Distributed Interactive Visualization using GPU-Optimized Spark

Sumin Hong, Junyoung Choi and Won-Ki Jeong, Member, IEEE.

Abstract—With the advent of advances in imaging and computing technologies, large-scale data acquisition and processing have become commonplace in many science and engineering disciplines. Conventional workflows for large-scale data processing usually rely on in-house or commercial software that are designed for domain-specific computing tasks. Recent advances in MapReduce, which was originally developed for batch processing textual data via a simplified programming model of the map and reduce functions, have expanded its applications to more general tasks in big-data processing, such as scientific computing and biomedical image processing. However, as shown in previous work, volume rendering and visualization using MapReduce is still considered challenging and impractical owing to the disk-based, batch-processing nature of its computing model. In this work, contrary to this common belief, we show that the MapReduce computing model can be effectively used for interactive visualization. Our proposed system is a novel extension of Spark, one of the most popular open-source MapReduce frameworks, which offers GPU-accelerated MapReduce computing. To minimize CPU-GPU communication and overcome slow, disk-based shuffle performance, the proposed system supports GPU in-memory caching and MPI-based direct communication between compute nodes. To allow for GPU-accelerated in-situ visualization using raster graphics in Spark, we leveraged the CUDA-OpenGL interoperability, resulting in faster processing speeds by several orders of magnitude compared to conventional MapReduce systems. We demonstrate the performance of our system via several volume processing and visualization tasks, such as direct volume rendering, iso-surface extraction, and numerical simulations with in-situ visualization.

Index Terms—MapReduce, Spark, GPU, Distributed Rendering, In-situ Visualization

1 INTRODUCTION

Modern high-throughput data acquisition devices produce vast amounts of data in many disciplines, such as medicine, biology, geoscience, and engineering. Following this ever-increasing trend toward large-scale data acquisition, there is a growing need for high-performance data processing and analysis technology. Open-source platforms based on Google’s MapReduce programming model [1], such as Hadoop [2] and Spark [3], have become the de-facto standard in large-scale data processing in the era of big data. The main strengths of the MapReduce programming model, which is based on functional programming, include its ease of use, high scalability, and reliability. Its simple programming model only requires users to program map and reduce functions, as other data-management and communication tasks in distributed computing are implicitly and automatically handled by the MapReduce system. Therefore, users can focus on the design of the data processing rather than the complicated parallel tasks of scheduling and synchronization, which makes the model easy-to-use for novices. Each map or reduce function provides data parallelism for high scalability in data-intensive computing tasks. The MapReduce model also offers a fault recovery mechanism for reliable, error-free processing of large-scale data. Owing to these features, and despite its originally intended use for the batch or stream processing of textual data, MapReduce has attracted recent attention for its applications in un-conventional, big-data processing tasks, such as high-performance computing (HPC) and machine learning [4].

There have also been research efforts to employ MapReduce as a parallel visualization platform [5], [6], [7], [8], [9]. A seminal work by Vo et al. [6] first demonstrated that how the MapReduce programming model can be effectively used for large-scale visualization and computing problems. In this work, they reported that the Hadoop-based visualization system can render a volume of 27 billion voxels (3072 × 3072 × 3027, 108 GB) on a 100-megapixel framebuffer using triangle rasterization in 19 minutes on a 60-node CPU Hadoop cluster. Although the raw data size is in the multi-gigabytes range, a rendering speed of several minutes per frame is less practical when compared to the visualization systems specifically developed for distributed rendering [10], [11]. This performance issue arises mainly from the fact that the Hadoop and Spark frameworks were originally developed for the batch processing of embarrassingly parallel tasks, and interactivity was not the main performance goal. The MapReduce frameworks suffer from various performance bottlenecks and overheads inherent to their system architecture, such as slow, disk-based shuffle communication and the memory management overhead of Java Virtual Machine (JVM). In addition, although visualization tasks are highly data-parallel, the original Hadoop and Spark systems [2], [3] are designed only for multi-core CPU clusters and do not natively support computing accelerators (e.g., GPUs or MICs). Therefore, using the computing accelerators in the existing MapReduce systems for visualization has been challenging. The MapReduce programming model is also considered simple but less flexible, as its computations must be modeled by decomposing data into key-value pairs and then reducing their values based on corresponding keys, which is neither straightforward nor efficient for certain visualization algorithms.

While recent studies have noted that the MapReduce model could benefit from the use of computing accelerators [4], [7], [12], [13], [14], [15], [16], [17], [18], [19], none have fully addressed the performance issues of MapReduce that warrant the adoption of interactive visualization. For example, recent work by Choi et al. [7] introduced GPU-accelerated Spark system for
Fig. 1: (a) An example of in-situ computing and rendering workflow of electron microscopy connectome data using our system, (b) volume rendering of the deep water impact asteroid simulation data, (c) surface rendering of the Mandelbulb data, (d) isosurface rendering of the solution to eikonal equation.

visualization, and reported that rendering a volume of 9.6 billion voxels \((1856 \times 1612 \times 3240, 38 \text{ GB})\) on a 4-megapixel framebuffer using ray casting took 3.5 minutes with an 8-node GPU Spark cluster. This seems a big improvement over Vo et al. [6], but is still far from acceptable performance for interactive visualization. Based on this observation, we propose a novel extension of the Spark framework that allows for interactive, distributed volume processing and visualization. The main contributions of our work\(^1\) are multi-fold. First, the extension supports NVIDIA’s CUDA and OpenGL application programming interfaces (APIs) for GPU-accelerated computing and raster graphics. The proposed system includes a GPU worker manager that runs concurrently with the conventional Spark system, thereby effectively managing Spark’s performance overhead. Second, we have optimized several points of intra- and inter-node communications in Spark, including the provision of GPU in-memory caching to minimize CPU-GPU data transfer for iterative workloads; the development of an MPI-based, node-to-node direct communication method to drastically reduce the shuffle performance overhead; and the introduction of CUDA-OpenGL buffer sharing for faster in-situ visualization using raster graphics. These optimizations allow for interactive framerates in distributed visualizations using MapReduce. Finally, we developed new APIs that are fully compatible with the current, Python-based Spark APIs, thus providing GPU acceleration with minimal modification of the existing Spark code. To the best of our knowledge, this is the first work that successfully demonstrates interactive visualization using an open-source framework for MapReduce. We assessed the performance of our system in several distributed, interactive, and in-situ visualization applications, including sort-last, parallel volume rendering; parallel iso-surface extraction; and iterative, numerical simulation using domain decomposition.

