GPU in-memory processing using Spark for iterative computation

Sumin Hong, Woohyuk Choi, Won-Ki Jeong
School of Electrical and Computer Engineering
Ulsan National Institute of Science and Technology, Ulsan, Republic of Korea
{*sumin246,whchoi,wkjeong*}*@unist.ac.kr*

***Abstract*—Due to its simplicity and scalability, MapReduce has become a de facto standard computing model for big data processing. Since the original MapReduce model was only appropriate for embarrassingly parallel batch processing, many follow-up studies have focused on improving the efficiency and performance of the model. Spark follows one of these recent trends by providing in-memory processing capability to reduce slow disk I/O for iterative computing tasks. However, the acceleration of Spark's in-memory processing using graphics processing units (GPUs) is challenging due to its deep memory hierarchy and host-to-GPU communication overhead. In this paper, we introduce a novel GPU-accelerated MapReduce framework that extends Spark's in-memory processing so that iterative computing is performed only in the GPU memory. Having discovered that the main bottleneck in the current Spark system for GPU computing is data communication on a Java virtual machine, we propose a modification of the current Spark implementation to bypass expensive data management for iterative task offloading to GPUs. We also propose a novel GPU in-memory processing and caching framework that minimizes host-to-GPU communication via lazy evaluation and reuses GPU memory over multiple mapper executions. The proposed system employs message-passing interface (MPI)-based data synchronization for inter-worker communication so that more complicated iterative computing tasks, such as iterative numerical solvers, can be efficiently handled. We demonstrate the performance of our system in terms of several iterative computing tasks in big data processing applications, including machine learning and scientific computing. We achieved up to 50 times speed up over conventional Spark and about 10 times speed up over GPU-accelerated Spark.**

***Keywords*-Spark, MapReduce, GPU, In-memory Computing**

I. INTRODUCTION

The MapReduce framework proposed by Google [1] has evolved into a standard high-level programming model for large-scale distributed data processing. It was originally proposed for unstructured text data processing, but its application has expanded to various fields, such as graph analysis, data mining, and machine learning. MapReduce decomposes a program into a collection of two abstract stages – *map* and *reduce* – that are borrowed from the semantics of functional programming. Many open-source implementations of the MapReduce programming model exist, and among them, Apache Hadoop [2] and Spark [3] are widely used in industry and academia. The strength of the MapReduce programming model lies in its usability, scalability, and reliability, as shown in Hadoop and Spark. All intermediate data communication is done via *shuffle*, which is a user-transparent data sorting and collecting process by key. The programmer only needs to focus on developing the *Mapper* and *Reducer* and does not need to consider complex data communication or task decomposition for the distributed system. Therefore, Hadoop and Spark have successfully adopted for many big data processing problems that can be easily fitted to the MapReduce programming model.

Although MapReduce is now a de facto standard for big data processing, the original proposal [1] was not designed for interactive data processing. Every transformation (e.g., the mapper or reducer execution) requires slow disk I/O to store and shuffle intermediate data objects, significantly impacting overall performance. This MapReduce computing model is better suited to batch or streaming processing of large data stored on disk. This constraint severely impairs the usability of the framework because many computing tasks require repetitive data access. To address this problem, Spark introduced caching data in memory so that memory objects can be directly passed between MapReduce operations. By using *in-memory* caching, low-latency computing employing distributed data and iterative computing using shared data can be effectively performed in memory without accessing the disk.

A recent trend in high-performance computing (HPC), in contrast to the approaches mentioned above, has involved leveraging many-core accelerators, such as graphics processing units (GPUs) and many integrated cores (MICs), to increase throughput. Such computing accelerators have wide single instruction multiple datastream (SIMD) lanes with many parallel computing cores. Thus, they map well to data-parallel tasks, that is, tasks that involve executing the same operation on many data elements concurrently. Following this trend, many highly ranked supercomputers in the recent top500 list [4] have been equipped with GPUs or MICs. Similarly, there have been recent research efforts to leverage GPUs in MapReduce systems. Earlier work focused on developing new GPU-based MapReduce systems [5], [6]; later, adopting GPUs in the Hadoop ecosystem became a major research trend [7]–[9]. Due to its growing popularity, Spark has recently received attention for the GPU-accelerated MapReduce framework [10]–[12]. The previous studies mostly focused on task offloading and speed-up issues related to GPUs – for example, how to manage GPU

tasks in a Java virtual machine (JVM) or how to leverage heterogeneous computing resources. Recently, Ohno *et al.* [13] introduced a Spark extension for caching intermediate array data into local and remote GPU device memory. However, efficient memory and execution management for iterative *in-memory* MapReduce computing on distributed GPU cluster systems has not yet been fully explored.

The main motivation of our work[1] stems from our recent attempt to use Spark as a distributed computing platform for scientific computing problems, such as computational fluid dynamics (CFD). Initially, we thought that Spark's in-memory processing would possibly map well to iterative algorithms for partial differential equation (PDE) solvers. However, we realized that such processing only works for *read-only* memory objects; direct (i.e., in-place) modification of a memory object, which is a common operation in iterative algorithms, is not feasible in Spark. Therefore, a naïve implementation of an in-memory iterative algorithm in Spark results in allocating a new memory object per iteration, causing a JVM stack overflow. Another performance issue we observed is that, although the previous literature reported reasonable performance gain when using GPUs in Spark, the major bottleneck of GPU-accelerated Spark came not from the computation but from the system overhead. For example, whenever a Spark transformation is executed, a new worker process is spawned per node, data and parameters are serialized and passed to the worker from the driver process, and the worker decodes them to execute. We also discovered that the JVM I/O operations result in an unusual system overhead, especially in PySpark. This Spark-oriented execution overhead could be in the order of tens of seconds, which is significant compared to the GPU kernel execution time. Finally, no previous work has addressed *GPU in-memory* processing, meaning that caching data in GPU memory for iterative computing. The GPU deepens the memory hierarchy of Spark because it has its own memory that explicitly managed by the user. Without careful management of GPU memory, iterative execution of the GPU program in Spark will introduce an additional performance bottleneck.

