GPU-Accelerated High-Throughput Online Stream Data Processing

Zhenhua Chen, Jielong Xu, Jian Tang, *Senior Member, IEEE*, Kevin A. Kwiat, Charles Alexandre Kamhoua, *Senior Member, IEEE*, and Chonggang Wang, *Senior Member, IEEE*

Abstract—The Single Instruction Multiple Data (SIMD) architecture of Graphic Processing Units (GPUs) makes them perfect for parallel processing of big data. In this paper, we present the design, implementation and evaluation of *G-Storm*, a GPU-enabled parallel system based on Storm, which harnesses the massively parallel computing power of GPUs for high-throughput online stream data processing. G-Storm has the following desirable features: 1) G-Storm is designed to be a general data processing platform as Storm, which can handle various applications and data types. 2) G-Storm exposes GPUs to Storm applications while preserving its easy-to-use programming model. 3) G-Storm achieves high-throughput and low-overhead data processing with GPUs. 4) G-Storm accelerates data processing further by enabling Direct Data Transfer (DDT), between two executors that process data at a common GPU. We implemented G-Storm based on Storm 0.9.2 and tested it using three different applications, including continuous query, matrix multiplication and image resizing. Extensive experimental results show that 1) Compared to Storm, G-Storm achieves over 7× improvement on throughput for continuous query, while maintaining reasonable average tuple processing time. It also leads to 2.3× and 1.3× throughput improvements on the other two applications, respectively. 2) DDT significantly reduces data processing time.

Index Terms—Stream data processing, GPU, parallel computing, big data infrastructure

1 Introduction

We are well underway in the era of big data. The amount of data created are getting bigger at an ever increasing pace due to the rapid rise of mobile and wearable devices, social media, etc. Despite various means to capture the data, it is very challenging to extract valuable and meaningful insights from huge-volume, fast and continuous data streams as soon as they become available. In the past decade, MapReduce [8] and Hadoop [11] have emerged as the *de facto* programming model and system for data-intensive applications respectively. However, MapReduce and Hadoop are not suitable for online stream data applications because they were designed for offline batch processing of static data, in which all input data need to be stored on a distributed file system in advance. Recently, several general-purpose distributed computing systems, such as Storm [27] and S4 [25], have emerged as promising solutions for online processing of unbounded streams of continuous data at scale. Their architectures are quite different from MapReduce-based systems and they employ different message passing, fault tolerance and resource allocation mechanisms that are designed specifically for online stream data processing.

A Graphics Processing Unit (GPU) is a massively-parallel, multi-threaded and many-core processor, whose peak computing power is considerably higher than that of a CPU. For example, a single NVIDIA GeForce GTX 680 GPU delivers a peak performance of 3.09 Tera Floating Point Operations Per Second (TFLOPS) [10], which is over 28 times faster than that of a 6-core Intel Core i7 980 XE CPU. The Single Instruction Multiple Data (SIMD) architecture of GPUs makes them perfect for parallel processing of big data. However, research on stream data processing with GPUs is still in its infancy.

Research efforts have recently been made to utilize GPUs to accelerate MapReduce-based systems and applications, such as Mars [12], MapCG [13], etc. However, these solutions cannot be applied to solve our problem since the programming model and system architecture of a MapReduce-based offline batch processing system are quite different from those of an online stream data processing system (such as Storm). However, scant research attentions have been paid to leveraging GPUs for accelerating online parallel stream data processing. In this paper, we present design, implementation and evaluation of *G-Storm*, a GPU-enabled parallel system based on Storm, which harnesses massively parallel computing power of GPUs for high-throughput online stream data processing.

A Storm application is modeled as a directed graph called a topology, which usually includes two types of components: spouts and bolts. A spout is a source of data stream, while a bolt consumes tuples from spouts or other bolts, and processes them in a way defined by user code. Spouts and bolts can be executed as many tasks in parallel on multiple

- *Z. Chen, J. Xu, and J. Tang are with the Department of Electrical Engineering and Computer Science, Syracuse University, Syracuse, NY 13244. E-mail: {zchen03, jxu21, jtang02}@syr.edu.*
- *K.A. Kwiat and C.A. Kamhoua are with the US Air Force Research Lab (AFRL), Rome, NY 13441. E-mail: {kevin.kwiat, charles.kamhoua.1}@us.af.mil.*
- *C. Wang is with InterDigital, Inc, King of Prussia, PA 19406. E-mail: cgwang@ieee.org.*

Manuscript received 2 Dec. 2015; revised 28 July 2016; accepted 14 Sept. 2016. Date of publication 9 Oct. 2016; date of current version 7 June 2018. Recommended for acceptance by J. Chen and H. Wang.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TBDATA.2016.2616116

executors (i.e., threads), workers (i.e., processes) and worker nodes (i.e., machines) in a cluster. A GPU has a quite different programming model and architecture, which poses a few challenges for integrating it into Storm that is originally designed to run only on CPUs. First, it is not trivial to expose GPUs to various Storm applications, which have no knowledge of the underlying hardware that actually carries workload. Second, Storm is a complex event processing system which processes incoming events (tuples) one at a time as it arrives at each component. Under heavy workload, a Storm cluster can become easily overloaded and processing can come to a halt. GPUs can be leveraged to offload workload to prevent overloading and enable high-throughput processing. To marry GPUs with stream data processing is very challenging and it requires a new computation framework. It can be seen intuitively that it is inefficient if the GPU worked on one tuple at a time. It is essential to achieve high GPU utilization to amortize data transfer overhead while keeping processing latency within a reasonable time. Third, it is not clear if new features provided by the new GPU architecture, such as Kepler [17], can be leveraged to speed up data processing in Storm. Kepler succeeds the Fermi architecture and has several new features such as multi-process service and dynamic parallelism. The design and implementation of G-Storm carefully addresses these challenges, which leads to the following desirable features:

1) G-Storm is designed to be a general (rather than application-specific) data processing platform as Storm, which can deal with various applications and data types.
2) G-Storm exposes GPUs to Storm applications while preserving its easy-to-use programming model by handling parallelization, data transfer and resource allocation automatically via a runtime system without user involvement.
3) G-Storm achieves high-throughput and low-overhead data processing by buffering multiple data tuples and offloading them in a batch to GPUs for highly-parallel processing.
4) G-Storm accelerates data processing further by supporting Direct Data Transfer (DDT) between two executors co-located on a common GPU, which eliminates unnecessary data transfers between host memory and device memory.

To the best of our knowledge, G-Storm is the first GPU-enabled online parallel stream data processing system that is built based on Storm. We believe the main ideas in our design, such as tuple batching, index building and DDT (See Section 3), can be applied to other parallel stream data processing systems, such as Yahoo!'s S4 [25], Google's MillWheel [2] and Microsoft's TimeStream [23], which have similar programming models and system architectures.

2 Background

For completeness of presentation, we briefly introduce Storm, GPU programming and JCUDA in this section.

2.1 Storm

Storm [27] is a real-time stream data processing system that is designed to be distributed, reliable, and fault-tolerant.

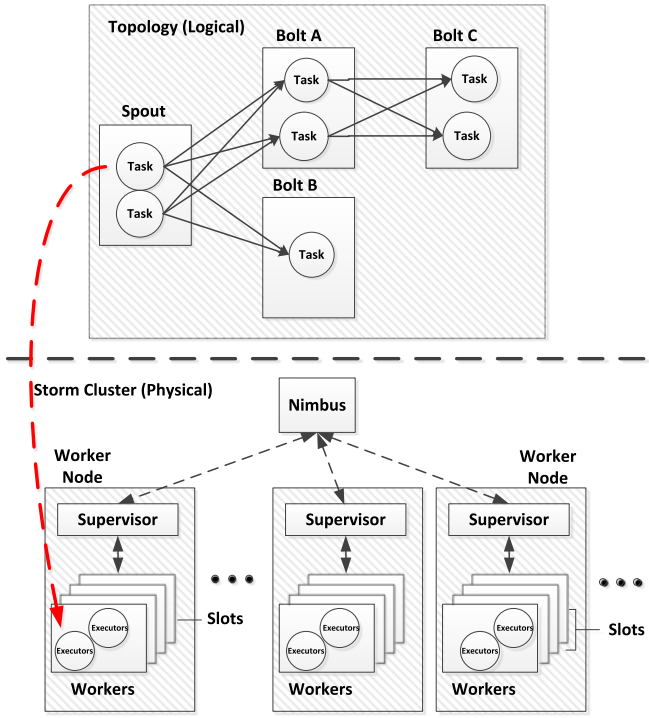


Fig. 1. The logical and physical views of Storm [31].

