memory.

Our Hadoop+ is implemented based on Hadoop 2.1.0 beta. We use the following big data applications for evaluation:

- *K-nearest neighbors (KNN).* Details of the algorithm have been discussed in Section 2.

  The base Hadoop version for comparison is implemented using the algorithm presented in [11]. For fairness, we also introduced a maximum heap in the version to select the top-K training examples before shuffling.

  The input data size is 960GB. The dimension of input is 128, the number of training examples is 977,054,400, the number of testing examples is 448, and K is 4. The split size is 1GB.

- *K-Means Clustering (Kmeans).* Kmeans partitions a number of objects into k clusters such that similar objects belong to the same cluster [12].

  The base Hadoop version for comparison is mahout-0.7.0 [1].

  The data set consists of 3,932,280,000 four-dimension vectors. The number of clusters (K) is set to 100. The split size is set as 1GB.

- *Row Similarity (RS).* The algorithm is typically used in recommendation systems. It takes a number of vectors as input, and computes the similarity between two vectors of the inner product space.

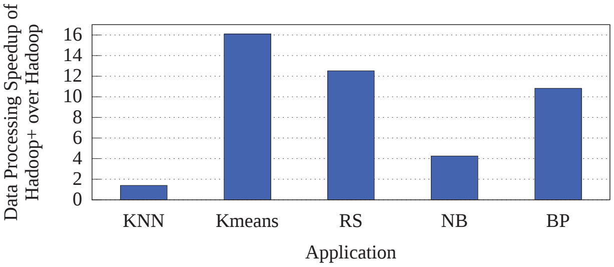

Figure 7: Speedup of Hadoop+ over Hadoop.

  The base Hadoop version for comparison is mahout-0.7.0 [1].

  The dimension of vectors is 16,777,216, the number of vectors is 8192, the sparsity is set to 0.0625, and the split size is set as 1GB.

- *Naive Bayes Classifier (NB).* This is a popular method for text categorization. It uses word frequencies as the features, and judges documents as belonging to one category of the other.

  The base Hadoop version for comparison is mahout-0.7.0 [1].

  The size of the input data is 96GB, and the number of classes is 20, and the split size is set as 1GB.

- *Back Propagation (BP).* This is a classical algorithm for training artificial neural networks. It calculates the gradient of a loss function with respects to all the weights in the network, and iteratively updates the weights to minimize the loss function.

  We implement the Hadoop version and Hadoop+ version using the approach in [25].

  The training data are 1,024,000 48-dimension vectors. The number of nodes in hidden layer and output layer is set to 65536 and 48 respectively, and the split size is set as 4MB.

For all the five applications we used, the map phase dominates the execution. Therefore, we write the $PMap$ functions with CUDA using the Hadoop+ interface for GPU acceleration, and keep using original *Reduce* functions for the reduce phase. Meanwhile, the original Map functions are also included for CPU tasks.

We evaluate Hadoop+ in two scenarios. First, the five applications are executed one by one in single-application scenario and we evaluate the performance for each application. Second, two of the five applications are launched simultaneously to emulate the multi-application scenario, and we evaluate the completion time for the two applications. Sections 5.2 and 5.3 present the performance results for the two scenarios respectively, Section 5.4 discusses the experimental results case by case, Section 5.5 verifies our performance prediction model, and Section 5.6 demonstrates how to find a cost-efficient resource consumption using our model.

5.2 Results in Single-App Scenario

In this section, we present the overall performance speedup for the five application implemented in Hadoop+ over Hadoop. To make a fair comparison, we also tune the optimal configuration for Hadoop applications, which is identical for all the five applications, i.e., 6 single-threaded CPU tasks.

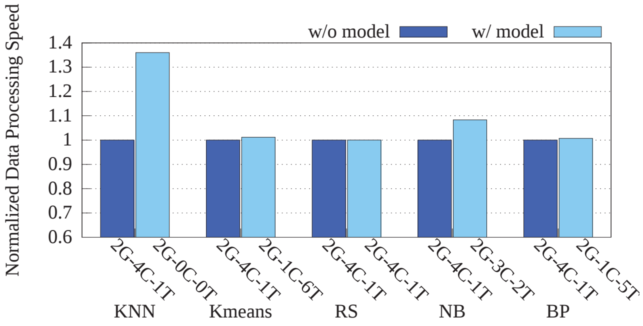

Figure 8: Performance improvement of configuration selection using the heterogeneity model.

