A Real-Time Big Data Analysis Framework on a CPU/GPU Heterogeneous Cluster

A Meteorological Application Case Study

Mohamed Hassaan
Computer and Systems Engineering
Alexandria University
Alexandria, Egypt
muhammad.aboelhassan@gmail.com

Iman Elghandour
Computer and Systems Engineering
Alexandria University
Alexandria, Egypt
ielghand@alexu.edu.eg

ABSTRACT

It is important to analyze and predict meteorological phenomena in real-time. Parallel programming by exploiting thousands of threads in GPUs can be efficiently used to speed up the execution of many applications. However, GPUs have limitations when used for processing big data, which can be better analyzed using distributed computing platforms such as Hadoop and Spark. In this paper, we propose DAMB a system that processes streamed data on a heterogeneous cluster of CPUs and GPUs in real-time. The core of DAMB is SparkGPU, a platform that extends Apache Spark to allow it to manage a heterogeneous cluster that has both CPUs and GPUs and to execute tasks on GPUs. DAMB also provides data visualization tools that present the analyzed data in an interactive way in real-time. As a case study, we focus on a meteorological application that analyzes lightening discharges. We show that DAMB can successfully process and analyze the meteorological data streamed to it and visualize the results in real-time on a cluster of size 12 nodes, each is equipped with one or more GPU cards. This is a speedup of two orders of magnitude as compared to a sequential program implementation for the same application.

CCS Concepts

•Computer systems organization → Heterogeneous (hybrid) systems; •Computing methodologies → *Distributed programming languages;*

Keywords

Heterogeneous clusters; GPU Programming; In-memory cluster computing

BDCAT'16, December 06-09, 2016, Shanghai, China
© 2016 ACM. ISBN 978-1-4503-4617-7/16/12...$15.00
DOI: http://dx.doi.org/10.1145/3006299.3006304

1. INTRODUCTION

The recent advancement in technology has enabled researchers to capture and hence monitor data about meteorological phenomena in real-time, for example, lightining [17], air quality [26], Tsunami [22], and precipitation [15, 23]. It is becoming very important to monitor, understand, and predict these various meteorological phenomena because it can help us reduce the damage they cause by taking additional precautions. For example, every year, lightning strikes kill people and wild animals, cause thousands of fires, and cause damage to electrical infrastructures such as power lines. The cost of these damages is estimated to be billions of dollars per year in USA [5].

A typical scenario for many meteorological applications is that devices such as interferometers [8, 11] and phased arrays [15, 23] are used to capture signals (e.g. lightning discharges) and digitize their values. Analysis and prediction techniques, which are compute intensive, are then performed on the collected data. In previous works, the main objective was the accuracy of the concluded results of the analsysis and/or prediction. However, it is also important to reduce the lagging time between capturing the data and computing the results of the analysis and/or prediction tasks [15] so the concluded results will not lose their value. This latter objective is also referred at as achieving real-time or near-real time analysis of the captured data. One common challenge that needs to be addressed by many of these applications is that they generate very large data per unit time interval, which is usually one or few seconds. Therefore, we need to use a system that can process large data streamed to the system in real-time.

FPGAs have long been used to efficiently analyze data in meteorological applications [6, 20]. However, FPGAs do not cope with the very large data sizes, and real-time analysis will not be guaranteed. Parallel programming by exploiting the thousands of threads of a GPU has been efficiently used in many applications [9, 14, 21]. However, it is not the best solution for processing big data, which are better analyzed using distributed computing platforms such as MapReduce [7] (open source implementation Hadoop available at [2]) and Spark [24] (open source available at [4]). Both of the parallel programming frameworks and the the distributed computing platforms have their pros and cons. Parallel programming frameworks that run on GPUs achieve very high parallelism. However, they are not suitable for executing large data sets because of the limited memory size of the on-chip memory of the GPUs. Additionally, the data transfer between the host and the GPU and vice-versa introduces high overhead, which is typically larger than the on-chip data transfer. On the contrary, the current available implementations of distributed computing platforms such as Hadoop and Spark have been efficiently used for processing very large data sets. However, the parallelism in Hadoop and Spark is limited by the number of CPU cores available at each node in the cluster.

A solution that integrates both parallel programming and distributed computing is expected to encapsulate the pros of both approaches and avoids their cons. However, new challenges are introduced. Platforms such as Hadoop and Spark are designed to manage a cluster of CPUs that have similar specification. Spark initiates tasks on worker nodes to execute an application on partitions of the data. If these tasks are executed on GPUs rather than CPUs, how can their code and data be transferred to the GPUs for execution. Therefore, a new communication mechanism between the distributed computing platform, namely Spark and a GPU is required to delegate code and data to be executed on the GPU.

In this paper, we present DAMB, an end-to-end solution that allows data to be streamed to it and process them on a heterogeneous cluster of CPUs and GPUs. The outputs of the analysis tasks are also visualized for better understanding of these results. The core component of DAMB that executes its analysis tasks in real-time, SparkGPU, is an extension of Spark that is capable of managing a cluster of CPUs and GPUs and delegating tasks to be executed on the GPU. Therefore, each time a new micro-batch of data is streamed to DAMB, new tasks are created and are executed on GPUs in the cluster.

We use as a case study an application that detects lightning discharges [17]. This application generates 1.6 GB of data per second. This data is logged into micro-batches and streamed to DAMB for analysis in real-time. Each micro-batch constitute the data generated per second. Upon the arrival of new streamed data, tasks are created and are executed on GPUs in the cluster. We note that a sequential implementation takes around 12 minutes to process the data generated per second.

The main contributions of this paper are as follows:

- An end-to-end solution that takes data streamed from weather measuring devices such as interferometers and phased array radars, analyze these data, and finally visualize the results.
- An extended version of Spark that is capable of managing a heterogeneous cluster of CPUs and GPUs and distribute tasks among them.
- Two approaches that allow applications written for Spark to submit tasks to be executed on a GPU.
- A real-time solution for the lightning application presented in [17].
- An experimental study that evaluates our proposed framework and approaches.

The rest of the paper is organized as follows. In section 2 we present an overview of the DAMB architecture. Next, we describe SparkGPU in Section 3. We then present the details of the lightning application that we use as our case study and how we use DAMB for executing it in Section 4. We present our performance evaluation study in Section 5. Finally, we summarize the related work in Section 6 and conclude in Section 7.