Unlike batch data processing systems such as Hadoop [11], Storm processes unbounded streams of data in real or near real time.

A Storm application is modeled as a directed graph called a topology, which consists of two types of components: spouts and bolts. Input data are emitted into the topology from spouts in the form of a stream, which is defined as an unbounded sequence of tuples. A tuple is simply a named list of values, which we call *arguments*, and the values can be of any data type. For example, an image tuple may contain arguments called *srcImg*, *filename*, *srcWidth*, whose values are of tpye *byte array*, *string*, and *integer*, respectively.

A spout can read data from external sources such as files or queues. A bolt consumes tuples from spouts or other bolts, and processes them according to user code. A bolt can pass resulting tuples to some other bolts for further processing, or store the data in storage devices. In the topology graph, these components are represented by vertices, where direct edges between the vertices indicate how data streams are routed. At runtime, spouts and bolts are executed as many parallel tasks, which may run on multiple machines in a cluster.

Storm expresses parallelism by using two levels of abstractions (logical and physical), which we illustrate in Fig. 1. In the physical layer, there is a master node in a Storm cluster that serves as a central control unit and a set of physical machines (called *worker nodes*) that actually process incoming data. A topology is executed in one or multiple worker processes (called *workers* for simplicity) running on one or multiple worker nodes. There is a daemon called *supervisor* that runs on each worker node that listens for any work assigned to it by the scheduler. Each worker node is configured with slots, which are ports used to receive messages. The number of slots is used to indicate the number of

workers that can be run on this worker node depending on hardware resources, and is usually pre-configured by the cluster operator. In most cases, it should be set to the number of cores on that worker node. Each worker runs in a Java Virtual Machine (JVM) that executes a subset of tasks defined in a topology. An *executor* is a thread spawned by a worker, which can contain a running instance of a spout or bolt. The master node controlling a Storm cluster runs a daemon called *Nimbus*. Nimbus is responsible for distributing user code around the cluster, and assigns executors of running topologies to workers and worker nodes. It also monitors the health of the cluster by restarting failed workers and re-assigning workload if needed. Storm uses *Zookeeper* [37] as a coordination service to maintain configuration information, naming, and distributed synchronization among worker nodes.

Storm is well suited for big stream data because of its distributed nature, fault-tolerance guarantee, and fast data processing capability that ensures low tuple processing latency.

2.2 GPU Programming and MPS

GPU programming is considerably different from programming applications to run on a CPU, because of dramatic differences in hardware. In 2006, NVIDIA introduced CUDA [7], a programming platform consisting of a runtime and APIs that extend C to ease GPU programming. CUDA allows a developer to write C code with CUDA extensions to run general-purpose computations on a GPU without having to possess intimate knowledge of the graphics API and the GPU architecture. The current GPU architecture features thousands of execution units called CUDA cores. These cores are organized in multiple streaming multiprocessors, each containing thousands of registers, several caches, and thread schedulers. A CPU host communicates with a GPU via a PCIe bus.

The GPU execution model attempts to map a hierarchy of threads to a hierarchy of execution cores on a GPU. A host program launches specialized C functions, known as *kernels*, which are executed in parallel by different threads on a GPU when called. Each thread block is scheduled to be run by a single multiprocessor, and a multiprocessor can run multiple thread blocks in a time-sliced fashion. A CPU host and its GPU device have different memory spaces. Before a kernel can be executed on a GPU, memory space needs to be allocated on the GPU, and data required by the kernel need to be copied from the host memory to the device (i.e., GPU) memory. After execution, output data produced by the kernel need to be transferred back to the host memory.

NVIDIA's Kepler architecture [17] is a successor to the Fermi architecture, which was released in 2012. Kepler was designed for fast double precision computing performance to accelerate professional HPC compute workloads [17]. Kepler introduces several new features, such as Dynamic Parallelism, Hyper-Q and Multi-Process Services (MPS). MPS enables multiple host processes to run CUDA kernels concurrently on a GPU. Before MPS, each host process creates its own CUDA context, and the GPU cannot be shared between two host processes because each CUDA context has its own address space. To increase GPU utilization, MPS allows kernel and memcopy operations from different processes to overlap on a common GPU. MPS allocates one copy of GPU memory and scheduling resources, which can then be shared by all host processes, eliminating swapping overhead and improving utilization. In G-Storm, we leverage this feature for enabling DDT to speed up data processing, which is discussed in Section 3.3.5.

2.3 JCUDA

Storm applications are written in Java. As discussed above, GPU programming is mostly done using CUDA, which is an extension to C. We need to first address the problem of how to expose GPU to a Storm application. The most common approach to access C or CUDA from Java is to use the Java Native Interface (JNI). However, this method is cumbersome and not productive, because it involves writing Java code, JNI stubs in C on a host, as well as CUDA code for the GPU [32]. To make it programmer-friendly, we examined various Java bindings to CUDA and chose JCUDA for our implementation because of ease of use. JCUDA is a set of libraries that give Java programs easy access to a GPU. JCUDA has a runtime API and a driver API that map to the CUDA runtime and driver APIs respectively. JCUDA uses the runtime API to interact with CUDA runtime libraries such as CUBLAS, CUFFT, etc, and it uses the driver API to invoke user CUDA kernels. We utilize JCUDA's driver API in G-Storm to launch user kernels.

3 Design and Implementation of G-Storm

3.1 Overview

Because of differences in programming models, in order to utilize GPUs in Storm, a novel framework is needed. It is not trivial to simply store incoming tuples in buffers and copy them to and from the GPU. The number of data copy operations needs to be minimized for each kernel launch to reduce overhead while the amount of data copied each time needs to be sufficiently large for high-throughput processing, which are achieved by the batcher described below. Each CUDA thread needs to know the type and location of input data and output space to operate on, and which tuple the data belong to. This requires knowledge of the precise location of each tuple in memory. We call this the index information of a tuple. G-Storm's index construction is both innovative and robust in that it can deal with any input data type and batching size, while using minimal amount of memory footprint.

As illustrated in Fig. 2, the overall design of G-Storm consists of a central control component called the GPU-executor, which runs as a thread on the CPU and launches user kernel to process incoming tuples on the GPU; and three additional modules that enable parallel processing of multiple tuples on a GPU, whose functionalities are summarized in the following:

1) *Batcher*: It works along with the Index Builder and Buffer Manager (described next) for critical operations to batch incoming tuples and extracting output tuples.
2) *Index Builder*: It constructs indices for tuple arguments, which are their locations in memory.
3) *Buffer Manager*: It is responsible for allocating and initializing buffers to store tuples.

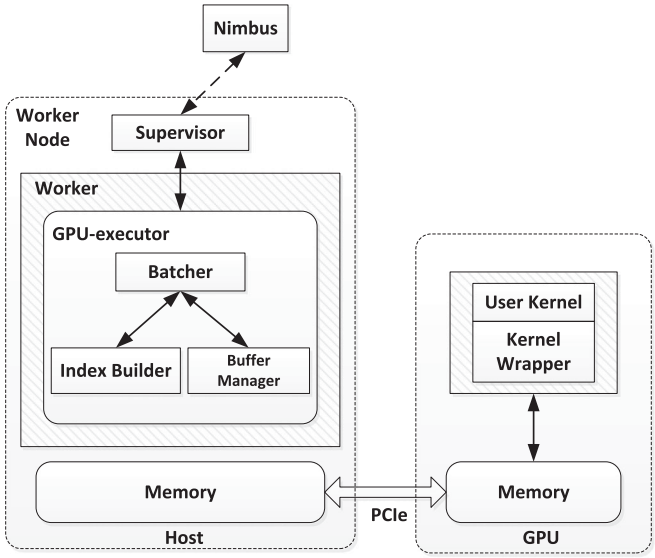


Fig. 2. The architecture of G-Storm.

