FlinkCL: An OpenCL-Based In-Memory Computing Architecture on Heterogeneous CPU-GPU Clusters for Big Data

Cen Chen, Kenli Li, *Senior Member, IEEE*, Aijia Ouyang, and Keqin Li, *Fellow, IEEE*

Abstract—Research on in-memory big data management and processing has been prompted by the increase in main memory capacity and the explosion in big data. By offering an efficient in-memory distributed execution model, existing in-memory cluster computing platforms such as Flink and Spark have been proven to be outstanding for processing big data. This paper proposes FlinkCL, an in-memory computing architecture on heterogeneous CPU-GPU clusters based on OpenCL that enables Flink to utilize GPU's massive parallel processing ability. Our proposed architecture utilizes four techniques: a heterogeneous distributed abstract model (HDST), a Just-In-Time (JIT) compiling schema, a hierarchical partial reduction (HPR) and a heterogeneous task management strategy. Using FlinkCL, programmers only need to write Java code with simple interfaces. The Java code can be compiled to OpenCL kernels and executed on CPUs and GPUs automatically. In the HDST, a novel memory mapping scheme is proposed to avoid serialization or deserialization between Java Virtual Machine (JVM) objects and OpenCL structs. We have comprehensively evaluated FlinkCL with a set of representative workloads to show its effectiveness. Our results show that FlinkCL improve the performance by up to $11\times$ for some computationally heavy algorithms and maintains minor performance improvements for a I/O bound algorithm.

Index Terms—Big data, GPGPU, heterogeneous cluster, in-memory computing, OpenCL

1 Introduction

1.1 Motivation

The increase in main memory capacity, together with the decreasing price of main memory and the explosion of big data has stimulated the development of in-memory big data management and processing. Over the past few years, in-memory cluster computing platforms, such as Flink [1] and Spark [2] have been widely adopted for analyzing large-scale data in both academia and industry because of their outstanding features compared to other traditional cluster computing frameworks, such as Hadoop. As their distributed in-memory execution model saves an enormous amount of time for disk I/O operations, these platforms are more efficient than traditional platforms, particularly for data mining and machine learning which require many iterative operations [2].

The use of heterogeneous CPU-GPU clusters is becoming mainstream now, especially in high-performance computing (HPC) due to their high computational power and energy efficiency. This shift is apparent in the current ranking of supercomputers on the market. Heterogeneous CPU-GPU computers dominate both the Top 500 (www.top500.org, 2015) and the Green 500 (www.green500.org, 2015). Many researchers have focused on implementing algorithms or accelerating algorithms using GPUs or hybrid CPU-GPU systems [3], [4], [5], [6].

However, open-source versions of in-memory cluster computing platforms such as Flink and Spark can only run on CPUs at this moment. That is, these platforms cannot utilize the available efficient computing resources of GPUs that may be present in the cluster. Furthermore, there exist many challenges for processing big data on large-scale heterogeneous CPU-GPU clusters and integrating GPUs into Flink, such as programmability, reliability and practicability.

Programmability. The current programming model of heterogeneous CPU-GPU clusters differs from that of Flink. In current programming models for heterogeneous CPU-GPU clusters (e.g., MPI [7] plus OpenMP [8], plus CUDA or OpenCL), programmers must manually tune low-level code based on the specific device architecture (e.g., cache misses and memory overflow errors), and handle the communication and distribution of workloads among the worker nodes in the cluster and many cores in a single node. These low-level and error-prone operations increase the burden on programmers.

Reliability. As systems scale up, the time between hardware failures tends to decrease. Reliability is thus the main driver for constructing our system, FlinkCL, on top of Flink. Flink has a robust job management system that uses replication and error detection to schedule around failures. The essence of Flink's fault tolerant mechanism lies in its ability

- *C. Chen and K. Li are with the College of Information Science and Engineering, National Supercomputing Center in Changsha, Hunan University, Changsha, Hunan 410082, China. E-mail: {chencen, lkl}@hnu.edu.cn.*
- *A. Ouyang is with the Department of Information Engineering, Zunyi Normal College, Zunyi, Guizhou 563006, China. E-mail: oyaj@hnu.edu.cn.*
- *K. Li is with the College of Information Science and Engineering, National Supercomputing Center in Changsha, Hunan University, Changsha, Hunan 410082, China, and also with the Department of Computer Science, State University of New York, New Paltz, NY 12561. E-mail: likq@hnu.edu.cn.*

Manuscript received 15 June 2017; revised 28 Apr. 2018; accepted 7 May 2018. Date of publication 23 May 2018; date of current version 7 Nov. 2018. (Corresponding author: Kenli Li.)
Recommended for acceptance by F. Douglis.
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/TC.2018.2839719

to take consistent snapshots of the distributed data stream and operator status. These snapshots server as consistent checkpoints to which the system can fall back in case of a failure [9].

Practicability. Practicability is an important issue in practical big data applications and a challenging issue for programmers. Current in-memory cluster platforms provide many practical functionalities, such as supporting the Hadoop Distributed File System (HDFS), batch and streaming processing, and easy programming models [1], [2].

1.2 Our Contributions

Our previous work [10] has already integrated CUDA into Flink. However, this work had many limitations. First, the framework proposed in the previous study only supported NVIDIA GPUs. Second, the same tasks could not be executed on both CPUs and GPUs simultaneously. Because of the different characteristics of CPUs and GPUs, each is more suited for certain tasks than the other. Thus, GFlink underutilized the computing power of both CPUs and GPUs in the cluster. Third, to utilize the proposed framework, programmers were required to write both Java programs and CUDA programs, which is relatively complicated. Fourth, all the intermediate results in the GPUs generated by the Map phases had to be transferred to the main memory prior to shuffling, thus requiring large-scale communication over the (Peripheral Component Interconnect Express) PCIe bus, especially for shuffle-intensive applications.

Considering these issues, in this study, we propose FlinkCL: an in-memory computing architecture on heterogeneous CPU-GPU clusters on top of Flink and OpenCL. OpenCL and CUDA have emerged as mainstream languages for programming on GPUs. Because OpenCL is a general-purpose standard, in contrast to CUDA, which can only run on NVIDIA GPUs, it is supported by many heterogeneous processors, such as CPUs, NVIDIA GPUs, AMD GPUs, Intel many-core devices (MICs) and field-programmable gate arrays (FPGAs). An important reason for selecting OpenCL as the basis for the entire framework is to enable future expansion, which can help us ultimately establish a unified in-memory computing architecture for heterogeneous clusters.

We have analyzed and identified the challenges for effectively using GPUs in current distributed in-memory data processing systems, and FlinkCL is carefully designed to address these challenges. Our proposed architecture is compatible with both the compile-time and runtime of Flink, thereby inheriting the existing outstanding features of Flink, including high reliability, good fault tolerance, easy programming model and distributed file system. In summary, this paper makes the following contributions.

- *Programmability*: Through our programming framework on the heterogeneous DataSet (HDST), programmers only need to program in Java utilizing simple interfaces provided by FlinkCL. They do not need to consider error-prone and low-level issues such as memory management, load balance and communication on clusters. The user-defined Java codes can be compiled to OpenCL kernels and executed on heterogeneous CPUs and GPUs automatically.
- *Efficient Communication Strategy*: We have identified communication as a critical consideration in in-memory heterogeneous CPUs and GPUs computing. Thus, we propose a novel data mapping scheme to avoid serialization and deserialization between the JVM objects and OpenCL structs. Furthermore, our proposed framework allows programmers to design the data layout themselves, and utilize the memory hierarchy in GPUs, thus improving memory throughput in GPUs. In addition, we propose a hierarchical partial reduction to reduce the communication overhead during the Shuffle phase.
- *Heterogeneous Task Management*: We use a heterogeneous task management scheme to allow CPUs and GPUs to cooperate and simultaneously execute tasks. To achieve good load balance among heterogeneous CPUs and GPUs, we propose an auto-tuning partitioning scheme and a dynamic load balancing scheme.

The remainder of this paper is organized as follows. Section 2 provides background information and reviews related work. Section 3 describes the design and architecture of FlinkCL. Section 4 presents the details of our core techniques. Section 5 shows the performance results of FlinkCL. Section 6 concludes the paper.

2 Related Work and Background

2.1 GPGPU and Aparapi

GPUs were extended to the general-purpose HPC area after the emergence of general purpose GPU (GPGPU) under support of several frameworks, including CUDA and OpenCL. OpenCL is the open standard for cross-platforms, and parallel programming of heterogeneous processors (such as CPUs, GPUs, and accelerated processing units). Both AMD and NVIDIA have released OpenCL implementations that support their respective GPUs. Programmers write OpenCL applications with two portions of code-functions to be executed on the CPU host and functions to be executed on the GPU device, called OpenCL kernel. Because the host and kernel codes are executed in two different memory spaces, explicit data transfers between the host and GPUs are necessary.

An important performance optimization method is to coalesce the global memory accesses generated by streaming multiprocessors (SMs). Many studies have focused on optimizing data layout to improve the performance [11], [12], [13]. Generally, there are three types of data layouts: Array-of-Structures (AoS), Structure-of-Arrays (SoA) and Array-of-Primitive (AoP). The efficiency of the same GPU application can drastically differs depending on the type of data layout used [14], [11].

Aparapi [15], an open-source tool developed by AMD, provides Just-In-Time (JIT) compilation of Java bytecode to OpenCL kernels and execution of those kernels on heterogeneous devices. The compilation and execution of OpenCL kernels, OpenCL memory allocation, data transfers and kernel invocation are handled by Aparapi and thus become transparent to Java programmers. However, Aparapi has several limitations (for instance, it only supports primitive types or single-dimensional arrays of primitives).

2.2 MapReduce on GPUs

Due to the high computational power and memory bandwidth of GPUs, many previous studies have focused on accelerating MapReduce with GPUs. Fang et al. [16] proposed Mars, a MapReduce runtime system accelerated with GPUs, and integrated it into Hadoop for cluster computing. Stuart et al. [17] presented GPMR, a MapReduce library that leverages the power of GPU clusters for large-scale computing. Chen et al. [18] accelerated the MapReduce model on a coupled CPU-GPU architecture to make full use of both CPU and GPU computing resources. Max et al. [19] integrated Aparapi into Hadoop to process big data. Helw et al. [20], [21] proposed a MapReduce framework Glasswing which uses OpenCL to exploit multi-core CPUs and accelerators. Glasswing involves a 5-stage pipeline that overlaps computation, communication between cluster nodes, memory transfers to compute devices, and disk access in a coarse grained manner. In addition, Glasswing utilizes fine-grained parallelism within each node to target modern multi-core and many-core processors.