To address such problems, we propose a novel Spark extension that specifically targets efficient GPU in-memory caching and processing. The proposed system introduces a GPU worker that maintains GPU context and caches data in GPU memory. Unlike Spark's CPU worker, which is dynamically spawned and terminated for each Spark operation, the proposed GPU worker persists over the entire lifetime of the Spark runtime system. Therefore, data in the GPU memory can be shared by multiple Spark operations without CPU-GPU and CPU-JVM data transfers, which greatly reduces the running time. In addition, the proposed system allows direct modification of GPU memory – that is, overwriting of existing GPU memory objects with new ones – so that iterative computation can be performed solely in the GPU memory. The GPU worker also supports direct inter-GPU communication based on a message-passing interface (MPI) so that many iterative algorithms based on a domain decomposition method can be efficiently parallelized in Spark without expensive data exchange via a disk-based shuffle process. We provide a set of new GPU-specific application program interfaces (APIs) for PySpark that can be used as a building block for further customization of Spark with minimal modification. We demonstrate the performance of the proposed system in several iterative computing applications that benefit from GPU in-memory processing, such as logistic regression, K-means clustering, and iterative numerical solvers for heat and Navier-Stokes PDEs.

[1]Source code is available at http://hvcl.unist.ac.kr/vispark/

II. Background

A. Spark and in-memory processing

Spark provides a novel data abstraction method called Resilient Distributed Dataset (RDD) [14]. An RDD is a read-only collection of data objects stored in a distributed memory system. It can be processed by two types of Spark operations: one is a *transformation* that converts an input RDD to a new RDD (e.g., `map`, `filter`, `reduceByKey`, `join`, etc.), and the other is an *action* that generates the output data from an input RDD (e.g., `reduce`, `collect`, `foreach`, etc.). When the user performs an action, the Spark scheduler examines the RDD's dependency graph for the action, also called a *lineage graph*. In a lineage graph, two types of dependencies exist between RDDs: *narrow* and *wide*. The narrow dependency is when the child RDD has only a single parent RDD, whereas the wide dependency is when the child RDD has multiple parent RDDs. Figure 1 shows an example of an RDD dependency graph. In this figure, `map` and `filter` are transformations resulting in a narrow dependency, whereas `join` and `groupByKey` result in a wide dependency. The Spark scheduler combines RDDs in a narrow dependency, called a pipelined RDD (Figure 1, dotted box), and processes them as a single fused transformation.

Another novel feature of Spark is that RDDs can be stored in the JVM's heap memory via `persist` or `cache`. Unlike other MapReduce systems (e.g., Hadoop) that store RDDs to disk after each operation, Spark can keep RDDs in memory and reuse them across multiple operations, which greatly reduce the running time. Spark's in-memory caching is especially useful when the task is memory-bound and is performing iterative accesses of the same RDDs (e.g., logistic regression).

B. Using GPUs in a MapReduce framework

Previous studies that leveraged the computing power of GPUs in a MapReduce framework had mainly focused

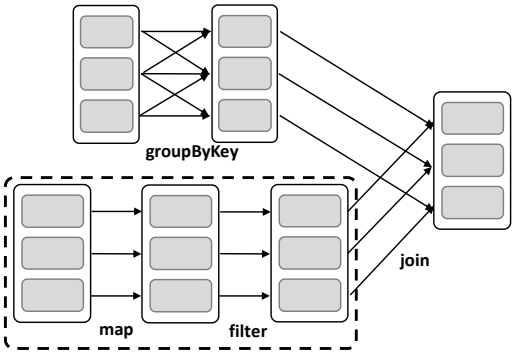


Figure 1: RDD transformations in Spark. Transformations in a narrow dependency, e.g., `map` and `filter` in this example, are pipelined and processed as a single transformation by the Spark scheduler.

on building a new GPU-enabled MapReduce system from scratch. However, as open-source MapReduce systems (e.g., Hadoop and Spark) and their ecosystems become popular, current studies focus more on bridging GPU code and Hadoop or Spark systems. A general approach to use GPUs in existing MapReduce systems is using GPU-wrapper APIs, for example, APARAPI [15] for Java or PyCUDA [16] for Python, to manually execute GPU code in the user-level MapReduce code. Listing 1 shows an example of a CUDA-enabled PySpark map function using PyCUDA APIs, which includes explicit handling of CUDA code and GPU memory.

Listing 1: A PyCUDA example of a CUDA kernel execution in a PySpark user code.

```
def map_func(data,args):
  import pycuda.autoinit
  import pycuda.driver as drv
  from pycuda.compiler import SourceModule

  mod = SourceModule(...)
  func = mod.get_function(...)

  dest = numpy.zeros_like(data)

  func(drv.Out(dest),drv.In(a),drv.In(args),
      block(...),grid(..))

  return dest
```

This method is a simple and fast approach to use GPU computing capabilities in an existing MapReduce framework without modification. Current Hadoop and Spark systems allow multiple CPU worker processes running in the same compute node; these processes are frequently created and terminated based on a system's scheduling policy. Because the above approach creates a GPU context in the user function, it has a limited lifetime bound by that of the process. This eventually causes frequent creation and termination of the GPU context as well. In addition, because each worker process has its own GPU context that cannot be shared

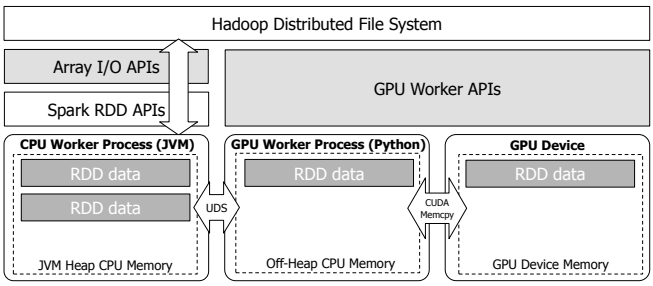


Figure 2: The software stack and architecture of the proposed system (a single GPU node is shown here). The arrows show the data flow from disk to worker processes and the GPU.

with other processes, the data in the GPU memory have to be transferred back to the host JVM memory whenever the worker process terminates. We observed that this results in inefficiency owing to frequent CPU-GPU communication and hinders the reuse of GPU memory over multiple operations. This motivated us to develop a system that supports a persistent GPU context that is independent of worker processes. Our proposed system is described in greater detail in the following sections.

