Accelerating Spark RDD Operations with Local and Remote GPU Devices

Yasuhiro Ohno, Shin Morishima, and Hiroki Matsutani
Dept. of ICS, Keio University,
3-14-1 Hiyoshi, Kohoku, Yokohama, Japan 223-8522
Email: {ohno,morisima,matutani}@arc.ics.keio.ac.jp

***Abstract*—Apache Spark is a distributed processing framework for large-scale data sets, where intermediate data sets are represented as RDDs (Resilient Distributed Datasets) and stored in memory distributed over machines. To accelerate its various computation intensive operations, such as reduction and sort, we focus on GPU devices. We modified Spark framework to invoke CUDA kernels when computation intensive operations are called. RDDs are transformed into array structures and transferred to GPU devices when necessary. Although we need to cache RDDs in GPU device memory as much as possible in order to hide the data transfer overhead, the number of local GPU devices mounted in a host machine is limited. In this paper, we propose to use remote GPU devices which are connected to a host machine via a PCI-Express over 10Gbps Ethernet technology. To mitigate the data transfer overhead for remote GPU devices, we propose three RDD caching policies for local and remote GPU devices. We implemented various reduction programs (e.g., Sum, Max, LineCount) and transformation programs (e.g., SortByKey, PatternMatch, WordConversion) using local and remote GPU devices for Spark. Evaluation results show that Spark with GPU outperforms the original software by up to 21.4x. We also evaluate the RDD caching policies for local and remote GPU devices and show that a caching policy that minimizes the data transfer amount for remote GPU devices achieves the best performance.**

***Keywords*—Apache Spark, RDD, GPU, CUDA, PCIe over 10GbE**

I. Introduction

Due to the recent advances on mobile and IoT devices, social networking services, and cloud computing, data sets to be stored and analyzed are continuously growing in size and diversity. Distributed data processing framework is thus a key component to analyze such large data sets.

Various distributed data processing frameworks are available. Apache Hadoop [1] is one of representative open-source products for this purpose. As a storage layer, it uses HDFS (Hadoop Distributed File System). For each processing, input data sets are read from the distributed file system and the processing results are stored again. This approach is fault-tolerant as data sets are partitioned and stored in multiple nodes or disks. However, for applications that iteratively perform operations on the same data sets (e.g., machine learning), since input/output data sets are read/written from/to the distributed file system for each iteration, it requires a number of disk accesses, resulting in a disk access bottleneck.

Apache Spark [2] is another representative open-source distributed data processing framework where intermediate data sets are stored in a distributed shared memory. More specifically, the data sets are represented as RDDs (Resilient Distributed Datasets) [3] and distributed over machines. Various operations can be performed on a given RDD as input data and the computation result is stored as another RDD. Thus, for applications that iteratively perform operations on the same data sets, we can drastically eliminate the disk access to read/write intermediate data. Spark framework provides various functions onto RDDs and they can be classified into two categories: Action and Transformation. The computation result of an Action function (e.g., Max, Sum) is a scalar or vector value, while that of a Transformation function (e.g., Sort) is another RDD.

Since these functions are computation-intensive, in this paper, we accelerate them by using GPU (Graphics Processing Unit) devices. As data transfer between a host and GPU devices is a performance bottleneck, we try to cache RDDs in GPU device memory as much as possible in order to hide the data transfer overhead. However, since the number of local GPU devices mounted in a host machine is limited, cached RDDs in GPU device memory would be frequently swapped in and out. To mitigate the overhead, we propose to use remote GPU devices which are connected to a host machine via a PCI-Express (PCIe) over 10Gbps Ethernet (10GbE) technology. We can increase the number of remote GPU devices and the amount of available memory for parallel processing. However, data transfer overhead of remote GPU devices is higher than that of local GPU devices. To use both local and remote GPU devices efficiently, we need to take into account the size and access frequency of RDDs when we cache them in local or remote GPU devices. In this paper, we propose three RDD caching policies for local and remote GPU devices [1].

The rest of paper is organized as follows. Section II overviews background knowledge. Section III illustrates our Array Cache design for GPU processing of Spark Action and Transformation functions, and Section IV extends it to local and remote GPU devices. Section V mentions the Array Cache implementation. Section VI evaluates various Spark operations using local and remote GPU devices. Section VII concludes this paper.

[1] An early stage of this work appears at HEART 2016 as a short presentation, which is not included nor published in the conference post proceedings.

II. Background and Related Work

A. Spark and RDD

Apache Spark is a distributed processing framework where data sets are represented as in-memory RDDs [3]. The framework is implemented in Scala language. Although HDFS is used as a data source, the intermediate results are stored as in-memory RDDs; thus it is efficient especially for applications that iteratively perform certain operations on the same data, such as machine learning. In Spark framework, a driver node distributes tasks and data sets to worker nodes and they perform the tasks on a given data. Then the driver node collects and aggregates the computation results from the worker nodes. In this paper, we focus on the worker nodes and modify them so that they invoke CUDA kernels when they perform computation intensive operations on RDDs.

RDD operations can be classified into two categories: Action and Transformation functions. Action functions perform an aggregation operation on a given RDD and the computation result is a scalar or vector value. Transformation functions perform a conversion operation on a given RDD and the result is stored as a new RDD. As Action functions, in this paper, we accelerate reduce() [2], max(), min(), and count() by using GPU devices. In addition, as Transformation functions, we accelerate sortByKey(), map(), and filter() by using GPU devices. Please note that although the above-mentioned functions do not cover all the functions provided by Spark framework, they are representative ones in Spark and we can implement GPU processing for the remaining functions in the same way. Since intermediate data sets are represented as RDDs, a series of Action and Transformation functions can be represented as a graph where a node is an RDD and an edge is an Action or Transformation function.