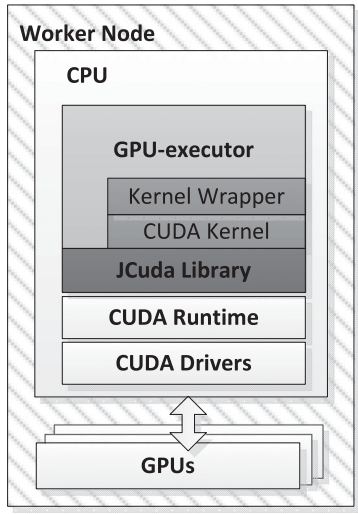


Fig. 3. Software stack of a GPU-executor.

The user needs to provide a PTX file containing a kernel that specifies how to process an incoming tuple on a GPU. Then a kernel wrapper needs to be generated to facilitate kernel execution across multiple tuples. Next, we summarize the workflow of tuple processing in a GPU-executor at runtime:

1) When a tuple arrives at the GPU-executor, its Batcher adds the new tuple to a dedicated buffer.
2) Once the desired number of tuples to be buffered (i.e., batching size) is reached, the Batcher allocates the necessary input and output spaces on the GPU (device) memory and copies the input data (from tuples) from the host memory to the device memory to prepare for processing.
3) The Batcher calls the Index Builder to construct the necessary index information for batching and extracting tuples (See Section 3.3.3). The index information is copied along with input data to the device memory so the kernel can extract the data and align threads to operate on the data.
4) The GPU-executor launches the kernel wrapper, which itself is a kernel function for locating input data and output memory space in the device memory.
5) The kernel wrapper launches the user kernel, utilizing multiple threads to process batched input tuples in parallel on the GPU.
6) After user kernel execution is complete, output data are stored in a pre-allocated output memory space, which is then copied back to the host at once to minimize transfer overhead.
7) The Batcher uses the (already generated) index information to extract individual output tuples.
8) The output tuples are emitted to the next stage in the topology as it is normally done.

Due to the way we design and implement G-Storm, it has several desirable features, which are explained as follows:

1) G-Storm is designed to be a general-purpose (rather than application-specific) data processing platform as Storm: It can deal with various applications and data types.
2) G-Storm exposes GPUs to Storm applications while preserving its easy-to-use programming model: a GPU-executor launches the kernel provided by the user for data processing on GPUs, and handles parallelization, data transfer and resource allocation automatically without user involvement (See Section 3.2).
3) G-Storm achieves high-throughput and low-overhead data processing: The Batcher buffers multiple data tuples and then offloads them in a batch to a GPU to achieve high resource utilization and low data transfer overhead (See Sections 3.3.2 and 3.3.3).
4) G-Storm accelerates data processing further by supporting DDT between two executors that process data at a common GPU: The GPU-executor eliminates unnecessary data transfers between host memory and device memory by utilizing the MPS of the Kepler architecture to pass data directly to another kernel on the same GPU (See Section 3.3.5).

3.2 G-Storm Programming Model

G-Storm's programming model is very similar to Stor-m's programming model, where a user builds a Storm application by defining a topology consisting of spouts and bolts, and implement the spout and bolt classes by specifying how to emit and process tuples. Additionally, the G-Storm model incorporates the GPU programming model as well by using JCUDA as a bridge to a GPU from Storm.

In G-Storm, we introduce a specialized bolt, called a GPU-bolt, which offloads tuple processing to a GPU. A GPU-executor is a running instance of a GPU-bolt. Fig. 3 shows the software stack of a GPU-executor.

To seamlessly integrate GPUs into Storm, we create a GPU bolt base class, called *BaseGPUBolt*, which inherits Storm's BaseRichBolt and implements its IRichBolt interface. *BaseGPUBolt* acts like a checklist and makes it convenient for a user to build his/her own bolts as it has to be done in Storm. To run an application on GPUs with G-Storm, the user needs to provide the GPU bolt code, a GPU kernel function file and a kernel wrapper file, in addition to those needed for the original Storm. Specifically, a user can create a GPU bolt class by inheriting *BaseGPUBolt* and

implementing its abstract methods for his/her custom application. Note that *BaseGPUBolt* also includes some concrete methods, such as *prepare* and *execute*, which leverages JCUDA for performing some basic operations (such as initialization and kernel launching) for a GPU-executor (See Section 3.3.1).

In addition, for a GPU-executor, a kernel function file and a kernel wrapper file need to be provided by the user to perform custom tuple processing for his/her application. A kernel wrapper runs on the GPU with the user kernel to facilitate the mapping of tuples to CUDA threads. The user kernel only needs to define how one thread should operate on incoming tuple(s), such as resizing an image, or multiplying two matrices. Once tuples have been buffered by the Batcher and copied to the device memory, it is the job of the kernel wrapper to extract input tuple arguments, and output memory space locations associated with each tuple from index information so that the user kernel can be run with the correct number of threads in parallel and each thread knows which data to operate on. Note that the kernel wrapper can be manually written for each application, or it can be automatically generated because all index information is known on the host before data are transferred to the GPU.

G-Storm also allows a user to configure some parameters (such as batching size) of the Batcher (See Section 3.3.2) via its methods, which may make an impact on the performance of an application. For example, a user can specify the batching size (i.e., the max number of tuples to be buffered before being sent to a GPU) by calling the *setMaxTupleCount* method of the Batcher. These parameters should be pre-configured according to application needs.

3.3 GPU-Executor

In this section, we describe the key modules of a GPU-executor at runtime and their implementations in details.

3.3.1 Basic Operations

The BaseGPUBolt class includes two methods: *prepare* and *execute*, which perform basic operations for a GPU-executor. The *prepare* method is called only once when the GPU-executor is initialized, which performs the following necessary operations to prepare the GPU for user kernel execution:

- Initialize the CUDA driver.
- Create a new CUDA device.
- Return a handle to the actual device.
- Create a new CUDA context on the device and associate it with the GPU-executor.
- Locate the user kernel file.
- Compile the kernel file into PTX format and load it.

Moreover, G-Storm offers the user a chance to add his/her additional preparation operations in the *userPrepare* method for flexibility.

The original *execute* method in Storm processes a single tuple at a time. The *execute* method in the BaseGPUBolt class uses the Batcher to buffer multiple incoming tuples and offload them in a batch to a GPU. When input data is resident on the GPU, it launches the kernel wrapper and user kernel for data processing. When kernel execution is complete, the output data are copied back to the host for further processing.

Note that all these operations are performed by the GPU-executor automatically without user involvement. In our implementation, we leverage the JCUDA library for conducting these operations. JCUDA provides methods for device and event management, such as allocating memory on a GPU and copying memory between the host and GPU. More importantly, it contains bindings to CUDA which allows for loading and launching CUDA kernels from Java [14].

3.3.2 Batcher

The *Batcher* is a key module of G-Storm. The batcher is started immediately when the GPU-executor is initialized. The Batcher works along with the Index Builder and Buffer Manager (described next) for critical operations to batch incoming tuples and extracting output tuples. Specifically, the Batcher performs the following operations:

- It calls the Buffer Manager to construct a tuple buffer of sufficient size in the host memory to hold incoming tuples.
- As each tuple arrives, again it uses the Buffer Manager to add the tuple to a pre-allocated tuple buffer.
- If the number of buffered tuples reaches the batching size, it first calls the Index Builder to construct an index buffer (described next) containing the working parameters of each tuple.
- It allocates memory on the GPU to hold the tuple buffer.
- It calls the Buffer Manager to copy the buffered tuples from the host memory to the device memory.
- After data processing on the GPU is complete, it copies the output data from the GPU back to the host.
- Using the index information constructed by the Index Builder, it extracts each individual output tuple from the output data.