III. Proposed System

A. System Overview

The proposed system is based on PySpark, which is a Python wrapper for Spark. To minimize the modification of Spark, we introduce only a few new functions to PySpark, all of which are written in Python. These new functions can be easily merged with the current PySpark as add-on features. One group of newly introduced functions is special I/O operations to support structured array data on the Hadoop distributed file system (HDFS). The other group of functions is APIs for GPU workers, which handle data communication between CPU and GPU workers and GPU kernel execution. Table I lists the GPU worker APIs newly introduced in the proposed system.

Figure 2 shows the overview of the proposed system architecture on a single computing node with a GPU. When the system starts, the GPU worker daemon (Python process) and Spark's CPU worker daemon (Java process) are launched. The proposed array data I/O APIs and Spark's default I/O APIs can be used to load data from the HDFS to create an RDD by spawning a new CPU worker process from the daemon. In this case, the RDD data are located in the JVM heap memory. Then the data are transferred from the CPU worker to the GPU worker via Unix domain socket (UDS) communication and then stored in the off-heap CPU memory (i.e., VRDD in Section III-B). To run the GPU kernel, the data in the GPU worker are copied to the GPU device memory using the CUDA memcpy command. The GPU workers are also bound by an MPI

communication system to ensure efficient communication among GPU worker processes in different nodes.

B. Virtual RDD for GPU: VRDD

We introduce an extension of Spark's RDD for GPU memory objects, named *VRDD*. There exist two types of VRDD; one is for the GPU worker and is stored in the off-heap CPU memory, and the other is for the GPU device and is stored in the GPU device memory. Because we implement GPU worker functions (see Table I) as Spark transformations, their output is also a Spark RDD object. However, GPU worker functions should manage memory objects either in the off-heap memory or in the GPU device memory, so we extend RDD to represent *Virtual* RDD whose data field is empty but other necessary information (e.g., information of data, parameters, etc) is stored. Because VRDD is virtual, there is no serialization or deserialization required for RDD transformation when a GPU worker function generates a VRDD. Instead, only a necessary direct memory transfer is performed either using UDS or CUDA memcpy commands. Figure 3 describes RDDs and VRDDs for a GPU in-memory caching example.

C. GPU worker

The proposed system manages GPU workers that are solely dedicated to GPU tasks, independently of the Spark's worker JVM. There are several benefits to having GPU workers outside the CPU worker's JVM, as listed below:

Permanent GPU context: In the Spark runtime system, the CPU worker is frequently forked and terminated as a Spark task is executed. If the GPU context is bound to a CPU worker process, it cannot be preserved easily owing to the dynamic process execution and termination. Because our system provides a GPU worker as a daemon, it can be permanent during the lifetime of the Spark runtime system and allow data caching in the GPU memory.

Independent memory management: In the Spark data caching system, RDD data are converted to a byte-array via serialization. This is an essential step for handling a large number of record data in Spark because serialization is useful for reducing the size of data when a cache is created in the memory system. Furthermore, the Spark system is built using JAVA, and thus, in-memory cached data are managed by JVM garbage collection. In this situation, large byte-array data are preferred over small-sized data records to reduce the JVM garbage collection cost. However, serialization and deserialization incur large CPU computing cost, especially for large array data used in Spark. In our system, GPU workers are managed independently of the Spark JVM as python processes. Thus, data can be cached in an off-heap memory directly, and it can be operated without additional serialization and deserialization steps using VRDDs.

Inter-GPU communication via MPI: Data communication in Spark can be performed using a disk-based shuffle operation, but this is not efficient in several applications. For example, in scientific computing on a distributed system, the domain decomposition method, which divides the computing domain into slightly overlapped sub-domains, is a commonly used parallelization method. To synchronize across boundaries of sub-domains, overlapped regions (i.e., *halo*) must be exchanged after computation. However, enabling halo communication based on a shuffle operation in Spark requires shuffling of the entire data. In particular, the Spark shuffling process depends on file reading and writing on slow hard disks; this becomes a critical obstacle to adopting HPC applications in the current Spark system. In our work, GPU workers are connected via MPI, which allows direct data communication among them. This is discussed in greater detail in Section III-G.

In our system, the GPU worker is managed externally from the Spark runtime system. Thus, we provide several new APIs to support GPU workers in the conventional Spark system. Table I lists up the new APIs introduced in our system. These APIs are used for specialized transformations on RDDs.

Table I: GPU worker APIs

Type	description
`send`	Copy data from Spark RDD to GPU worker.
`recv`	Copy data GPU worker to Spark RDD.
`htod`	Copy data from GPU worker to GPU memory.
`dtoh`	Copy data GPU device memory to GPU worker.
`exec`	Execute GPU kernel function.
`cache`	Make consistent copy in GPU memory.
`uncache`	Eliminate consistent copy from GPU memory.
`halo_extract`	Extract halo data in domain decomposition.
`halo_shuffle`	MPI communication of extracted halo data.
`halo_append`	Append halo data for synchronization.
`vmap`	High-level API to execute GPU kernel function.