B. GPU Processing on Spark Framework

HeteroSpark [4] is based on a Spark framework and it utilizes GPU devices in addition to CPUs. cuSpark is another framework similar to Spark which performs parallel processing using GPU devices on a single machine [5]. However, detail implementations regarding how to process and cache RDDs for GPU devices are not discussed in detail. In this paper, we introduce Array Cache which is an array structure extracted from a given RDD and will be cached in the GPU device memory.

Other than Spark, conventional MapReduce frameworks that employ GPU devices have been studied so far [6][7][8]. Mars [6] is an example of such GPU-based MapReduce frameworks. To utilize the massive thread parallelism of GPU devices, it provides a small set of APIs similar to CPU-based MapReduce framework. Optimization techniques that utilize the shared memory have been proposed in [7]. An OpenCL MapReduce framework optimized for AMD GPUs has been proposed in [8]. Our work is specialized for the in-memory RDDs of Spark and we propose to cache them in local and remote GPUs' device memory.

In [9], a document-oriented database is accelerated by utilizing GPU devices. To process a number of BSON documents with GPU devices, [9] proposes DDB Cache which is an array structure, where documents for the same field are extracted and represented as a Compressed Row Storage (CRS) format. A similar array structure based on CRS format is used for accelerating a graph database by using GPU devices [10]. Our Array Cache used in this paper for GPU acceleration of Spark also uses a similar array structure based on CRS format.

[2]Spark built-in function is denoted as "function()" in this paper.

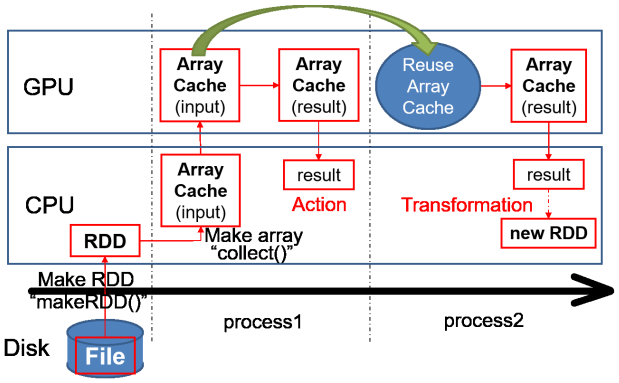


Fig. 1. Making Array Cache for GPU processing

In addition, we propose to use remote GPU devices which are connected to a host machine via PCIe over 10GbE, in order to increase the number of remote GPU devices when needed. We employ NEC ExpEther [11][12] as a PCIe over 10GbE technology (see Figure 3). Using ExpEther, PCIe packets for hardware resources (e.g., GPU devices) are transported in 10GbE by encapsulating the packets into an Ethernet frame. Please note that there is software service based on client-server model which provides GPU computation for clients [13], while we employ a PCIe over 10GbE technology for connecting a lot of GPU devices directly. Remote GPUs devices via such a PCIe over 10GbE technology are used for database processing [14], while they have not been studied for Spark nor MapReduce. Another contribution of this paper is to explore RDD caching policies based on a static analysis of a given Spark application for local and remote GPU devices.

III. Spark Framework with GPU

A. System Overview

Figure 1 illustrates a flow of performing Action and Transformation functions of Spark using GPU device. First, a given data set is transformed into an RDD by using makeRDD() function of Spark. Then, the RDD is converted to an array structure which is suitable for GPU processing by using collect() function of Spark. We call this array structure Array Cache. To perform Action and Transformation functions using GPU device, the Array Cache is transferred to the GPU device memory and a CUDA kernel is invoked. To reduce the conversion overhead, once Array Caches are created, they reside in a host memory so that they can be reused for the subsequent processing. In addition, once Array Caches are transferred to GPU device memory, they reside in a GPU device memory until they are replaced with newly-transferred Array Caches. A CUDA kernel corresponding to a given Action or Transformation function is performed for the Array Cache transferred in the GPU device memory and

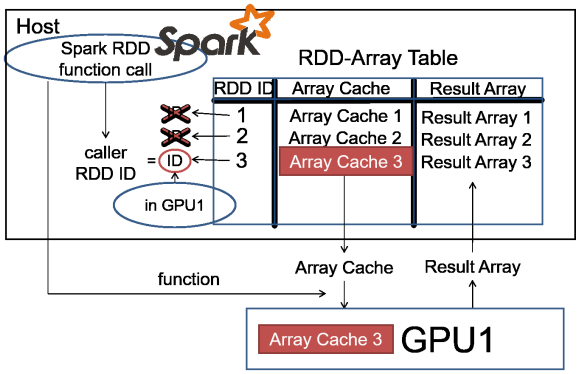


Fig. 2. Invoking GPU kernel from Spark framework

the computation result is returned to the host. Array Cache implementation is mentioned in Section V.

B. Array Cache of RDD

Figure 2 illustrates how RDDs are cached as Array Caches and how a GPU kernel is invoked when Action or Transformation function is called. As shown, the used Array Caches reside in GPU device memory and can be reused for the subsequent processing. Here we introduce a table that manages a relationship between RDDs and Array Caches. We call this table RDD-Array Table. We implemented RDD-Array Table in Spark framework. When an Array Cache is created from an RDD, their IDs are stored in RDD-Array Table. When Transformation functions are performed, the result (i.e., new RDD) is recorded as a Result Array in the table. We implemented read/write functions of RDD-Array Table in Spark framework. When an Action or Transformation function that utilizes GPU device is called, RDD-Array Table is checked to see whether the target RDD has been transformed as Array Cache. If the Array Cache does not exist, the target RDD is then transformed into Array Cache and registered in RDD-Array Table. If it exists, the table is further checked to see whether the Array Cache resides in GPU device memory. If it does not reside in the GPU device, it is then transferred to a GPU device.