FlinkCL has several differences and advantages compared to the other strategies that can be used to implement MapReduce on GPUs and integrate GPUs into Hadoop. First, FlinkCL inherits the advantages of in-memory computing provided by Flink. Second, FlinkCL provides a series of high-level transformation and action interfaces (e.g., Map, Reduce, Join, Group and Count) that simplify the work of programmers. Third, studies which did not integrate GPUs into Hadoop, had poor fault tolerance and usability. Flink has a robust job management system that uses replication and error detection to schedule around failures [9].

2.3 In-Memory Cluster Computing on GPUs

Yuan et al. [22] proposed a system based on Spark called Spark-GPU to accelerate in-memory data processing on clusters. In Spark-GPU, a scheme named GPU-RDD integrates GPUs into Spark. GPU-RDD only aims at conduct data processing on GPUs; that is, the same tasks cannot be executed on both CPUs and GPUs concurrently. Second, the data to be processed in GPUs must be transferred from memory in JVM to the native memory and then from the native memory to the device memory of the GPUs, thus incurring extra copy overhead. Third, although the data can be cached in the GPUs, the data generally needs to be transferred to the GPUs in most cases due to the limited size of the device memory of GPUs.

Li et al. [23] proposed a heterogeneous CPU-GPU Spark platform for machine learning algorithms. However, the communication between CPU memory and GPU memory is based on remote method invocation (RMI), including serialization and deserialization. This scheme introduces large extra overhead. Most importantly, to utilize the proposed framework, programmers must write not only Java programs but also CUDA programs, increasing the complexity of the entire process.

3 Design and Architecture

3.1 Challenges for Integration

Generally, applications on Flink usually feature rich data parallelism, which matches GPU's single instruction multiple data (SIMD) execution model. However, because the properties of GPUs and Flink differ, it is non-trivial to integrate GPUs into Flink efficiently. The challenges for integration are listed below.

Communication. The CPU and GPU work in master-slave mode, with the CPU as the master and the GPU as the slaves. The CPUs and GPUs have separate memory spaces. Therefore, it is necessary to explicitly transfer data between the main memory and the device memories of the GPUs. Because of the JVM's memory management mechanisms, including the JVM's garbage collection (GC) function, the virtual addresses and actual physical addresses of objects in the JVM are not fixed and are invisible to programmers. Therefore, data or objects in JVMs cannot be directly transferred to GPUs by PCIe bus. With regards to transferring arrays of primitives to and from the GPUs, Aparapi uses a number of type-specific methods. For example, *Kernel.put(int[] arr)* and *Kernel.get(int[] arr)* are used to transfer integer arrays. The natural method involves first converting JVM objects into arrays of primitives manually. Once the results are transferred from the GPUs to JVMs, the primitive arrays must be transformed into objects. Manual serialization and deserialization can result in large-scale overhead.

The Shuffle phase in MapReduce performs an all-to-all copying of intermediate data from the Map phase to the Reduce phase. It involves intensive communication between Mappers and Reducers, which can significantly delay job completion [24], [25], [26]. However, the PCIe link connecting the main memory and the device memory of the GPUs has limited bandwidth. This limited bandwidth can often form a bottleneck for the computations [27]. For example, PCIe Gen 3 has a theoretical maximum throughput of 15.75 GB/s, far lower than the memory bandwidth on the CPU-side. Therefore, the intensive communication involved in the Shuffle phase can greatly affect performance.

Data Layout. Data layout is the form in which data should be organized and accessed in memory when operating on multi-valued data. Different GPU kernels are appropriate for different types of data layouts [14], [28]. Global memory accesses always coalesce when using the columnar format (SoA). However, [14] and [12] have shown that AoS is a better choice than SoA in some applications in which the structure sizes map to one of several intrinsic vector types on the GPU (e.g., int4, float4) that are fully supported for coalesced read/write by the underlying GPU global memory system. Moreover, for GPU local memory, only those AoS cases that have a stride that does not create so-called local memory bank conflicts can function without performance loss. The selection of an appropriate data layout is crucial in the development of GPU-accelerated applications.

Programmability. Another challenge is that the programming model of GPGPU is different from that of Flink. Using Flink, programmers only need to implement certain high-level interfaces (e.g., Map and Reduce) without considering factors such as communication, fault tolerance, and data synchronism. In contrast, in the programming model of GPGPU (e.g., OpenCL), programmers must write host code and kernel code. In particular, as in big data processing, distributed computing is a necessity. However, OpenCL is just a programming mode on a single node.

Task Management. In a heterogeneous environment, the computing power of CPUs differs from that of GPUs. In

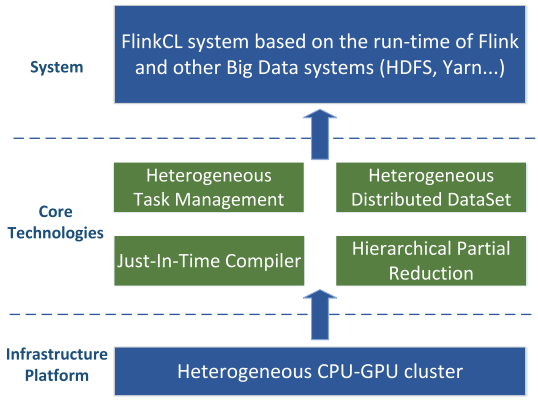

Fig. 1. Core techniques.

addition, the computing power also varies with the nature of the computations performed by different applications. We need to investigate how to effectively utilize both CPUs and GPUs to accelerate a single MapReduce application to achieve good load balance and thus ultimately improve the performance.

3.2 Core Techniques

To solve the problems described above and in Section 1.1, we designed four core techniques and integrated them into Flink as shown in Fig. 1. These techniques ensure that FlinkCL can run on heterogeneous and is compatible with both the compile-time and the runtime of Flink.

Heterogeneous Distributed Data Model. A heterogeneous abstract data model DST (HDST) based on Flink's existing abstract data model DST is proposed to combine the computing model of GPUs and Flink. The HDST objects denote the data stored in the distributed managed memory in FlinkCL, in the form of user-defined HJavaBean. A novel data mapping scheme is proposed to allow the raw data bytes of HJavaBean to be stored in the distributed managed memory; these layouts of raw bytes can then be directly mapped to the OpenCL structs. Therefore, these raw bytes can be transferred to the GPUs without modification. This scheme can avoid serialization and deserialization between the JVM objects and OpenCL structs, thus greatly enhancing the performance.

JIT Compiling Scheme. We improved the Aparapi's code-generator functionality to support the data mapping scheme and automatically transform HJavaBeans to OpenCL structs. Next, we designed an Aparapi bridge to fill the gap between hMapper/hReducer and Aparapi's Runnable interface. This scheme allows the user-defined Java programs based on our provided APIs to be compiled to OpenCL kernels during the runtime, which can be executed on almost all available GPUs.

Hierarchical Partial Reduction (HPR). A partial reduction scheme is utilized to reduce the communication overhead during the Shuffle phase. The main idea is to conduct Reduce as soon as some of the intermediate results are generated by the mappers when the reduction operations are commutative and associative. In accordance with the hierarchical communication in the CPU-GPU cluster, an HPR strategy is proposed.

Heterogeneous Task Management Scheme. Our proposed heterogeneous task management is responsible for managing CPU computing and GPU computing (e.g., OpenCL kernel execution, data transfers and environment management for GPUs). To improve the performance, an auto-tuning partitioning algorithm and a dynamic load balancing scheme are proposed to achieve good load balance among the heterogeneous CPUs and GPUs.

3.3 Overall Architecture

The overall architecture is shown in Fig. 2. Analogous to the existing architecture of Flink, our proposed FlinkCL architecture also contains a client, a master and workers, all of which are based on the runtime of Flink, thereby preserving the compatibility with existing platforms, as well as many practical features, such as fault tolerance, distributed file system, in-memory computing and streaming computing.

When the FlinkCL system is started, it brings up one JobManager in the master, which serves as the coordinator of the FlinkCL system. Developers only need to write one version of the driver programs using the interfaces provided by FlinkCL. Next, the data to be processed will be stored in the distributed memory in raw bytes through the proposed HDST scheme. Meanwhile, the tasks defined in the driver program will be scheduled to the workers in the cluster by the

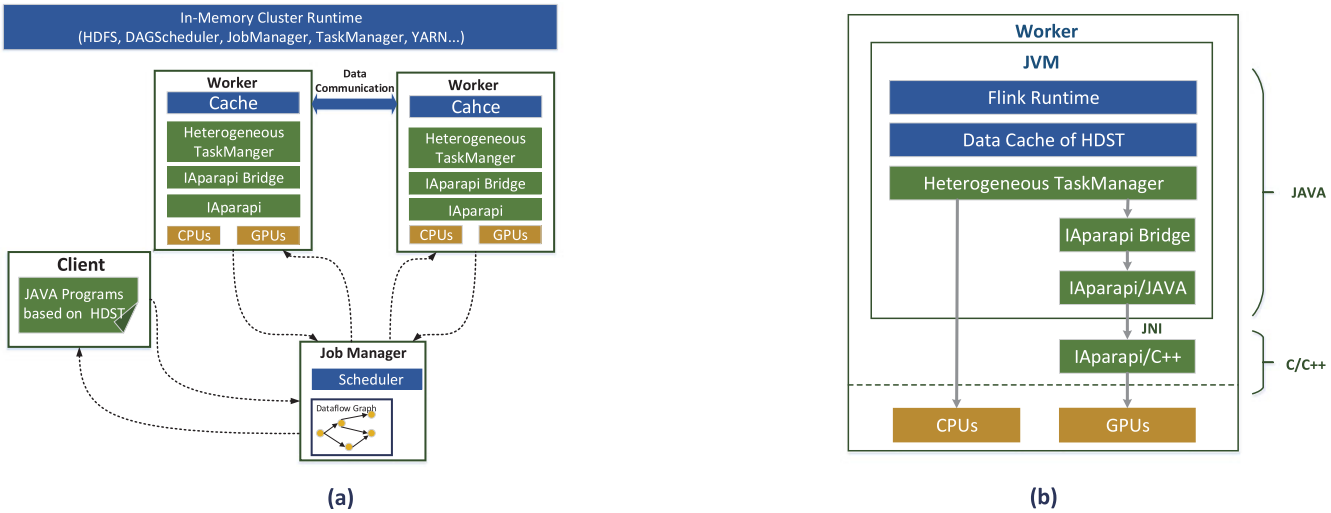

Fig. 2. Architecture of FlinkCL (a) Overall architecture. (b) System components in workers.

```
//Example for AOS layout
public class PointAOS extends HJavaBean_4 {
    @Field(order = 0, array = false)
    public float x;

    @Field(order = 1, array = false)
    public float y;
}
HDataSet<PointAOS> points;

//Example for SOA layout
public class PointSOA extends HJavaBean_4 {
    @Field(order = 0, array = true, length = 1000)
    public float x[1000];

    @Field(order = 1, array = true, length = 1000)
    public float y[1000];
}
HDataSet<PointSOA> points;
```

Fig. 3. Definition of HJavaBean.

master. The tasks will then be scheduled to be executed on the CPUs or GPUs by our proposed heterogeneous TaskManager. If the tasks need to be executed on GPUs, the user-defined programs will be compiled to OpenCL kernels by our proposed JIT compiling scheme and then invoked to be further executed on GPUs by the heterogeneous TaskManager.