2 RELATED WORK

MapReduce Framework The MapReduce programming model, which compiles all programming steps into two functions, \texttt{map} and \texttt{reduce}, was first proposed by Google [1]. The core idea of this programming model is a simplified data processing using key-value pairs. Many open-source frameworks for the MapReduce programming model have been developed. Among these, Hadoop is considered the most representative [2], and various software tools have been created in the Hadoop ecosystem, such as the Hadoop distributed file system (HDFS) [20] and a Hadoop resource manager (YARN). Spark has recently emerged as another representative framework for big-data processing via MapReduce [3]. Spark has a novel data abstraction called a resilient distributed dataset (RDD) [21] and supports in-memory caching capabilities; therefore, Spark can achieve much faster performance in iterative computing tasks when compared to Hadoop. In addition, Spark is also compatible with the Hadoop ecosystem, and has greater usability as it supports both a functional language (Scala) and an interpreted language (Python).

GPU-Accelerated MapReduce The computational power of the GPU has already been established as a key part of building

1. Source code is available at https://github.com/hvcl/spark_in_situ
HPC systems. As such, there have been many attempts to combine MapReduce with the GPU’s computing capabilities. Earlier studies focused on implementing a standalone MapReduce framework based on GPU devices [13], [14], [22], [23]. Some researchers tried to develop a heterogeneous framework that supported both CPU and GPU computing [12]. Later works focused on implementing a GPU accelerator in a more popular MapReduce ecosystem, such as Hadoop [15], [16], [17], [24] or Spark [4], [7], [18], [25]. However, these systems concentrate only on accelerating the calculation tasks of the map and reduce functions. Two studies proposed a more advanced technique that allows for GPU in-memory caching of intermediate data [19], [25]. Another study demonstrated that not only could GPU acceleration be achieved, but the cost of data shuffling could be reduced using a separate socket-based communication rather than disk-based shuffling [7], but its performance was still not sufficient for interactive visualization.

**MapReduce for Visualization and Scientific Computing**

In the field of data visualization, MapReduce is an uncommon model; nonetheless, some pioneers have shown the potential of MapReduce for large-scale data visualization. Stuart et al. [5] proposed a new multi-GPU MapReduce system (built from scratch using C++) for volume rendering, which is not compatible with Hadoop or Spark. Another study implemented a rendering system for meshes based on the MapReduce algorithm [8]. Several researchers have implemented rendering in Hadoop [6], as well as Spark [7]. Vo et al. [6] first demonstrated visual computing (e.g., iso-surface extraction and mesh simplification) on the Hadoop framework, and Eldawy [9] demonstrated a visualized analysis method for spatial data in Hadoop. Choi et al. [7] proposed visual computing using Spark with GPUs. However, none of the existing MapReduce systems achieved visualization at interactive rates.

There have also been research directions to apply the MapReduce model to the domain of general computational science for dealing with more complicated, multi-dimensional data formats other than image data. Buck [26] introduced a NetCDF dataset for Hadoop that enables the processing of multidimensional, array-based scientific data. Palamttam [27] developed the sRDD and APIs for Spark, which extends its original RDD to scientific data formats. There have also been attempts to compute complex scientific data in Hadoop [28] and Spark [29]. Two recent works attempted to process large-scale 3D volumes using a domain decomposition method in Spark [7], [19].

**Large-scale Visualization Systems and Languages**

The Visualization Toolkit (VTK) [30] is one of the most popular open-source software libraries for visualization. ParaView [10] and VisIt [11] are well-known, widely-used software systems for large-scale distributed visualization. ParaView supports distributed and parallel visualization based on the Image Composition Engine for Tiles (IceT) library as well as in-situ visualization via the Catalyst plugin [31]. Recently introduced VTK-m extended the functions of the original VTK for fine-grained parallel computing using modern parallel architectures, such as GPUs and MICs [32], [33]. VisIO [34] provides a new I/O library containing locality-aware algorithms that can efficiently use non-Posix distributed file systems (e.g., Hadoop Distributed File System) in traditional visualization systems such as ParaView. Several GPU-friendly distributed visualization systems [35], [36], [37] have also been proposed. Several researchers proposed domain-specific languages for visualization and parallel processing leveraging GPUs [38], [39], [40]. OSPRay [41] is a CPU-based distributed scientific visualization tool. It is built on the Intel SPMD Program Compiler (ISPC) [42] and takes full advantage of various vector instruction set architectures (ISAs). Therefore, OSPRay is optimized for multi-CPU based HPC resources as well as Xeon Phi.

**3 BACKGROUND**

**3.1 MapReduce and Spark**

MapReduce is an easy-to-use programming and computing model for Big Data processing on large-scale distributed systems. The main idea of MapReduce is to hide all the details of complicated parallel execution and data management from the users and expose only two-level primitive programmable stages to the users: map and reduce. The map function generates key-value pairs from the input data. Then, the intermediate data from the map function are grouped or sorted based on the key assigned to each value, which is called shuffle. Once intermediate data are sorted, then the reduce function finally collects the data and performs computation to generate the final result. This simplified programming model allows users to easily write parallel computing code that leverages distributed computing resources and scales to large data sets. The fault recovery mechanism with redundant copies of data provided in many MapReduce systems adds more stability to large-scale distributed computing. However, the slow disk-based I/O performance of MapReduce becomes a major hurdle for interactive or iterative tasks, such as volume rendering. Figure 2 is a pictorial description of the conventional MapReduce model.

Spark is one of the popular open-source MapReduce platforms, which provides several unique features that conventional MapReduce systems lack. One is a data abstraction called Resilient Distributed Datasets (RDD), which is a read-only data object to manage distributed data in Spark. There are two types of operations that can be applied to RDDs in Spark. One is the transformation to create a new RDD, and the other is the action to produce a result (for example, return an array to the driver program). Transformation can be further classified into two types: one is for narrow dependency (map, filter, etc.) based on a simple serial dependency between the parent and child RDDs, and the other is for wide dependency (reduce, group, sort, etc.) where a child RDD has multiple parent RDDs. A sequence of transformations in a narrow dependency can be combined into a single fused transformation stage, called a pipelined RDD. This is useful to minimize the data I/O by reducing the number of operation stages. Another important feature of Spark is in-memory caching, which allows storing RDDs in memory and accessing them multiple times in
different stages. These in-memory caching and data reusing features make Spark more efficient for iterative computing compared to other conventional disk-based MapReduce systems (e.g., Hadoop). Our proposed system further extends Spark to support GPU in-memory caching (i.e., data persistently resides in GPU memory) such that CPU-GPU data communication overhead is minimized.

3.2 MapReduce for Distributed Visualization

Earlier work on MapReduce-based visualization attempted to map the raster graphics pipeline to the MapReduce computing model [6]; the mapper generates a per-fragment color (value) and assigns a corresponding pixel location (key), the shuffle stage groups fragments using pixel locations, and the reducer generates the final pixel color from multiple fragments assigned to each pixel location. Such a fine-grain MapReduce model generates too many intermediate key-value data, which results in significantly slowing down the shuffle stage and makes the rendering process impractical.