At the GPU device side, a CUDA kernel corresponding to a given Action or Transformation function is executed. We employ jcuda [15] in order to invoke CUDA kernels from Spark framework implemented in Scala and Java languages.

C. GPU Processing for Action

As Action functions, here we illustrate GPU processing for Sum, LineCount, and Max (Min) programs as examples.

To calculate the sum of elements in a given RDD, reduce() function of Spark framework is used. In the original Spark framework, the sum of the elements is calculated as follows.

1) An RDD is created.
2) reduce() function is performed for the RDD to calculate the sum of elements in the RDD.

We implemented the corresponding CUDA kernel based on [16]. In our Spark framework that supports GPU processing, the procedure is changed as follows.

1) An RDD is created.
2) An Array Cache is created from the RDD and their IDs are registered in RDD-Array Table.
3) The Array Cache is transferred to a GPU device memory and a CUDA kernel corresponding to reduce() function is performed for the Array Cache.

To count the number of lines in a given RDD (i.e., LineCount), count() function of Spark framework is used. To find the maximum (minimum) value in a given RDD, max() (min()) is used. We implemented their corresponding CUDA kernels. Their procedures are similar to that of the Sum mentioned above.

D. GPU Processing for Transformation

Transformation functions can be accelerated by GPU devices as well as Action functions. A major difference between Action and Transformation functions is that the computation result of Transformation is a new RDD. Thus, the computation result of GPU devices is transformed into an RDD by using makeRDD() function of Spark. Here we illustrate GPU processing for SortByKey, PatternMatch, and WordConversion programs as examples.

To sort key-value pairs by their keys in a given RDD, sortByKey() function of Spark is used. In the original Spark framework, key-value pairs are sorted by their keys as follows.

1) An RDD is created.
2) sortByKey() function is performed for the RDD to sort key-value pairs by their keys in the RDD.
3) The computation result is stored as a new RDD.

We implemented a CUDA kernel of Bitonic sort algorithm. The procedure is changed as follows.

1) An RDD is created.
2) An Array Cache is created from the RDD and their IDs are registered in RDD-Array Table.
3) The Array Cache is transferred to a GPU device memory and a CUDA kernel corresponding to sortByKey() function is performed for the Array Cache.
4) The computation result of GPU is transformed into an RDD by using makeRDD().

To extract the lines which are matched to a given pattern from a given text (i.e., PatternMatch), filter() function of Spark framework is used. To convert an input text to a specific output based on a given rule (i.e., WordConversion), map() function of Spark framework is used. We implemented their corresponding CUDA kernels.

E. GPU Processing for Complex Processing

We can combine Action and Transformation functions to perform more complex processing on a given RDD. Such compound processing can be accelerated by GPU devices. Here we illustrate GPU processing for calculating Variation of elements in a given RDD as an example.

To calculate a variance of elements in a given RDD, both reduce() and map() functions are used. In the original Spark framework, a variance is calculated as follows.

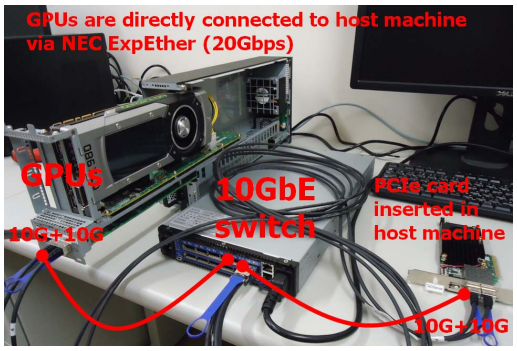


Fig. 3. Remote GPU devices connected via 10GbE

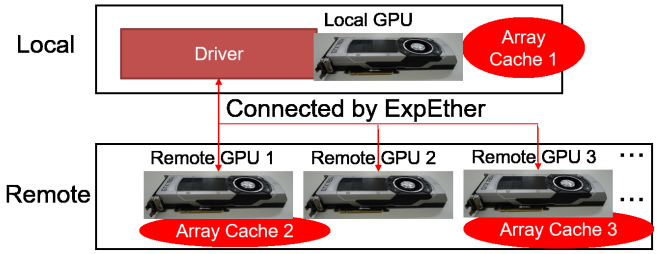


Fig. 4. Spark with local and remote GPU devices

1) An RDD is created.
2) reduce() function is performed for the RDD to calculate an average value of elements in the RDD.
3) map() function is performed for the same RDD to calculate a square of the difference between each element value and the average value.
4) The computation result is stored as a new RDD.
5) reduce() function is performed for the new RDD to calculate an average value of elements in the new RDD.

With GPU processing, the procedure is changed as follows.

1) An RDD is created.
2) An Array Cache is created from the RDD and their IDs are registered in RDD-Array Table.
3) The Array Cache is transferred to a GPU device memory and a CUDA kernel corresponding to reduce() function is performed for the Array Cache.
4) A CUDA kernel corresponding to map() function is performed for the Array Cache.
5) A CUDA kernel corresponding to reduce() function is performed for the Array Cache.

An input RDD is transferred to a GPU device only once (Step 3). Then the three CUDA kernels are performed for the data stored in the GPU device memory (Steps 3, 4, and 5).