3.4 Main System Components in a Worker

We modified the original Flink's functionality to realize our proposed data caching scheme for HDST and added three main system components to each worker: the Improved-Aparapi (IAparapi), IAparapi Bridge and Heterogeneous TaskManager, as shown in Fig. 2.

IAparapi. We improved the Aparapi's code-generator functionality so that the user-defined HJavaBeans can be transformed to OpenCL structs automatically to support our proposed data mapping scheme. Furthermore, we also modified Aparapi's original execution flow so that we can identify the specific device by API to execute the OpenCL kernel. IAparapi contains two components: IAparapi in Java and IAparapi in C++. These two components communicate with each other through Java Native Interface (JNI).

IAparapi Bridge. The IAparapi Bridge is used to fill the gap between hMapper/hReducer and Aparapi's runnable interfaces. IAparapi and IAparapi Bridge cooperate to fulfill the JIT compiling functions.

Heterogeneous TaskManager. The heterogeneous TaskManager is in charge of managing the CPU computing and GPU computing (e.g., OpenCL kernel execution, data transfers and automatic environment management for GPUs). The scheduling strategy contains an auto-tuning partitioning algorithm and a dynamic load balancing scheme that reside in the heterogeneous TaskManager to achieve load balance between the heterogeneous CPUs and GPUs.

3.5 Programming Framework

3.5.1 Programming on Heterogeneous Distributed DataSet (HDST)

In-memory cluster computing (e.g., Flink and Spark) provides an abstract model for distributed data (e.g., the Resilient Distributed Dataset (RDD) [2] in Spark, and the DataSet (DST) in Flink). This abstract data model offers user-defined functions (UDFs) composed of a series of high-level transformation interfaces and action interfaces (e.g., Map, Reduce, Join and Group), thereby enabling programmers to work more easily. DSTs represent a collection of distributed items of user-defined type (UDT) that can be concurrently manipulated across many computing nodes. The kernel computation in Flink and Spark is a MapReduce model that contains two high-level stages: Map and Reduce.

Our proposed HDST is based on the abstract data model DST of Flink and inherits all of the existing functions of DST from Flink. Programmers can define HDST objects and create these objects from HDFS or other HDST objects on top of the user-defined HJavaBean.

The main goal of our proposed FlinkCL is to improve the computational performance of applications on Flink by leveraging the GPU's high computing power, meanwhile retaining the convenient programming model. To leverage the GPU's computing resources to process big data by utilizing FlinkCL, we need to follow several steps that are similar to those for Flink, as follows.

- Define HDST objects based on our proposed HJavaBean, and then define hMappers, hReducers for these HDST objects (in Java).
- Implement hMappers and hReducers (in Java).

3.5.2 Heterogeneous JavaBean (HJavaBean)

HJavaBean is similar to JavaBean. However, in contrast to JavaBean, the position and the length of the array must be addressed in HJavaBean. The member variables can be defined as primitive types (int, $float$, $long$, $double$, and so on), or object references for primitive types, other HJavaBeans or arrays of these types. Utilizing HJavaBean, programmers can organize the data layout and define the type of alignment. The raw bytes of the user-defined HJavaBean are stored in a sequential manner, as defined by HJavaBean. By default, the data layout is AoS. Programmers can define arrays in the user-defined HJavaBean and then the data layout will become SoA, similar to the columnar format. One SoA is a sub-region, and all the sub-regions in the cluster constitute an entire dataset. If the arrays in SoA are separated, the data layout will become AoP.

Coalesced memory access has long been advocated as one of the most important off-chip memory access optimizations for modern GPUs. Programmers can design the data layout by themselves depending on the actual situation. Fig. 3 illustrates how to define HJavaBean for AoS and SoA data layouts, as well as for HDST. In the definition of $PointAOS$, $order = 0$ is used to indicate that the position of the member variable x is 0. For the definition of $PointSOA$, $array = true, length = 1000$ is used to indicate that the position of member variable $x[1000]$ is an array, the length of which is 1000. $HJavaBean\_4$ indicates that the size of the alignment is 4 bytes.

HJavaBean has several limitations. Inheritance function is not supported. That is, no HJavaBean can inherit other superclasses. Moreover, some built-in structures (such as ArrayList and Map) are not supported by HJavaBean.

3.5.3 Heterogeneous Mapper and Reducer

Flink provides UDF with user-implemented transformation functions (e.g., Map, Reduce, FlatMap, and Join) and action functions (e.g., Count and Save) for the abstract model DST.

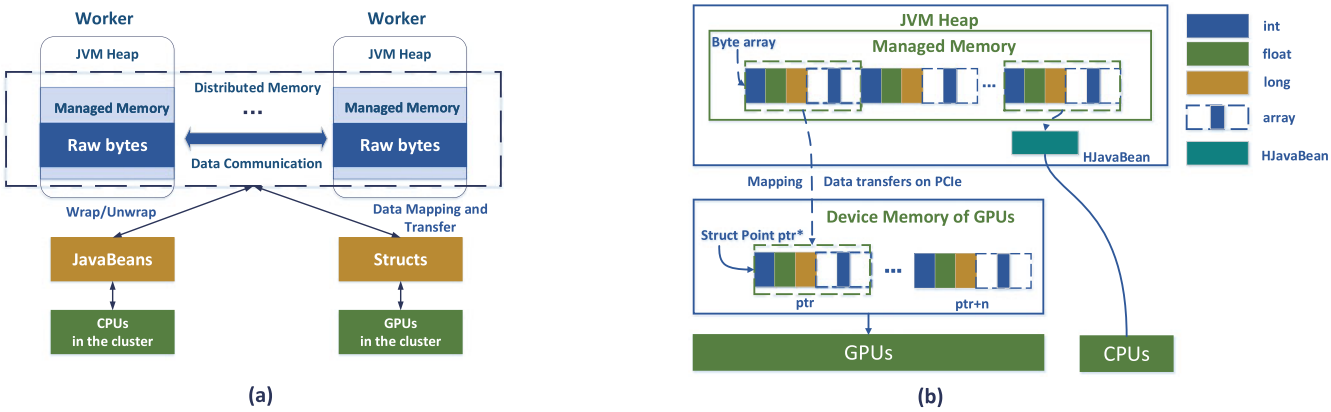

Fig. 4. Efficient communication strategy (a) Overview of HDST in a cluster. (b) Details of data mapping scheme in a work node.

In practical applications, most computing tasks are encapsulated in user-implemented Mappers and Reducers. Therefore, we have added heterogeneous user-implemented Map and Reduce interfaces (e.g., hMap, hReduce, and hFlatMap) and heterogeneous bulk execution interfaces (e.g., hMapBulk and hReduceBulk). In terms of executions on GPUs, user-defined programs are complied to OpenCL kernels via JIT and further executed on GPUs using kernel launches. Other functions such as Join and Count can also be implemented in GPUs but are beyond the scope of this paper.

3.5.4 Bulk Execution for the Heterogeneous Mapper and Reducer

A GPU is a massively parallel co-processor that executes GPU kernels in a single instruction multiple threads (SIMT) manner. To maximize a GPU's performance, each GPU kernel should be launched with a large number of GPU threads that can utilize GPU computing resources and hide GPU memory access latency to achieve high throughput. To meet this requirement, the system must support a bulk processing model that processes one block of data elements at a time. However, Flink does not satisfy these two requirements. Flink adopts the iterator model and computes one element at a time using the row format, which significantly underutilizes GPU resources. In this case, to efficiently harness GPU resources, it is essential to transfer the data to GPUs in chunks and invoke block processing simultaneously.

Therefore, we recommend storing many elements in a HJavaBean and utilizing the bulk execution interfaces provided by FlinkCL (hBulkMap and hReduce). We also provide an automatic scheme to group a bulk of HJavaBean objects in an array (such as the AoS data layout) and invoke bulk execution on both CPUs and GPUs.

4 Implementation Details

4.1 Heterogeneous Distributed DataSet

The abstract layer of the JVM object references also renders the physical addresses and data layout of objects invisible and transparent to programmers. However, the OpenCL kernel can only process primitives and structs, or arrays of primitives and structs, rather than the JVM objects, while Aparapi merely supports primitives and arrays of primitives. A natural solution would be to initially transform JVM objects into primitives or arrays of primitives manually. In our experiments, we have discovered that the manual conversions between JVM objects and primitives or arrays of primitives not only incur heavy burdens for programmers but also result in serious overheads.

Flink itself provides a distributed managed memory scheme. Thanks to this scheme, the pressure on the GC function provided by JVM is reduced, thus improving and smoothing performance. The managed memory spaces are allocated as $byte[]$ segments in the JVM heap or off the JVM heap in each worker when FlinkCL is started. We have utilized and enhanced this scheme.

4.1.1 Data Caching of HDST

HDST objects denote the data that is cached in the distributed managed memory in FlinkCL in the form of user-defined HJavaBeans. Fig. 4a illustrates that the data is cached in distributed managed memory in a cluster. In our scheme, the raw bytes generated in accordance with the layout of the user-defined HJavaBeans are stored in the distributed managed memory. As discussed in Section 3.5.1, programmers can define HDST based on the user-defined HJavaBeans. All accepted data, including the data from HDFS, transferred from networks and produced by CPUs or GPUs, are transformed to $byte[]$ based on the layout of the user-defined HJavaBean and further stored in the distributed managed memory spaces. Prior to processing, the raw bytes to be processed in CPUs are wrapped as HJavaBeans, whereas the data to be processed in GPUs are transferred to GPUs in the form of $byte[]$ over the PCIe bus.