Another approach is the coarse-grain MapReduce model [7], which is based on sort-last distributed parallel rendering. In this model, the mapper generates an intermediate, partially-rendered image and assigns a key for the compositing order. Then, the shuffle stage groups intermediate images based on the key. The reducer combines the intermediate images into the final rendered image using either depth comparison for mesh rendering or alpha compositing for volume rendering. In this model, only a few key-value pairs are generated, and thus the shuffle process is much less computationally intensive compared to the fine-grain model. Figure 3 describes the MapReduce model for sort-last distributed rendering. Even a coarse-grain MapReduce model is expensive for interactive rendering in a conventional disk-based shuffle process. In this paper, we introduce MPI-based direct communication between nodes with GPU-accelerated OpenGL rasterization to further optimize the coarse-grain MapReduce model to achieve the interactive rendering performance.

4 Proposed System

4.1 Overview

The main goal of our system is to provide easy-to-use MapReduce APIs with interactive rendering of distributed visualization tasks. To achieve this goal, we implemented a GPU worker that runs independently from Spark’s JVM and is dedicated to GPU-specific computing tasks. The proposed GPU worker was written in C++ to avoid the inefficiencies of JVM and to allow for concurrent execution with Spark workers. We also introduced new GPU-worker APIs for memory management and execution, such as CUDA-kernel and OpenGL execution, GPU in-memory caching, CPU-GPU data transfer, and halo communication. These APIs were implemented using optimized inter- and intra-node communications via the CUDA-OpenGL interoperability and MPI-based direct node communication, which further reduces the latency of the Spark framework. This enables the interactive rendering of distributed in-situ visualization using OpenGl and CUDA on MapReduce. Our system is fully compatible with Python-based Spark (PySpark) so that users can easily adopt our new APIs to leverage GPU acceleration and raster graphics. As the OpenGL context is based on EGL [43], it is easy to install in a headless environment and enables data sharing through the CUDA-OpenGL interoperability. Figure 4 depicts the software stack and the architecture of the proposed system.

Fig. 3: Sort-last parallel rendering using a coarse-grain MapReduce model. Input data are split and stored in the distributed storage, and map functions generate partially rendered intermediate images from each local data. Those intermediate images are grouped in the shuffle stage, and the reduce functions perform image compositing to generate a final rendering result.

Fig. 4: The software stack and architecture of our system. It is based on the standalone cluster mode of Spark, and has additional GPU worker processes and newly-introduced APIs for coordinating between CPU and GPU workers.

4.2 GPU Workers and APIs

Each GPU task is processed by a GPU worker process, and our system provides a GPU worker manager that handles the execution and coordination of GPU workers independently of the Spark executors. The proposed GPU worker manager was built using C++, which allows GPU workers to use off-heap memory while Spark’s CPU workers use JVM’s heap memory. The main idea behind this design choice is to maintain a GPU context on the GPU worker process so that it is not affected by Spark’s main scheduler, thereby allowing permanent data caching in GPU memory regardless of the lifetime of Spark’s CPU workers (note that Spark’s CPU workers are dynamically forked and terminated as new Spark tasks are processed). Our system therefore provides GPU in-memory caching via a series of RDD transformations and without CPU-GPU data transfers. Decoupling the GPU workers from Spark’s CPU workers allows for direct communication between GPUs without the use of slow, disk-based shuffling. Direct communication enables the
cached data in the GPU memory to be efficiently transferred between GPU workers. Table 1 shows a list of the APIs for GPU task execution and the GPU workers’ interactions with Spark’s GPU workers. Listing 1 is a usage example of the GPU worker APIs for the in-situ visualization task. In this example, the volume data is first loaded (line 7), copied to the GPU (line 9), and then processed by the computing code written in CUDA (line 11). The result is then rendered using OpenGL shaders (line 13). During these stages, the data are processed by the computing code written in CUDA and OpenGL shaders.

Listing 1: Code example of in-situ computing and rendering using CUDA and GL shaders.

```python
from vislab.gpuAPIs import *

cuda_code = "..cuda_source_file..
frag_code = "..shader_source_file..
vtx_code = "..shader_source_file..

volume = sc.binaryFiles("..path...")
mesh = volume.map(lambda (key, value): (key, execGPU(value)))
image = mesh.map(lambda (key, value): (key, execGL(value, src=cuda_code, func_name="computing", etc'))) 
result = image.map(lambda (key, value): (key, recvGPU(value))).collect()
```

4.3 GPU Execution Model and In-memory Caching

The GPU worker APIs were built in the C++ environment and support NVIDIA CUDA and Khronos EGL. Our system was designed to receive and execute user code written in CUDA and GL shaders on the fly. To achieve this, the GPU worker APIs support Just-In-Time (JIT) compilation of the CUDA source code. To use this function, the user first passes the necessary data and information to the system, such as the CUDA kernel code, the name of the user function to call in the GPU, and the information of the output as well as other function parameters. The GPU worker then automatically allocates the GPU memory, executes the kernel function, and initializes CUDA’s multi-stream architecture. In the current version of our system, we implemented a single input and output array for GPU execution, which is common in the MapReduce computing model. OpenGL shaders work in a similar fashion by providing the shader source code and related parameters to the GPU worker.

Spark has a unique execution model (i.e., pipelined RDD) for minimizing the cost of reading and writing data via lazy evaluation, and the GPU execution of our system was designed to follow this model. First, our GPU worker APIs are compatible with Spark’s pipelined RDD because they are implemented as functions in Spark’s mapper. Lazy evaluation is also used for communication between the CPU and GPU workers, as the GPU workers in our system run independently of the CPU workers and communicate instead with the socket. Frequent communication between the CPU and GPU workers results in a large network overhead, and thus minimizing CPU–GPU communication is a key for optimization. Among our GPU worker APIs, recvGPU works as a type of action, similar to collect in Spark. When other GPU worker APIs are called, such as sendGPU, execGPU, and execGL, they do not execute the corresponding operation immediately, instead generating a pipelined GPU command. Figure 5 depicts the data flow in the example code of Listing 1, showing a pipelined RDD generated by execGPU and execGL. The pipelined RDD given in this example is executed via collect in line 18. This example contains four GPU worker APIs, including two executions; however, the CPU and GPU workers communicate only once, when recvGPU is executed in Spark’s pipelined RDD.