2. OVERVIEW OF DAMB

We used the following components to build DAMB and to allow it to process streamed data in real-time: Apache Kafka [3], spark streaming [4, 25], Spark [4, 24], Tachyon [12], and D3.js library. Figure 1 shows the main components of our solution and the flow of the data between them. The data are generated by receiver devices such as antennas that are located in an observation site. These data are then streamed to an on-site server that is equipped with a digitization card. The role of this card is to convert the collected data into digital form (bytes) and stream them to our DAMB cluster. We assume that data are streamed from the on-site server to the DAMB cluster every unit interval (e.g., a second) as one chunk.

The streamed data are stored in Apache Kafka [3], which allows us to partition the data and organize them into messages that are distributed across the cluster machines. Kafka is fully integrated with spark streaming [25], which employs a technique called discretized streams or micro-batches. Spark streaming reads Kafka messages each certain time interval as their input and trigger tasks to execute the running application on them. Using Kafka and Spark streaming enable us to partition the data streamed to DAMB into micro-batches and divide and distribute them among the nodes of the cluster to process them.

SparkGPU, which lies at the heart of DAMB, is responsible for executing applications on the streamed data. Every time spark streaming submits a chunk of data to a node of the cluster, a Spark task starts to process it. SparkGPU extends Spark by enabling it to execute its tasks on the GPUs available on the cluster nodes. The main extensions to Spark are: (1) a new interface that allows communication between spark tasks and GPUs; and (2) a new programming style of the Spark applications to specify the functions that are executed on GPUs. Therefore, the data assigned to each Spark task is executed on a GPU if one is available on the same node that this task is running. Finally, the execution output of each streamed micro-batch is persisted in the distributed storage (e.g. HDFS) for further referencing.

The final phase of DAMB is to plot the outputs of analyzing the data on charts. This helps scientists using the meteorological application to better understand the analysis results. Figure 2 shows one of the visualization charts that we plot for the lightning application that we discuss in more details in Section 4. These charts are the same ones used in the studies presented in [11, 17]. However, our objective is to display the results on the chart in real-time, and therefore, we use the sophisticated web development library D3.js. Moreover, to pipeline the output between the SparkGPU cluster and the visualization tool, we use Tachyon [12], which is now distributed as Alluxio [1]. Tachyon is a memory centric distributed storage system, and therefore it allows us to perform in-memory sharing of the analysis result between the SparkGPU cluster and the visulaization tool. Using Tachyon has enabled us to plot the analysis result while it is being computed using the SparkGPU tasks, leading to improved performance. Moreover, we have enabled the fol-

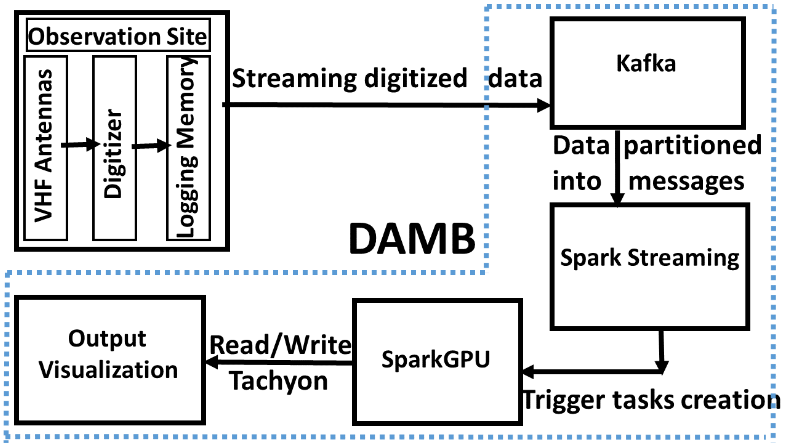


Figure 1: The flow of the data between the components of DAMB architecture.

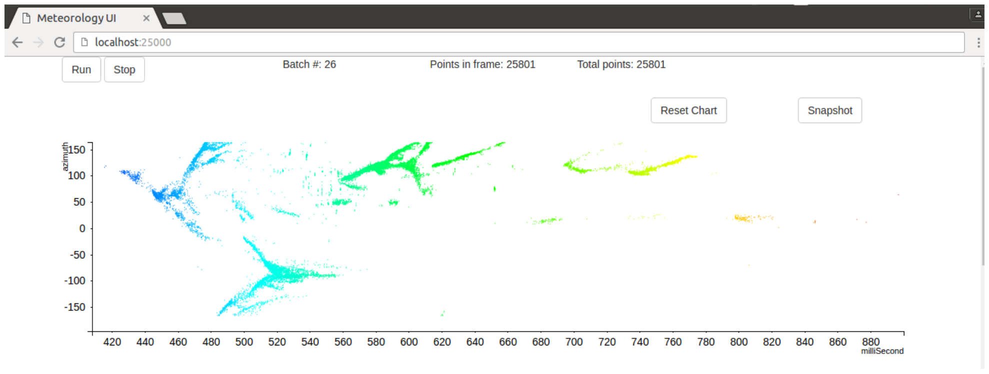


Figure 2: An example of the charts that we plot to represent the analyzed output.

lowing features in our visualized charts: zooming, panning, and snapshots. These features give the users more flexibility when they try to understand the results of the analysis [1].

3. SPARKGPU: A HETEROGENEOUS REAL-TIME BIG DATA ANALYSIS PLATFORM

At the heart of DAMB is the SparkGPU cluster. Every time SparkStreaming submits a micro-batch of new data, SparkGPU is triggered to schedule new tasks to process the streamed micro-batch of data on its worker nodes. SparkGPU extends Spark by adding functionality to it to manage a cluster of worker nodes that are equipped with GPUs in addition to the CPUs and to assign tasks to be executed on these GPUs.

Next, we give an overview of the SparkGPU architecture in Section 3.1, and we then present the two approaches that we propose to enable SparkGPU to assign tasks to the GPUs that are available on the nodes of the cluster that it manages.

3.1 Overview of the SparkGPU

Similar to Apache Spark, SparkGPU has one master node that runs a SparkContext and several worker nodes, each of them runs an Executor. Any application has to run within a SparkContext and when it is initialized, it is assigned resources from the cluster in the form of Executors running at worker nodes. Each executor is configured with the CPU cores and GPUs that are available as resources on the node that it runs on and that it can use to execute tasks.