4.1.2 Details of the Data Mapping Scheme

Fig. 4b details the data mapping scheme in a worker node with the example given in Section 3.5.1. The raw bytes of the user-defined HJavaBean are stored in the managed memory in a worker's JVM heap in a sequential manner, as defined by HJavaBean. These raw bytes match the layout of the struct whose definition is generated by the JIT compiler as described in Section 4.2.1. Therefore, the raw bytes can be transferred to the device memory of GPUs without any modifications by the JNI and over PCIe. After the data transfers, the struct and the members in the struct can be accessed by the struct $pointer$, as described in Section 4.2.1. For executions on the CPUs, the raw bytes are wrapped as HJavaBeans.

```
#pragma pack (4)
typedef struct PointAOS{
    float x;
    float y;
}PointAOS;

typedef struct PointSOA{
    float x[1000];
    float y[1000];
}PointSOA;
```

Fig. 5. Data structure generation.

This scheme effectively inhibits the occurrence of serialization and deserialization between JVM objects and OpenCL structs, thus greatly improving the performance and reducing the burden on programmers. It is worth noting that, serialization and deserialization processes occur between raw bytes and HJavaBeans when executing hMappers or hReducers on the CPUs. Nevertheless, the scheme is similar to the original implementation in Flink. In Flink, DSTs are stored in the distributed managed memory in the form of raw bytes. Therefore, when executing functions on CPUs, the raw bytes must be wrapped as UDTs. After execution on CPUs, the contents (in the form of UDT objects) are transformed to raw bytes and stored in the distributed managed memory.

4.2 JIT Compiling Scheme

The main goal of our proposed FlinkCL is to improve the computational performance of applications on Flink by utilizing the GPUs' high computing power while retaining the easy programming model. Utilizing our proposed JIT compiling scheme, programmers only need to write hMapper/hReducer in Java using the interfaces provided by FlinkCL. If the tasks are scheduled to be executed on GPUs, the user-defined hMapper/hReducer will be compiled to the OpenCL kernel automatically.

One of the user-friendly usable aspects of Aparapi is the way in which the body of a parallel region is defined, which differs from that in Flink. Both require programmers to extend a kernel interface similar to Java's runnable class. Moreover, Aparapi only supports primitives and arrays of primitives. This limitation requires programmers to transform JVM objects to primitives and arrays of primitives manually if Aparapi is blended with Flink. Therefore, we have improved the original Aparapi named as IAparapi and designed an IAparapi Bridge to fill the gap between hMapper/hReducer and Aparapi's runnable interface. The IAparapi includes data structure generation and function generation, and addresses some of these limitations, while reusing some of the code-generation framework of Aparapi.

4.2.1 Data Structure Generation

Our proposed JIT compiler can generate HJavaBeans to OpenCL structs automatically. The positions of member variables are specified explicitly in the definition of HJavaBean. Through Java annotation and reflection techniques, the details and layout of HJavaBean and Tuple can be obtained during the runtime. For each field in the JVM object, a similar typed field is created during the definition of the struct auto-generated by our code generator. Fig. 5 shows an example of the struct generated by the *Point* class previously presented in Section 3.5.1.

```
public mapBulk(PointSOA input, PointSOA_out output,
               int gid)
{
    float x = input.x[gid];
    float y = input.y[gid];
    for(i = 0; i < 5; i++){
        dis = distance(x, y, cen[i].x, cen[i].y);
        if(dis < min){
            min = dis;
            minIndex = i;
        }
    }
    out.index[gid] = i;
    out.x[gid] = x;
    out.y[gid] = y;
  return;
}
```

Fig. 6. Java codes for KMeans.

The packed attribute is utilized to set the type of alignment. The struct data (in the form of raw bytes, which matches the layout of the OpenCL struct) is passed by value to the accelerator kernel using OpenCL's *clSetKernelArg* API. Their values are fetched from the JVM using the JNI APIs. The order of the field metadata of the struct matches that specified in HJavaBean. In the GPUs, this struct can be referenced using an appropriately typed pointer, and Public fields in the Point object can be loaded using the -> operator.

We classify the data structures supported by our JIT compiler into four categories: primitives, non-array HJavaBeans, arrays of primitives, and arrays of HJavaBeans. HJavaBean also supports other HJavaBean objects as its member variables.

4.2.2 Function Generation

In terms of function code generation, we attempted to enhance Aparapi to support functions for HJavaBean or arrays of HJavaBeans. Specifically, the OpenCL standard does not include dynamic memory allocation as a supported operation. Therefore, we treat *new* opcode in Java as definition of variables.

In addition to a proper design of the data layout, utilizing memory hierarchy in GPUs to improve the memory bandwidth is another important optimization method for programming on GPUs. Other optimization methods include designing multi-dimensional ranges, selecting appropriate sizes of work groups and controlling synchronization. We reuse the mechanisms provided by Aparapi and integrate them into FlinkCL.

In the original implementation of Flink, a broadcast variable functionality was used to broadcast the small amounts of data among all the workers in the cluster. Generally, broadcast variables are small. Using local memory on OpenCL can improve performance because the cost of fetching from local memory is much lower than that from global memory. FlinkCL provides an automatical functionality to transfer broadcast variables to local memories that are shared by all the work items executing in the same work group. If we set a variable as broadcast, the contents of the variable will be transferred to GPUs in the form of local memory., and the prefix of the variable in the OpenCL code will be set as local. Programmers can also define a variable in local memory using a local suffix.

Fig. 6 shows a Java code snippet of the Map phase in KMeans utilizing the SoA layout and the hBulkMap

```
__kernel void run(
    __global PointSOA *input,
    __global PointSOA_out *out
    __local  Point *cen){
    int gid = get_global_id(0);
    float x = input->x[gid];
    float y = input->y[gid];
    for(i = 0; i < 5; i++){
        dis = distance(x, y, cen[i].x, cen[i].y);
        if(dis < min){
            min = dis;
            minIndex = i;
        }
    }
    out->index[gid] = i;
    out->x[gid] = x;
    out->y[gid] = y;
    return;
}
```

Fig. 7. Generated OpenCL kernel for KMeans.

interface. The centroids are treated as a broadcast variable. The definitions of broadcast variables resemble those in the original Flink. The codes for defining broadcast variables are not presented. Java codes can be executed on CPUs. For conducting executions on CPUs, FlinkCL increases *gid* automatically and invokes the hMapBulk function in Java in a sequential manner. To conduct executions on GPUs, the Java codes will be compiled to the OpenCL kernel codes, which is presented in Fig. 7.

4.3 Hierarchical Partial Reduction

As discussed in Section 3.1, in some applications, the communication overhead during the Shuffle phase greatly affects the performance. In addition, a large amount of memory may be required to hold all the intermediate results from the Shuffle phases before Reduce begin to consume them. Hence, a hierarchical partial reduction schema (as shown in Fig. 8) is proposed to improve the performance in heterogeneous CPU-GPU clusters. The main idea of the proposed hierarchical partial reduction schema is to reduce the PCIe communication and network-transfer time. The proposed method is based on the assumption that the operations in the Reduce stage are usually associative and commutative, which covers most of the MapReduce applications, with some exceptions such as the sorting application [18].

4.3.1 Partial Reduction for Bulk Execution

The pseudo-code for the MapReduce paradigm and the improved partial reduction paradigm are listed in the list below. The original MapReduce expresses the computation as two UDFs: Map and Reduce. The map function takes a set of input instances and generates a set of corresponding intermediate output $(key, value)$ pairs. During the Shuffle phase, all the intermediate values associated with the same key are grouped and shuffled to the Reduce function. The Reduce function, which is also written by the user, accepts a key and a set of values associated with that key. It then merges these values together to form a potentially smaller set of values.

In the partial reduction based implementation model, the map function, which is defined by the user, works in the same way as in the traditional MapReduce. However, each key-value pair is inserted into the reduction object (RO) at the end of each map operation. Thus, every key-value pair will be merged to the output results immediately after it is generated. Every time a key-value pair arrives, the RO will

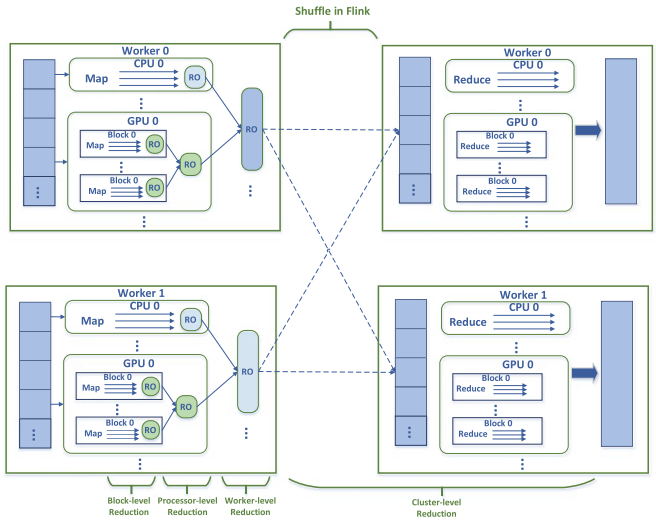

Fig. 8. Hierarchical partial reduction.

locate the index corresponding to the key, and reduce this key-value pair to the index in the RO. The RO is transparent to the users, who writes the same Map and Reduce functions as in the original MapReduce. This approach is advantageous mainly because it avoids the communication overhead of a large number of key-value pairs across worker nodes and large memory requirements for storing intermediate results. This occurs because the key-value pairs are reduced immediately after they are generated.

4.3.2 Hierarchical Partial Reduction

We consider the following hierarchy in a heterogeneous CPU-GPU cluster environment: local memory in each SM of GPUs, device memory in each GPU, main memory in each worker and communication over networks. Based on this hierarchy, we propose a hierarchical partial-reduction scheme, as shown in Fig. 8. In this method, we include four-level partial reduction that is transparent to programmers. The optimization is exposed as an additional user callback function that the user can set if desired.