Another advanced feature of the proposed system is its GPU in-memory caching. Because CPU-GPU communication is expensive, we developed an API for data caching in the GPU memory, cacheGPU. cacheGPU is another type of action, similar to recvGPU, which executes pipelined GPU commands; however, whereas recvGPU delivers the output to the Spark driver, cacheGPU preserves the data in the GPU memory, delivering just 24 bytes of access code. Listing 2 provides an example of the data caching and reuse implemented using our GPU worker APIs. This code consists of two stages, which are marked by the calling of cacheGPU on line 7 and recvGPU on line 13. During these stages, the data are
When data are cached in GPU workers, Spark’s shuffle operations extend Spark’s shuffle operations to include MPI direct communication, which incurs very high communication costs. In our system, cache() which is saved in Spark by cacheGPU (or CPU worker processes) located in the same node. Inter-node executors transfer data using the slower, disk I/O method, and only a small amount of access code remains in Spark’s pipelined RDD, which is saved in Spark by cache(). Afterward, execGPU andrecvGPU refer the data to the RDD, which is cached on line 7. Using cacheGPU, it is therefore possible to preserve and reuse data in the GPU memory. This is useful for reducing communication costs, especially when iteratively accessing large data, as it prevents data transfer to the GPU memory in every iteration.

Listing 2: GPU data caching and reuse example.

```python
from vislab.gpuAPIs import *
data = sc.binaryFiles("..path..")
data = data.map(lambda (key,value) : (key,cacheGPU(value))).cache()
while True:
    image = data.map(lambda (key,value) :
        (key,execGPU(value,'..etc..'))) 
    result = image.map(lambda (key,value) :
        (key,recvGPU(value))).collect()
```

4.4 Direct Shuffle between GPU Workers

The biggest performance bottleneck in the MapReduce model comes from its slow shuffling. Spark currently supports faster shuffling using memory caching, but only among multiple executors (or CPU worker processes) located in the same node. Inter-node executors transfer data using the slower, disk I/O method, which incurs very high costs in communication. In our system, we extended Spark’s shuffle operations to include MPI direct communication between all GPU workers, which works on data cached in the GPU memory by cacheGPU. When data are cached in the GPU memory, only a small snippet of their identification code remains in Spark’s pipelined RDD, as the actual data exists in the GPU memory. If the identification code is moved to another computing node via Spark Shuffle, the actual data must be migrated together. This migration currently uses a high-speed MPI transmission system. To support this, we introduced a simplified, external GPU shuffle manager, the use of which is demonstrated in Listing 3.

Listing 3: Example of shuffle for GPU cached data

```python
mng = GPUshufflemanager()
RDD = RDD.reduceByKey()
mng.gpuShuffle(RDD)
```

If a shuffle operation (e.g., reduceByKey or groupByKey) is called, only a small amount of shuffling is executed for key-value pairs, without using the actual data, which can be completed quickly (see Figure 6). The GPU shuffle manager then examines how the data are moved between the nodes via Spark Shuffle and makes a transfer plan for migrating cached data in the GPU memory between GPU workers. Once the transfer plan is established, it is delivered to each GPU worker in the network, which performs MPI direct communication according to the established plan. One critical point in this process is to ensure the data are transferred correctly, which can be achieved by creating a synchronization barrier between Spark Shuffle and the MPI transfer. This command does not generate a pipelined RDD as do other GPU worker APIs, and instead operates as an action, such as collect.

4.5 Built-in Viewer

Our system provides a built-in viewer written in QT and C++, (Figure 7). The provided viewer runs independently from the Spark system, and they are communicating via a network (TCP/IP). Therefore, it is possible to split the entire system to a back-end Spark system running on a headless cluster and a user client system runs remotely. After connecting the viewer and the Spark driver, the resulting images are displayed in the viewer by using the command openViewer. This command works in both cases; one is when the data are loaded onto the Spark driver after calling collect, and the other is when the data are still cached on the GPU worker. For the latter case, openViewer replaces recvGPU and the data are directly transferred to the viewer without going through the Spark driver. The built-in viewer can generate viewer-specific parameters, such as model-view matrix and transfer functions, based on the user’s interaction. Such viewer-specific parameters are exchanged between the viewer and the Spark driver by calling connViewer. We demonstrate a simple usage example of the built-in viewer provided in our system (Listing 4). In line 7, viewing parameters are collected from the interactive viewer to the local (Spark driver) variable func_args by calling connViewer. Then, rendering is performed in line 8–9, and the resulting image is sent to the viewer process via openViewer without calling Spark’s action (such as collect) in line 10–11.

4.6 Implementation

GPU Workers: In the current version of our system, NumPy Python data types can be used with the GPU worker APIs, as they are converted to binary format and sent to the C++-based GPU workers. We designed the system so that each computing
Listing 4: Example code for the built-in interactive viewer.

```python
from vislab.gpuAPIs import *

viewer = viewermanager()

// Rendering loop
while True:
    func_args = connviewer(viewer)
    image = data.map(lambda (key, value): 
                     (key, execGPU(value, func_args)))
    image.map(lambda (key, value): 
               openviewer(value)).collect()
```

node manages one GPU worker process and multiple CPU worker processes. They are running concurrently, and CPU workers communicate with a GPU worker via TCP/IP sockets. Our system uses a special data structure, command package, to encapsulate the commands and parameters (i.e., function arguments). Then, when a task is executed, the CPU worker sends the data with the corresponding command package to the GPU worker. If the data is already cached on the GPU, only the cache ID and the command package are sent to the GPU worker to avoid the redundant CPU-GPU data transfer. For each CPU to GPU data transaction, a unique CUDA stream is created and cudaMemcpy is called asynchronously. By doing this, multiple CPU workers send the data to a single GPU worker concurrently, which minimizes the communication latency.

Temporarily cached data in GPU memory using cacheGPU is managed using a Least Recently Used (LRU) cache replacement algorithm when GPU memory is running low. persistGPU is similar to cacheGPU, but it makes a permanent copy that is not swapped out by the LRU algorithm but remains in memory until unpersistGPU is called. User-written CUDA source code passed to execGPU is converted to CUDA PTX code via JIT compilation. Once CUDA PTX code is generated, it is cached and reused if the identical CUDA source code is passed to execGPU again (for example, the same GPU code is executed multiple times) to avoid redundant compilation, which may cause a significant performance bottleneck.

OpenGL support: To use OpenGL in our system (Linux-based Spark), we need to activate an X server in each computing node. The X server typically requires a connection to a working display; however, it is not easy to connect and manage physical displays on all computing nodes in a cluster environment. To resolve this issue, we employed Khronos EGL, which allows creating a GL context without a display attached. We implemented two functions for OpenGL support; initGL generates a EGL context and framebuffer object, and execGL creates actual GL buffer objects and executes the user-provided GL shaders. All intra-node computing and GL rendering tasks can be done without moving data between memory buffers by leveraging CUDA-OpenGL interoperability (i.e., sharing the same memory buffer between CUDA and GL contexts), which is crucial for in-situ visualization.