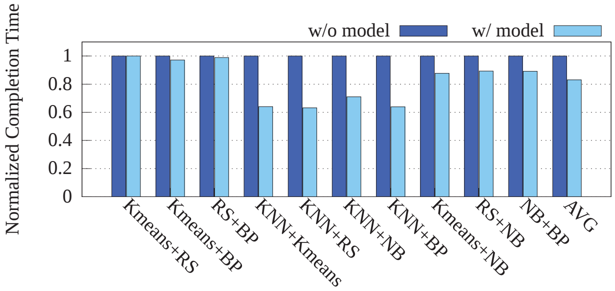

Figure 9: Results in a multi-app scenario.

Figure 7 shows that Hadoop+ can achieve 1.4x to 16.1x speedups over Hadoop. The benefit depends on the features of applications, and we will further discuss it in Section 5.4.

Figure 8 shows the performance contribution of our heterogeneity model, with the obtained optimal task configurations annotated on the horizontal axis. For each application, the left bar shows the performance (normalized to 1) of the default configuration, i.e., 2G-4C-1T, and the right bar shows the normalized performance with the optimal configuration obtained using our heterogeneity model. For RS, the obtained optimal configuration is identical to the default value. For Kmeans and BP, the optimal configuration brings only 1.1% and 0.7% performance improvement. However, for KNN and NB, the optimal configuration would bring 36.0% and 8.3% performance improvement respectively.

5.3 Results in Multi-App Scenario

Using the 5 applications, we generate $C_5^2 = 10$ pair-wise combinations to evaluate the multi-application scenario. Without our heterogeneity model, FIFO scheduler is used to allocate GPUs among the simultaneously running applications. With the heterogeneity model, we can allocate GPUs to the applications for which GPU can bring more performance gain. In particular, for two simultaneously running applications A and B, if A is predicted to be able to get more performance gain from GPUs than B, A would be given high priority to obtain GPUs.

Figure 9 shows the normalized completion time for each combination, with each group representing for one combination. For each combination, the first bar represents the completion time when FIFO scheduler is used without the heterogeneity model, and the second bar represents the completion time when GPUs are allocated using the heterogeneity model. As shown in Figure 9, for the 10 combinations, our performance model would reduce the completion time up to 36.9%, with the average of 17.6%.

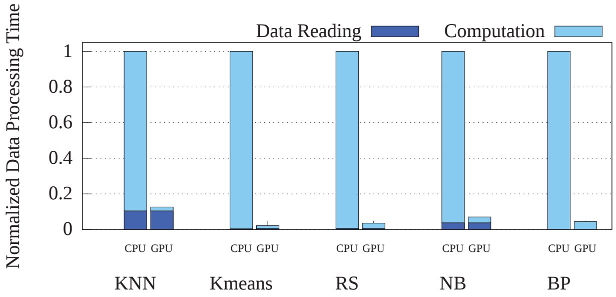

Figure 10: Normalized data processing time.

From Figure 9, we make the following observations:

- For a given pair-wise combination, if each application can get significant benefit from GPUs, the GPU allocation strategy would not bring significant performance improvement for the combination. In particular, the combinations of Kmeans+RS, Kmeans+BP, and RS+BP belong to this category.
- For a given pair-wise combination, if one application can get significant benefit from GPUs while the other cannot, the GPU allocation strategy would bring significant performance improvement. The other 7 combinations belong to this category.

5.4 Case Study

As we discussed earlier, the performance gain from leveraging GPUs depends on two issues: the ratio between data reading and computation, and the speedup of GPU computation over CPU computation. In this section, we use experimental results to demonstrate the analysis.

To analyze the application features, we use Figure 10 to break down the data reading time and computation time for GPU and CPU tasks. For each application, the left bar shows the execution time of a CPU_base task, which is normalized to 1, and the right bar shows the execution of a GPU_base task, normalized by the CPU_base's execution time. Furthermore, both bars are broken down into data reading time (the dark blue section on the bottom) and computation time (the light blue section on the top).