IV. Local and Remote GPU Devices

A. System Overview

In this section, we extend our Spark framework to use multiple GPU devices connected to a host via PCIe (i.e., local GPU devices) and PCIe over 10GbE (i.e., remote GPU devices as shown in Figure 3). Figure 4 illustrates the framework with local and remote GPU devices. Since the number of GPU devices mounted in a host machine is limited by the available PCIe slots, the number and size of RDDs which can be cached in GPU device memories are also limited. If the GPU device memory capacity is not enough, Array Cache of RDDs may be frequently swapped out and in from GPU devices. Here we propose to combine a few local GPU devices and a number of remote GPU devices connected via PCIe over 10GbE so that we can expand the GPU device memory capacity for Array Cache.

There are interesting and important design options regarding where RDDs should be cached, local or remote GPU devices. Actually, while the number of remote GPU devices can be expanded, the remote GPU devices may suffer communication overhead since they are connected to a host via 10GbE. While the number of local GPU devices is limited, they do not suffer a 10GbE communication overhead. Thus an efficient use of the local and remote GPU devices is a key for accelerating the Spark framework.

B. Caching Policies for Local and Remote GPUs

Caching policies of RDDs that take into account the characteristics of local and remote GPU devices affect the performance. For example, we may improve the performance when the number of CUDA kernel executions on local GPU devices is maximized. Alternatively, we may improve the performance when the amount of data transfer between a host and remote GPU devices is minimized. In this paper, we explore the following three RDD caching policies.

1) Policy-1 that caches Array Caches on local/remote GPU devices so that the most-used RDD is cached in a local GPU device.
2) Policy-2 that caches Array Caches on local/remote GPU devices so that the number of data transfers between a host and remote GPU devices is minimized.
3) Policy-3 that caches Array Caches on local/remote GPU devices so that the total amount of data transfers between a host and remote GPU devices is minimized.

Policy-1 tries to reduce the CUDA kernel execution overheads for remote GPU devices. Policy-2 and Policy-3 try to reduce the data transfers between a host and remote GPU devices, while Policy-2 focuses on the number and Policy-3 focuses on the amount. The data transfers can be classified into 1) input data transfers from a host to remote GPU devices before CUDA kernel execution and 2) output data transfers from remote GPU devices to a host after CUDA kernel execution. The output data transfer size depends on the type of function executed. Action functions return only a scalar or vector value, while Transformation functions return a new RDD. If CUDA kernels are performed on the same input data, the input data transfers can be eliminated except for the input data transfer for the first execution. Otherwise, the input data transfers are needed when Array Cache of RDD is swapped out/in from/to a GPU device memory.

The three RDD caching policies are performed based on a static analysis of a given application source code running on Spark framework. More specifically, an RDD graph where a node is an RDD and an edge is a relationship between two

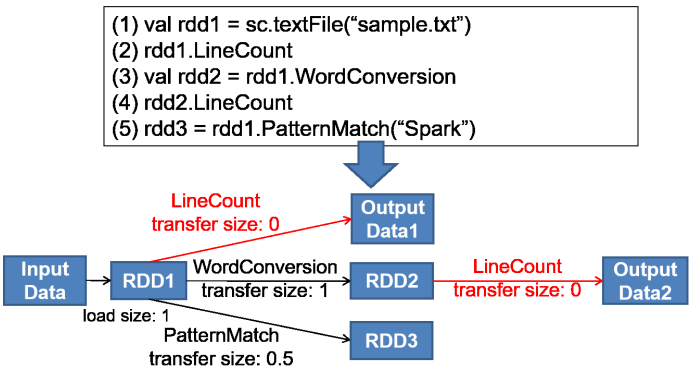


Fig. 5. Creation of RDD graph

RDDs (in the case of Transformation) or RDD and a value (in the case of Action) is extracted from the Spark application source code by using our code analysis program. Based on the RDD graph, we can estimate the number of accesses for each RDD in a given application by simply counting the number of edges on each node.

Each node (i.e., RDD) has a size property in our RDD graph. Our static analysis program simply estimates a result size of each edge (i.e., Transformation or Action function) as 1) scalar-size, 2) smaller-size, 3) equal-size, or 4) larger-size. Actually the result size estimation highly depends on a given function and input data. For example, Action functions return a value, which is very small compared to an input RDD. As Transformation functions, sortByKey() returns a same-sized RDD compared to an input RDD, while filter() returns a smaller-sized RDD compared to an input RDD. In the case of filter(), the application programmers can feed an expected match rate as a hint for estimating the result size of each edge for better accuracy, while in this paper we simply estimate the result size as one of the above four patterns.

Figure 5 illustrates an example of RDD graph extracted by our static analysis program. As shown, the number of accesses to RDD1 is three and that to RDD2 is one. When an input data set is loaded, the input data size is marked as "1." As a result size of Action functions, "0" indicates that the result is a scalar value. As result sizes of Transformation functions, "0.5", "1", and "1.5" indicate that the results are smaller than, equal to, and larger than an input RDD, respectively. For example, since LineCount program is based on an Action function, its result size is marked as "0." WordConversion program uses a Transformation function that performs a word conversion on a given text. Since the result size can be expected to be almost the same as an input text, its result size is marked as "1." PatternMatch program uses a Transformation function that extracts words from a given text based on a given rule, and the result size is assumed to be small but exact size is not known; thus its result size is marked as "0.5." Since the input data and the results of both the WordConversion and Filter programs performed on RDD1 should be saved for future use, the cumulative size of RDD1 is "2.5."

To simplify code analysis, the i-th RDD used in evaluations is denoted as "RDD(i)." Here all the GPU programs and their result sizes are detected beforehand. Since lines including "RDD(i).(program name)" in source codes are counted by our code analysis program, the number of RDD accesses and the sum of the result sizes are analyzed.

Based on the number of accesses and input/output sizes for each RDD in the RDD graph, our proposed RDD caching policies are performed to determine where the Array Cache is stored (i.e., local or remote GPU devices). For example, Policy-1 places RDD1 to a local GPU device as RDD1 is the most frequently-used one. We will evaluate the three RDD caching policies in terms of application execution times in Section VI.