The applications running on SparkGPU use the Spark programming interface. The Spark driver program is structured as follows:

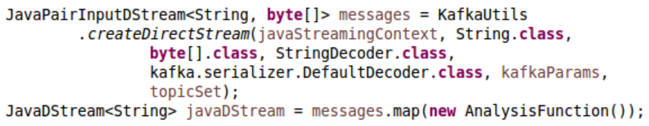
- We create a SparkContext object, which coordinates the spark tasks that run on the cluster.

- We setup `KafkaUtil`, which is a class provided by the Spark streaming library with the parameters required to allow Spark streaming to stream data from Kafka. These parameters include port numbers, Kafka topics, and number of partitions. Figure 3 shows an example of how we setup Kafka in the Spark driver code with the parameters that are named `kafkaParams`.

- Data are stored on the Kafka cluster in the form of partitions, which are a sequence of data records. We us the Spark streaming API to connect to Kafka and stream data from its partitions. We setup the cluster so the Kafka data partitions are locally read by the Spark tasks.

- Spark streaming discretizes the data partitions read from Kafka and creates RDDs from them.

[1]Demo of DAMB is available at: https://youtu.be/NKpVrGh67i8.



```
JavaPairInputDStream<String, byte[]> messages = KafkaUtils
        .createDirectStream(javaStreamingContext, String.class,
                byte[].class, StringDecoder.class,
                kafka.serializer.DefaultDecoder.class, kafkaParams,
                topicSet);
JavaDStream<String> javaDStream = messages.map(new AnalysisFunction());
```

Figure 3: A code fragment from the Spark driver showing the setup of the Kafka cluster and the calling of the `AnalysisFunction` that processes the RDDs on GPUs.

- We execute our application on the RDD as UDF (User Defined Function) in a map function. In Figure 3, the RDD name is `message` and the application that we run is implemented in the `AnalysisFunction`. For allowing the code to be executed on GPU, we have researched two approaches that we discuss next. The implementation of the `AnalysisFunction` differs according to the approach that we use.

3.2 Communication between Spark and GPUs via JNA

Spark applications are written in Java, python, or Scala, and the code running on the GPU are written in CUDA or openCL. Therefore, the Spark tasks requires an interface to communicate with the GPU and to delegate its work to it. One of the approaches that SparkGPU adopts to delegate the Spark task work to a GPU on the same cluster node uses the Java Native Access (JNA) library. We first compile the functions of the application that we would like to run on the GPU (written in CUDA) into a native shared library. Next, we write the Spark application to call functions from the shared native library using JNA. Therefore, we define a Java interface that defines the access to the shared native library as shown in Figure 4. Moreover, we implement `AnalysisFunction`, which will run on each worker node as follows:

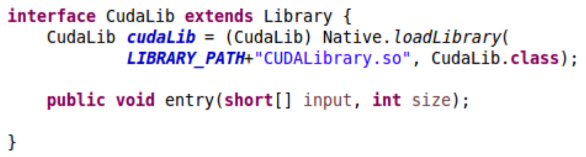
- We create a new configuration object that includes information about the size of data and the location of the CUDA shared library.

- We use the Java native access interface (Figure 4) to call the `entry` method defined in the interface. This `entry` method calls the suitable function from the JNA shared library that we want to execute for a specific application. Figure 5 shows an example calling for the `entry` method.

- The code compiled and stored as shared library defines the CUDA kernel code, copies the data from the memory of the node to the memory of the GPU, starts the kernel code on the GPU, and finally copies the result to the memory of the node.

- We store the final results on Alluxio.

Figure 6 shows the architecture of SparkGPU that uses JNA to interface with the GPUs available on the cluster nodes. When a spark task starts running, each time it invokes a function from the native shared library, a new process is created with a CUDA context. For multiple tasks running on the same node and trying to call functions from the native shared library to execute on the GPU available

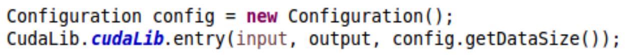


```
interface CudaLib extends Library {
    CudaLib cudaLib = (CudaLib) Native.loadLibrary(
            LIBRARY_PATH+"CUDALibrary.so", CudaLib.class);

    public void entry(short[] input, int size);

}
```

Figure 4: A code fragment showing how we use JNA as our interface to call CUDA native shared library.



```
Configuration config = new Configuration();
CudaLib.cudaLib.entry(input, output, config.getDataSize());
```

Figure 5: A code fragment showing how we call the native shared library.

on this node, each one of them will create a separate process with a separate CUDA context. Therefore, these processes will compete to access the GPU, only one of them will manage to access it at any time, and the other processes will have to block. Given that Spark is designed to assign small blocks to each task, the GPU is underutilized and an unnecessary overhead is introduced due to switching between CUDA contexts. To overcome these challenges we introduce another approach for establishing communication between Spark tasks and the GPU in the next section.

3.3 Communication between Spark and GPUs via a Node Manager

To avoid the overhead introduced by using JNA for the communication between SparkGPU and the GPUs on the same cluster node, we introduce a new approach that leverages the multiple kernels execution capability of GPUs. Modern GPUs are able to simultaneously run different operations from different threads in different execution streams. This can significantly improve the utilization of the GPU cores. To adopt this approach in SparkGPU, we need to guarantee that all the concurrent Spark tasks that delegate their work to a GPU are starting threads to execute on the GPU from one process rather than starting a separate process per task. This would guarantee that all these threads are running within the same CUDA context.

For each executor at a worker node, we create a new process called `node manager`. The node manager listens on a certain port waiting for messages coming from different Spark tasks. Each message consists of the chunk of data to be processed and the CUDA kernel function to execute. The node manager is now responsible for starting threads to concurrently run on the GPU. Each thread is assigned to run on one of the GPUs available on the nodes is running on. Besides, It defines the kernel function, copies the data from its memory to the GPU memory, starts running the CUDA kernel function on the GPU, and copies the result back to the memory of the node. This requires that a user of SparkGPU to write the CUDA kernel functions and add their declaration to a special header file that is linked by the node manager. Figure 7 shows the architecture of SparkGPU that uses a node manager to interface with the GPUs available on the cluster nodes.