We take KNN for the first case study, the speedup of Hadoop+ with GPU over Hadoop is 1.4x, since the data reading time dominates the data processing time for the GPU task, i.e., the data reading takes 14.1 seconds while the computation takes only 2.9 seconds. Even if the GPU task exhibits a significant speedup over CPU computation (2.9 seconds vs 120.2 seconds), the high proportion of I/O reading time makes the GPU speedup concealed from the perspective of the whole application execution. In particular, Hadoop issues 6 CPU single-threaded tasks to run simultaneously, and can complete 6 splits in 138.9 seconds, while Hadoop+ can process one split in 17.0 seconds. Thus the overall speedup is only 1.4x.

Kmeans, RS and BP exhibit different behaviors in Figure 10, for which the GPU task spends very little time on data reading, thus the tasks are dominated by GPU computations. For Kmeans, RS and BP, the data reading time occupies less than 0.6% of the total execution time. Meanwhile, the computation on GPU obtains significant speedups

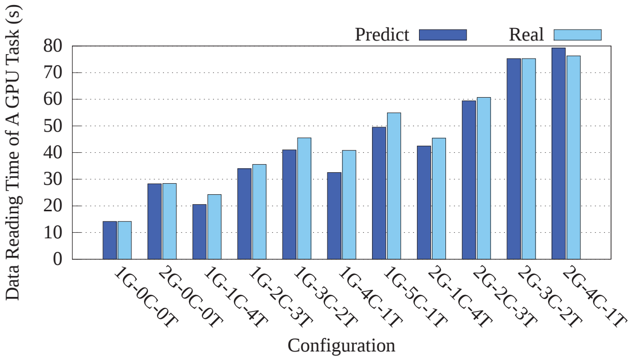

Figure 11: KNN's predicted data reading time.

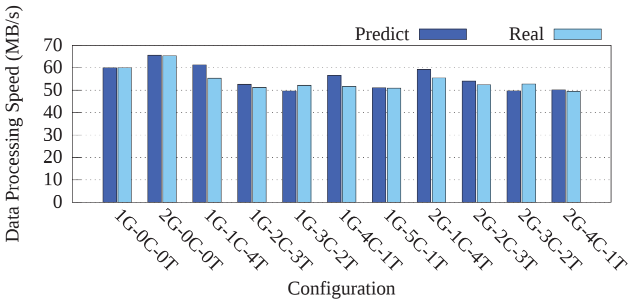

Figure 12: KNN's predicted processing speed.

over CPU, i.e., 61.6x, 34.0x, and 22.6x for Kmeans, RS and BP respectively. Therefore, they obtain 16.1x, 12.5x and 10.8x speedups over Hadoop respectively.

NB is an application in between the above two types. Its data reading time occupies 53.9% of the total execution time for GPU tasks, and the computation of GPU obtains 26.7x speedup. Therefore, its performance is also in between the above two types, i.e., 4.2x over Hadoop.

5.5 Model Verification

We use the configuration schemes in Figure 1 for our model verification. Figure 11 shows the predicted and real data reading time under different resource configurations for KNN, with the left bars representing the predicted values, and the right for the real values. The average error is only 5.9%. Furthermore, our model can accurately predict KNN's overall data processing speed of the whole cluster under different resource configurations (For clarity, each node in the cluster shares the same resource configuration). Figure 12 shows the predicted and real data processing speed of the cluster, with an average error of 3.8%.

Using the same approach, we can create the data processing model for the other 4 applications. Due to space limitations, we select five 2-GPU configurations to verify our model. Figure 13 presents the experimental results for these applications, and it shows that our model can accurately predict the data processing speed under different resource configurations, with an average error of 2.5%.

5.6 Cost Efficiency

In this section, we leverage our model to help users select the most cost-effective resource configuration. We use the official price of our CPU and GPU card to define the cost per time unit [4, 28]. In particular, the price of a CPU is \$408, with each core priced \$68, and the price of a GPU card is \$2499. Therefore, if the cost per time unit of one CPU core is normalized to 1, the cost per time unit of one GPU card would be \$36.8. To evaluate the cost efficiency, we use the 11 configurations in Figure 1, together with 6 extra configurations with GPUs disabled.