V. Array Cache Implementation

To process an RDD by using GPUs, the RDD is converted to an Array Cache by using collect() function of Spark. Array Cache with array size, the original RDD ID, and the original data type are cached in a host main memory as a Java object. We prepare an array for these Java objects in a host main memory. When a Java object is created, it is recorded in the array. We call the set of these records RDD-Array Table. Since each record has an RDD ID and an Array Cache, we can easily find the target Array Cache. The registration and search times of RDD-Array Table are negligible compared to computation and transfer times.

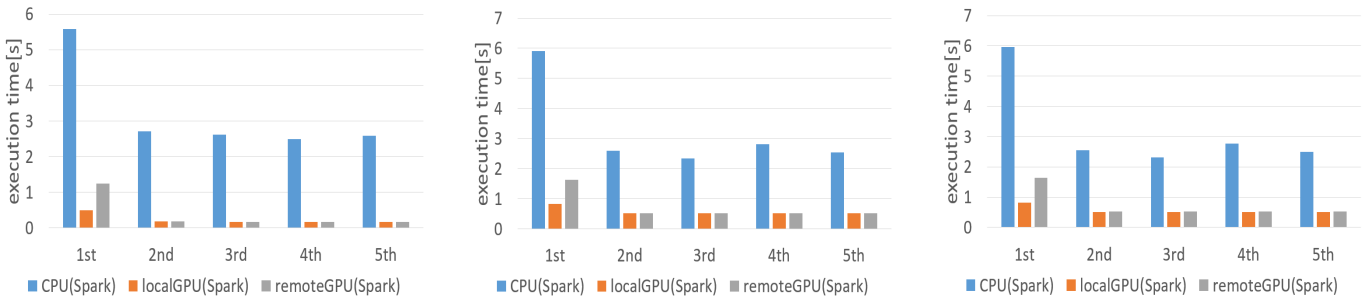
When a Spark function that utilizes GPUs is called, Array Cache corresponding to the target RDD is selected from RDD-Array Table and its size is compared to that of the GPU device memory. If the Array Cache size is smaller, it is transferred to the GPU device memory by using jcuda. Then, it is partitioned into blocks, each of which is stored in the shared memory of each thread block. They are processed by a CUDA kernel function corresponding to the Spark function called. Intermediate and/or final results are stored in the other array so that the original array can be reused for future computations. Finally, the result array is returned to the host and all the intermediate data are deleted from the device memory. The result array can be reused if needed.

If the target Array Cache size is larger than the GPU device memory, it is divided into smaller sub-arrays, each of which is fit to the GPU device memory. Assuming, for example, the target Array Cache size is 10GB and GPU device memory size is 3GB, the Array Cache is divided into three 3GB sub-arrays and a single 1GB sub-array. These sub-arrays are processed by a single GPU device sequentially or, if multiple local and/or remote GPU devices are available, multiple sub-arrays are transferred to the available GPU devices and processed by them in parallel. Thus RDDs larger than GPUs' device memory can be efficiently processed by utilizing multiple local and remote GPU devices. The host machine then gathers the partial results from these GPU devices and merges them to build the final result which will be stored as a new RDD.

VI. Evaluations

A. Performance of Action

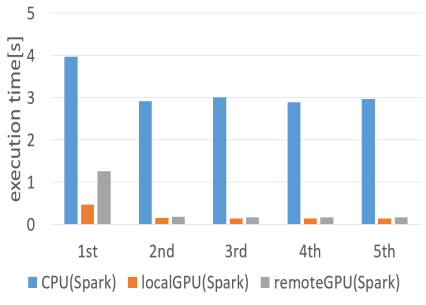
Here, we compare Spark framework with CPU, local GPU, and remote GPU in terms of execution times of Action functions. Table I shows the evaluation environment. Java heap size was set to 200GB. In the remote GPU case, each GTX 980 Ti GPU is connected to the host via two SFP+ direct attached cables with NEC ExpEther10G.



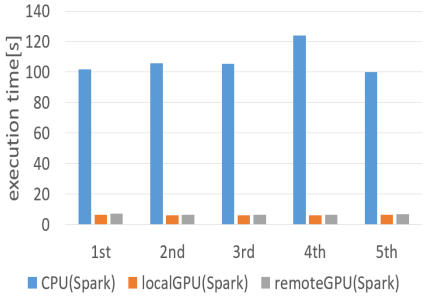
(a) Sum (overhead: 15.6s)

(b) Max (overhead: 15.4s)

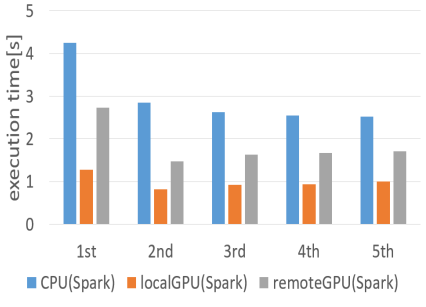
(c) Min (overhead: 15.6s)



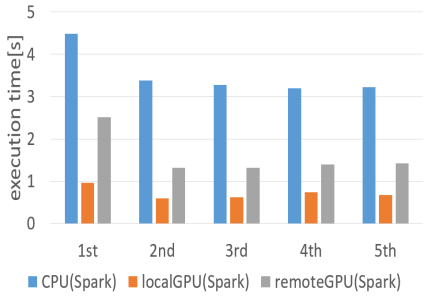
(d) LineCount (overhead: 28.3s)