Note that the `node manager` handles the assignment of received message on multiple GPUs if the cluster node that it runs on is equipped with more than one GPU. All the tasks

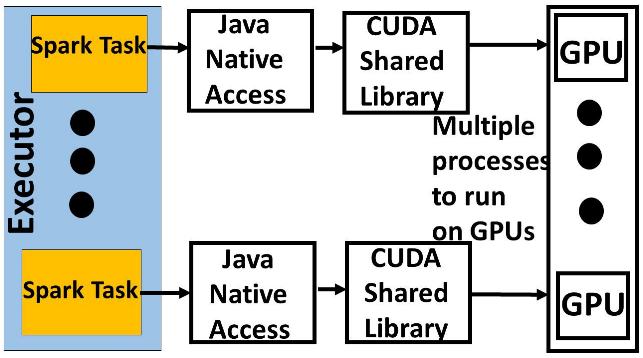


Figure 6: System architecture of SparkGPU that uses JNA to interface with GPUs in the cluster.

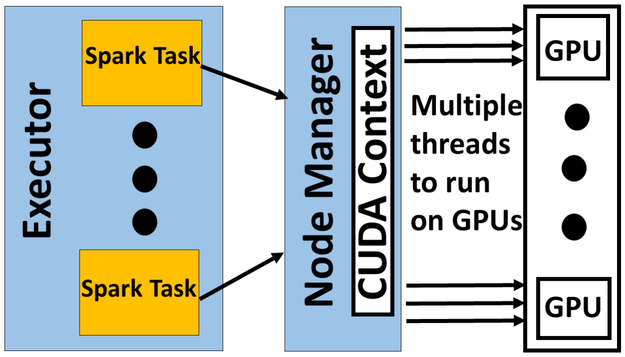


Figure 7: System architecture of SparkGPU that uses a node manager at each worker node.

are expected to take an equal execution time on the GPU. Therefore, for balancing the load across all the GPUs, we use round robin to assign the requests that the node manager receives on all the GPUs.

In this approach, we implement `AnalysisFunction` as follows:

- We create a new configuration object that includes information about the size of data and the port that the `node manager` listens to.
- We send the data to the `node manager`.

4. CASE STUDY: MONITORING AND ANALYZING LIGHTNING

In this section, we describe the lightning analysis application [17] that we use as our case study. We show that DAMB is capable of executing this application in real-time on gigabytes of data streamed per second to it. We first describe the application and its C implementation in Section 4.1. Next we discuss two simple parallel implementations in Section 4.2. Finally, we describe our solution using DAMB in Section 4.3.

4.1 Overview of Detecting Lightning Discharges

Lightning is associated with static electrical discharges. Several interferometric lightning mapping have been developed at Osaka University [11, 17, 19]. The technique employed by these systems is based on gathering broadband VHF (Very High Frequency) lightning discharges using multiple antennas that are carefully aligned with certain angles between each pair of them. Given the data read by each antenna and the setup information about the locations of these antennas, the phase differences between these antennas is calculated and then the azimuth and elevation are calculated. The calculated values indicate how the lightning is progressing through its occurrence time interval, which is usually less than a second. In our case study, we focus on the broadband digital VHF interferometers (DITF) proposed in [17].

In [17], to identify whether an electrical discharge captured by the antennas represent the occurence of lightning or not, two assumptions are made [17]: (1) discharges captured by one antenna within a short time (nanoseconds) are generated by the same lightning strike source; and (2) a lightning discharge that has many branches or is horizontally spread across a long distance is simultaneously captured by multiple antennas. Therefore, to confirm the occurrence of lightning, a lightning discharge has to be captured by multiple antennas within a very short time. The proposed approach continuously records signals from three antennas aligned on the apexes of obtuse angle triangle. Given that each antenna reading can be represented in 2 bytes, the total amount of data accumulated per second is 1.6 GB. To analyze this data, they are divided into windows of 128 records each. Each of these records consists of three readings, one from each antenna, and a time occurrence value.

The analysis algorithm applies the following operations on each sample window (128 records):

- The sample data is assembled into three vectors, each represent the readings from one antenna during the window interval.
- Fast Fourier Transform (FFT) is computed for each vector.
- The phase difference is calculated between each pair of antennas for each frequency component.
- We are only concerned about the frequencies in the bandwidth 30 to 80MHz, therefore, the slope of phase difference for each frequency component in that bandwidth is calculated.
- Finally, the azimuth and elevation are calculated for each frequency component.

Optimally, when the signal captured by all antennas is a lightning discharge, the calculated phase difference is the same for all frequency components. However, to consider the effect of noise, the algorithm calculate standard deviation of the phase differences of all frequency components, and therefore, only those signals associated with small standard deviation of their phase difference are considered to indicate lightning discharges.

Note that the computations are performed on each window of data independent from other windows, and that each window is pretty small (128 records, each is composed of readings from three antennas). This deems the algorithm as being embarrassingly parallelizable. However, since the original solution presented in [17] used a sequential C program for analyzing the data, 1473.96 s (24 minutes) were

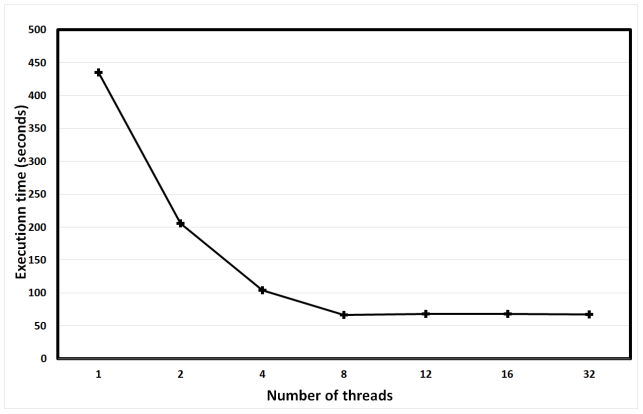


Figure 8: The performance of running the multi-thread version of the lightning application on an EC2 machine with eight cores.