We use a single tuple buffer for multiple tuples to reduce the number of data transfers over the slow PCIe bus. The front of the tuple buffer contains input tuples. The next portion is the index buffer containing the index information that is generated by the Index Builder (explained next). Following the index buffer is the output space, which is used to store output data. Note that batching size may greatly affect throughput and latency. Larger size usually leads to higher throughput but longer latency. We discuss this tradeoff in greater details with experimental results in Section 4.

3.3.3 Index Builder

The Index Builder is responsible for constructing indices for tuple arguments, which are their locations in memory. We use a dedicated buffer called the *index buffer* to store the information. The size of the index buffer is calculated based on the batching size and the number of arguments of each tuple. The design of the index is essential to the batching process because the Index Builder needs to provide to the kernel wrapper at runtime the precise indices for arguments of input tuples, and locations of output tuples of varying batching sizes.

The structure of the index buffer is shown in detail in Fig. 4. The first portion of the index buffer stores the total number of threads to be used during user kernel execution. The next

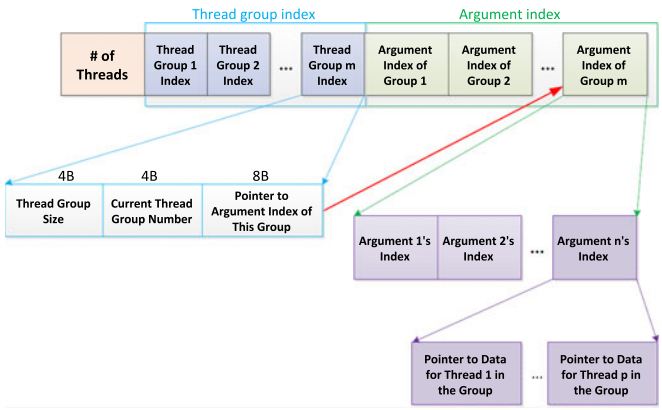


Fig. 4. Index buffer.

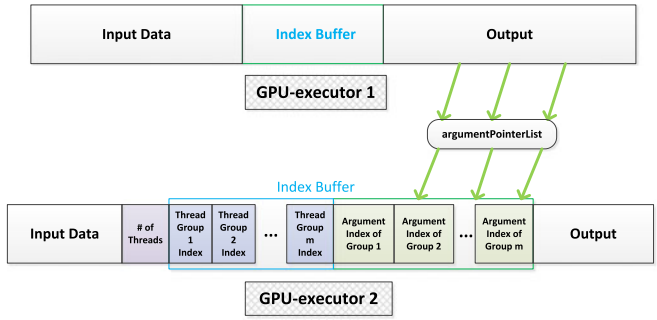


Fig. 5. DDT between co-located executors.

portion is the thread group index. A thread group is a flexible sized group of threads, which can be adjusted for different applications. Because the tuple format in different applications may be different, in some cases one thread may operate on one tuple, in other cases (such as large matrices), one thread may operate on one element (e.g. two numbers in matrices) of the tuple. Thread group sizes should be multiples of a GPU streaming processor's warp size, which is usually 32. It may take multiple thread groups to map a tuple to threads. Within each thread group, we store three values needed to extract the tuple information by the kernel wrapper. First, how many threads are to be used within each group, a thread group may not have all threads in use. For example, if a thread group size is 32, and a tuple has 40 elements, the second thread group only needs to use 8 threads for execution, not all 32. Extra threads in the kernel can return immediately if there is no work for them to do. Second, since a tuple may span multiple thread groups, we need to know which thread group maps to which tuple. In the previous example, there are two thread groups that map to a single tuple and we store their group numbers as 1 and 2. Last, we store a pointer to where argument pointers are stored for each thread group. In this space, pointers to arguments of the tuple are stored contiguously for each thread group so they also can be easily extracted in the kernel wrapper. These pointers point to actual locations where input arguments and output arguments are located in memory. This design leads to a structure that is small in size, while providing all the necessary information to the kernel wrapper for processing input tuples of any batching size.

3.3.4 Buffer Manager

The Buffer Manager is responsible for allocating and initializing buffers to store tuples. As described above, when the Batcher starts, the first thing it does is to call the *initBuffers* method to allocate a single tuple buffer of sufficient size calculated based on the batching size and tuple argument information known *a priori*. The tuple buffer has to be large enough not only to hold input data, but also the expected output data. As each tuple arrives, the Batcher asks the Buffer Manager for the appropriate buffer to store the tuple. Arguments of each tuple are stored directly in the single tuple buffer. A parameter that may be common to all tuples is stored into an additional buffer so we don't have unnecessary waste of memory space. For example, if a tuple is a matrix, and the application is to transform it by adding a value x to each element in the matrix, then x can be stored in the additional buffer. Note that in some cases, we may need to store a common vector of values into the additional buffer. By storing such parameters separately, we do not need to replicate them for each tuple, which can significantly save space.

For output data, we also need to reserve memory space in the tuple buffer. In addition to input tuples and output space, the index buffer also needs to be inserted into the tuple buffer. The Buffer Manager first reserves the space for the index buffer and later when the indices have been calculated by the Index Builder, it is called to insert the index buffer to form the complete tuple buffer. The actual copying of data from a host to a GPU is also handled by the Buffer Manager.

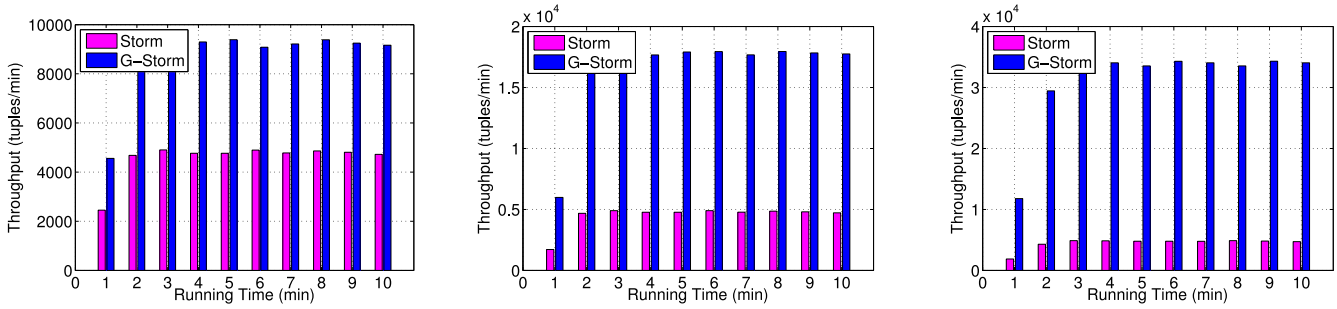
3.3.5 Direct Data Transfer

It is very common to have a chain-like topology with two (or more) consecutive bolts, where the output of one bolt is directly connected to the input of the next bolt. This presents a source of optimization in G-Storm.

In the case where both bolts can be accelerated by a GPU, and both GPU-executors have been assigned to the same GPU worker node at runtime, the ideal solution is to let the GPU-executor of the second bolt directly utilize the intermediate data in the GPU memory. However, this is not possible with the current GPU programming model, because the output of kernel execution is often copied back to the host, therefore, the output of the first bolt is copied back to the host memory, and the following bolt needs to again copy the same data back to the GPU for execution, which is not efficient at all. We design G-Storm with a desirable feature called *Direct Data Transfer* to alleviate this inefficiency.

Using Storm's topology configuration and assignment information, we design G-Storm's *BaseGPUBolt* class with a method called *ifOutTasksSameWorker* to check if the output data of the current executor is assigned to other executor(s) on the same worker node. If so, this executor will bypass the unnecessary memory transfer back to the host, and instead, it will only pass pointers to the output data that's already on the device and its associated tuple index information so that the next executor(s) can directly operate on the data. We illustrate DDT between co-located executors in Fig. 5.