Block-Level. The GPU is composed of a certain number of SMs. Each SM contains a set of simple cores that perform in-order processing of the instructions. A block of threads is mapped to and executed on a SM. Local memory in the GPUs is private to each SM and supports much faster read and write operations. Therefore, we make use of the local memory to conduct partial reduction in each block in the GPUs. The results of the mappers generated by threads in a block are combined together in the local memories. Overflow handling must be considered because the capacity of local memory is very limited. We utilize the implementation of RO, overflow handling and swapping mechanism proposed in [29].

Processor-Level. Processor-level partial reduction is responsible for combining Map results generated in CPUs or GPUs. For GPUs, the results for a large amount of data are combined in the device memory in the GPUs, and the mechanism utilized is similar to the methods proposed in [29]. For CPUs, the intermediate results are combined in the main memory. The hash bucket is utilized to conduct processor-level partial reduction for CPUs, as proposed in [30].

Node-Level. Node-level partial reduction is responsible for combining the Map results generated by different

processors, including CPUs and GPUs. The implementation is similar to the node-level partial reduction for CPUs.

Cluster-Level. Cluster-level partial reduction is responsible for combining the Map results generated by different worker nodes in the cluster. We use the Shuffle implementation in the original Flink to conduct the combination before invoking the Reduce phase.

If the operations during the Reduce stage are not associative and commutative, the original MapReduce process as presented in Algorithm 4.1 (line 2 to line 8) will be adopted. In this case, the Reduce stage begins after the completion of the Shuffle phase, and the Shuffle phase performs an all-to-all copying of intermediate data from the Map phase to the Reduce phase.

Algorithm 4.1. Partial Reduction for Bulk Execution

1: Original MapReduce:
2: **for** Each *element* e **do**
3: $(key, val) \leftarrow map(e)$;
4: **end for**
5: shuffle (key, val) over key to create ROs;
6: **for** each RO r in ROs **do**
7: $out = reduce(r)$;
8: **end for**
9:
10: Partial Reduction:
11: **for** Each *chunk* ck **do**
12: **for** Each *element* e in ck **do**
13: $(key, val) \leftarrow map(e)$;
14: $RO(key) \leftarrow reduce(RO(key), val)$;
15: **end for**
16: **end for**
17: shuffle $RO(key)$ over key to create ROs;
18: **for** each RO r in ROs **do**
19: $out = reduce(r)$;
20: **end for**

4.4 Heterogeneous Task Management

This section describes our task management scheme for the heterogeneous CPUs and GPUs.

4.4.1 Overview

The runtime of the original Flink in a worker (including JobManager, DAGScheduler and TaskManager) can be considered to be a producer that produces tasks. The JobManager and DAGScheduler in Flink are responsible for managing the dependencies of tasks. Therefore, the tasks produced during the runtime of Flink can be regarded as independent tasks. Our proposed heterogeneous TaskManager is a consumer that consumes tasks in a first-in-first-out (FIFO) style.

There is a scheduler, a thread pool and a GPU manager in the heterogeneous TaskManager. The scheduler is in charge of scheduling tasks among CPUs and GPUs. The threads in the thread pool are responsible for executing tasks on the coupled CPUs and GPUs. The GPU manager is utilized to manage the GPU devices and cache the environmental variables of the GPUs.

The computing power of CPUs and GPUs differs for executing different applications, and one may be preferable for certain tasks. We investigate how to effectively utilize both CPUs and GPUs to accelerate a single application. An auto-tuning partitioning scheme and a dynamic load balancing scheme are proposed to achieve good load balance among the heterogeneous CPUs and GPUs based upon the differences between hMappers and hReducers.

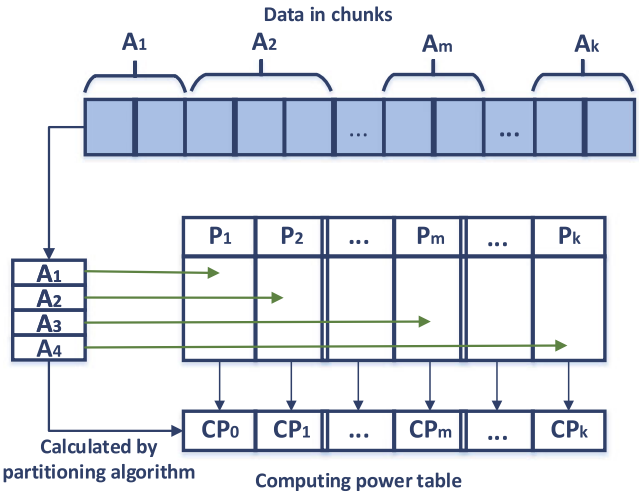

Fig. 9. Auto-tuning partitioning scheme.

4.4.2 Environmental Cache

The GPU manager is responsible for caching and managing the environmental variables of GPUs. In Flink, the data is processed by chunks sequently. Moreover, iterative data-intensive applications perform highly similar computations over a number of iterations and play an essential role in in-memory big data processing. Our experiments demonstrate that there is large overhead associated with the operations (discovering and initializing the device, compiling the Java codes to OpenCL kernel codes, creating a command queue, and loading the corresponding libraries). For example, the time spent during the matrix vector multiplication case is nearly 20-fold greater than the total transfer and execution time when a chunk is processed. Therefore, a cache scheme is proposed to store the execution environment, compiled OpenCL kernels, and created command queues. Thus, the overhead is only incurred once when processing the first chunk.

4.4.3 Auto-Tuning Partitioning Scheme

An analysis of many applications shows that most hMappers are regular. In most cases, the time complexity involved when utilizing a specific hMapper to process a pair is stable. In addition, the data size of a pair is generally the same. That is, if the computing power of a processor is invariable, the processing time of a specific hMapper varies linearly with the size of the data to be processed. Therefore, an auto-tuning partitioning scheme is proposed to address this situation.

Our proposed auto-tuning partitioning scheme contains an auto-tuning approach that learns the computing power of the GPUs and multicore CPUs using our proposed analytical model during the first iteration or the beginning of processing small datasets for specific applications. Subsequently, we use a partitioning algorithm to partition the datasets based on the computing power.

The auto-tuning data partitioning process is presented in Fig. 9. First, we create a computing power table (CPT) to save the computing power of all the processors for specific

applications. In the CPT, the computing power of processor j is denoted as CP_j. After calculating the computing power of each processor, the auto-tuning partitioning algorithm presented in Algorithm 4.2 is adopted to partition the datasets for the heterogeneous GPUs and CPUs. The partitioning results are stored in the partitioning table. A_j is the number of chunks assigned to processor j.

Algorithm 4.2. Partitioning Algorithm

Input: The K computing powers $CP_1 \leq CP_2, \ldots, \leq CP_K$; the total computing power CP; the number of chunks M.
Output: The partitioning results $A_1, A_2, \ldots, A_K$

```
First stage:
for i = 1 to K do
    P ← M × CP_i/CP;
    A_i ← ⌊P⌋;
    if P − ⌊P⌋ = 0 then
        R_i ← 1/CP_i;
    else
        R_i ← (⌈P⌉ − P)/CP_i;
    end if
    Put R_i into R;
    Total ← Total + A_i;
end for

Second stage:
Remain ← M − Total;
i ← 1;
while i ≤ Remain do
    Select the smallest one R_x from R;
    A_x ← A_x + 1;
    R_x ← 1/CP_i;
end while
return A_1, A_2,...,A_K.
```

The goal of Algorithm 4.2 is to find a partitioning scheme for distributing all the chunks among the K heterogeneous processors to achieve the shortest execution time. The partitioning algorithm has two stages. First, the actual number of chunks of each processor P is calculated. P changes linearly with the computing power calculated by our proposed method and is always a decimal number. However, each bulk can only be computed initially. Therefore, $\lfloor P \rfloor$ chunks of each processors are distributed at first.

The second stage of Algorithm 4.2 involves allocating the remaining number of chunks $Remain$. R_i refers to the execution time of processing a chunk in processor i at present. If the P of processor i is an integer, R_i can be calculated as: $1/CP_i$. If the P of processor i is an decimal, R_i can be calculated as: $(\lceil P \rceil - P)/CP_i$. The method of the second stage is greedy. We select the processer with the smallest execution time for processing a new chunk and then allocate a chunk to it until all the remaining chunks are distributed.

Next, we detail the calculation of the computing power table. Assume that there are K processors including CPUs and GPUs, whose computing powers are CP_1, $CP_2, \ldots,$ CP_K. In most cases, the scale of a task assigned to a processor is measured by the size of the data, which should be linearly proportional to the computing power of the processors. In Flink, the data is processed in chunks, each of which is stored in a memory segment. The total computing power is calculated as follows:

$$CP = \sum_{i=1}^{K} CP_i = \sum_{i=1}^{K} S_i/T_i, \quad (1)$$

where S_i is the size of the data that has been processed by processor i and T_i is the total actual processing time.

Based on these definitions, the parameter to be learned is the actual processing time of a chunk. By adopting the environmental cache scheme, the time required to initialize the environment (such as creating command queues, compiling codes) can be avoided. As discussed in Section 3.5.4, the data is processed as similarly sized chunks. The computing power of the GPU with the processor index i and the environmental cache scheme is given by

$$CP_{g\_i} = 1/T_{g\_i} = 1/(T_{gp\_i} + T_{gm\_i} + T_{gf\_i}). \quad (2)$$

The computing power without the environmental cache scheme is given by

$$CP_{g\_i} = 1/T_{g\_i} = 1/(T_{gp\_i} + T_{gm\_i} + T_{go\_i} + T_{gc\_i} + T_{gf\_i}), \quad (3)$$

where T_{g_i} denotes the execution time of a chunk processed by the CPU with processor number i, T_{gp_i} indicates the data processing time on the GPU with the processor number i, T_{gm_i} denotes the moving data buffers between the CPU and the GPU, T_{gf_i} denotes the fixed time for invoking OpenCL kernel, T_{go_i} represents the initialization time (to create a command queue and to create the kernel), and T_{gc_i} denotes the time required to compile the Java code to the OpenCL kernel.

4.4.4 Dynamic Load Balancing

Our proposed auto-tuning partitioning scheme is appropriate for dealing with load balance for regular tasks. However, if it is used to partition data for irregular tasks (for example, most hReducers are irregular), the partitioning results can be inaccurate. Therefore, we develop a dynamic load balancing scheme. The bulks of a specific application are treated as a task pool. When a processor finishes a bulk, it will obtain another bulk from the task pool. Before obtaining a bulk from the task pool, the thread must lock the task pool. After obtaining a bulk, the thread must unlock the task pool. Clearly, the process of obtaining the tasks, locking or unlocking the task pool and waiting for unlocking the task pool will frequently cause extra overhead. An extra CPU thread is required to conduct the dynamic scheduling.