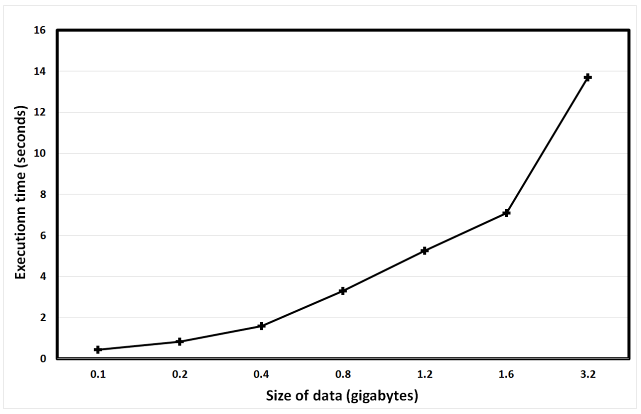


Figure 9: The performance of running the lightning application on different data sizes on one GPU.

taken to process the data captured by the antennas in two seconds [11] [2].

We conclude that using a sequential program is far from a real-time solution and that a better solution can be achieved through processing data in different windows in parallel. Next, we discuss why simple parallel solutions will not reach a real-time processing performance and we then present our solution using DAMB.

4.2 Parallel Algorithms for the Lightning Detection Application

As discussed in the previous section, the data captured by the antennas are partitioned into windows of 128 records each, and the records in each window are analyzed independent from the other windows. We study two approaches for parallelizing the analysis of the data.

The first approach is to use threads. We create `N` threads. Each thread, reads a window of data (128 records) and performs the analysis steps described in Section 4.1. Using threads has better performance as compared to the sequential implementation because it exploits the multiple cores on the machine running the application. However, this solution is limited by the number of cores on the machine. Figure 8 shows the execution time of the lightning application on a machine with eight cores[3]. Note that the execution time of the application reduces when the number of threads is increased until it reaches the number of cores of the machine. For eight cores, the analysis time of the data captured in one second by the antennas took 67 seconds. This solution is certainly limited and will not lead to real-time execution unless there are hundreds of cores on the machine.

Since the multithread implementation for the lightning application can reduce the execution time but is limited to the number of cores on the machine running the application, we explore using GPUs to exploit their thousands cores. We use the CUDA framework to execute the analysis operations for each window of data in a separate thread. The execution time of data captured in one second, which has the size of 1.6 GB, took 13.7 seconds on an EC2 machine that has one GPU (details of the setup of machines are presented in Section 5.1). This approach also has its limitation. GPUs do not scale well with data and do not guarantee fault tolerance.

To increase the parallelism provided by GPUs, we can divide the data into chunk and execute each chunk on a separate GPU. Figure 9 shows the time needed to execute the lightning application on data of different sizes on one GPU. This experiment gives us an indication about the size of a data chunk to send to each GPU to reach real-time processing performance for the entire data generated in one second by the application. The experiment shows that a chunk size has to be smaller than 200MB to be suitable for real-time processing. Next, we show how the solution can be scaled using DAMB to leverage multiple GPUs.

4.3 Real-Time Solution for the Lightning Detection Application using DAMB

We show in the previous section that using GPUs allow employing many parallel threads at any time, however this solution does not scale well with the increase of data size [4]. Furthermore, distributed computing platforms such as Spark guarantee scalability and fault tolerance but do not provide the required degree of parallelism for our solution due to the limited number of cores on CPUs. We describe the architecture of DAMB, which leverages the advantages of GPUs and distributed computing platforms in Section 2. It is worth noting that DAMB is also capable of stream processing of data and therefore it perfectly fits our solution.

The data captured by the antennas are logged and sent in micro-batches to DAMB. We choose the batch size to be one second because this is the minimum streaming rate by Spark streaming. Therefore, a 1.6 GB of data are streamed

[2] We made simple optimization to the code to eliminate unnecessary disk reads/writes and matrix operations, and we managed to reduce the execution time to around 14.5 minutes.

[3] More details about the setup of the experiments is presented in Section 5.

[4] We note that even though the memory of a modern GPU can be enough for executing the entire data streamed per second for the application we are studying, we were able to enhance the performance of the application and reach real-time analysis performance by dividing the work among multiple GPUs. Moreover, our objective is to build a generic framework that is capable of executing hundreds of gigabytes of data in real-time.

to DAMB every second. Upon their arrival, new tasks are created to execute the lightning application.

We write the application using Java, which is one of the programming interfaces for Spark. The code that is run on the GPU is written in C using the CUDA framework. For communication between SparkGPU, which is the distributed execution platform of DAMB, and the GPUs, we implemented the two approaches described in Section 3. To prepare the code for each of them, we do the following:

- Using JNA. In this approach, we write the analysis code for each window of data in CUDA and we compile it as a CUDA native shared library (`*.so`). Our Spark code calls this library as illustrated by the code fragment shown in Figure 4.

- Using Node Manager. We write the lightning analysis code as CUDA Kernel functions. The node manager code is written in C and therefore it can simply link to the CUDA kernel functions and call them from its code. When the node manager receives a message from a Spark task with the data and function to run on the GPU, the node manager copies the data in the message to the GPU and calls the requested Kernel function.

After finishing the analysis of each partition of the data, we write its result (time of occurrence, azimuth, and elevation) to a file stored on Tachyon. This output is momentarily pulled by the visualization tool and plotted as shown in Figure 2.

5. EXPERIMENTS

5.1 Experimental Setup

We implemented DAMB using Kafka version 0.8.2 and SparkGPU that extends Spark version 1.5.1. We conducted our experiments on AWS EC2 instances[5] of two types:

- g2.2xlarge instances. Each instance has 8 virtual CPUs (vCPUs), 15 GB of memory, and one Nvidia GPU with 1536 CUDA cores and 4 GB of video memory.

- g2.8xlarge instances. Each instance has 32 virtual CPUs (vCPUs), 60 GB of memory, and four Nvidia GPU with 1536 CUDA cores and 4 GB of video memory.

Each instance runs Amazon Linux and uses the Nvidia CUDA compiler version 6.5. We setup Spark and SparkGPU on the instances in the standalone mode.

For evaluating the performance of DAMB, we use the lightning detection application described in Section 4. We got a real dataset of lightning signals and the sequential C implementation that are described in details in [17] from Osaka University. Each record in the data represent readings from three antennas and a GPS reading to identify the time. Each reported result is based on the average execution time of three runs. We mainly rely on the measured execution time as our metric to compare the performance of various implementation versions of the DAMB platform. The execution times reported are also for processing data streamed to DAMB in one second, which has the size of 1.6 GB.

[5]Available at: https://aws.amazon.com/ec2/.