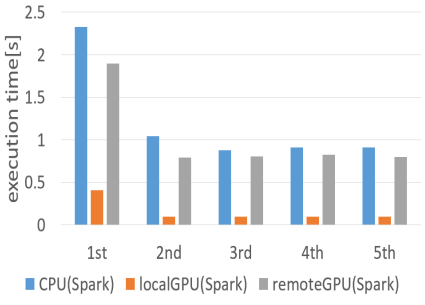
(e) SortByKey (overhead: 206.2s)



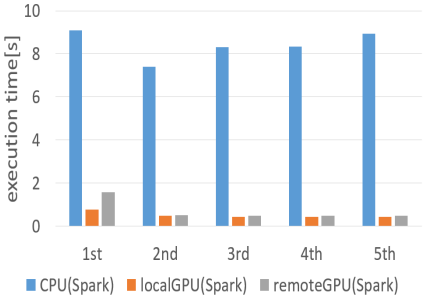
(f) PatternMatch 10% (overhead: 28.4s)



(g) PatternMatch 1% (overhead: 28.2s)



(h) WordConversion (overhead: 28.6s)



(i) Variance (overhead: 9.9s)

Fig. 6. Execution times of programs with CPU, local GPU, and remote GPU

TABLE I. EVALUATION ENVIRONMENT

CPU	Intel Xeon E5-2637 v3
Number of CPU cores	4
Operating frequency	3.50GHz
Memory capacity	256GB
GPU	GeForce GTX 980 Ti
OS	CentOS 6.7
CUDA version	7.5
jcuda version	0.7.5
Spark version	1.6.0 (Stand alone mode)
Scala version	2.10

We implemented Sum, Max, Min, and LineCount programs for the performance evaluations of Action functions. Five continuous executions of a program (i.e., 1st, 2nd, ..., 5th executions) are defined as a continuous execution set, and we performed 50 continuous execution sets for each program. Average execution times over the 50 sets are calculated. For the 1st execution, the execution time includes a Spark start-up time and an RDD creation time. An Array Cache creation time is also needed for the GPU executions. Currently, Array Cache creation is not optimized and it takes a relatively long time (we will discuss this issue in Section VII). In Figure 6, a Spark start-up time, an RDD creation time, and Array Cache creation time are not included in the bars because they are required only when the target RDD is newly created. Among these three overheads, Array Cache creation time is only shown as "overhead" in a caption of each graph.

Figure 6(a) shows the execution times of Sum program by CPU, local GPU, and remote GPU. A 1GB 32-bit signed integer array with random values is used as input data. As the start-up overheads, it takes 1.9 seconds, 8.4 seconds, and 15.6 seconds for Spark start-up, RDD creation, and Array Cache creation, respectively. Please note that a GPU execution time is a sum of a GPU processing time and a data transfer time between host and GPU device. It thus takes a longer time for the 1st execution since the Array Cache is transferred to the GPU device memory, while the execution times are much reduced for the subsequent executions since the transferred Array Cache can be reused. As shown in the graph, local GPU outperforms CPU by 11.4x for the 1st execution. It also outperforms CPU by 14.4-15.7x for the subsequent executions. Remote GPU is comparable to local GPU for the subsequent executions, while its performance gain is reduced for the 1st execution due to the data transfer overhead.

Figures 6(b) and 6(c) show the execution times of Max

and Min programs by CPU, local GPU, and remote GPU, respectively. The 1GB integer array with random values is used as input. Figure 6(d) shows the execution times of LineCount program. A 1GB randomly-generated text file is used as input. Their results are similar to Sum program (Figure 6(a)). Local GPU outperforms CPU by up to 21.4x in LineCount program.

B. Performance of Transformation

We implemented SortByKey, PatternMatch, and WordConversion programs for the performance evaluations of Transformation functions.

Figure 6(e) shows the execution times of SortByKey program by CPU, local GPU, and remote GPU. A 1GB key-value pairs array (512MB for keys and 512MB for values in total) is used as input. Please note that, for the GPU executions, only the key part (i.e., 512MB) is transferred to the GPU device memory for SortByKey program. An Array Cache creation time is quite long, because extraction of keys from a given RDD is performed in a naive manner with Scala language, which should be further optimized.

For the GPU executions, as explained, the Array Cache is transferred to GPU device memory for the 1st execution, while it can be reused and the data transfer overheads can be eliminated for the subsequent executions. In Figure 6(e), however, there is almost no difference in the results of the 1st and subsequent executions in the local and remote GPU cases. This is because, since the GPU processing time of SortByKey program is large, the data transfer time becomes relatively small. As a result, local GPU outperforms CPU by 16.0x for the 1st execution (if we do not consider the Array Cache creation time) and by 15.7-20.0x for the subsequent executions. Also, remote GPU outperforms CPU by 14.2x for the 1st execution and by 14.6-18.9x for the subsequent executions.

Figures 6(f) and 6(g) show the execution times of PatternMatch program with different hit rates (i.e., 10% and 1%). The 1GB text file in which 10% or 1% of words are matched to a given pattern is used as input. Figure 6(h) shows the execution times of WordConversion program. The 1GB randomly-generated text file is used as input. Due to the data transfer overhead is relatively large compared to computation time, performance gain of remote GPU over CPU is limited.

C. Performance of Complex Processing

We implemented Variance program for the performance evaluation of a compound processing.

Figure 6(i) shows the execution times of Variance program. The 1GB integer array with random values is used as input. The Array Cache creation time is 9.9 seconds. Local GPU outperforms CPU by 11.5x for the 1st execution and by 15.2-19.9x for the subsequent executions. Remote GPU outperforms CPU by 5.8x for the 1st execution and by 14.3-18.5x for the subsequent executions.