D. GPU execution model

To execute the GPU kernel function in our framework, Spark RDD uses a combination of GPU API transformations. For example, most general sequences used for GPU kernel execution for Spark RDD data are serial transformations of `send` → `htod` → `exec` → `dtoh` → `recv`. In this process, RDD data is sequentially transferred, transformed, and processed following the operations. When an RDD is transferred to the GPU via `send` and `htod` commands, it becomes a VRDD and is assigned to a unique CUDA stream. In the `exec` operation, the enclosed GPU kernel function code is compiled just-in-time by the GPU worker using a CUDA compiler, and the compiled GPU kernel is executed (all are orchestrated by PyCUDA). Once GPU kernel execution is finished and the resulting VRDD is copied back to the Spark RDD from the GPU worker, the assigned GPU stream is deleted. Unlike Spark, which generates a new RDD for each transformation without freeing previously generated

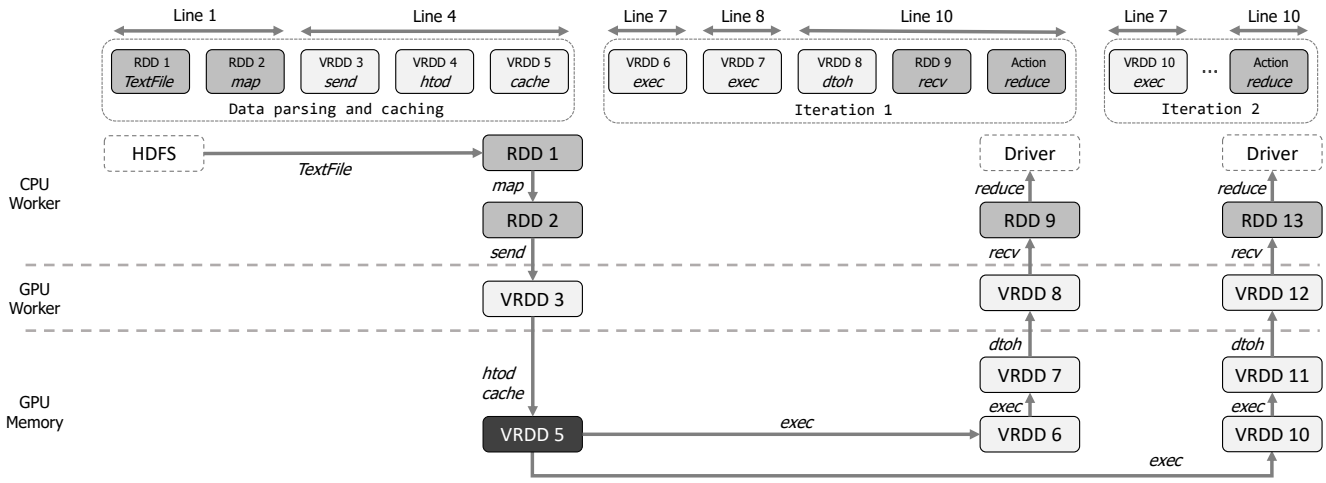


Figure 3: GPU in-memory caching example on logistic regression. The first row shows pipelined RDDs, and below rows show data transformation between CPU and GPU (note how RDDs are converted to VRDDs and vice versa). VRDD5 is cached in the GPU memory and reused across iterations without re-loading from the CPU worker.

RDDs, our GPU execution keeps only the most recent (i.e., current) result and delete previous VRDDs located in the GPU memory for better GPU memory utilization unless they are cached explicitly. The up-to-date VRDD in the GPU memory can be used as an input to the following transformation (in case of GPU in-memory processing) or copied back to the CPU worker. In case of node failures, we rely on the fault recovery mechanism provided by Spark to restore RDDs in the CPU worker level, not VRDD in the GPU worker or device level.

E. Lazy evaluation

Lazy evaluation is one of the most important techniques for achieving in-memory processing in Spark. Spark RDD transformation without wide dependencies such as `map` and `filter` only creates a pipelined RDD without actual execution, and these pipelined RDDs are executed in once. This is useful to reduce frequent I/O overheads, so our system also provides similar methods. To minimize communication between the CPU worker and the GPU worker, most operations, except `recv`, are not directly sent to the GPU worker. Instead, they create pipelined-GPU APIs along with Sparks pipelined RDD system. Then, the pipelined-GPU APIs – `send`, `htod`, `exec`, `dtoh`, and `recv` operations – are sent to the GPU worker with one combined package. At this time, the communications between the CPU worker and the GPU worker are limited to single CPU-TO-GPU and single GPU-TO-CPU communication. This is especially useful when using several `exec` operations consecutively. This lazy evaluation is depicted in Figure 3, for example, transformations from VRDD5 to RDD9 are performed at the end of a pipelined RDD created by the `reduce` action in iteration 1 (a pipelined RDD is shown as a dotted round box in the top row).

F. GPU in-memory cache

By reusing cached data in the GPU device memory, we can reduce the communication cost between the CPU and the GPU or the JVM and the CPU workers. As in the case of Spark in-memory caching, the GPU in-memory cache can persist in a single GPU device by using the `cache` operation.

Listing 2 shows the implementation of logistic regression using low-level GPU worker APIs, and Figure 3 shows the actual data movement and RDD-VRDD transformations in this example. Listing 2 lines 1–4 load the data and cache it in the GPU memory, and lines 7–8 are two GPU kernel executions (one is used for prediction on the training data, and the other is a local reduction performed to extract parameters of the logistic regression model). Line 10 is the global reduction performed by Spark to collect the final result.

Listing 2: Implementation of logistic regression using low-level GPU worker APIs. Functions and arguments are simplified for convenience.

```
1  points = sc.wholeTextFile(DataPath).map('
       parsing')
2
3  #GPU in-memory cache
4  points = points.send().htod().cache()
5
6  for i in range(iterations):
7    new_m = points.exec('exec_args')
8    new_w = new_m.exec('exec_args')
9
10   w = new_w.dtoh().recv().reduce('...')
```

In Figure 3, three pipelined RDDs are shown in the top row (indicated by dotted round boxes). The first pipelined

RDD (lines 1–4) is for loading, parsing, and caching data to the GPU. For this purpose, five Spark map functions are pipelined (i.e., `wholeTextFile` and `map` are Spark native functions, and `send`, `htod`, and `cache` are low-level GPU worker APIs that are newly introduced in this paper). Note that this pipelined RDD starts to process at the end of this pipeline via *lazy evaluation*, when the `cache` is called (serves as an action). After processing this pipelined RDD, the resulting VRDD5 is cached in the GPU memory. In the following iterations, VRDD 5 can be directly accessed by the GPU kernel within the GPU (line 7) to generate VRDD6 and VRDD10 (see the arrows labeled as *exec* from VRDD5 to VRDD6 and to VRDD10 in Figure 3). As shown above, the cached VRDD5 will be reused many times without being copied from the CPU worker as the iterations proceed. Furthermore, note that the user actually needs to write codes in high-level APIs, as shown in Listing 3, which is similar to conventional Spark code except a few minor additions, such as `vmap` for VRDD. We provide a code-to-code translator that automatically converts user code such as shown in Listing 3 to a low-level code such as shown in Listing 2 at runtime. In this example, the translator replaced `vmap` to `exec` and automatically added data transfer APIs such as `send, htod, recv` to the user code.