5.2 The Execution Performance of DAMB on One Machine

In this section, we demonstrate the effectiveness of executing the lightning application using DAMB as compared to alternative approaches. We compare DAMB with the sequential C version and the multithreaded version of the lightning application discussed in Sections 4.1 and 4.2, respectively. We also compare two versions of DAMB: (1) the executing platform is Spark; and (2) the executing platform is SparkGPU. We chose to execute this experiment on one machine because the sequential C version and the multithreaded version of the lightning application does not scale-out to multiple nodes. In this experiment we use one EC2 instance of type g2.8xlarge that has 32 vCPUs. For this experiment, we vary the number of threads of the multithreaded implementation of the application between one thread and 32 threads, and we vary the configuration of Spark to use one core to 32 cores. For the SparkGPU setup, we set it to use only one GPU.

Figure 10 shows a comparison of the execution time of each implementation of the lightning application. The execution time of the sequential C implementation is the worst. Additionally, this execution time does not improve when we increase the number of cores on the machine, therefore it is not a scalable solution. The multithreaded implementation scales very well when the number of cores on the machine increases. However, this solution does not scale-out and is limited by the maximum number of cores that can exist on any machine. When DAMB uses Spark at its core, the execution time improves when the number of cores increase on the machine, because Spark can efficiently exploit these cores. Finally, the SparkGPU implementation achieved performance that is better than any of the other implementations when the processing of the data is done on one GPU.

It is also shown in Figure 10 that the performance of the multithreaded implementation of the application is very close to that of the Spark implementation of the application. This is mainly because Spark tasks are executed as threads within one Executor on one machine. Besides, the memory is big enough to hold the entire input data streamed in a steaming window (set to one second in the experiments) and any intermediate results. As for SparkGPU, even though it also runs multiple tasks as an inherited feature from Spark, its performance is limited by the number of GPUs allocated for it as all these tasks compete to delegate their work to these GPUs.

Thus, we note that DAMB with Spark or SparkGPU executing the applications have the best performance and can also scale-up and scale-out well. The results of this experiment show that we are still far from achieving real-time analysis, therefore, we need to scale-out the computation across many nodes that are equipped with GPU cards.

5.3 Performance of DAMB when Scaled-Out to Multiple Machines

In this section, we evaluate the performance of SparkGPU as compared to Spark when executing the lightning analysis application. We use the two implementations of SparkGPU that use JNA and a node manager to interface with the GPUs as described in Sections 3.2 and 3.3, respectively. In the experiment, we use EC2 g2.2xlarge instances to setup the Spark and SparkGPU clusters, and we vary the number of nodes in these clusters from one to 12.

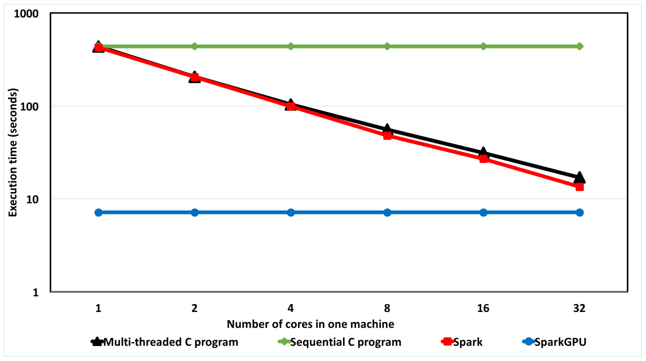


Figure 10: Comparing the execution performance of the lightning application when using a sequential C program, a multithreaded C program, DAMB running Spark and DAMB running SparkGPU.

Figure 11 shows that the execution time required to analyze the data of the lightning application streamed to DAMB per second is reduced when the number of the nodes in the cluster increases. However, when the core of DAMB is Spark, this performance improvement is limited and does not lead to real-time performance when the number of nodes in the cluster reaches 12 and each node has 8 cores.

When the executing platform of DAMB is SparkGPU, the execution time is an order of magnitude less than Spark. Moreover, the performance improves fast to achieves real-time execution. This result is expected, because adding one node equipped with a GPU can achieve performance equivalent to adding several CPU cores.

In this experiment, we also compare the two communication interfaces between SparkGPU and the GPUs. Figure 11 shows that using a node manager lead to better performance. This is because of the following: (1) the node manager uses multiple threads to concurrently access a shared GPU while the JNA uses multiple processes to access the GPU leading to lots of context switching; and (2) the data transfer between Spark and JNA is much slower than the data transfer between Spark and the node manager. We discuss the latter reasoning in more details in the next experiment. However, the figure shows that the performance difference between SparkGPU two communication approaches is not that significant. The main reason is that when there are many GPUs in the cluster the opportunity of many Spark tasks delegating their work to the same GPU reduces. Moreover, the execution time of each task submitted to the GPU is very short, and therefore, other tasks waiting for their turn to run on the GPU will not block for a long time.

Finally, note that we were able to achieve real-time analysis of the lightning application when we scaled-out our cluster and use the SparkGPU platform for executing the analysis tasks. This was not a feasible solution for other alternative implementations.

5.4 Data Transfer in SparkGPU

In this section, we discuss the effect of changing the partition size on the performance of SparkGPU. In the previous section, we demonstrated that the two approaches we studied to enable SparkGPU to communicate with the GPUs in the cluster lead to very close performance, and using a node manager was slightly better than using JNA. We designed this experiment to closely examine the effect of changing the block size on the data transfer overhead in both cases. We change the size of blocks in SparkGPU and measure the time required to transfer this block from the task running by SparkGPU to the GPU. This transfer time is expected to vary based on the communication interface that we use and that are described in Section 3.

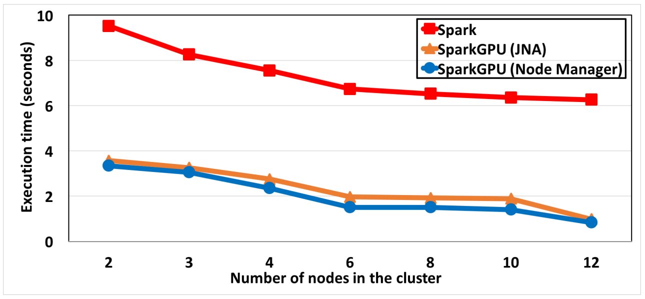


Figure 11: Comparing the execution performance of the lightning application when using Spark, SparkGPU (JNA), and SparkGPU (Node Manager).