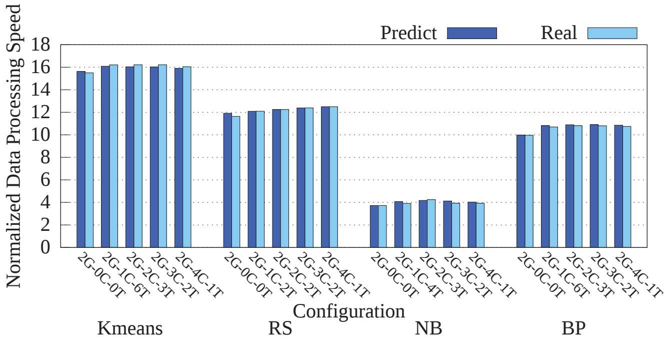

Figure 13: Predicted vs. measured data processing speeds. (Normalized to dps on Hadoop.)

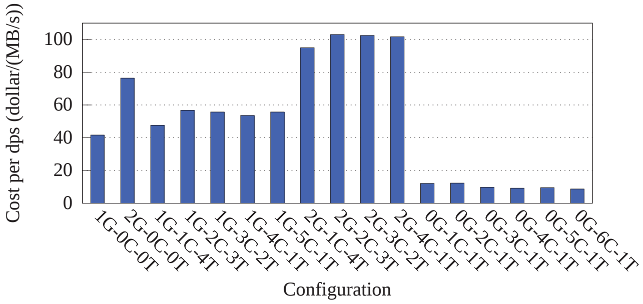

Figure 14: Cost efficiency for KNN.

Now we seek for the most cost-effective resource configuration for each application. Given a MapReduce application and a platform with p CPU cores and n GPUs, for a given resource configuration, its cost-efficiency is defined as the cost divided by the performance:

$$\begin{aligned} cost\_efficiency(k) &= cost(k)/performance(k) \\ &= (G*g+C*c*t)*(V/dps)/dps \end{aligned} \quad (8)$$

where G and C represent the cost per time unit for a GPU card and a CPU core respectively. For a given configuration k, it uses g GPUs and $c*t$ CPU cores, thus $G*g+C*c*t$ is the accumulated cost per time unit of the configuration k, and V/dps is the execution time. So $(G*g+C*c*t)*(V/dps)$ represents the total cost for completing the application, and dps is used for representing the performance.

Figure 14 presents the cost efficiency for KNN, which shows that the most cost-effective configuration is using only CPUs (0G-6C-1T). Table 2 lists the most cost-effective configurations obtained using our model for the 5 applications. NB is similar to KNN, with the 0-GPU configuration being the most cost-effective. For Kmeans, BP and RS, leveraging GPU would increase the cost efficiency, with Kmeans and BP using 1 GPU, and RS using 2 GPUs.

Table 2: The most cost-effective configurations.

Benchmark	g	c	t
KNN	0	6	1
Kmeans	1	0	0
RS	2	0	0
NB	0	4	1
BP	1	2	3

6. RELATED WORK

There has been a lot of work on developing MapReduce frameworks on modern architectures, especially for GPU-involved heterogeneous platforms/clusters.

On single-node GPGPU platform, Catanzaro et al presented a MapReduce framework which executes user-provided mapper and reducer CUDA kernels on a GPU [7]. He et al proposed Mars, which provides a similar MapReduce programming API and introduces a sort stage to sort the intermediate (key, value) pairs [20]. Hong et al proposed MapCG, another framework which can port user-provided MapReduce programs to either CPU or GPU [21]. Comparing with Mars, MapCG proposed a GPU-based hash table, to avoid sorting intermediate (key, value) pairs before reduce [21]. Furthermore, Ji et al, Chen et al proposed to use shared memory to optimize MapReduce framework on GPUs [22, 8]. All these related work are single-node MapReduce frameworks, requiring the data to be processed can fit into a single GPU's memory.

On GPU clusters, Stuart and Owen proposed GPMR, a stand-alone C++-based MapReduce framework, which enables user-written CUDA kernels. Meanwhile, GPMR introduced a number of intermediate stages, such as partial reduction and accumulation to minimize inter-node communication. Furthermore, GPMR also supports multiple GPUs per node with dedicated processes for each GPU [32].