Halo communication: The domain decomposition method, splitting structured data into blocks of a specific shape, is a commonly used technique to deal with large-scale volumetric data in a distributed computing environment. Since each block is processed independently from the others, there should be a synchronization method especially on the boundary of block (i.e., each block needs to access its immediate neighboring voxel values on the boundary), which is called halo communication. In our implementation, we employed a simple domain decomposition strategy by splitting 3D volume data along the z-direction, which allows transferring a contiguous chunk of memory for halo communication. We implemented two low-level functions for halo communication; extractHalo generates the halo data, and appendHalo attaches the halo data to the current block after data shuffling. To hide the details of splitting and attaching halo, we provide a high-level function gpshuffle with an argument "halo" (see Listing 5).

Listing 5: Example of halo communication
```
1. mng = GPshufflemanager()
2. mng.gpshuffle(RDD, "halo")
```

5 Examples

5.1 Volume Rendering

Our first example is sort-last distributed volume rendering (see Listing 6; in this code snippet, detailed information of data parsing, function implementation, and function arguments are removed for simplicity). We assume that the volume is decomposed into blocks and stored in the HDFS as independent files, where the filename represents the location of the block in the original volume (i.e., block index). In Listing 6 line 14, blocks is initialized by loading the file. When initializing blocks, key is assigned with...
the block index from each file name, and value is assigned with the data from the current block. There exist two GPU functions for volume rendering: render and composite. They are executed by execGPU and execGPUseq, respectively. This code also has two additional CPU functions: getDepth, for calculating the depth of the intermediate rendered image from each block, and sortByDepth, for generating the sorted value list with respect to the depth. Here, for simplicity, we use the map function for sequential data arranged by groupByKey and sortByDepth rather than using a reduce function.

Figure 8 shows the data flow in the implementation of distributed volume rendering (Listing 6). The code contains two actions, gpshuffle and collect. Once the input volume is permanently cached by persistGPU, it is reused without additional CPU-GPU communication (from RDD3 to RDD9, and the following iteration).

5.2 Isosurface Extraction
The second example is isosurface extraction using halo communication (Listing 7). The proposed implementation is based on the marching cube algorithm [44] with a domain decomposition method for distributed processing. Isosurface extraction is a one-time process (similar to a simple data filtering), but each subdomain needs to exchange the data on its boundary with its neighboring subdomains for correct computation, which is called halo communication. Once each sub-region is loaded and cached on the GPU memory (line 3), the shuffle manager mng is created (line 6) using the domain information provided by the user (e.g., list of blocks, size of entire domain, split information and size of halo). After that, calling gpshuffle synchronizes the halo data (line 9). Finally, the GPU marching cube code is executed on each sub-domain by calling execGPU and recvGPU (lines 12–15).
5.3 Numerical Simulation

The last example is a numerical simulation application that requires iterative computation. We implemented an Eikonal equation solver using a finite difference method. The Eikonal equation is a first-order hyperbolic partial differential equation defined as follows:

\[
H(x, \nabla \phi) = |\nabla \phi(x)|^2 - \frac{1}{f^2(x)} = 0, \forall x \in \Omega \subset R^n
\]

\[
\phi(x) = 0, x \in \Gamma \subset \Omega
\]

where \( \Omega \) is the computational domain in \( R^n \) (\( n = 3 \) in our experiment, so \( \Omega \) is defined as a uniform 3D rectilinear grid), \( \Gamma \) is the collection of seed points (i.e., boundary condition), \( \phi(x) \) is the travel time or the distance from the seed region to the grid location \( x \), and \( f(x) \) is a positive speed function defined on \( x \). We solve this equation using a first-order Euler method by iteratively updating the solution. Listing 8 is the implementation of the proposed method. The main while loop iteratively updates the RDD blocks (lines 8–15). Whenever blocks are updated, the halo data should be exchanged by calling gpuShuffle to ensure the correct result. Note that gpuShuffle for the halo exchange (line 9) is operated as an action, followed by execGPU and cacheGPU for updating blocks (lines 12–15). Note also that blocks is in the GPU memory but is not read-only in this case; therefore, cacheGPU is used instead of persistGPU (lines 4 and 15).

6 Performance Analysis

Our proposed system is developed on Spark version 2.1.0, Python 2.7.13, NVIDIA CUDA 8.0 and MVAPICH2 2.3. All the experiments were conducted on a cluster system with a master node and 16 compute nodes. The master node is equipped with two deca-core Intel Xeon CPUs and 128 GB main memory. Each compute node is equipped with either an octa-core or deca-core Intel Xeon CPU, 64 GB memory, and an NVIDIA Titan X (GM200 or GP102) or GeForce 1080Ti (GP102) GPU. All nodes are connected via a QDR Infiniband network and communicate using IPoIB (IP over Infiniband). We constructed the distributed experimental environment on a HDFS and a standalone cluster-mode of PySpark. We used eight CPU cores per computing node for the experiments. Each benchmark is evaluated on four different Spark implementations: a conventional CPU-based Spark system (labeled as Spark-C), a naive GPU support (task offloading) in Spark (labeled as Spark-G), Spark with GPU in-memory cache support (labeled as Spark-G-IM), and Spark with GPU in-memory cache and MPI-based direct GPU shuffle system (our proposed system, labeled as Spark-G-SH). We configured the data block size in an HDFS and the number of blocks per node depending on the Spark implementation. For the CPU-based Spark, we assigned the number of blocks identical to the available number of computing cores per node (i.e., eight blocks per node). For the GPU-based Spark, because there is only one GPU per node, we used the largest block size allowed in the Spark system (i.e., 2 GB per block) and reduced the total number of blocks. For example, when experimenting with 64 GB data using 16 compute nodes (with a total of 128 CPU cores and 16 GPUs), the CPU-based Spark used 128 0.5-GB blocks and the GPU-based Spark used 32 2-GB blocks. We repeated same experiment twenty times and computed the average for all running time measurement. Initial data loading and memory caching times are common to all Spark implementations, and thus we do not include them in the comparison result.

6.1 Volume Rendering

The first experiment is direct volume rendering (ray casting) using the Deep Water Impact Asteroid Simulation data\(^2\) (Figure 9). For this experiment, we generated rectilinear structured volumes on different scales, from \( 840 \times 520 \times 448 \) (8.0GB) to \( 2136 \times 1304 \times 1116 \) (127.4GB), by resampling the original unstructured grid. Our volume rendering is sort-last parallel rendering using binary tree compositing. Because the number of blocks is different depending on the Spark implementations, the number of compositing steps is also different (for example, for the binary tree compositing, seven compositing steps are required for 128 blocks in the CPU-based Spark, but only four compositing steps are used for 16 blocks in the GPU-based Spark). We did not use any advanced compositing algorithms (e.g., binary-swap compositing [45]) in our experiments.

Figure 11 shows a comparison of the rendering time of our MapReduce volume ray casting on four different Spark systems using the various combinations of input data and output screen

---

\(^2\) http://oceans11.lanl.gov/deepwaterimpact/
Fig. 10: In-situ rendering of neuron segmentation from mouse brain electron microscopy (EM) data.