5 Performance Evaluation and Analysis

We evaluate the performance of FlinkCL using GPU instances in Amazon AWS and other state-of-the-art GPUs. Our architecture is based on Flink 1.3.1, which is also provided by AWS's EMR service.

5.1 Overall Speedup Analysis

In this section, we build a brief time-cost model to evaluate the overall speedup achieved by GPU acceleration and other performance improvements that arise from adopting the proposed optimizations in FlinkCL.

The core execution in Flink is MapReduce, although some other convenient functions are provided (e.g., Count and Join). To simplify the analysis, only the MapReduce

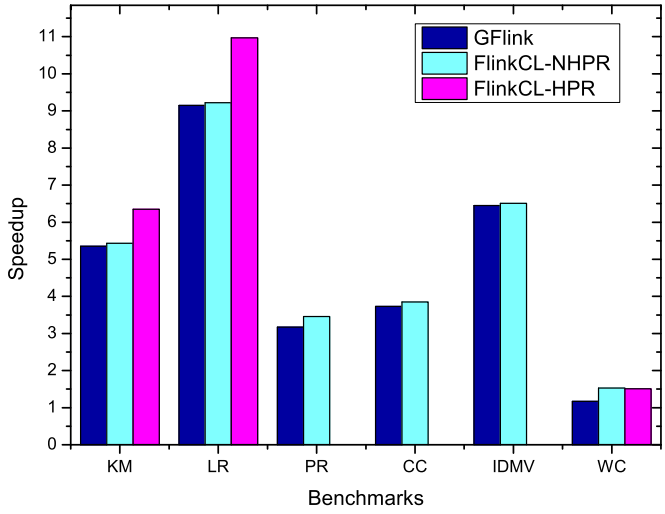

Fig. 10. Overview of the performance results.

phases are taken into consideration. The overall speedup and the total time required to execute an application in Flink can be derived using Equations (4) and (5), as follows:

$$Speedup = \frac{T_{Flink\_total}}{T_{FlinkCL\_total}} \tag{4}$$

$$T_{total} = \sum_{i=1}^{n} (T_{map\_i} + T_{reduce\_i} + T_{shuffle\_i}) + T_{submit} + T_{IO} + T_{schedule}, \tag{5}$$

where $Speedup$ represents the speedup obtained by FlinkCL, T_{Flink_total} denotes the total execution time of an application in Flink, $T_{FlinkCL\_total}$ denotes the total execution time of an application in FlinkCL, n denotes the number of MapReduce phases. T_{map_i}, $T_{shuffle\_i}$ and T_{reduce_i} denote the time cost of ith Map, Shuffle and Reduce phase separately. T_{submit}, T_{IO} and $T_{schedule}$ denotes the time cost of submitting an application, reading or writing data to HDFS or other systems (e.g., HBase), and scheduling the job, respectively. During some MapReduce phases, the Reduce phase and the Shuffle phase can be omitted.

Suppose that there are K CPUs and L GPUs in a cluster. The speedup of the ith Map or the ith Reduce on FlinkCL can be denoted by

$$Speed = \frac{\max(T_{c\_1}, \ldots, T_{c\_K})}{\max(T_{c\_1}, \ldots, T_{c\_K}, T_{g\_1}, \ldots, T_{g\_L})}, \tag{6}$$

where T_{c_K} is the execution time of the Map/Reduce phase processed by the Kth CPU and T_{g_L} is the execution time of the Map/Reduce phase on the Lth GPU.

These two equations yield the following observations.

Observation 1. The speedup of an application in FlinkCL depends on the characteristics of the application. The execution time of the Shuffle phase affects the speedup of an application. If other parameters are fixed, a Shuffle phase occupying a larger space will result in a smaller speedup. Our proposed HPR scheme can reduce the communication overhead during the Shuffle phase, thus increasing the speedup.

Observation 2. Equation (6) shows that achieving greater speedup requires shortening the total execution time of the GPUs, and distributing the data to the CPUs and GPUs in a reasonable manner so that the maximum value of the time cost by all the processors can be decreased.

Observation 3. Equations (2) and (3) indicate that if the time required to transfer the data between the GPU and the JVM, the initialization time and the compiling time will decrease, the total execution time will get decreased. Our proposed data mapping scheme and environmental cache scheme can also decrease the time.

Observation 4. If a small amount of data is to be processed, the T_{submit}, T_{IO} and $T_{schedule}$ will occupy a large part of the total execution time. That is, in general, the speedup of processing small datasets will be smaller than that of processing large-scale datasets.

5.2 Effectiveness of FlinkCL

In this section, we will evaluate the performance of FlinkCL with 6 benchmarks and 10 slave nodes compared with Flink and GFlink. During this case, Amazon's p2xlarge instances are utilized each of which contains 4 Intel Xeon E5-2686 v4 CPU, 61GB memory and 1 NVIDIA K80 GPU. Each GPU has 2496 CUDA processor cores and 12 GB global memory.

5.2.1 Speedup Overview

We selected benchmarks from the in-memory version of HiBench [31], which has been widely used to evaluate the Spark and Hadoop frameworks, and 2 extra benchmarks to evaluate the performance of our proposed FlinkCL. Overall, we use four types of benchmarks: machine learning (KMeans and LinearRegression), graph processing (PageRank and ComponentConnect), micro benchmarks (WordCount) and numerical calculation (matrix vector multiplication called IMV). WordCount is a batch workload that contains only one-pass processing, whereas KMeans, LinearRegression, PageRank, ComponentConnect and IMV are iterative workloads. The overall speedup results are presented in Fig. 10. *FlinkCL-NHPR* denotes the speedup of FlinkCL without the HPR scheme, while *FlinkCL-HPR* denotes the speedup of FlinkCL with the HPR scheme.

Machine Learning. KMeans is a widely used clustering algorithm. The algorithm has two steps in an iteration: (1) finding the closest center for each data point and using a map function and (2) grouping data points to their center to further obtain a new cluster center at the end of each iteration using a reduce function. In the implementation of FlinkCL, 210 million points and 50 cluster centers are utilized. A total of 200000 points are grouped as a bulk in the SoA layout, and hBulkMap is utilized to process a bulk. Cluster centers are treated as broadcast variables and stored in local memory to speed up the reading of cluster centers. Fig. 10 shows that the speedup in FlinkCL without the HPR scheme ($5.53\times$) slightly exceeds that of GFlink (5.35), because in FlinkCL, both CPUs and GPUs can be utilized to conduct the Map and Reduce functions. However, the effects of utilizing CPUs for Map and Reduce are not obvious because there are only 4 CPUs in a worker node. Nevertheless, one CPU must be utilized to control the execution of one GPU, and thus this CPU

cannot be adopted to conduct the computation. Greater speedups are obtained by adopting the HPR scheme ($6.35\times$). This increase occurs because the communication overhead in the Shuffle phase is reduced due to the small number of keys during the shuffle.

Logistic regression is a commonly used classification method in statistical analysis. The algorithm finds a value v that can best separate a data set. In each iteration, a logistic function is applied to every data point in the data set. Next all the results are aggregated to update v. Similar to KMeans, 210 million points are utilized and 200000 points are grouped as a bulk in the SoA layout. FlinkCL with the HPR scheme hase significantly improved the performance, by nearly $10.97\times$. The reason for this improved performance is that the linear regression is bounded by calculations on each data point and thus can benefit from the GPU's high computation power. Similar to KMeans, the effect of utilizing CPUs to accelerate the Map and Reduce functions is not obvious. Nevertheless, adopting the HPR scheme can greatly improve performance.

Micro Benchmarks. Word Count is a simple metric for measuring article quality by counting the total number of occurrences of each word. There is only one MapReduce phase in WordCount. In this case, the dataset contains 40GB. The speedup in WordCount is not high. WordCount is a batch application without iterative execution. Therefore, the I/O overhead of reading data from HDFS forms the bottleneck. Furthermore, the hMapper and hReducer in WordCount are not compute-intensive. Since there are many types of keys (words), adopting the HPR scheme does not strongly reduce the communication overhead.

Numerical Calculation. Iterative matrix vector multiplication (IMV) is an iterative workload. When implementing IMV in FlinkCL, a row is grouped as an HJavaBean object and hMapper is utilized to multiply a grouped row and the vector. This phase can be accelerated significantly by GPUs. The speedup of implementing IMV on GFlink is nearly $6.43\times$, whereas the speedup for FlinkCL is $6.49\times$. The HPR scheme is not utilized because it only contains an hMapper.

Graph Processing. PageRank is a graph algorithm that ranks a set of elements according to their references. ComponentConnect provides an important topological invariant of a graph. The speedup in these two benchmarks is greater for FlinkCL than for GFlink, and the effects of using both CPUs and GPUs to accelerate these two benchmarks are more obvious than for other benchmarks. Because Shuffle involves very large-scale numbers of key types (up to millions), we did not adopt the HPR scheme.

In summary, these results demonstrate that using FlinkCL can improve the performance of various workloads especially for iterative computation and computationally intensive applications.

5.2.2 Evaluation of Programmability

There are three execution modes of hMap and hReduce: CPU-only, GPU-only and hybrid CPU-GPU. We also provide two means of using the hMap and hReduce: unified and separate. If the unified strategy is adopted, programmers only need to write one version of the code in Java. The user-defined programs will be automatically executed in CPUs or GPUs. For the separate strategy, programmers can program for the CPUs and GPUs separately. To facilitate programming, programmers can utilize the unified strategy with HJavaBean and hMap/hReduce. To improve the performance on CPUs and GPUs, the separate strategy can be utilized. There are three code portions when programming on Flink: organizing UDFs, defining UDTs and implementing UDFs (Map/Reduce). When utilizing the unified strategy, the amount of code is almost identical to that required to use the original Flink. For the separate strategy, programmers must implement hMap/hReduce for CPUs and GPUs separately.

For programming on GFlink, the same UDF cannot be executed on CPUs and GPUs simultaneously. GFlink attempts to shorten the codes for allocating/deallocating memory to GPUs, data transfer and environment management. However, programmers must implement UDFs in Java, register CUDA kernels as GExecutor and define UDFs in Java. They must also write CUDA kernels and define structs in CUDA. We evaluate the amount of code for KMeans and PageRank in GFlink and FlinkCL. The amount of code required to execute applications on GPUs in GFlink is nearly twofold greater compared to utilizing the unified strategy in FlinkCL. In contrast, for the separate strategy, the amount of code in GFlink is slightly greater than in FlinkCL. These six benchmarks for evaluation are all implemented utilizing the unified strategy.