In this figure, GPU-executor 1 uses its Batcher and Index Builder to buffer tuples and construct the tuple buffer as described above. Its Index Builder also builds a list called *argumentPointerList*, storing the index information of output data, which are argument indices of thread groups as



(a) Batching Size = 128 (b) Batching Size = 256 (c) Batching Size = 512

Fig. 6. Throughput on the continuous query topology.

shown in Fig. 4. If GPU-executor 1 detects that GPU-executor 2 is assigned to work on the same GPU, it will alter the output workflow by emitting *argumentPointerList* to GPU-executor 2. Then GPU-executor 2 will copy the index information in the list to its own index buffer as shown in the figure. It will no longer need to batch individual tuples, allocate memory space on the GPU, or copy the data from the host to the GPU. Its Index Builder will use the content of the *argumentPointerList* to locate input data from GPU-executor 1's output space. In this way, data are resident on the GPU and do not need to be copied back and forth.

The NVIDIA MPS feature of the Kepler architecture [17] plays a key role in the implementation DDT. As mentioned above, MPS creates a single CUDA context that can be shared among multiple host processes. With MPS, different GPU-executors can access the same memory space on the GPU, which enables DDT among them.

4 Performance Evaluation

In this section, we first describe the experimental settings, and then present results and the corresponding analysis.

We implemented the proposed G-Storm system based on Storm 0.9.2 release (obtained from Storm's repository on Apache Software Foundation [28]), JCUDA [14] and CUDA 6.5 [7], and installed it on top of Ubuntu Linux 12.04. We performed real experiments with the NVIDIA Quadro K5200 GPU, which supports MPS that is needed for implementing G-Storm's DDT feature.

We evaluated the performance of G-Storm in terms of throughput and tuple completion time. We utilized Storm's default performance reporting system, Storm's UI, to collect many useful performance statistics such as the number of tuples emitted, acked, and failed. The throughput is the number of tuples fully acknowledged within one minute interval. The second metric is the tuple completion time, which is the amount of time G-Storm takes to complete the processing of n tuples. n varies with applications since the number of tuples processed within a certain amount of time in different applications could be quite different.

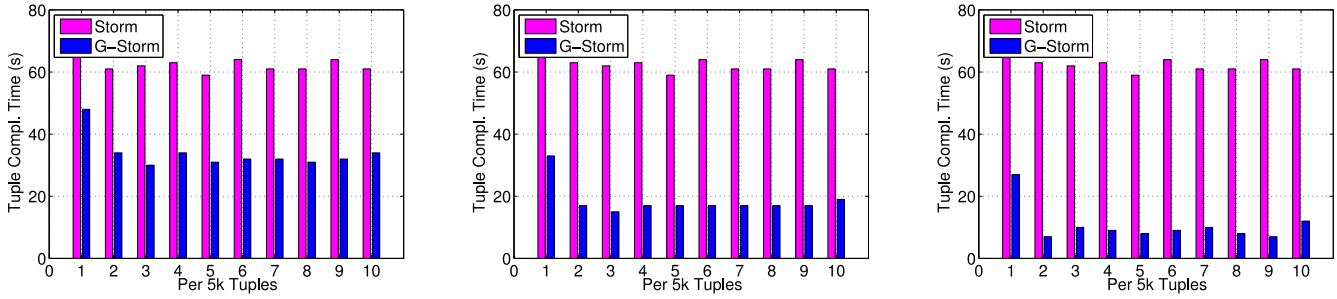
We compared G-Storm with the original Storm that runs only on CPUs by conducting extensive experiments using three typical online data processing applications (topologies), namely, *Continuous Query*, *Matrix Multiplication* (stream version) and *Image Resizing* (stream version). In the following, we describe these applications and how they were run in details, and then present and analyze the corresponding experimental results.

4.1 Continuous Query

A simple select query works by initializing access to a database table and looping over each row to check if there is a hit [4]. To implement this application in Storm, we designed a topology consisting of a spout and two bolts. The spout continuously emits a query, which is randomly generated. The second component is the query bolt, which can be either a CPU or a GPU bolt. The table is placed in memory on the host for Storm or on the GPU for G-Storm. This bolt takes the query from the spout and iterates over the table to see if there is a matching entry. A hit on a matching record is then emitted to the last component, a printer bolt, in the topology. This bolt simply takes the matching record and writes it to file. In our experiment, we randomly generated a database table with vehicle plates, their owners' information (such as names and SSNs), and we randomly generated queries to search the table to find the information of the owners of those speeding vehicles (vehicle speed is randomly generated and included in every tuple).

We performed several experiments on both Storm and G-Storm using this application and presented the corresponding results in Figs. 6 and 7. The running time for each experiment is 10 minutes. In the first experiment, we set G-Storm's batching size to 128. From Fig. 6a, we can see the first minute of execution shows relatively low throughput for both G-Storm and Storm. After the first minute, G-Storm is able to achieve a stable throughout. At the 10th minute, G-Storm achieves a throughput of 9,163 tuples/min, compared to Storm at 4,722 tuples/min, which represents a $1.94\times$ improvement. Over the entire running time, G-Storm consistently outperforms Storm and achieves an average of $1.9\times$ improvement on throughput. Fig. 7a shows the performance of G-Storm and Storm in terms of the completion time of 5 K tuples (i.e., $n = 5$ K). Specifically, the first group of bars show the completion time of processing the first 5 K tuples, the second group of bars show the completion time of processing the second 5 K tuples, and so on. G-Storm offers an average 5 K tuple completion time of 32 s, compared to Storm's 62 s.

In the second experiment, we increased the batching size to 256. At the 10th min, G-Storm has a throughput of 17,750 tuples/min, which gives an increase by a factor of 3.76. From Figs. 6b and 7b, we can see that on average, G-Storm outperforms Storm by 3.7 times in terms of throughput; and



(a) Batching Size = 128 (b) Batching Size = 256 (c) Batching Size = 512

Fig. 7. 5 K tuple completion time on the continuous query topology.

the completion time per 5 K tuples is only 17 s for G-Storm. In the third experiment, as shown in Figs. 6c and 7c we further increased the batching size to 512. We observed that, on average, the batching size increase leads to 7× improvement on throughput over Storm and reduces the 5 K tuple completion time from 17 to 8.9 s.

From these results, we observe that 1) G-Storm consistently outperforms Storm in terms of throughput and tuple completion time. 2) Increasing batching size leads to more significant improvement.

In G-Storm, multiple tuples are processed in a batch by a GPU, which leads to higher throughput but may introduce delay for individual tuples since they have to wait for a little bit before being offloaded to the GPU and moreover, data transfers cause additional latency. To show the trade-off between batching size and latency, we again used the UI to collect tuple processing time of individual tuples, and present the average tuple processing time in Fig. 8. As expected, larger batching size leads to longer average tuple processing time due to longer waiting time in the buffer on the host. However, we notice that the cost of increasing batching size on individual tuple processing time is rather negligible, compared to the significant improvement on throughput shown in Fig. 6. Moreover, we can see that when G-Storm stabilizes, the average tuple processing time is generally below 0.9 s, which is acceptable for online data processing.

4.2 Matrix Multiplication

The second application is a typical numerical computing application, Matrix Multiplication. For this application, we first prepared input files containing matrices. We generated two files each containing around 9 K randomly generated 80×80 double-precision matrices. To feed the matrices into the topology, we used an open-source tool called LogStash [21] that can read data from file and push them into a Redis queue. Two LogStash instances are used to read two files into two Redis queue channels. In the topology, the first component is a *ReadMatrix* spout that continuously extracts data from the queue channels to form a tuple. Each tuple contains two matrices. The spout then emits the tuple to a bolt. Similarly, this bolt can be either a CPU bolt or a GPU bolt, which extracts the two input matrices and perform the multiplication operation on a CPU or a GPU. The last stage of this topology is a printer bolt which writes the results out to a file.