Fig. 11: Comparison of single frame rendering time (in a log scale) of MapReduce volume rendering for various screen sizes and system implementations.

sizes; the input volume sizes are 16, 32, 64, and 128 GB, and the output screen sizes are $512 \times 512, 1024 \times 1024, 2048 \times 2048,$ and $4096 \times 4096$. The per-frame rendering time of Spark-G ranges from 6 to 64 seconds, which is, on average, approximately $2.4 \times$ faster than Spark-C. However, Spark-G also stores input volume data in the heap space of JVM, as does Spark-C; therefore, the entire volume data must be transferred from the CPU to GPU via the PCI-e bus for every frame. In addition, Spark-C and Spark-G are based on JVM, and thus the system overhead of data processing and memory management become more dominant as the data size increases. Therefore, when rendering a 128 GB volume, the performance difference between Spark-C and Spark-G becomes less severe (around $1.47 \times$, Figure 11d). If we employ GPU in-memory caching (Spark-G-IM), the JVM system overhead is effectively reduced because the data reside in the GPU memory.

As a result, Spark-G-IM is, on average, approximately $18 \times$ faster than Spark-G for the small screen sizes ($N = 512$). However, when the screen size increases ($N = 4096$), the difference between Spark-G and Spark-G-IM is reduced to approximately $2.5 \times$. Even though Spark-G-IM employs GPU in-memory caching, the compositing step exchanges intermediate images via Spark disk-based shuffle, which becomes the main bottleneck as the screen size increases (up to $70\%$ of the total time for Spark-G-IM). For example, each intermediate rendered image for a $4096 \times 4096$ screen size is 256 MB (RGBA float buffer for color and alpha accumulation); therefore, the total shuffle data size reaches up to 32 GB for 64 blocks. Spark-G-SH significantly reduces this shuffle cost by using MPI-based direct GPU communication. As a result, on average, Spark-G-SH is $3.3 \times$ faster than Spark-G-IM and $17 \times$ faster than Spark-C. Our MapReduce volume rendering runs at an interactive rate (i.e., a few seconds per frame) for most cases. A more in-depth analysis of the volume rendering performance of Spark-G and Spark-G-SH, by breaking down the entire process into three stages (rendering, data shuffle, and compositing), is given in the supplementary material (Table 6).

### 6.2 Isosurface Extraction with In-situ Rendering

The second experiment is isosurface extraction and in-situ surface rendering using OpenGL. We used two different datasets for this experiment. The first dataset is the labeled volume for the neuron segmentation from mouse brain electron microscopy (EM) data (Figure 10). In this dataset, each voxel value represents the index of a corresponding neuron. The dimension of the volume is $848 \times 848 \times 848$ (each voxel is a float value, total 2.27 GB in size), and there are 97,000 different neuron labels in total. Table 2 shows the details of the experimental result. In this experiment, we compared the computing time for extracting the surfaces of 100, 500, 1000, and 5000 neurons (we excluded disk I/O times for initial data loading and the final result storing to compare only the computing time). The generated surface is a triangular mesh where each face (triangle) consists of 108 bytes of information for geometry, normal, and color per vertex. This task is compute-bound because the marching cube time dominantly increases as the number of extracted neurons increases. Thus, Spark-G has the advantage of using the GPU for computation over Spark-C, which results in approximately $145 \times$ maximum speed-up (see Table 2a). Spark-
TABLE 2: (a) Execution time (in seconds) of the isosurface extraction from the mouse brain EM dataset for various output sizes (number of labels) and Spark systems, (b) in-situ rendering time (in seconds) of various resulting mesh sizes on \textit{Spark-G-SH}. The input data is a 848\(^3\) float volume (2.27GB in total size) and the output screen size is 2048 \times 2048.

![Figure 12: Comparison of volume rendering time (single frame) of Spark-G-SH, ParaView and OSPRay for various input volume and screen sizes.](image)

Fig. 12: Comparison of volume rendering time (single frame) of Spark-G-SH, ParaView and OSPRay for various input volume and screen sizes.

TABLE 3: Execution time (in seconds) of the isosurface extraction from the Mandelbulb fractal dataset for various resulting mesh sizes and Spark systems.

G relies on Spark’s shuffle system for halo communication but \textit{Spark-G-SH} uses an MPI-based direct communication system, and thus \textit{Spark-G-SH} shows better performance on shuffle and further shortens the running time by approximately 1.8 seconds. Table 2b shows the in-situ rendering results. The triangular meshes extracted by CUDA are rendered with Phong shading via OpenGL (the screen size is 2048 \times 2048), and the intermediate rendered images (color and depth) are finally composited using depth comparison. As shown in this result, the marching cube computing time dominates the total time and the rendering time stays almost constant (less than a second). The main rendering performance benefit is from the direct CUDA-OpenGL buffer sharing.

The second dataset is synthetically generated Mandelbulb fractal volume data (Figure 1c), chosen to assess the performance for larger input and output data. Table 3 shows the isosurface computing time for various input and output data sizes. Note that the input volume is binary (1 for object and 0 for empty); therefore, even though the input data are much bigger (the largest input volume is 3200\(^3\)) the computing time is shorter (similar to extracting a single neuron in the previous result). For the binary data, \textit{Spark-G-SH} took only 1.46 seconds to generate approximately one billion triangles from a volume of 3200\(^3\); otherwise it took nearly two minutes using a conventional Spark system (Spark-C).

6.3 Numerical Simulation with In-situ Rendering

The last experiment is the Eikonal equation solver as an example of an iterative numerical simulation with in-situ visualization (Figure 1d). This example was chosen to demonstrate the performance of GPU in-memory caching and fast halo communication using direct GPU shuffle on iterative tasks where both computing and communication time become the major bottlenecks. Our solver iteratively updates the solution of Eikonal equation defined on the computational domain, and the current solution can be visualized using in-situ rendering of level sets. Table 4 shows the results of computing and in-situ rendering of the proposed Eikonal solver. The input volumes are 1344\(^3\) (9.0GB), 1696\(^3\) (18.1GB), and 2144\(^3\) (36.7GB), and the output screen size is 1024 \times 1024. Note that the Eikonal solver, marching cube, and OpenGL rendering take place on the GPU without CPU-GPU communication, and the dominant time is used for computing rather than rendering (halo shuffle and Eikonal solver), which demonstrates the benefit of our system for in-situ rendering for iterative tasks.