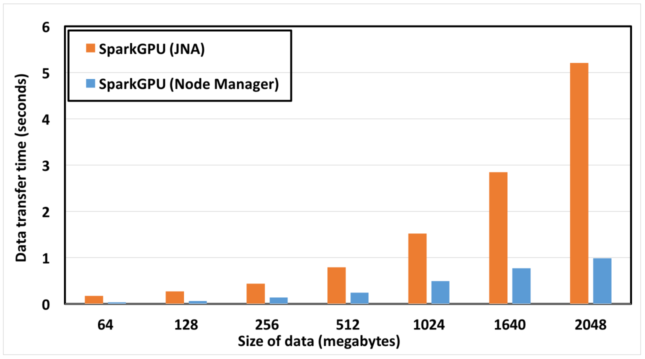


Figure 12: Comparing the time required for transferring the data between SparkGPU and the GPU when using JNA and a node manager for communication.

Figure 12 shows that the data transfer time linearly increases when the block size increases. It also shows that the time required for transferring the data is reduced when we use a node manager. This means that the overhead introduced by using JNA is much higher and that using the node manager can always lead to better performance. Note that even though the difference in the execution time is not significant as shown in Figure 11, saving milliseconds is still important for a real-time application.

5.5 Executing SparkGPU on Nodes with Multiple GPUs

One of the benefits of the node manager described in Section 3.3 is that it allows multiple Spark tasks to concurrently run on one GPU. Besides, if the cluster node is equipped with multiple GPUs, it distribute the Spark tasks to work on them. On the contrary, when we use the JNA approach described in Section 3.2, a Spark task is queued and is assigned to one of the GPUs on the node when it becomes available. The node manager guarantees that the tasks are

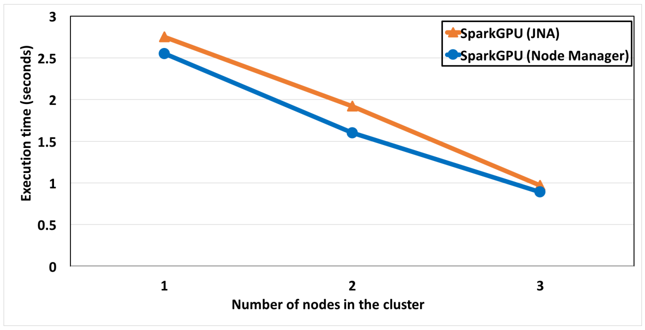


Figure 13: Comparing the JNA and a node manager approached when SparkGPU runs on instances with 4 GPUs.

executed concurrently of the GPUs, therefore we expect that they perform better as compared to using JNA for the communication between the Spark tasks and GPUs. In the experiment described in Section 5.3, we compared using node manager and JNA when running sparkGPU on a cluster of nodes where each only has one GPU. In this experiment, we compare both approaches when we use cluster instances that have multiple GPUs. Therefore, we use EC2 g2.8xlarge instances that have 4 GPUs.

As expected, Figure 13 shows that real-time performance is reached for this application when the number of cores reaches 12 (3 nodes each has 4 GPUs). This indicates that SparkGPU can successfully manage a heterogeneous cluster where each node can be equipped with one or more GPUs, and it can successfully exploit all the resources in the cluster. Moreover, both JNA and node manager approaches have a close performance. However, when using the node manager approach, the execution time is relatively lower as compared to when using the JNA approach. This is mainly due to the concurrency achieved at each node. The difference is not big because the execution time of each task on the GPU is very short and therefore blocking spark tasks until the one running on the GPU finishes and the GPU becomes available would not significantly increase the execution time.

6. RELATED WORK

There has been several work that studied meteorological phenomena for better understanding and prediction [11, 17, 19] studied analyzing lightning using the interferometric approach. Other work focus on predicting precepitation [15, 23], air quality [26], and Tsunami [22]. However, the focus of all this work is the accuracy of the solution rather than the performance. In this paper, our main objective is to achieve a real-time performance of the analysis process. Even though we focus on the application proposed in [17], we can claim that DAMB is capable of processing other streaming applications in real-time.

The LIVE system introduced in [11] proposes hardware and a new less accurate analysis approach for analyzing the lightning. The objective of this work is to achieve real-time performance. However, the time required to analyze the data generated per second takes 28 seconds to 106 seconds based on the quality of the results.

GPUs have successfully been used in simulating and modeling lightning discharges [10, 16]. These work show how the simulations can now be performed on commodity machines with GPUs rather than requiring supercomputer power while achieving good accuracy. In this paper, we show how we use DAMB for processing lightning data that are continuously streamed in real-time.

Few research works have studied employing Spark to manage the execution of applications on a heterogeneous cluster [13, 18]. HeteroSpark [13] is a big data analysis framework that extends Spark to allow it to manage a heterogeneous cluster of CPUs and GPUs. It benefits from the presence of GPUs on worker machines to accelerate the computations. Users of the system write their analysis code in the form of Java code using the normal Spark APIs while relying on the Java remote method invocation (RMI) mechanism to invoke tasks execution on GPUs available locally on each machine or available on other machines in the cluster through the network. The native methods implemented as a shared native library can then be called to accelerate the execution of code on the GPU. The system should work transparently accelerating the performance depending on the available resources in the cluster. The main component of DAMB that performs the parallel execution is the SparkGPU. Its architecture is close to HeteroSpark. However, we optimize this architecture in Section 3.3. Moreover, we compare these two approaches in the experiment section: (1) communicating between Spark and GPU using JNA (similar to HeteroSpark); and (2) communicating between Spark and GPU through a node manager. We show that the latter approach performed better. SparkCL [18] is another framework to provide heterogeneous computing platform. SparkCL uses a more general approach to what can be integrated into the existing big data frameworks by designing the system to have the ability to also use FPGAs and DSPs in addition to GPUs. SparckCL provides a new abstract layer that can call and work with the APIs of Apache Spark and the hardware devices that can be used as accelerators. Users of the system write the analysis code using the higher level layer and then the framework will handle the communications between this high level API with the underlying APIs of spark and OpenCL. There is a trade-off between flexibility as provided by SparkCL and tailoring an optimized code as provided by SparkGPU. We relied on the latter approach because our goal was achieving real-time execution performance, and allowing the users to write their code optimized for execution of GPUs appeared to us as a better alternative. Moreover, we note that DAMB is a platform for parallel processing of streamed data that has SparkGPU at its heart.