Listing 3: Implementation of logistic regression using high-level APIs.

```
1  points = sc.wholeTextFile(DataPath).map('
       parsing')
2
3  #GPU in-memory cache
4  points = points.cache("GPU")
5
6  for i in range(iterations):
7    new_m = points.vmap('exec_args')
8    new_w = new_m.vmap('exec_args')
9
10   w = new_w.reduce('...')
```

G. Halo communication

To solve large-scale scientific problems on a distributed system, the domain decomposition method is commonly used. This method splits a domain into small overlapping subdomains so that each subdomain can be processed independently on a different computing node. To maintain correctness across the domain boundaries, the overlapped region (i.e., halo) must be synchronized (Figure 4).

Radenski [17] proposed a domain decomposition method using MapReduce. This method uses the index of the subdomain as the key in the MapReduce processing. Therefore, each mapper gets the pair of the subdomain's index and its data values as an input. However, this halo communication method relies on MapReduce join operations to communicate key-value pairs between sub-domains. This method significantly increases the disk I/O because both the halo regions and the original grid data need to be written to the disk for shuffling. In addition, scientific computing problems generally require many iterations during computation, which significantly impairs the performance.

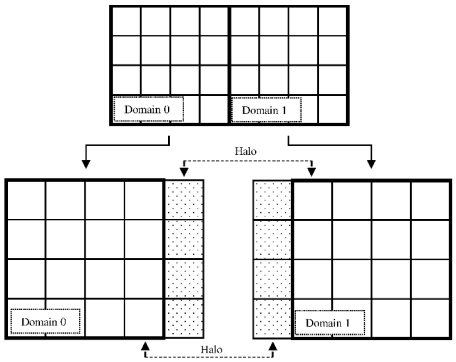


Figure 4: Halo at the boundary of adjacent sub-domains

Listing 4: Code example of halo communication and GPU in-memory processing for iterative computing.

```
1  grids = grids.cache("GPU")
2
3  for i in range(10):
4    #Halo Communication
5    grids.halo_extract().halo_shuffle()
6    grids = grids.halo_append()
7
8    #Computation
9    grids = grids.exec("..do_stencil..").cache()
```

To remove the disk I/O overhead in halo communication, we introduce domain-specific transformations for halo synchronization via efficient MPI communication (see Table I). `halo_extract` collects halo data from the GPU memory and transfers them to the off-heap CPU memory (owned by the Python worker process). `halo_shuffle` exchanges the collected halo data among GPU workers via MPI communication. This process must be synchronized between GPU workers, and therefore, it is performed as a Spark `action` operation. `halo_append` attaches the exchanged halo data to the current region.

Listing 4 shows an example code of halo communication written in the new halo APIs. From the cached data in line 1, line 5 extracts halo data and shuffles them among GPU workers. After the shuffling is complete, the grid is updated using the new halo data (line 6) and the GPU kernel function is executed (`exec` in line 9). Furthermore, note that the `cache` in line 9 provides the most recent VRDD as the input to the next iteration when the same VRDD name is shared in the iteration; this is called *in-memory processing* (see Figure 5, VRDD6 is used as the input for the next iteration). Owing to halo communication, small-sized halo data have to be copied to the GPU worker. However, this overhead is small because the halo size is usually small compared to the domain size and the communication can be performed effectively via MPI.

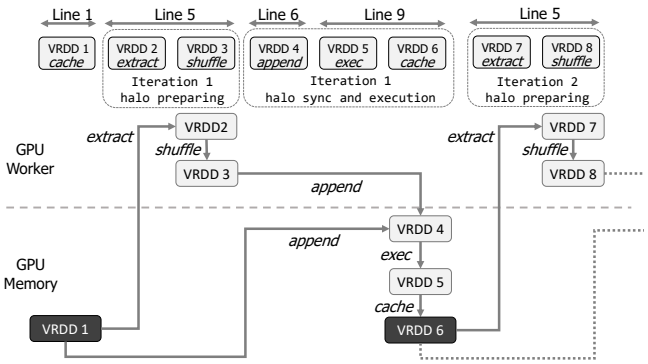


Figure 5: GPU in-memory processing (Spark-G-IM) for iterative computing with halo communication.

IV. Experimental Evaluation

Our proposed framework is developed using Spark version 1.2.1, Python 2.7.6, PyCUDA 2014.1, MPI4Py 2.0.0, and NVIDIA CUDA 7.5. All experiments are conducted on an eight-node GPU cluster system in which each node is equipped with an octa-core Intel Xeon CPU, 64GB main memory, and an NVIDIA GTX Titan X (GM200) with 3072 CUDA cores and 12GB device memory. All eight nodes are connected via a QDR Infiniband network. We selected four representative benchmarks: logistic regression and K-means clustering from machine learning for in-memory caching, and heat equation and Navier-Stokes equation from scientific computing for in-memory processing. All experiments are tested on Python, HDFS (Hadoop Distributed File System), and standalone cluster-mode PySpark environments. To avoid the extra overhead caused by the distributed file system, we adjusted the block size and numbers so that all data access can be performed locally. Each benchmark is evaluated in three different implementations: Spark's in-memory cache with CPU computation (labeled as *Spark-C*), a naïve GPU implementation in Spark using JVM memory to cache data (labeled as *Spark-G*), and our proposed GPU in-memory processing in Spark (labeled as *Spark-G-IM*). For heat transfer simulation, we performed comparisons with one more implementation, *Spark-C-MPI*, to differentiate from the conventional Spark implementation based on shuffle communication. We repeated same experiment 5 times and computed the average for all running time measurement. We also assumed that whole data are cached to CPU and GPU memory (so there is no memory swapping for caching).