On CPU/GPGPU hybrid clusters, HAPI [29] and HadoopCL [19] are representative work. Both of them use APARAPI [18] to automatically generate GPU OpenCL kernels from Java code. In particular, HAPI provides a heterogeneous mapper class, including preprocess, gpu and gpuprocess, which are provided by users. After users provide these class implementations in Hadoop, HAPI can automatically generate the corresponding OpenCL mapper kernel on GPU using APARAPI. At runtime, the HAPI would invoke the generated OpenCL kernel via JNI. HadoopCL further enhances HAPI to provide more friendly programming interface for programmers. Glasswing [16] is another MapReduce framework using OpenCL to exploit multi-core CPUs and accelerators. It uses pipeline to overlap computation, communication, memory transfer and disk access, and exploits fine-grained parallelism within each stage by taking advantage of a variety of devices.

HAPI and HadoopCL are most closely related to our work. However, our work has the following differences: 1) Hadoop+ takes the shared resource contention into consideration and provides a heterogeneity model, to guide users to select their desired resource utilization for the purposes of improving performance or cost-efficiency, or to guide the GPU allocation across multiple simultaneously running MapReduce applications. Thus users can select specific computing resources under different purposes. 2) Hadoop+ enables users to provide explicitly parallel functions as plug-ins, thus it can easily integrate existing high-performance libraries into a large-scale data-processing framework, thus facilitating exploitations of GPUs in the Hadoop framework. 3) Hadoop+ enables the original *Map* and *Reduce* functions in Hadoop to co-exist with user-provided explicitly parallel *PMap* and *PReduce* functions. Thus users can take existing CUDA/ OpenCL codes, plug it into existing Hadoop programs, to make the applications coordinately run on both CPUs and GPUs. For example, in our evaluation, we directly download the CUDA code of K-means from [2] and plug it into the Hadoop implementation of K-means in mahout [1], thus facilitating programming for heterogeneous clusters.

Performance degradation caused by shared resource contention is another related area of our work. There has been a lot of work addressing contentions on shared cache [10, 23, 38], memory bandwidth [15, 37, 36], memory subsystem [26, 27, 40, 17, 33, 34, 39], and I/O resource [6, 35, 24]. Meanwhile, GPUPerf [31] has been developed to predict the performance and understand bottlenecks of GPGPU applications. Our work focuses on MapReduce applications and models their behaviors in heterogeneous clusters.

7. CONCLUSION AND FUTURE WORK

This paper presents a heterogeneous MapReduce framework, Hadoop+, which enables user-provided explicit parallel functions written in native languages to be embedded in Hadoop as plug-ins. Furthermore, we leverage Hadoop+ to model the heterogeneity for a MapReduce application, thus determining the optimal or most cost-effective resource configuration for an application. We evaluate the model by using 5 real-world big data applications and demonstrate that Hadoop+ enables users to exploit GPU capability, thus achieving 1.4x–16.1x speedups over Hadoop for the 5 real applications. Moreover, the heterogeneity model can be used to allocate GPUs among multiple simultaneously running MapReduce applications, bringing up to 36.9% (17.6% in average) performance improvement.

In future work, we plan to take the non-data-local tasks into consideration. And we will extend this work to model the heterogeneous resource contention for more distributed computing applications, e.g., MPI applications, and for more architectures, e.g., closely integrated on-chip GPUs. Furthermore, we will consider to automatically generate CUDA codes for MapReduce applications [13].

Acknowledgment

This research is supported by the National High Technology Research and Development Program of China under grant No. 2012AA010902 and 2015AA011505; the NSFC under grants No. 61202055, 61221062,61303053, 61432016 and 61402445; and the National Basic Research Program of China under grant No. 2011CB302504; Australian Council Research (ARC) Grants, DP110104628 and DP130101970. We would like to thank all the reviewers for their valuable comments and suggestions.

8. REFERENCES

[1] Apache mahout. http://mahout.apache.org/.

[2] Efficient algorithms for k-means clustering. http://www.cs.umd.edu/ mount/Projects/KMeans/.

[3] Hadoop Yarn. In *http://hadoop.apache.org/docs/r2.3.0/ hadoop-yarn/hadoop-yarn-site/YARN.html*.

[4] Official Price for Intel Xeon E5-2620. http://ark.intel.com/products/64594/Intel-Xeon-Processor-E5-2620-15M-Cache-2_00-GHz-7_20-GTs-Intel-QPI.

[5] Apache Hadoop. In *http://lucene.apache.org/hadoop/*, 2006.

[6] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross. Understanding and improving computational science storage access through continuous characterization. In *MSST*, 2011.

[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] L. Chen and G. Agrawal. Optimizing mapreduce for gpus with effective shared memory usage. In *HPDC*, 2012.