5.2.3 Evaluation of FlinkCL with Different Numbers of Slave Nodes

In this section, the performance with different scales of slave nodes is considered. The number of slave nodes is gradually increased from 4 to 24. The detailed running time of KMeans is presented in Fig. 11a, while the speedup in KMeans, ComponentConnect and IMV is presented in Figs. 11a, 11b and 12a, respectively.

As shown in Fig. 11a, the runtime of KMeans on Flink decreases rapidly as the number of slave nodes increases, whereas the runtime on GFlink and FlinkCL decreases more gradually as the number of slave nodes increases. Next, we evaluate the speedup in KMeans on GFlink and FlinkCL compared with Flink under different numbers of slave nodes; the results are presented in Fig. 11b. We find that the speedup in KMeans on FlinkCL with the HPR scheme compared to Flink increases as the number of slave nodes increases. This result shows that adopting the HPR scheme enhances scalability. When increasing the number of slave nodes, the speedup in KMeans on GFlink and FlinkCL compared to Flink decreases gradually.

Figs. 11c and 12a show the speedup in ComponentConnect and IMV, respectively, for Flink, GFlink and FlinkCL with varying numbers of slave nodes. In these figures, the speedup in applications on Flink with 4 slave nodes is set to 1. The figures show that both the speedup in ComponentConnect and IMV on Flink, GFlink and FlinkCL increase as the number of slave nodes increases. GFlink and FlinkCL show similar scalability. IMV shows the best scalability; ComponentConnect shows the worst scalability. That is because there is no Shuffle phase in our implementation of IMV. Nevertheless, the proportion of the Shuffle phase is the greatest in ComponentConnect. In addition, the speedup in ComponentConnect and IMV on GFlink and

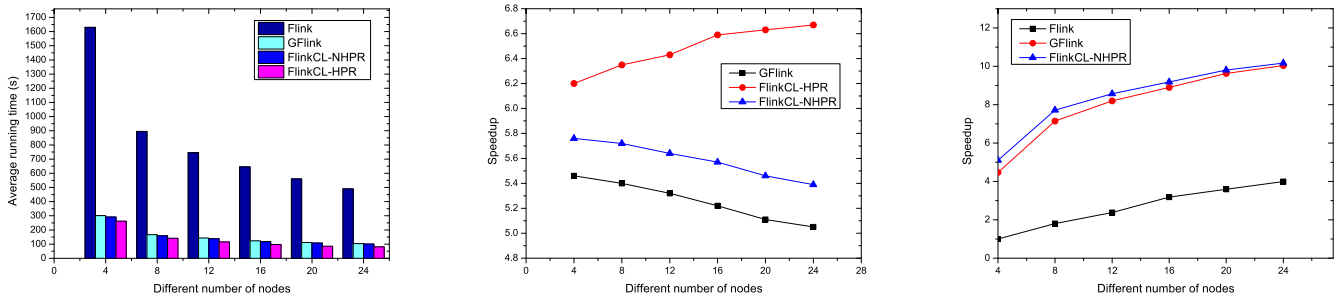

Fig. 11. Performance results with different numbers of slave nodes. (a) Running time of KMeans. (b) Speedup of KMeans. (c) Speedup of component connect.

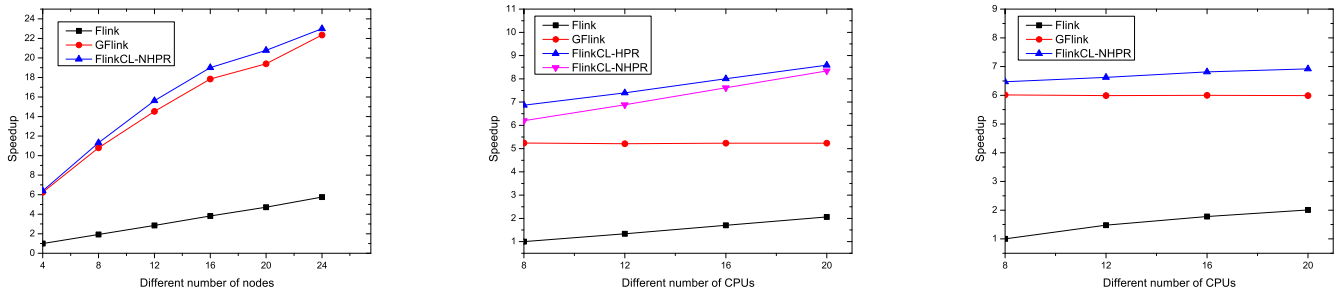

Fig. 12. (a) Speedup in IMV. (b) Evaluation of KMeans with different numbers of CPUs. (c) Evaluation of IMV with different numbers of CPUs.

FlinkCL compared to Flink decreases as the number of slave nodes increases. This behavior occurs because, for implementation on the GPUs, the computation is significantly accelerated. Therefore, the overhead caused by I/O, communication over networks, task scheduling and system invoking forms a bottleneck rather than the computation.

5.2.4 *Evaluation of FlinkCL with Different Numbers of CPUs*

In this section, we use Amazon's p2.8xlarge instances and 5 slave nodes. We vary the number of CPUs to evaluate the performance of KMeans and IMV on FlinkCL compared with GFlink, as shown in Figs. 12b and 12c. Two GPUs are utilized in this implementation, and the number of CPUs in a node is varied from 8 to 20. The HPR scheme is adopted in KMeans and not in IMV. The data sizes are as described in Section 5.2.1.

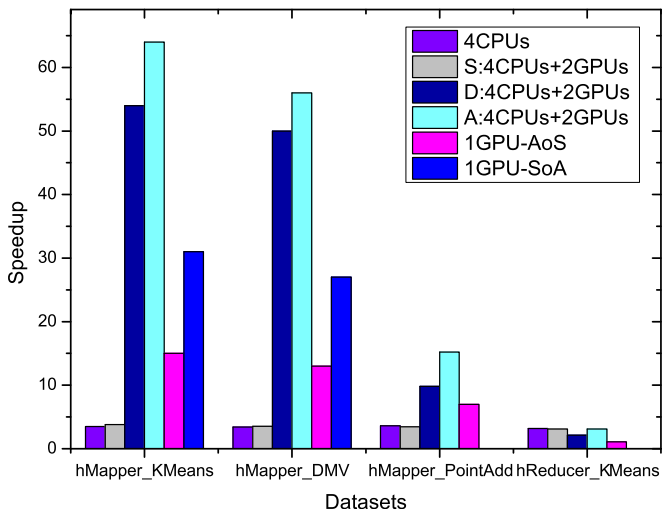

Fig. 13. Detailed speedup in hMapper and hReducer on FlinkCL.

In these figures, the speedup of applications on Flink with 8 CPUs within a node is set to 1. The speedup in applications on GFlink remains unchanged because GFlink does not utilize CPUs to conduct Map and Reduce computation. In contrast, the speedup in applications on Flink and FlinkCL increases gradually as the number of CPUs within a node increases. Fig. 12b shows that the speedup in KMeans on Flink increases more rapidly with the HPR scheme than without it. It demonstrates that the HPR scheme also improves the scalability on CPUs.

5.3 Evaluation of Optimization

In this section, we evaluate the performance impact of our proposed optimization techniques.

5.3.1 *Evaluation of Heterogeneous Task Management and Data Layout*

In this section, we evaluate the speedup of some hMappers and hReducers accelerated by FlinkCL utilizing Tesla C2050 GPUs and Intel(R) Core(TM)i5-4590 CPUs. The dynamic load balancing scheme and auto-tuning partitioning scheme are employed to evaluate the effects of these two scheduling schemes. We omit other phases, such as reading data from HDFS, submission of the job and scheduling of the job. For the executions on CPUs, we use the hBulkMap function of the original Flink, in which all elements are traversed by the iterator in JAVA and processed by the Map function one by one.

The detailed speedup of hMapper and hReducer on FlinkCL is presented in Fig. 13. In this figure, *S:4CPUs +2GPUs* indicates that the static load balancing is used. In other words, the tasks are evenly distributed to all the processors, which include 4CPUs and 2GPUs. *D:4CPUs+2GPUs* indicates the use of dynamic load balancing, and *A:4CPUs +2GPUs* indicates the use of auto-tuning partitioning scheme.

From Fig. 13, some observations can be obtained. First, the speedup in the hMapper of KMeans and IMV is much higher

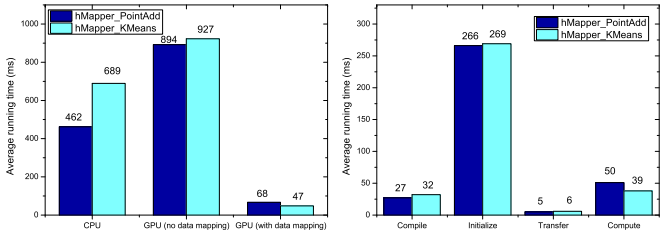

Fig. 14. (a) Evaluation of the data mapping scheme. (b) Evaluation of the environmental cache scheme.

than the overall speedup in these applications. The speedup of hMapper of IMV is lower than that in KMeans. However, the overall speedup in IMV is higher than that in KMeans. That is because other overheads also exist in these applications. According to the Amdahl rule, the speedup is higher than the overall speedup. Second, the speedup is nearly identical for *4CPUs* and *S:4CPUs+2GPUs* because the original Flink also distributes the tasks among CPUs evenly. One CPU must be utilized to control the execution of one GPU, and thus this CPU cannot be adopted to conduct the computation. Third, as for hMappers, *A:4CPUs+2GPUs* has the best speedup, and the speedup in *D:4CPUs+2GPUs* is smaller than that of *A:4CPUs+2GPUs*. This result occurs because during dynamic load balancing, obtaining tasks from the task pool and subsequent synchronization can incur extra overhead. Finally, the hReducer in FlinkCL cannot obtain good speedup because it is not computationally intensive.