We set the batching size to 16 and used completion time of processing 1 K tuples as the performance metric (i.e., $n = 1$ K) since it deals with much less tuples compared to the first application. As shown in Figs. 9 and 10, we can see G-Storm performs consistently better than Storm. At 180 seconds, G-Storm achieves a throughput of 4,341 tuples/min, compared to Storm at 1,777 tuples/min, which represents a 2.4× improvement. On average, G-Storm offers a 2.3× improvement over Storm on throughput, and it reduces the 1 K tuple completion time from 36 s (given by Storm) to 17 s.

This application demonstrates the robustness of G-Storm in that it can work with different applications by adaptively adjusting the batching size. In this type of applications, the batching size can be set to a relatively small value because G-Storm can map an element of each tuple to a CUDA thread so that it can still fully utilize resources on a GPU even with small number of tuples.

4.3 Image Resizing

The third application is a typical image processing application, which resizes each input image. This application has a non-trivial topology, which is shown in Fig. 11.

For this application, we designed a bolt that uses the *mortennobel* image processing library [22] at its core. To stream images into the topology, we again used a combination of

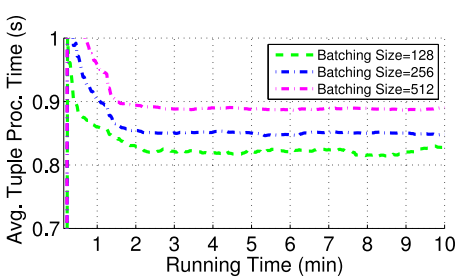


Fig. 8. Average tuple processing time.

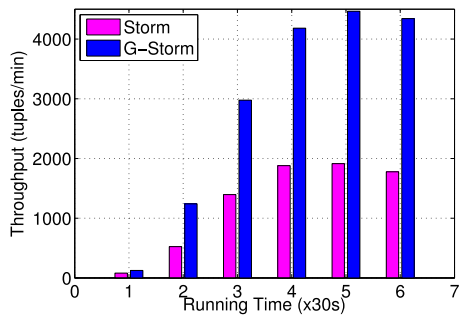


Fig. 9. Throughput on the matrix multiplication application (Batching size = 16).

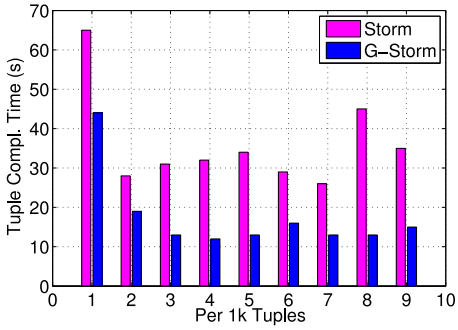


Fig. 10. Tuple completion time on the matrix multiplication application (Batching size = 16).

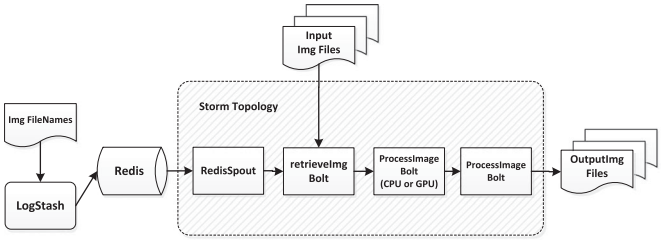


Fig. 11. Topology of the image resizing application.

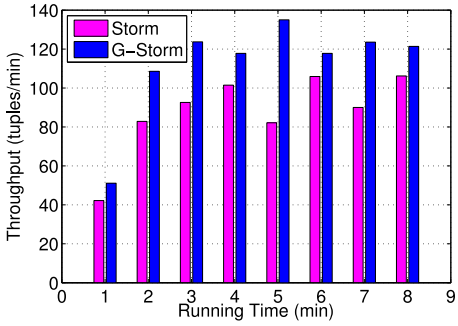


Fig. 12. Throughput on the image resizing topology (Batching size = 16).

LogStash and Redis queue. As shown in Fig. 11, we used LogStash to read image file names into the Redis queue. Inside the topology, the first component is a RedisSpout that listens to the Redis queue and consumes the file names one line at a time for input. The spout then emits each file name as a tuple to the next component. The next component is a RetrieveImage bolt that opens each image file and creates a *BufferedImage* object from it. The image object, along with the file name, forms the next tuple that is emitted to the next component. The next component is either a CPU or a GPU bolt that actually processes images. In the CPU bolt, we directly used the *mortennobel* Java library to perform resizing on each image. For fair comparisons, in the GPU bolt, we implemented the resizing function in the *mortennobel* library with CUDA so that it can be executed on the GPU. The last component of the topology is an output bolt which simply writes the result images back to files. We used completion time of processing 100 tuples as the performance metric (i.e., $n = 100$) since this application involves much less but bigger tuples. We show the corresponding results in Figs. 12 and 13.

In these figures, we observe that G-Storm achieves an average of $1.3\times$ improvement on throughput, compared to

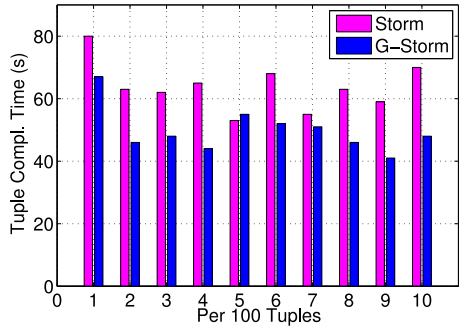


Fig. 13. 100 tuple completion time on the image resizing topology (Batching size = 16).

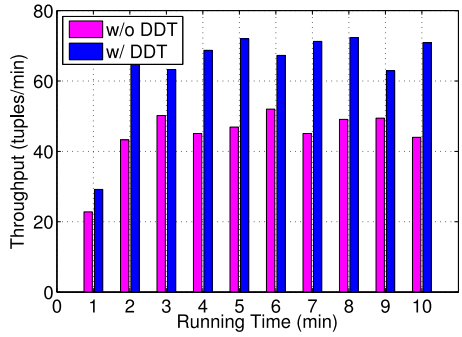


Fig. 14. Throughputs with and without DDT.

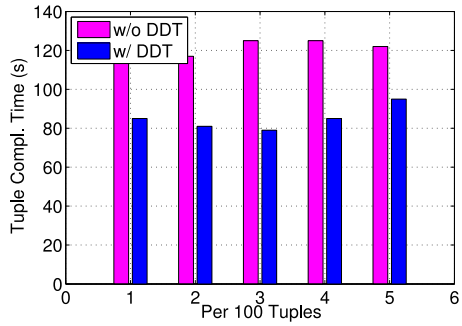


Fig. 15. 100 tuple completion time with and without DDT.

Storm, while reducing the average 100 tuple completion time from 63 s (given by Storm) to 48 s.

4.4 Speedup with DDT

To show the performance gain that can be achieved by the DDT feature of G-Storm, we also used the image resizing application, which involves large amounts of data transfers. The topology is almost the same as in Fig. 11, except that it has an additional ProcessImage bolt (GPU bolt), which simply resizes the input image one more time. We conducted two experiments to observe the speedup. The DDT feature was turned off in the first experiment and turned back on in the second one. In both experiments, we set the batching size to 4. The corresponding results are shown in Figs. 14 and 15.

Fig. 14 shows DDT consistently leads to higher throughput. It achieves an average throughput of 64 tupes/min, compared to G-Storm without DDT at 44 tuples/min, which represents an improvement of 45 percent. Similarly, in Fig. 15, we can see that on average, G-Storm with DDT can process 100 tuples within 85 s, while G-Storm without DDT

needs 122 s. These results justify that DDT can significantly speed up data processing for G-Storm.

5 Related Work

In this section, we provide a comprehensive review for the existing systems and solutions that are related to G-Storm, and identify the literature gap.