D. RDD Cache Policies

In the previous subsections, we showed the measured execution times of various programs by CPU, local GPU, and remote GPU. Using these measured execution times as parameters, we perform simulations of RDD caching policies proposed in Section IV-B. Here we assume three applications that use the Action and Transformation programs we implemented. Since the first and second applications are simple, we assume that one local and one remote GPU devices are available. However, the third application is large and thus we assume that one local and seven remote GPU devices are available. We also assume that only a single Array Cache, which has been converted from a given RDD, can be cached at a time in each GPU device. If RDD j depends on RDD i, their two computations are performed sequentially, while if these two RDD do not have any dependencies, they are performed in parallel.

In the first application, 1GB integer data, 1GB text data, and 1GB text data are loaded as RDD1, RDD2, and RDD3, respectively. Then the following steps are performed.

1) RDD1.Sum
2) RDD2.LineCount
3) RDD1.Variance
4) RDD4 = RDD3.WordConversion
5) RDD1.Max
6) RDD5 = RDD2.WordConversion
7) RDD1.Min

In this application, the same RDD (i.e., RDD1) is used repeatedly. The most used one is RDD1.

In the second application, 1GB key-value pair data, where key is a random integer value and value is a random string value, are loaded as RDD1. The following steps are performed.

1) RDD2 = RDD1.SortByKey
2) RDD3 = RDD2.values.WordConversion
3) RDD2.keys.Max
4) RDD2.keys.Min
5) RDD3.LineCount
6) RDD3.Min

In this application, RDDs are created in a sequential manner along a linear RDD graph. The most used one is RDD2.

To evaluate the RDD caching policies with a larger application, randomly-selected 50 functions are performed for randomly-selected RDDs, as the third application. The number of source RDDs is set to three. The randomly-generated RDDs have 1-13 edges as a result. Size of each RDD is set to 1GB. Tables II, III, and IV show the simulation results of the three applications.

In the first application, we assume a situation that there is no dependency between RDDs. RDDs without dependency can be executed in parallel by using remote GPU devices, and the performance can be improved. When Policy-1 is adapted, all operations on RDD1 can be performed in local GPU device without swapping. However, since RDD2 and RDD3 are processed in remote GPU devices, the total data transfer amount between a host and remote GPU devices becomes large. As the execution time by remote GPU devices is 2x longer than that of local GPU device, the total performance gets worse. When Policy-2 is adapted, the number of data transfers between a host and remote GPU devices is only two. As a result, the data transfer overhead is reduced compared to Policy-1 and the performance gets better. When Policy-3 is

TABLE II. SIMULATION RESULTS OF THE FIRST APPLICATION

RDD caching policy	RDD1	RDD2	RDD3	Execution time
Policy-1	Local	Remote	Remote	4.987 (sec)
Policy-2	Local	Local	Remote	3.504 (sec)
Policy-3	Remote	Local	Local	2.777 (sec)
Local GPU only	Local	Local	Local	4.223 (sec)

TABLE III. SIMULATION RESULTS OF THE SECOND APPLICATION

RDD caching policy	RDD1	RDD2	RDD3	Execution time
Policy-1	Remote	Local	Remote	8.258 (sec)
Policy-2	Remote	Local	Local	8.387 (sec)
Policy-3	Local	Local	Remote	7.450 (sec)
Local GPU only	Local	Local	Local	7.579 (sec)

TABLE IV. SIMULATION RESULTS OF THE THIRD APPLICATION

RDD caching policy	Execution time
Policy-1	20.278 (sec)
Policy-2	19.811 (sec)
Policy-3	18.576 (sec)
Local GPU only	46.479 (sec)

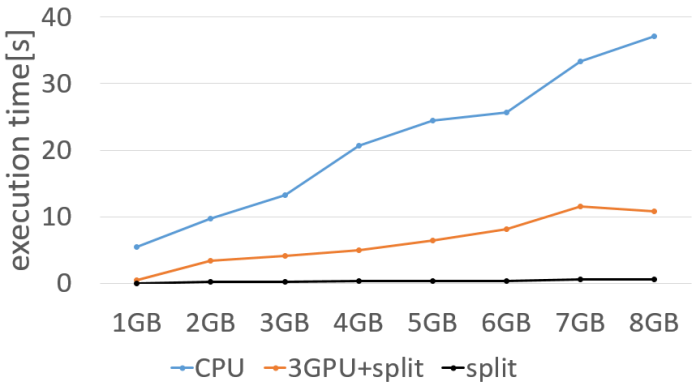


Fig. 7. Execution times of Sum program on different RDD sizes

adapted, RDD2 and RDD3 are processed in local GPU device and the total data transfer amount is minimized. As a result, Policy-3 achieves the best performance.

In the second application, we assume a situation that there are dependencies between RDDs. The dependencies introduce waiting time during parallel processing and benefit of adding more remote GPU devices is reduced. As a result, "Local GPU only" that uses only local GPU device achieves a good performance. Nevertheless, Policy-3 achieves the best performance, because RDD2 and RDD3 can be processed in parallel by local and remote GPU after RDD3 has been created.

In a large application such as the third application, Array Cache and parallel processing by using remote GPU devices lead to a performance improvement. In this situation, Policy-3 achieves the best performance. Based on the above results, for parallel executable applications, adding more remote GPU devices can reduce the number of swap out of RDDs from GPU device memory and the data transfer overhead can be reduced.

E. Performance of Large RDD

As mentioned in Section V, an RDD larger than a GPU device memory is divided into sub-arrays and processed by one or more GPU devices in parallel. Assuming a GPU device memory size is 1GB for simplicity, here we evaluate the execution times of Sum program on an RDD (32-bit randomly-generated integer array) with different sizes from 1GB to 8GB. We use three GPU devices (i.e., one local and two remote GPU devices, denoted as "3GPU") for this evaluation; thus, up to three 1GB sub-arrays are processed by one local and two remote GPU devices in parallel. Such a parallel execution is repeated until all the sub-arrays have been completed.