7 DISCUSSION

7.1 Comparison with Existing Systems

Even though our system is not designed to compete with (or replace) the existing software systems optimized specifically for distributed visualization (such as ParaView [10], VisIt [11] and OSPRay [41]), it is worth comparing to those systems to see how our proposed system performs. Figure 12 compares the
distributed volume rendering performance of our method to that of ParaView and OSPRay. We used ParaView’s default setting for distributed volume rendering with Image Composition Engine for Tiles (IceT) for parallel compositing. Because OSPRay is a CPU-based distributed visualization system, we used a total of 128 CPU cores for 16 nodes instead of GPUs. For a small screen size (512 × 512), ParaView and OSPRay outperformed our method, by up to 4.3 × and 12.1 ×. However, as the screen size increases, our method starts to outperform ParaView, by up to 1.65 × on the 8192 × 8192 screen size. OSPRay is superior to our method in all cases, but we observed the trend that the performance gap decreases as the screen size increases.

We compared our method with ParaView, one of the widely used distributed visualization systems, for isosurface in-situ visualization (Table 7). In this scenario, we first applied distributed marching cube to generate the triangular mesh on the fly, and then it was rendered using an OpenGL surface renderer. In our method, once the data are loaded to the GPU in each compute node, the GPU marching cube and OpenGL rendering can be performed without CPU-GPU communication using GPU in-memory caching. In ParaView, we tested two contour filters; one is the default VTK isocontour filter running on the CPU, and the other is the GPU-accelerated isocontour filter in VTK-m [33]. In the case of the ParaView CPU filter, there is a huge performance penalty due to computation speed and communication overhead between the CPU and GPU as shown in Table 7. The VTK-m accelerated filter significantly reduced the running time of the contour filter, but the improvement in communication overhead is still limited and does not scale to large data over 633M of faces. We found the result promising because, to the best of our knowledge, this is the first successful attempt to use a general-purpose big-data processing framework (e.g., Spark) for distributed in-situ visualization with the performance comparable to those of existing visualization software systems.

We also compared our method with Vispark [7] for a volume rendering example (Figure 13, Table 5). Vispark is an earlier work of Spark extension supporting GPUs with socket-based inter-node communication, which shares some similarities with our system. However, our system is more advanced and optimized to eliminate the overheads in the conventional Spark system, which is crucial for interactive performance. For a fair comparison, we used the same Zebrafish data (1856 × 1612 × 3240, about 38 gigabytes in size) and the system setting (eight GPU nodes) as shown in [7]. Spark-C is a pure CPU-based MapReduce system built on Scalar-based Spark while Vispark is based on Python-based Spark (PySpark). Therefore, in Spark-C, the volume rendering time is slower but the other system overhead is much smaller. Spark-G uses the independent GPU worker system developed in C++, and the system overhead of data (de-) serialization was reduced, which greatly improved both rendering (from 60s to 12.1s) and compositing (from 155s to 6.1s) times. By introducing GPU in-memory support, the rendering time of Spark-G-IM is further reduced to sub-second (0.9s), which is about 60 × speed up compared to Vispark. Introducing MPI-based direct GPU shuffle in Spark-G-SH further reduces the compositing time (from 5.2s to 1.6s). As shown in this experiment, Spark-G-SH greatly reduced the overhead of Spark and made the system practically useful for visualization tasks; we believe this increase in performance makes interactive use cases feasible in a way that was not possible with predecessor approaches.

Table 5: Execution time (in seconds) of direct volume rendering (ray casting) for a zebrafish volume dataset.

<table>
<thead>
<tr>
<th>Number of partitions</th>
<th>Vispark</th>
<th>Spark-C</th>
<th>Spark-G</th>
<th>Spark-G-IM</th>
<th>Spark-G-SH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rendering time</td>
<td>60</td>
<td>12.1</td>
<td>0.9</td>
<td>0.8</td>
<td></td>
</tr>
<tr>
<td>Composition time</td>
<td>155</td>
<td>35.4</td>
<td>5.2</td>
<td>5.2</td>
<td>1.6</td>
</tr>
</tbody>
</table>

7.2 Fault-Tolerance

One of the nice features of Spark and other MapReduce systems is the automatic fault recovery. Spark’s fault recovery system stores multiple copies of intermediate results in the distributed file system. When a node fails, the other node having the intermediate result in its local disk takes over the task from the failed node and continues. In case the intermediate result of the failed node does not exist in the distributed file system (for example, a node fails during in-memory processing), then Spark re-runs the task from scratch to recover the lost partition by loading the input data from the disk. Our system simply relies on the fault recovery model of Spark. We use -disable-auto-cleanup option in the MVAPICH to prevent automatically cleaning up of the other GPU nodes in the MPI communicator when a node failure occurs.
Figure 14 shows the fault-tolerance experiment of volume rendering. As shown in the graph, after loading the data from HDFS to GPU workers (frame 1), the running time drops to 2 to 4 seconds per frame. We intentionally made a node failed at frame 6, which caused a rolling-back to the closest checkpoint stored on the disk and re-execution of the task from the stored checkpoint. The system immediately returned its steady state at frame 7 after recovery. Note that our GPU caching and GPU checkpoint. The system immediately returned its steady state at frame 6, which caused a rolling-back to the closest checkpoint stored on the disk, which may cause a longer recovery time. The current system makes the best effort to keep the cluster alive then a node failure occurs by skipping automatic clean up function provided in the current implementation of our system.

### 7.3 Limitations

Spark manages all RDD transform information in the form of lineage for fault tolerance. This lineage delegates Spark to migrate tasks to other workers when a computing worker is down. However, there is a limit to the amount of information that this lineage can hold. We observed that more than 500 iterations of successive transformation execution using GPU in-memory caching causes a memory overflow in Spark. To circumvent this, we offload the intermediate data from GPU memory to HDFS once every few hundreds of iterations. Addressing this problem in Spark’s system level is out of the scope of this study (because it requires the modification of Spark’s main system).

The current system relies on Spark’s default fault-tolerance mechanism. Therefore, any system fault during GPU in-memory processing will be reverted to the nearest Spark checkpoint stored on the disk, which may cause a longer recovery time. The current system makes the best effort to keep the cluster alive then a node failure occurs by skipping automatic clean up function provided in