A. Logistic regression

Logistic regression is a representative example used by Spark for demonstrating its in-memory caching performance (for example, one argued that Spark in-memory processing is 100 times faster than Hadoop disk-based processing for logistic regression [18]). It is an iterative algorithm that repeatedly accesses the input data with relatively simple computations, and therefore it is considered a memory-bound problem. The dataset used in our experiment contains 32 million 41-dimensional points stored as an 8 GB ASCII file in HDFS.

Figure 6 shows the running time of logistic regression in three different implementations. In this experiment, Spark spent 48 s for in-memory caching, including the time required for disk I/O and data caching in the JVM memory. Once the in-memory cache is used, every iteration takes 10 s. In contrast, conventional Hadoop requires data loading from HDFS in every iteration (i.e., caching time). Thus, this demonstrates that Spark's in-memory caching improves the running time by up to around six times compared to disk-based MapReduce processing. If the computation code is replaced with that of the GPU (Spark-G), then the iteration time decreases further to 5 s while the caching time stays constant. If we exploit the proposed GPU in-memory caching (Spark-G-IM), then both the caching time and iteration time decrease further to 33 s and 0.4 s, respectively. This is almost 25 times faster than Spark in-memory processing and 145 times faster than disk-based MapReduce processing. As shown here, logistic regression is a memory-bound, and the performance gain mainly results from in-memory caching.

Figure 7 shows an in-depth analysis of Spark-G. In the caching process, data parsing, JVM caching and reading, HDFS disk I/O, and the Spark system's task management-related overheads require the maximum time. The iteration time shows that noncomputing tasks require the maximum time, such as JVM read (76%) and Spark's overhead (10%), because logistic regression is memory-bound (this becomes even more severe in Spark-G because the GPU further reduces the computing time). Only less than 0.6 s is actually spent for GPU execution, including CPU-GPU communication. Therefore, this naïve GPU task offloading without reducing other overheads, especially the JVM overheads, is not an effective solution. In contrast, our proposed method (Spark-G-IM) can remove the JVM overhead entirely from

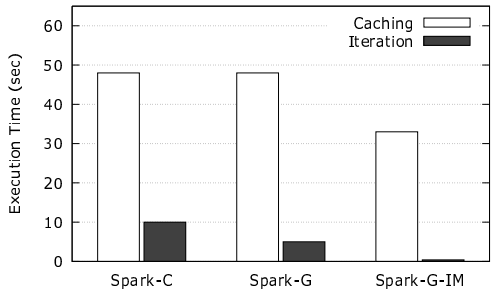
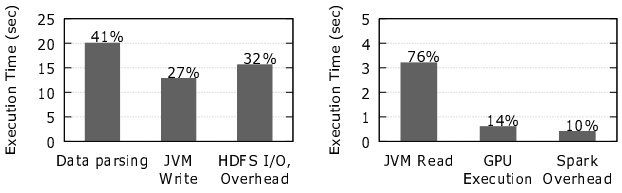


Figure 6: Execution time of a single iteration of logistic regression in various implementations. *Caching* is the running time for data loading and memory caching. *Iteration* is the average per-iteration running time.

the iteration time by caching the data in the GPU device memory, which results in speed-up by more than an order of magnitude (5 s vs. 0.4 s).



(a) Caching (b) Iteration

Figure 7: An in-depth analysis of the caching and the running times of Spark-G for logistic regression.

B. K-means clustering

K-means clustering is another iterative algorithm that is frequently used as an example of Spark in-memory caching. Unlike logistic regression, the computational cost of the K-means algorithm depends on the number of clusters, *K*, and therefore, we can control the bottleneck of the algorithm: it is either memory- or compute-bound. Figure 8 shows the running times of the K-means clustering algorithm as tested on a 9 GB dataset of a collection of 784-dimensional points. We performed tests using K values of 40, 80, and 160. Data caching time only depends not on the number of clusters (i.e., K) but on the data size, so it is same for all K (shown as a single graph on the left in Figure 8).

Compared to logistic regression, the GPU implementation shows higher speed-up as K increases because the problem becomes more compute-bound. For Spark-C, Spark-G, and Spark-G-IM, the iteration times for K = 40 are respectively 34 s, 8 s, and 1 s and those for K = 160 are respectively 104 s, 12 s, and 2 s. This result indicates that the speed-up factor of Spark-G-IM compared to Spark-C increases from 34 times to 50 times as K increases owing to the GPU acceleration. The estimated maximum speed-up of Spark-G-IM compared to disk-based MapReduce processing (i.e., Spark-C's caching + iteration) is around 100 times.

C. Heat transfer simulation

Heat transfer simulation is a representative example of a scientific computing problem that is suitable for in-memory processing with halo communication. The simulation iteratively solves the heat PDE (Equation 1), in which the derivatives are approximated by using a central difference discretization scheme.

$$\frac{\partial u}{\partial t} = h^2 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right) \quad (1)$$

We ran our experiment on a $2000 \times 1000 \times 1000$ three-dimensional rectilinear grid, that has a raw data size of 8 GB for a single-precision floating point per grid node. To distribute tasks across the eight-node cluster system, the grid is divided into 64 subgrids of size 128 MB each.

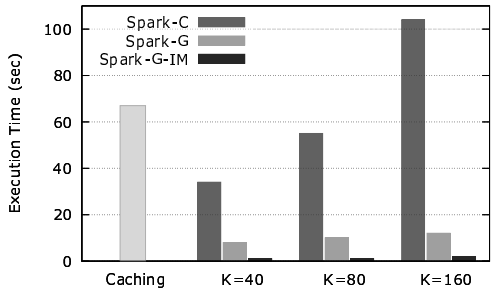


Figure 8: Execution time of a single iteration of K-means clustering algorithm in various implementations. *Caching* measures the data loading and memory caching time, and *K* is the number of clusters.