[9] R. Chen, H. Chen, and B. Zang. Tiled-mapreduce: Optimizing resource usages of data-parallel applications on multicore with tiling. In *PACT*, pages 523–534, 2010.

[10] S. Cho and L. Jin. Managing distributed, shared L2 caches through OS-level page allocation. In *MICRO*, 2006.

[11] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Trans. Inf. Theor.*, 13(1):21–27, Sept. 2006.

[12] H. Cui, G. Ruan, J. Xue, R. Xie, L. Wang, and X. Feng. A collaborative divide-and-conquer k-means clustering algorithm for processing large data. In *CF*, 2014.

[13] H. Cui, L. Wang, J. Xue, Y. Yang, and X. Feng. Automatic library generation for blas3 on gpus. In *IPDPS*, 2011.

[14] J. Dean and S. Ghemawat. ”mapreduce: simplified data processing on large clusters”. In *OSDI*, 2004.

[15] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. Bandwidth Bandit: Quantitative characterization of memory contention. In *CGO*, 2013.

[16] I. El-Helw, R. Hofman, and H. E. Bal. Scaling mapreduce vertically and horizontally. In *SC'14*, pages 525–535, 2014.

[17] A. Fedorova, M. Seltzer, and M. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *PACT*, 2007.

[18] G. Frost. Aparapi in amd developer websize. In *”http://developer.amd.com/tools/heterogeneous-computing/aparapi/”*.

[19] M. Grossman, M. Breternitz, and V. Sarkar. ”hadoopcl: Mapreduce on distributed heterogeneous platforms through seamless integration of hadoop and opencl”. In *IPDPSW*, 2013.

[20] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. ”mars: A mapreduce framework on graphics processors”. In *PACT*, 2008.

[21] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin. Mapcg: writing parallel program portable between cpu and gpu. In *PACT*, 2010.

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

[23] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *HPCA*, 2008.

[24] Y. Liu, R. Gunasekarany, X. Ma, and S. S. Vazhkudai. Automatic Identification of Application I/O Signatures from Noisy Server-Side Traces. In *FAST*, 2014.

[25] Z. Liu, H. Li, and G. Miao. Mapreduce-based backpropagation neural network over large scale mobile data. In *ICNC*, 2010.

[26] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In *MICRO*, 2011.

[27] J. Mars, L. Tang, and M. L. Soffa. Directly characterizing cross-core interference through contention synthesis. In *HiPEAC*, 2011.

[28] NVIDIA. Official Price for Tesla C2050. In *http://www.nvidia.com/object/io_1258360868914.html*.

[29] S. Okur, C. Radio, and Y. Lin. ”hadoop+aparapi: Making heterogeneous mapreduce programming easier”. In *”https://netfiles.uiuc.edu/okur2”*.

[30] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. ”evaluating mapreduce for multi-core and multiprocessor systems”. In *HPCA*, 2007.

[31] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc. A performance analysis framework for identifying potential benefits in gpgpu applications. In *PPoPP*, 2012.

[32] J. A. Stuart and J. D. Owens. Multi-gpu mapreduce on gpu clusters. In *IPDPS*, 2011.

[33] L. Tang, J. Mars, and M. L. Soffa. Compiling For Niceness Mitigating Contention for QoS in Warehouse Scale Computers. In *CGO*, 2012.

[34] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The Impact of Memory Subsystem Resource Sharing on Datacenter Applications. In *ISCA*, 2011.

[35] A. Uselton, M. Howison, N. Wright, D. Skinner, N. Keen, J. Shalf, K. Karavanic, and L. Oliker. Parallel I/O performance: From events to ensembles. In *IPDPS*, 2010.

[36] D. Xu, C. Wu, P. Yew, J. Li, and Z. Wang. Providing Fairness on Shared-Memory Multiprocessors via Process Scheduling. In *SIGMETRICS*, 2012.

[37] D. Xu, C. Wu, and P.-C. Yew. On mitigating memory bandwidth contention through bandwidth-aware scheduling. In *PACT*, 2010.

[38] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In *EuroSys*, 2009.

[39] J. Zhao, H. Cui, J. Xue, X. Feng, Y. Yan, and W. Yang. An empirical model for predicting cross-core performance interference on multicore processors. In *PACT*, 2013.

[40] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ASPLOS*, 2010.