Figure 7 shows the 1st execution times of CPU, 3GPU+split, and split. Here "split" denotes the overhead time to divide a given RDD into 1GB sub-arrays. We performed 10 continuous execution sets, and an average of the 1st execution times is shown in the graph. As shown, the execution time of 3GPU+split increases as the data size increases. The split time is negligible compared to the computation time. Due to the performance asymmetricity of the three GPU devices, the execution times are slightly fluctuated especially when a remainder sub-array is processed by a remote GPU device.

VII. SUMMARY AND FUTURE WORK

To accelerate various computation intensive operations of Spark, we modified Spark framework to invoke CUDA kernels when computation intensive operations are called. We tried to cache RDDs in GPU device memory as much as possible. In addition, we proposed to use remote GPU devices which are connected to a host machine via a PCIe over 10GbE to expand the device memory capacity and use GPU devices in parallel. We proposed three RDD caching policies for local and remote GPU devices. We implemented Sum, Max, Min, LineCount, SortByKey, PatternMatch, WordConversion, and Variance programs on Spark framework with local and remote GPU devices. We evaluated the program execution times of local and remote GPU devices assuming the same operation is performed iteratively. The evaluation results showed that our modified Spark that utilizes local GPU device outperforms the original Spark by up to 21.4x. Performance degradation of remote GPU device compared to the local GPU device is not significant except for the first execution during the iteration or executions with large results. Due to the data transfer overhead for the remote GPU devices, evaluation results of the three RDD caching policies showed that the RDD caching policy that tries to minimize the data transfer amount to remote GPU devices outperforms the other caching policies. Although we used a single host machine with local and remote GPU devices, we can extend it to multiple host machines that maintain more local and remote GPU devices.

Although the aim of this paper is explore how to utilize local and remote GPU devices from Spark framework, the Array Cache creation time should be reduced as a future work. We are currently using a built-in collect() function to build Array Cache from a given RDD for ease of implementation. To reduce the overhead, we are planning to implement our dedicated RDD-to-Array conversion function in our framework. Although we implemented the relatively simple applications for the demonstration purposes in this paper, we are currently planning to implement sophisticated applications and demonstrate the benefits of our approach as our future work.

Acknowledgements This work was supported by SECOM Science and Technology Foundation and JST PRESTO.

REFERENCES

[1] "Apache Hadoop," http://hadoop.apache.org/.

[2] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *Proceedings of the USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*, Jun. 2010, pp. 10–10.

[3] 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 USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*, Apr. 2012, pp. 15–28.

[4] P. Li, Y. Luo, N. Zhang, and Y. Cao, "HeteroSpark: A Heterogeneous CPU/GPU Spark Platform for Machine Learning Algorithms," in *Proceedings of the International Conference on Networking, Architecture and Storage (NAS'15)*, Aug. 2015, pp. 347–348.

[5] "cuSpark - a Functional Data Processing Framework," http://www.yaomuyang.com/cuspark/.

[6] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: A MapReduce Framework on Graphics Processors," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'08)*, Oct. 2008, pp. 260–269.

[7] F. Ji and X. Ma, "Using Shared Memory to Accelerate MapReduce on Graphics Processing Units," in *Proceedings of the International Symposium on Parallel and Distributed Processing (IPDPS'11)*, May 2011, pp. 805–816.

[8] M. Elteir, H. Lin, W. chun Feng, and T. Scogland, "StreamMR: An Optimized MapReduce Framework for AMD GPUs," in *Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS'11)*, Dec. 2011, pp. 364–371.

[9] S. Morishima and H. Matsutani, "Performance Evaluations of Document-Oriented Databases using GPU and Cache Structure," in *Proceedings of the International Symposium on Parallel and Distributed Processing with Applications (ISPA'15)*, Aug. 2015, pp. 108–115.

[10] ——, "Performance Evaluations of Graph Database using CUDA and OpenMP-Compatible Libraries," *ACM SIGARCH Computer Architecture News (CAN)*, vol. 42, no. 4, pp. 75–80, Sep. 2014.

[11] J. Suzuki, Y. Hidaka, J. Higuchi, T. Yoshikawa, and A. Iwata, "ExpressEther - Ethernet-Based Virtualization Technology for Reconfigurable Hardware Platform," in *Proceedings of the IEEE Annual Symposium on High-Performance Interconnects (HOTI'06)*, Aug. 2006, pp. 45–51.

[12] J. Suzuki, Y. Hayashi, M. Kan, S. Miyakawa, and T. Yoshikawa, "End-to-End Adaptive Packet Aggregation for High-Throughput I/O Bus Network Using Ethernet," in *Proceedings of the IEEE Annual Symposium on High-Performance Interconnects (HOTI'14)*, Aug. 2014, pp. 17–24.

[13] J. Duato, A. Pena, F. Silla, R. Mayo, and E. Quintana-Orti, "rCUDA: Reducing the Number of GPU-based Accelerators in High Performance Clusters," in *Proceedings of the International Conference on High Performance Computing and Simulation (HPCS'10)*, Jun. 2010, pp. 224–231.

[14] S. Morishima and H. Matsutani, "Distributed In-GPU Data Cache for Document-Oriented Data Store via PCIe over 10Gbit Ethernet," in *Proceedings of the International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar'16)*, Aug. 2016.

[15] "jcuda.org," http://www.jcuda.org/.

[16] "Optimizing Parallel Reduction in CUDA," https://docs.nvidia.com/cuda/samples/6_Advanced/reduction/doc/reduction.pdf.