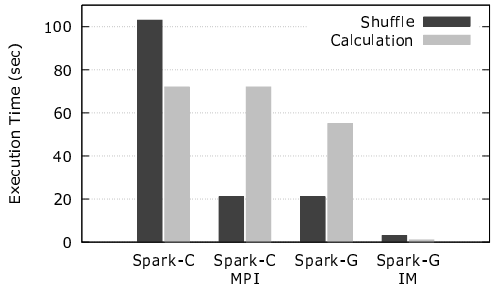
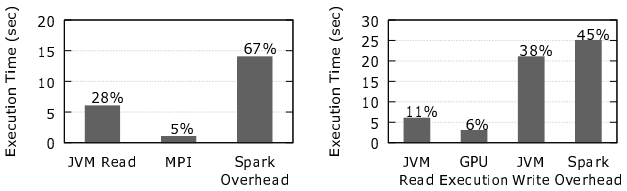


Figure 9: Shuffle and calculation times for a single iteration of the heat transfer simulation.

Unlike machine learning applications, heat transfer simulation requires halo communication between adjacent subgrids. To evaluate and compare the communication cost in greater detail, we introduce another Spark implementation, *Spark-C-MPI*, that exploits MPI for node-to-node direct communication in Spark and bypasses the expensive disk-based shuffle process. Note that Spark-G and Spark-G-IM also use MPI for halo communication in this experiment. Figure 9 shows the halo communication (i.e., shuffle) and calculation time for a single iteration of the heat transfer simulation in various implementations. In Spark-C, the halo communication took 107 s, which is greater than the computation time of 72 s. In Spark-C-MPI, owing to the direct communication of MPI, the shuffle time greatly reduces by a factor of five to 21 s. Naïve GPU implementation, Spark-G, further reduces the computation time to 55 s (24% reduction compared to CPU-based computation). Our GPU in-memory implementation (Spark-G-IM) requires only 3 s for the halo communication and 1 s for the calculation. This result represents speed-up of around 45 times compared to the conventional Spark implementation (Spark-C).

Figure 10 shows an in-depth analysis of the running time of a heat transfer simulation on Spark-G. As in the case of the logistic regression, the JVM read/write and Spark



(a) Shuffle (b) Calculation

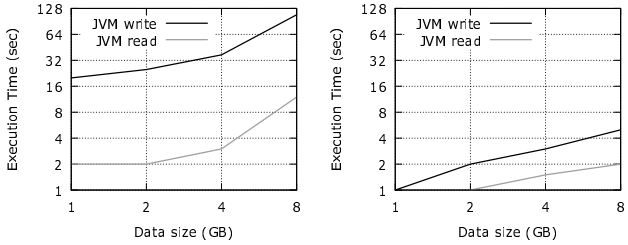
Figure 10: Detailed analysis of the running time of Spark-G for the heat transfer simulation.

overhead take up most of the time and MPI communication and GPU execution times were relatively small. In particular, we observed unusually large Spark overheads of up to 25 s for both the shuffle and the calculation processes, which is unlike the overhead in the logistic regression. In the logistic regression, Spark overheads are large in the caching stage (about 60% among the HDFS I/O and overhead), but small in the iteration time. On the other hand, in the heat transfer simulation, both the shuffle and the calculation times include a large Spark overhead. In the heat transfer simulation, data is cached in the CPU worker memory (i.e., JVM write) owing to Spark in-memory processing. This is similar to the caching stage in the case of the logistic regression. This implies that the Spark overhead increases if there exist a large number of JVM writes.

In every iteration of the logistic regression, only JVM read was performed and the Spark overhead was not large. However, the shuffle stage of the heat transfer simulation only has JVM read owing to direct communication via MPI; nonetheless, it shows a large Spark overhead. The only difference between these cases is that the logistic regression uses smaller data compared to the heat transfer simulation (the Spark system reports that the logistic regression uses 5 GB whereas the heat transfer simulation uses 10 GB internally). This also implies that there seems to be a correlation between the number of JVM I/O operations and the Spark overhead.

D. Discussion

To examine the overhead issue in greater detail, we conducted controlled experiments in which only data I/O operations to the JVM without computation were performed on PySpark and Scalar-based Spark (Figure 11). As shown in this figure, PySpark introduces a significant overhead of up to 130 s (i.e., increased running time) as the JVM I/O size increases. This is an unusual Python-specific overhead; Scalar-based Spark shows a maximum overhead of only 5 s (Figure 11 (b)). In addition, we also measured the timestamp of each worker process to see how well they are synchronized. We found that the CPU workers did not start at the same time in PySpark; in contrast, in Scalar Spark, all worker processes are exactly synchronized. We suspect this is due to the socket communication and data conversion between the JVM and PySpark. In summary, we confirm that there exists a significant system overhead in PySpark that is correlated with the number of JVM I/O operations; this issue is completely avoided in our proposed system because we bypass the JVM I/O and directly cache RDD data to the GPU memory.



(a) Python (b) Scala

Figure 11: Spark-G overhead analysis for various data size on different Spark platforms.

E. Application

To assess the applicability and performance of the proposed GPU in-memory processing method on a more complicated example, we implemented a Navier-Stokes equation solver. The Navier-Stokes equation describes the dynamics of fluid, and it is defined as follows:

$$\frac{\partial \mathbf{u}}{\partial t} = \nu \nabla^2 \mathbf{u} - (\mathbf{u} \cdot \nabla)\mathbf{u} - \frac{1}{\rho_0} \nabla p \qquad (2)$$

Solving the Navier-Stokes equation requires a large number of iterative updates on several buffers that represent the properties of the fluid, such as velocity and pressure. The grid dimension we used is 32768×8192, and each grid node stores six float values. The data size is 6 GB, and it is distributed evenly across eight computing nodes. Each inner iteration took 21.2 s for 24 mapper executions and 24 halo communications. To the best of our knowledge, this is the first Spark implementation that successfully executed a large number of iterative computations *in-GPU-memory* at this scale. We believe that without the proposed GPU in-memory processing framework, only a few iterations of the Navier-Stokes solver would cause the Spark system to fail owing to the accumulation of RDD objects.