Distributed/Parallel Stream Data Processing. As introduced above, Storm [27] is an open-source, distributed, reliable and fault-tolerant system that is designed particularly for online processing of unbounded stream data. In [31], Xu et al. proposed a traffic-aware scheduling algorithm and integrated it into Storm. In [20], Li et al. presented a predictive scheduling framework for speeding up distributed stream data processing and conducted experiments for performance evaluation in Storm. S4 [25] is another general-purpose distributed system that has a similar programming model and runtime. Time-Stream [23] is a distributed system designed by Microsoft specifically for low-latency continuous processing of big streaming data in the cloud. It employs a powerful new abstraction called resilient substitution that caters to the specific needs. MillWheel [2] is a platform for building low-latency data processing applications, which is widely used at Google. In MillWheel, users specify a directed computation graph and application code for individual nodes, and the system manages persistent states and the continuous flows of data, all within the envelope of the framework's fault-tolerance guarantees. Discretized Streams (D-Streams) [33] and closely related Spark Streaming [34] were developed for stream data processing, which, however, have a different processing model. It treats a streaming computation as a series of deterministic batch computations on small time intervals. In addition, Apache Flink [35] is an open-source framework for distributed data processing, supporting both batching and streaming modes. Apache Samza [36] is another distributed stream processing framework.

A few other distributed systems were developed to extend MapReduce/Hadoop to support stream data processing, including Hadoop-based M^3 [1], C-MR [3], Google's FlumeJava [5], WalmartLab's Muppet [19] and a system based on IBM's System S [18].

G-Storm is a general-purpose parallel online stream data processing system just like those proposed in these related works. However, we aim to accelerate data processing with GPUs, which has not be exploited by any of them.

GPU-Accelerated Data Processing. Mars [12] was the first known GPU-accelerated MapReduce system, which includes new user-defined and system APIs and a runtime system (based on NVIDIA GPUs). It needs to spend extra time on counting the size of intermediate results due to lack of dynamic memory allocation on a GPU at the time when the paper was written. Another MapReduce framework, MapCG [13] was developed to improve the performance of Mars, which employs a light-weight memory allocator to enable efficient dynamic memory allocation as well as a hash table for storing intermediate key-value pairs to avoid the shuffling overhead. StreamMR [9] is a similar system, which, however, was developed particularly for AMD GPUs. Efficient methods were introduced in [6], [15] to leverage shared memory on a GPU for further speedup. The authors of [29] introduced the development of a MapReduce-based system on a GPU cluster and presented several optimization mechanisms such as partial reduction and accumulation to reduce the communication overhead between different GPUs. A recent paper [16] presented the MATE-CG system, which is a MapReduce-like framework developed for data-intensive applications in a GPU cluster. The authors introduced a generalized reduction model and developed a novel auto-tuning strategy to determine the best data partitioning parameter values. In [30], the authors presented a processing element aware MapReduce based framework, Pamar, which allows users to utilize different kinds of processing elements (e.g., CPUs or GPUs) collaboratively for data processing. Pamar was implemented based on Hadoop.

All these works focused on MapReduce-based batch processing and MapReduced-specific issues (such as shuffling, data partitioning optimization, etc). G-Storm was built based on Storm for online stream data processing, which has a quite different programming model and architecture.

6 Conclusion

In this paper, we presented design, implementation and evaluation of *G-Storm*, a GPU-enabled parallel platform based on Storm that leverages massively parallel computing power of GPUs for high-throughput online stream data processing. G-Storm was designed to be a general-purpose data processing platform as Storm, which can support various application and data types. G-Storm seamlessly integrates GPUs into Storm while preserving its easy-to-use programming model. Moreover, G-Storm achieves high-throughput and low-overhead data processing with GPUs by buffering reasonable number of tuples and sending them in a batch to a GPU. In addition, G-Storm speeds up data processing further by enabling DDT between two executors processing data at a common GPU. We implemented G-Storm based on Storm 0.9.2 with JCUDA, CUDA 6.5, and the NVIDIA Quadro K5200 GPU. We tested it with three different applications, including continuous query, matrix multiplication and image resizing. It has been shown by extensive experimental results that 1) Compared to Storm, G-Storm achieves over $7\times$ improvement on throughput on continuous query while maintaining reasonable average tuple processing time. It also leads to $2.3\times$ and $1.3\times$ throughput improvements on the other two applications, respectively. 2) DDT significantly reduces data processing time.

Acknowledgments

This research was supported by AFOSR grant FA9550-16-1-0077. The information reported here does not reflect the position or the policy of the federal government. The paper has been Approved for Public Release; Distribution Unlimited: 88ABW-2015-0763, Dated 02 Mar. 15. This is an extended and enhanced version of a paper presented at IEEE BigData 2015.

References

[1] A. M. Aly, et al., "M3: Stream processing on main-memory MapReduce," in *Proc. IEEE 28th Int. Conf. Data Eng.*, 2012, pp. 1253–1256.

[2] T. Akidau, et al., "MillWheel: Fault-tolerant stream processing at internet scale," *Proc. VLDB Endowment*, vol. 6, pp. 1033–1044, 2013.
[3] N. Backman, K. Pattabiraman, R. Fonseca, and U. Cetintemel, "C-MR: Continuously executing MapReduce workflows on multi-core processors," in *Proc. 3rd Int. Workshop MapReduce Appl. Date*, 2012, pp. 1–8.
[4] P. Bakkum and K. Skadron, "Accelerating SQL database operations on a GPU with CUDA," in *Proc. 3rd Workshop General-Purpose Comput. Graph. Process. Units*, 2010, pp. 94–103.
[5] C. Chambers, et al., "FlumeJava: Easy, efficient data-parallel pipelines," in *Proc. 31st ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2010, pp. 363–375.
[6] L. Chen and G. Agrawal, "Optimizing MapReduce for GPUs with effective shared memory usage," in *Proc. 21st Int. Symp. High-Performance Parallel Distrib. Comput.*, 2012, pp. 199–210.
[7] CUDA C programming guide. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/
[8] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. 6th Conf. Symp. Opearting Syst. Des. Implementation*, 2004, pp. 10–10.
[9] M. Elteiry, H. Liny, W. Fengy, and T. Scogland, "StreamMR: An optimized MapReduce framework for AMD GPUs," in *Proc. IEEE 17th Int. Conf. Parallel Distrib. Syst.*, 2011, pp. 364–371.
[10] GeForce Kepler GK100 basic specs leaked. [Online]. Available: http://fudzilla.com/26459-geforce-kepler-gk110-basic-specs-leaked
[11] Apache Hadoop. [Online]. Available: http://hadoop.apache.org/
[12] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: A MapReduce framework with graphics processors," in *Proc. 17th Int. Conf. Parallel Archit. Compilation Techn.*, 2008, pp. 260–269.
[13] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin, "MapCG: Writing parallel program portable between CPU and GPU," in *Proc. 9th Int. Conf. Parallel Archit. Compilation Techn.*, 2010, pp. 217–226.
[14] Java bindings for the CUDA runtime and driver API. [Online]. Available: http://www.jcuda.org/jcuda/JCuda.html
[15] F. Ji and X. Ma, "Using shared memory to accelerate MapReduce on graphics processing units," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2011, pp. 805–816.
[16] W. Jiang and G. Agrawal, "MATE-CG: A MapReduce-like framework for accelerating data-intensive computations on heterogeneous clusters," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2011, pp. 644–655.
[17] NVIDIA's next generation CUDA compute architecture Kepler GK110. [Online]. Available: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf
[18] V. Kumar, H. Andrade, B. Gedik, and K.-L. Wu, "DEDUCE: At the intersection of MapReduce and stream processing," in *Proc. 13th Int. Conf. Extending Database Technol.*, 2010, pp. 657–662.
[19] W. Lam, L. Liu, S. Prasad, A. Rajaraman, Z. Vacheri, and A. Doan, "Muppet: MapReduce-style processing of fast data," *Proc. VLDB Endowment*, vol. 5, pp. 1814–1825, 2012.
[20] T. Li, J. Tang, and J. Xu, "A predictive scheduling framework for fast and distributed stream data processing," in *Proc. IEEE Int. Conf. Big Data*, 2015, pp. 333–338.
[21] Logstash-Open source log management. [Online]. Available: http://logstash.net/
[22] Morten Nobel's Blog. [Online]. Available: http://blog.nobel-joergensen.com/2010/08/21/java-image-scaling-0-8-5-released/
[23] Z. Qian, et al., "TimeStream: Reliable stream computation in the cloud," in *Proc. 8th ACM Eur. Conf. Comput. Syst.*, 2013, pp. 1–14.
[24] Quadro for desktop workstations. [Online]. Available: http://www.nvidia.com/object/quadro-desktop-gpus.html
[25] S4. [Online]. Available: http://incubator.apache.org/s4/
[26] P. Bakkum and K. Skadron, "Accelerating SQL database operations on a GPU with CUDA," in *Proc. 3rd Workshop General-Purpose Comput. Graph. Process. Units*, 2010, pp. 94–103.
[27] Storm. [Online]. Available: http://storm-project.net/
[28] Storm 0.9.2 on Apache software foundation. [Online]. Available: https://storm.apache.org/2014/06/25/storm092-released.html
[29] J. A. Stuart and J. D. Owens, "Multi-GPU MapReduce on GPU clusters," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2011, pp. 1068–1079.
[30] Y. S. Tan, B-S. Lee, B. He, and R. H. Campbell, "A Map-Reduce based framework for heterogeneous processing element cluster environments," in *Proc. 12th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2012, pp. 57–64.
[31] J. Xu, Z. Chen, J. Tang, and S. Su, "T-Storm: Traffic-aware online scheduling in Storm," in *Proc. IEEE 34th Int. Conf. Distrib. Comput. Syst.*, 2014, pp. 535–544.
[32] Y. Yan, M. Grossman, and V. Sarkar, "JCUDA: A programmer-friendly interface for accelerating Java programs with CUDA," in *Proc. 15th Int. Euro-Par Conf. Parallel Process.*, 2009, pp. 887–899.
[33] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proc. 24th ACM Symp. Operating Syst. Principles*, 2013, pp. 423–438.
[34] Spark Streaming. [Online]. Available: http://spark.apache.org/streaming/
[35] Apache Flink. [Online]. Available: https://flink.apache.org/
[36] Apache Samza. [Online]. Available: http://samza.apache.org/
[37] ZooKeeper. [Online]. Available: http://zookeeper.apache.org/