5.3.2 Evaluation of the Data Mapping Scheme

This case evaluates the effects of our proposed data mapping scheme in HDST. Fig. 14 shows the detailed execution time of processing a chunk by hMapper in KMeans and IMV on a CPU, on a GPU without the data mapping scheme and on a GPU with the data mapping scheme respectively. The average running time of executions on GPUs without the data mapping scheme is much longer than that of executions on CPUs. That is because, without the data mapping scheme, it is necessary to go through the entire dataset to transform Java objects to buffers before execution on GPUs. After execution on the GPUs, the results must be transformed to Java objects. If the data mapping scheme is adopted, the average running time of executions on GPUs can be significantly shortened. Thus, our proposed data mapping scheme substantially improves the performance.

5.3.3 Evaluation of the Environmental Cache Scheme

Fig. 14 shows the detailed results of the main stages of hMapper in KMeans and IMV on a GPU for processing a chunk. The overhead caused by compiling and initializing (including loading libraries and creating command queues) is much greater than that of data transfers and execution. If the environmental cache scheme is adopted, the overhead for compilation and initialization will only be aroused once. Our proposed environmental cache scheme thus clearly enhances the performance.

6 Conclusion and Future Work

GPUs have become efficient accelerators for HPC. This paper has proposed FlinkCL, which harnesses the high computational power of GPUs to accelerate the in-memory cluster computing with an easy programming model. FlinkCL is based on four proposed core techniques: an HDST), a JIT compiling scheme, an HPR scheme and a heterogeneous task management strategy. By using these techniques, FlinkCL remains compatible with both the compile-time and the runtime of the original Flink.

To further improve the scalability of FlinkCL, a pipeline scheme similar to that introduced in [20] could be considered. We can utilize this pipeline to overlap the communication between cluster nodes and the computation in a node. In addition, by using an asynchronous execution model, transfer on PCIe and executions on GPUs can also be overlapped. In the current implementation, data in the GPU memory must be moved into the host memory before it can be sent over the network. A future research direction could involve enabling GPU-to-GPU communication using GPUDirect RDMA to further improve the performance. Another optimization measure could be a software cache scheme that can cache intermediate data in GPUs to avoid unnecessary data transfers on PCIe.

In current design, hMap and hReduce function are compiled to kernels separately. Actually, we can fuse these kernels together if possible in our JIT compiler. By adopting this scheme, the time for kernel invoking can be decreased and some data transfers on PCIe can be avoided.

Acknowledgments

The authors would like to express their gratitude to the editor and three anonymous reviewers for their constructive comments. The research was partially funded by the National Outstanding Youth Science Program of National Natural Science Foundation of China (Grant No. 61625202), the International (Regional) Cooperation and Exchange Program of National Natural Science Foundation of China (Grant No. 61661146006), the Singapore-China NRF-NSFC Grant (Grant No. NRF2016NRF-NSFC001-111), the National Natural Science Foundation of China (Grant Nos. 61370095, 61472124, 61662090, 61602350), and the Key Technology Research and Development Programs of Guangdong Province (Grant No. 2015B010108006), the Outstanding Graduate Student Innovation Fund Program of Collaborative Innovation Center of High Performance Computing, the scholarship from China Scholarship Council (CSC) under (Grant No. 201706130121).

References

[1] Flink programming guide, 2016. [Online]. Available: http://flink.apache.org/, Accessed on: Nov. 1, 2016.

[2] 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, pp. 2–2.

[3] W. Yang, K. Li, Z. Mo, and K. Li, "Performance optimization using partitioned SpMV on GPUs and multicore CPUs," *IEEE Trans. Comput.*, vol. 64, no. 9, pp. 2623–2636, Sep. 2015.

[4] Z. Zhong, V. Rychkov, and A. Lastovetsky, "Data partitioning on multicore and multi-GPU platforms using functional performance models," *IEEE Trans. Comput.*, vol. 64, no. 9, pp. 2506–2518, Sep. 2015.

[5] K. Li, W. Yang, and K. Li, "Performance analysis and optimization for SpMV on GPU using probabilistic modeling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 1, pp. 196–205, Jan. 2015.

[6] C. Chen, K. Li, A. Ouyang, Z. Tang, and K. Li, "GPU-accelerated parallel hierarchical extreme learning machine on Flink for big data," *IEEE Trans. Syst. Man Cybern.: Syst.*, vol. 47, no. 10, pp. 2740–2753, Oct. 2017.
[7] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Comput.*, vol. 22, no. 6, pp. 789–828, 1996.
[8] L. Dagum and R. Enon, "OpenMP: An industry-standard API for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Jan.–Mar. 1998.
[9] P. Carbone, G. Fra, S. Ewen, S. Haridi, and K. Tzoumas, "Lightweight asynchronous snapshots for distributed dataflows," arXiv preprint arXiv:1506.08603, 2015.
[10] C. Chen, K. Li, A. Ouyang, Z. Tang, and K. Li, "GFlink: An in-memory computing architecture on heterogeneous CPU-GPU clusters for big data," in *Proc. Int. Conf. Parallel Process.*, 2016, pp. 542–551.
[11] C. Li, Y. Yang, Z. Lin, and H. Zhou, "Automatic data placement into GPU on-chip memory resources," in *Proc. IEEE/ACM Int. Symp. Code Generation Optimization*, 2015, pp. 23–33.
[12] N. Fauzia and P. Sadayappan, "Characterizing and enhancing global memory data coalescing on GPUs," in *Proc. IEEE/ACM Int. Symp. Code Generation Optimization*, 2015, pp. 12–22.
[13] T. Ben-Nun, E. Levy, A. Barak, and E. Rubin, "Memory access patterns: The missing piece of the multi-GPU puzzle," in *Proc. SC-Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2017, Art. no. 19.
[14] I. J. Sung, G. D. Liu, and W. M. W. Hwu, "DL: A data layout transformation system for heterogeneous computing," in *Proc. Innovative Parallel Comput.*, 2012, pp. 1–11.
[15] Aparapi in AMD developer website, 2016. [Online]. Available: http://developer.amd.com/tools-and-sdks/opencl-zone/aparapi/, Accessed on: Apr. 1, 2016.
[16] W. Fang, B. He, Q. Luo, and N. K. Govindaraju, "Mars: Accelerating MapReduce with graphics processors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 4, pp. 608–620, Apr. 2011.
[17] J. Stuart, J. D. Owens et al.,"Multi-GPU MapReduce on GPU clusters," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2011, pp. 1068–1079.
[18] L. Chen, X. Huo, and G. Agrawal, "Accelerating MapReduce on a coupled CPU-GPU architecture," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2012, Art. no. 25.
[19] M. Grossman, M. Breternitz, and V. Sarkar, "HadoopCL2: Motivating the design of a distributed, heterogeneous programming system with machine-learning applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 3, pp. 1–1, Mar. 2016.
[20] I. El-Helw, R. Hofman, and H. E. Bal, "Scaling MapReduce vertically and horizontally," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2014, pp. 525–535.
[21] I. El-Helw, R. Hofman, and H. E. Bal, "Glasswing: Accelerating MapReduce on multi-core and many-core clusters," in *Proc. Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2014, pp. 295–298.
[22] Y. Yuan, M. F. Salmi, Y. Huai, K. Wang, R. Lee, and X. Zhang, "Spark-GPU: An accelerated in-memory data processing engine on clusters," in *Proc. IEEE Int. Conf. Big Data*, 2016, pp. 273–283.
[23] P. Li, Y. Luo, N. Zhang, and Y. Cao, "HeteroSpark: A heterogeneous CPU/GPU spark platform for machine learning algorithms," in *Proc. IEEE Int. Conf. Netw. Archit. Storage*, 2015, pp. 347–348.
[24] M. Li, L. Zeng, S. Meng, J. Tan, L. Zhang, A. R. Butt, and N. Fuller, "MRONLINE: MapReduce online performance tuning," in *Proc. Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2014, pp. 165–176.
[25] Y. Guo, J. Rao, D. Cheng, and X. Zhou, "iShuffle: Improving Hadoop performance with shuffle-on-write," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 6, pp. 1649–1662, Jun. 2017.
[26] S. Liu, H. Wang, and B. Li, "Optimizing shuffle in wide-area data analytics," in *Proc. IEEE Int. Conf. Distrib. Comput. Syst.*, 2017, pp. 560–571.
[27] R. Mokhtari and M. Stumm, "BigKernel-high performance CPU-GPU communication pipelining for big data-style applications," in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp.*, 2014, pp. 819–828.
[28] J. Jenkins, J. Dinan, P. Balaji, T. Peterka, N. F. Samatova, and R. Thakur, "Processing MPI derived datatypes on noncontiguous GPU-resident data," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 10, pp. 2627–2637, Oct. 2014.
[29] L. Chen and G. Agrawal, "Optimizing MapReduce for GPUs with effective shared memory usage," in *Proc. Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2012, pp. 199–210.
[30] T. Gao, Y. Guo, B. Zhang, P. Cicotti, Y. Lu, P. Balaji, and M. Taufer, "Mimir: Memory-efficient and scalable MapReduce for large supercomputing systems," in *Proc. Parallel Distrib. Process. Symp.*, 2017, pp. 1098–1108.
[31] S. Huang, J. Huang, J. Dai, and T. Xie, "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis," in *Proc. IEEE Int. Conf. Data Eng. Workshops*, 2010, pp. 41–51.

Cen Chen is working toward the PhD degree in computer science at Hunan University, China. His research interests include parallel and distributed computing systems, machine learning on big data. He has published several research articles in international conference and journals of machine learning algorithms and parallel computing.

Kenli Li received the PhD degree in computer science from the Huazhong University of Science and Technology, China, in 2003. His major research areas include parallel computing, high-performance computing, grid, and cloud computing. He has published more than 200 research papers in international conferences and journals such as the *IEEE Transactions on Computers*, the *IEEE Transactions on Parallel and Distributed Systems*, and ICPP. He serves on the editorial board of the *IEEE Transactions on Computers*. He is a senior member of the IEEE.

Aijia Ouyang received the PhD degree in computer science from Hunan University, China, in 2015. His research interests include parallel computing, cloud computing, and big data. He has published more than 20 research papers in international conference and journals of intelligence algorithms and parallel computing.

Keqin Li is a SUNY distinguished professor of computer science with the State University of New York. His current research interests include parallel computing and distributed computing. He has published more than 550 journal articles, book chapters, and refereed conference papers, and has received several best paper awards. He is currently serving or has served on the editorial boards of the *IEEE Transactions on Computers*, the *IEEE Transactions on Parallel and Distributed Systems*, the *IEEE Transactions on Cloud Computing*, and the *IEEE Transactions on Services Computing*. He is a fellow of the IEEE.