<table>
<thead>
<tr>
<th># of blocks</th>
<th>Screen Size</th>
<th>Shuffle Amount (GB)</th>
<th>Render</th>
<th>Shuffle</th>
<th>Composite</th>
<th>Render</th>
<th>Shuffle</th>
<th>Composite</th>
</tr>
</thead>
<tbody>
<tr>
<td>16GB</td>
<td>512</td>
<td>0.12</td>
<td>3.2</td>
<td>0.6</td>
<td>0.2</td>
<td>0.1</td>
<td>1.1</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td>0.47</td>
<td>3.5</td>
<td>1.2</td>
<td>0.9</td>
<td>0.2</td>
<td>1.2</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>2048</td>
<td>3.88</td>
<td>3.0</td>
<td>3.1</td>
<td>1.4</td>
<td>0.2</td>
<td>1.4</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>7.50</td>
<td>4.2</td>
<td>16.3</td>
<td>4.5</td>
<td>0.2</td>
<td>2.2</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>8192</td>
<td>30.00</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.3</td>
<td>5.3</td>
<td>0.1</td>
</tr>
<tr>
<td>32GB</td>
<td>512</td>
<td>0.12</td>
<td>5.4</td>
<td>0.7</td>
<td>0.2</td>
<td>0.1</td>
<td>1.2</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td>0.47</td>
<td>5.9</td>
<td>1.3</td>
<td>0.3</td>
<td>0.2</td>
<td>1.2</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>2048</td>
<td>1.88</td>
<td>5.7</td>
<td>5.1</td>
<td>1.4</td>
<td>0.3</td>
<td>1.4</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>7.50</td>
<td>5.7</td>
<td>17.3</td>
<td>4.5</td>
<td>0.3</td>
<td>2.1</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>8192</td>
<td>30.00</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.4</td>
<td>5.3</td>
<td>0.1</td>
</tr>
<tr>
<td>64GB</td>
<td>512</td>
<td>0.24</td>
<td>11.1</td>
<td>0.8</td>
<td>0.2</td>
<td>0.4</td>
<td>1.4</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td>0.97</td>
<td>10.8</td>
<td>2.1</td>
<td>0.4</td>
<td>0.3</td>
<td>1.6</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>2048</td>
<td>3.88</td>
<td>10.0</td>
<td>5.5</td>
<td>1.7</td>
<td>0.4</td>
<td>2.3</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>15.50</td>
<td>10.1</td>
<td>26.8</td>
<td>5.5</td>
<td>0.4</td>
<td>3.2</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>8192</td>
<td>62.00</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.6</td>
<td>8.6</td>
<td>0.1</td>
</tr>
<tr>
<td>128GB</td>
<td>512</td>
<td>0.49</td>
<td>35.1</td>
<td>1.5</td>
<td>0.3</td>
<td>0.4</td>
<td>2.1</td>
<td>0.2</td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td>1.9/</td>
<td>36.1</td>
<td>3.3</td>
<td>0.5</td>
<td>0.4</td>
<td>2.3</td>
<td>0.2</td>
</tr>
<tr>
<td></td>
<td>2048</td>
<td>7.88</td>
<td>38.4</td>
<td>9.5</td>
<td>1.9</td>
<td>0.8</td>
<td>2.9</td>
<td>0.2</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>31.50</td>
<td>33.3</td>
<td>42.3</td>
<td>6.4</td>
<td>0.9</td>
<td>6.2</td>
<td>0.2</td>
</tr>
<tr>
<td></td>
<td>8192</td>
<td>126.00</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1.0</td>
<td>11.2</td>
<td>0.2</td>
</tr>
</tbody>
</table>

### TABLE 6: In-depth analysis of the execution time (in seconds) of volume rendering using Spark-G and Spark-G-SH for various screen and data sizes. Note that Spark-G failed to run on the 8192 × 8192 screen size due to the memory limitation of Spark shuffle system, which is not a problem in Spark-G-SH due to MPI-based direct GPU shuffle communication.

<table>
<thead>
<tr>
<th>Spark System Overhead</th>
<th>Rendering (OpenGL)</th>
<th>Contour Filter (GPU)</th>
<th>Marching Cube (CUDA)</th>
<th>Halo Sync (MPI)</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.40</td>
<td>0.38</td>
<td>0.45</td>
<td>0.31</td>
<td>0.21</td>
<td>0.26</td>
</tr>
</tbody>
</table>

### TABLE 7: Analysis of in-situ visualization performance of Spark-G-SH and conventional (no in-situ) visualization performance of ParaView on the Mandelbulb dataset (in seconds). Note that the conventional ParaView scales well but slow and the Paraview with VTK-m is faster but does not scale over large data. Our Spark-G-SH system scales data between CUDA and OpenGL contexts via buffers, which allows in-situ visualization without data transfer between the CPU and GPU. By doing this, the system scales well over large data as the conventional Paraview with high performance faster than VTK-m.
MPI. However, this does not guarantee the stability of the system for execution of longer tasks. Developing more sophisticated fault recovery method, such as a GPU-level fault-tolerance mechanism (e.g., storing intermediate GPU cache to CPU memory or disk for recovery) or robust management of the MPI communicator for a node failure, is left for the future work.

8 Conclusion and Future Work

In this paper, we introduced a novel extension of the Spark framework for distributed-in-situ visualization using MapReduce. The proposed system provides GPU in-memory caching and MPI-based direct shuffle to drastically reduce the overhead of the conventional Spark system, which enables GPU-accelerated in-situ visualization using raster graphics in Spark. We demonstrated that the proposed system performs several orders of magnitude faster than the conventional MapReduce system on visualization tasks, which is comparable to existing distributed visualization systems. The proposed system also provides high-level APIs, which are fully compatible with the Python-based Spark so that current Spark users can easily employ the new functions in their existing codes.

Even though we demonstrated visualization applications in this paper, we believe that the proposed system has a potential as a powerful computing platform for other compute-intensive applications as well, such as machine learning. In the future, we plan to leverage GPU-Direct RDMA over MPI for further accelerating the communication between GPU workers across multiple nodes. Scale out experiment on a large-scale cluster system is another future research direction.

Acknowledgments

This work was partially supported by the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2017R1D1A1A09000841) and the Ministry of Science and ICT (NRF-2019M3E5D2A01063819, NRF-2014K1A3A1A05034557), and by the Korea Health Industry Development Institute (KHIDI) funded by the Ministry of Health Welfare (HI118C0316). The authors would like to thank Prof. Hanspeter Pfister and his group at Harvard University for providing the connectome dataset.

References

distributed computing to weather event detection and tracking,” in Big Data (Big Data), 2015 IEEE International Conference on. IEEE, 2015, pp. 2020–2026.


Junyoung Choi received a BS degree from Ulsan National Institute of Science and Technology (UNIST), Korea in 2017. He is currently working towards the PhD degree in the school of electrical and computer engineering, UNIST. His research interest includes visualization, machine learning and extended reality.

Won-Ki Jeong is currently an associate professor in the department of computer science and engineering at Korea University. He was an assistant and associate professor in the school of electrical and computer engineering at UNIST (2011-2020), a visiting associate professor of the neurobiology at Harvard Medical School (2017–2018), and a research scientist in the Center for Brain Science at Harvard University (2008–2011). His research interests include visualization, image processing, and parallel computing.

He received a Ph.D. Degree in Computer Science from the University of Utah in 2008, and was a member of the Scientific Computing and Imaging (SCI) institute. He received the NVIDIA Graduate Fellowship in 2007 and hosted the NVIDIA GPU Research Center at UNIST in 2014.