7. CONCLUSION

In this paper we present DAMB, an end-to-end framework that process streamed data on a heterogeneous cluster of CPUs and GPUs in real-time and visualizes the result. DAMB leverages Kafka and Spark streaming for providing an efficient and fault tolerant mechanism to read streamed data and start tasks to process them. DAMB employs SparkGPU to execute the applications on cluster nodes that are equipped with GPUs. Therefore, SparkGPU enabled the Spark tasks to delegate their work to be executed on GPUs instead of CPU cores. We present two approaches for the communication between Spark applications and GPUs to delegate tasks to them. We use DAMB to run a lightning application and show that using our platform, we are able to reduce the execution time of the applications from few minutes to less than one second.

Acknowledgment

We would like to thank Prof. Zen Kawasaki (Osaka University and HITACHI Critical Facilities Protection PTE LTD) and Dr. Lotfy Samy (National Research Institute of Astronomy and Geophysics – NRIAG) for sharing lightning datasets collected at Osaka University and the C analysis code presented in [17] with us and for their support and continuous feedback while developing our proposed framework.

8. REFERENCES

[1] Alluxio: Open Source Memory Speed Virtual Distributed Storage. Available at: http://www.alluxio.org/.

[2] Apache Hadoop. Available at: http://hadoop.apache.org/.

[3] Apache Kafka. Available at: http://kafka.apache.org/.

[4] Apache Spark. Available at: https://spark.apache.org/.

[5] Lightning Costs and Losses from Attributed Sources. Available at: http://lightningsafety.com/nlsi_lls/nlsi_annual_usa_losses.htm.

[6] J. Chen, Y. Wu, and Z. Zhao. The new lightning detection system in China: Its method and performance. In *Asia-Pacific Int. Symp. on Electromagnetic Compatibility*, pages 1138–1141, 2010.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. USENIX Conf. on Operating Systems Design and Implementation (OSDI)*, pages 137–150, 2004.

[8] L. Elbaghdady, M. Akita, Z. Kawasaki, and M. Ragab. One site three dimensions lightning location system using VHF broadband interferometers. *Journal of Atmospheric electricity*, 33(2):91–105, 2013.

[9] M. W. Govett, J. Middlecoff, and T. Henderson. Running the NIM next-generation weather model on GPUs. In *Proc. IEEE/ACM Int. Conf. on Cluster, Cloud and Grid Computing (CCGrid)*, pages 792–796, 2010.

[10] A. Gulyás and I. Kiss. The use of low-cost, efficient GPU-based parallel computing in lightning modelling. *Electric Power Systems Research*, 113:41–47, 2014.

[11] Z. Kawasaki, M. Stock, T. Ushio, and M. Stanley. Lightning imaging via VHF emission. In *AGU Fall Meeting*, 2015.

[12] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proc. ACM Symp. on Cloud Computing (SoCC)*, pages 1–15, 2014.

[13] P. Li, Y. Luo, N. Zhang, and Y. Cao. HeteroSpark: A heterogeneous CPU/GPU Spark platform for machine learning algorithms. In *IEEE Int. Conf. on Networking, Architecture and Storage (NAS)*, pages 347–348, Aug 2015.

[14] J. Michalakes and M. Vachharajani. GPU acceleration of numerical weather prediction. *Parallel Processing Letters*, 18(04):531–548, 2008.

[15] S. Otsuka, G. Tuerhong, R. Kikuchi, Y. Kitano, Y. Taniguchi, J. J. Ruiz, S. Satoh, T. Ushio, and T. Miyoshi. Precipitation nowcasting with three-dimensional space–time extrapolation of dense and frequent phased-array weather radar observations. *Weather and Forecasting*, 31(1):329–340, 2016.

[16] G. Pyrialakos, T. Zygiridis, N. Kantartzis, and T. Tsiboukis. GPU-based three-dimensional calculation of lightning-generated electromagnetic fields. In *Int. Conf. on Numerical Electromagnetic Modeling and Optimization for RF, Microwave, and Terahertz Applications (NEMO)*, pages 1–4, 2014.

[17] L. Samy, Y. Nakamura, A. Allam, T. Ushio, and Z. Kawasaki. Ten minutes continuous recording lightning using broadband VHF interferometer. *Advances in Space Research*, 56(10):2218–2234, 2015.

[18] O. Segal, P. Colangelo, N. Nasiri, Z. Qian, and M. Margala. SparkCL: A unified programming framework for accelerators on heterogeneous clusters. *arXiv preprint arXiv:1505.01120*, 2015.

[19] X. Shao, D. Holden, and C. Rhodes. Broad band radio interferometry for lightning observations. *Geophysical Research Letters*, 23(15):1917–1920, 1996.

[20] M. Showerman, J. Enos, A. Pant, V. Kindratenko, C. Steffen, R. Pennington, W.-m. Hwu, et al. QP: a heterogeneous multi-accelerator cluster. In *LCI Int. Conf. on High-Performance Clustered Computing*, 2009.

[21] W. Vanderbauwhede and T. Takemi. An investigation into the feasibility and benefits of GPU/multicore acceleration of the weather research and forecasting model. In *Int. Conf. on High Performance Computing and Simulation (HPCS)*, pages 482–489, 2013.

[22] Y. Wei, A. V. Newman, G. P. Hayes, V. V. Titov, and L. Tang. Tsunami forecast by joint inversion of real-time tsunami waveforms and seismic or GPS data: Application to the tohoku 2011 tsunami. *Pure and Applied Geophysics*, 171(12):3281–3305, 2014.

[23] E. Yoshikawa, T. Ushio, Z. Kawasaki, S. Yoshida, T. Morimoto, F. Mizutani, and M. Wada. MMSE beam forming on fast-scanning phased array weather radar. *IEEE Transactions on Geoscience and Remote Sensing*, 51(5):3077–3088, 2013.

[24] 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 *USENIX Conf. on Networked Systems Design and Implementation (NSDI)*, 2012.

[25] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proc. ACM Symp. on Operating Systems Principles (SOSP)*, pages 423–438, 2013.

[26] Y. Zheng, F. Liu, and H.-P. Hsieh. U-Air: When urban air quality inference meets big data. In *Proc. ACM Int. Conf. on Knowledge Discovery and Data Mining (SIGKDD)*, pages 1436–1444, 2013.