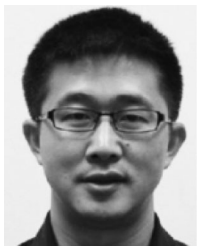
Zhenhua Chen received the bachelor's and master's degrees from Syracuse University. He is working toward the PhD degree in the Department of Electrical Engineering and Computer Science, Syracuse University. His research interests include stream data processing, GPGPU computing, and computer vision. He also has extensive experience in computer systems administration.

Jielong Xu received the bachelor's degree in computer engineering from Sun Yat-Sen University, China, and the master's degree in computer engineering from Syracuse University. He is working toward the PhD degree in the Department of Electrical Engineering and Computer Science, Syracuse University. His research interests include cloud computing, big data, and stream data processing.

Jian Tang received the PhD degree in computer science from Arizona State University, in 2006. He is an associate professor in the Department of Electrical Engineering and Computer Science, Syracuse University. His research interests lie in the areas of cloud computing, big data, and wireless networking. He has published more than 90 papers in premier journals and conferences. He received an NSF CAREER award in 2009, the 2016 Best Vehicular Electronics Paper Award from IEEE Vehicular Technology Society, and Best Paper Awards from the 2014 IEEE International Conference on Communications (ICC) and the 2015 IEEE Global Communications Conference (Globecom) respectively. He has been an editor of the *IEEE Transactions on Wireless Communications* since 2016, the *IEEE Transactions on Vehicular Technology* since 2010, and the *IEEE Internet of Things Journal* since 2013. He served as a TPC co-chair for the 2015 IEEE International Conference on Internet of Things (iThings) and the 2016 International Conference on Computing, Networking and Communications (ICNC). He also served as a TPC member for many international conferences, including IEEE Infocom 2010-2016, ICDCS'2015, ICC 2006-2016, Globecom 2006-2016, etc. He is a senior member of the IEEE.

Kevin A. Kwiat received the BS degree in computer science and the BA degree in mathematics from the Utica College of Syracuse University, and the MS degree in computer engineering and the PhD degree in computer engineering from Syracuse University. He has been with the U.S. Air Force Research Laboratory (AFRL), Rome, New York, for more than 32 years. Currently is assigned to the Cyber Assurance Branch. He holds four patents. In addition to his duties with the Air Force, he is an adjunct professor of computer science with State University of New York, Utica/Rome, an adjunct instructor of computer engineering with Syracuse University, and a research associate professor with the University at Buffalo. He is an advisor for the National Research Council. He has been by recognized by the AFRL Information Directorate with awards for best paper, excellence in technology teaming, and for outstanding individual basic research. His main research interest is dependable computer design.

Charles Alexandre Kamhoua received the BS degree in electronic from the University of Douala (ENSET), Cameroon, in 1999, and the MS degree in telecommunication and networking and the PhD degree in electrical engineering from Florida International University (FIU), in 2008 and 2011, respectively. In 2011, he joined the Cyber Assurance Branch of the U.S. Air Force Research Laboratory (AFRL), Rome, New York, as a National Academies postdoctoral fellow and became a research electronics engineer, in 2012. Prior to joining AFRL, he was an educator for more than 10 years. His current research interests include the application of game theory to cyber security, survivability, cloud computing, hardware Trojan, online social network, wireless communication, and cyber threat information sharing. He has more than 60 technical publications in prestigious journals and International conferences along with a Best Paper Award at the 2013 IEEE FOSINTSI. He has mentored more than 40 young scholars at AFRL counting Summer Faculty Fellow, postdoc, and students. He has been invited to more than 40 keynote and distinguished speeches in the USA and abroad. He has been recognized for his scholarship and leadership with numerous prestigious awards including 30 Air Force Notable Achievement Awards, the 2016 FIU Charles E. Perry Young Alumni Visionary Award, the 2015 AFOSR Windows on the World Visiting Research Fellowship at Oxford University, UK, 3 AFOSR Basic Research Awards of more than $2M, the 2015 Black Engineer of the Year Award (BEYA), the 2015 NSBE Golden Torch AwardPioneer of the Year, selection to the 2015 Heidelberg Laureate Forum, and the 2011 NSF PIRE Award at the Fluminense Federal University, Brazil, to name a few. He is currently an advisor for the National Research Council, a member of the ACM, the FIU alumni association, NSBE, and a senior member of the IEEE.



Chonggang Wang received the PhD degree from Beijing University of Posts and Telecommunications (BUPT), Beijing, China, in 2002. He is currently a member technical staff/senior manager in the Innovation Lab, InterDigital Communications, King of Prussia, Pennsylvania, with a focus on Internet of Things (IoT) R&D activities including technology development and standardization. His current research interest includes IoT, machine-to-machine (M2M) communications, mobile and cloud computing, and big data management and analytics. He was a co-recipient of the National Award for Science and Technology Achievement in Telecommunications in 2004 on IP Quality of Service (QoS) from the China Institute of Communications. He received the InterDigital Innovation Award in 2012 and 2013. He served as a panelist/reviewer for USA US National Science Foundation and Canada NSERC. He was the vice-chair of IEEE ComSoc Multimedia Technical Committee (2012-2014). He is the founding editor-in-chief of the *IEEE Internet of Things Journal* (2014-2016), an advisory board member of The Institute-IEEE (2015-2017), and on the editorial board for several journals including the *IEEE Transactions on Big Data* and the *IEEE Access*. He has been selected as an IEEE ComSoc Distinguished Lecturer (2015-2016). He is a senior member of the IEEE.