V. Related Work

The concept of MapReduce was first introduced by Google [1], and the two most popular open-source MapReduce implementations are Hadoop [2] and Spark [3]. There exists some previous literature on GPU acceleration in the Hadoop system [7]–[9] in which performance gain is mainly realized by accelerating the computation using GPUs. Some more recent studies have focused on leveraging GPUs in Spark [10]–[12]. These studies only focused on the computational aspect of the system; no previous study has

focused on GPU-level in-memory caching and processing as we proposed in this paper. Some studies have combined GPUs with Hadoop and Spark, and others have attempted to implement the proprietary GPU MapReduce framework [5], [6], [19]. Some previous studies have also tried to apply the MapReduce system to nontext data processing applications. Buck *et al.* proposed SciHaoop [20] to bring NetCDF data on Hadoop. Radenski *et al.* [17] introduced a domain decomposition method on the key-value pair system in MapReduce. Vo *et al.* [21] and Choi *et al.* [12] used the MapReduce system for visualization and scientific computing applications.

VI. CONCLUSION

We propose a novel Spark extension for efficient GPU in-memory caching and processing. We discovered unusual system overheads in the PySpark system, and resolved this problem by introducing an external GPU worker that avoids expensive JVM I/O operations and manages off-heap and GPU memory directly. Our proposed system also supports direct inter-GPU communication based on an MPI to enable handling iterative scientific computing problems based on the domain decomposition method efficiently. We provide a set of new GPU-specific APIs for customizing Spark with minimal modifications. As a result, our system is almost 25 times faster compared to Spark in-memory processing and 10 times faster compared to a GPU-accelerated Spark implementation for the logistic regression benchmark. In addition, our proposed system can solve large-scale scientific computing problems such as the heat equation and Navier-Stokes equation efficiently; these are otherwise difficult to solve using the original Spark system.

In future work, we plan to improve the performance of the JVM-Python overhead through in-depth analysis and optimization. We will support remote direct memory access (RDMA) among GPU devices to enable direct peer-to-peer communication. We also plan to develop memory swapping methods for GPU in-memory caching to handle the data larger than GPU's physical memory size. Developing fault-tolerant features for GPU in-memory processing could be another interesting future research direction.

ACKNOWLEDGMENT

This work is partially supported by the Institute for Information & communications Technology Promotion (IITP) grant (No. R0190-16-2012, High Performance Big Data Analytics Platform Performance Acceleration Technologies Development) and the Bio & Medical Technology Development Program of the National Research Foundation of Korea (NRF-2015M3A9A7029725) funded by the Korean government (MSIP), and the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2014R1A1A2058773).

REFERENCES

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10.

[2] Apache. Hadoop. [Online]. Available: https://hadoop.apache.org/

[3] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10.

[4] "TOP500 Supercomputer Site." [Online]. Available: http://www.top500.org

[5] W. Fang, B. He, Q. Luo, and N. K. Govindaraju, "Mars: Accelerating mapreduce with graphics processors," Parallel and Distributed Systems, IEEE Transactions on, vol. 22, no. 4, pp. 608–620, 2011.

[6] J. A. Stuart and J. D. Owens, "Multi-GPU MapReduce on GPU Clusters," in Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium, ser. IPDPS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 1068–1079.

[7] A. Abbasi, F. Khunjush, and R. Azimi, "A preliminary study of incorporating GPUs in the Hadoop framework," in Computer Architecture and Digital Systems (CADS), 2012 16th CSI International Symposium on. IEEE, 2012, pp. 178–185.

[8] M. Grossman, M. Breternitz, and V. Sarkar, "HadoopCL: MapReduce on distributed heterogeneous platforms through seamless integration of hadoop and OpenCL," in Proceedings - IEEE 27th International Parallel and Distributed Processing Symposium Workshops and PhD Forum, IPDPSW 2013, 2013, pp. 1918–1927.

[9] J. Zhu, J. Li, E. Hardesty, H. Jiang, and K. C. Li, "GPU-in-Hadoop: Enabling MapReduce across distributed heterogeneous platforms," in 2014 IEEE/ACIS 13th International Conference on Computer and Information Science, ICIS 2014 - Proceedings, 2014, pp. 321–326.

[10] P. Li, Y. Luo, N. Zhang, and Y. Cao, "HeteroSpark: A heterogeneous CPU/GPU Spark platform for machine learning algorithms," in Proceedings of the 2015 IEEE International Conference on Networking, Architecture and Storage, NAS 2015, 2015, pp. 347–348.

[11] D. Manzi and D. Tompkins, "Exploring GPU acceleration of apache spark," in Proceedings - 2016 IEEE International Conference on Cloud Engineering, IC2E 2016: Co-located with the 1st IEEE International Conference on Internet-of-Things Design and Implementation, IoTDI 2016, 2016, pp. 222–223.

[12] W. Choi, S. Hong, and W.-K. Jeong, "Vispark: GPU-accelerated distributed visual computing using spark," SIAM Journal on Scientific Computing, vol. 38, no. 5, pp. S700–S719, 2016.

[13] Y. Ohno, S. Morishima, and H. Matsutani, "Accelerating spark rdd operations with local and remote gpu devices," in Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS16), 2016.

[14] 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 Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 2–2.

[15] AMD. Aparapi. [Online]. Available: http://aparapi.github.io/

[16] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation," Parallel Computing, vol. 38, no. 3, pp. 157–174, 2012.

[17] A. Radenski, "Big data, high-performance computing, and MapReduce," in Proceedings of the 15th International Conference on Computer Systems and Technologies. ACM, 2014, pp. 13–24.

[18] "Putting Spark to Use: Fast In-Memory Computing for Your Big Data Applications." [Online]. Available: https://blog.cloudera.com/blog/2013/11/putting-spark-to-use-fast-in-memory-computing-for-your-big-data-applications/

[19] C. Basaran and K.-D. Kang, "Grex: An efficient MapReduce framework for graphics processing units," Journal of Parallel and Distributed Computing, vol. 73, no. 4, pp. 522–533, 2013.

[20] J. B. Buck, N. Watkins, J. LeFevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, and S. Brandt, "SciHadoop: Array-based query processing in Hadoop," in Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. ACM, 2011, p. 66.

[21] H. T. Vo, J. Bronson, B. Summa, J. L. D. Comba, J. Freire, B. Howe, V. Pascucci, and C. T. Silva, "Parallel visualization on large clusters using MapReduce," in Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on. IEEE, 2011, pp. 81–88.