G-Storm: GPU-enabled High-throughput Online Data Processing in Storm

Zhenhua Chen, Jielong Xu, Jian Tang, Kevin Kwiat and Charles Kamhoua

***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. We implemented G-Storm based on Storm 0.9.2 and tested it using two different applications: continuous query and matrix multiplication. Extensive experimental results show that compared to Storm, G-Storm achieves over 7x improvement on throughput for continuous query, while maintaining reasonable average tuple processing time. It also leads to 2.3x throughput improvement for the matrix multiplication application.**

I. Introduction

In the past decade, with the rapid rise of mobile and wearable devices, social media, just to name a few, data being created are getting bigger and bigger at an ever increasing pace. It is very challenging to extract valuable and meaningful information from huge-volume, fast and continuous data in a timely manner. MapReduce [5] and Hadoop [8] 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 [17] and S4 [16], 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) [7], 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 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 [9], MapCG [10], etc. However, these solutions cannot be applied to solve our problem since the programming model and system architecture of an online stream data processing system are quite different from those of a MapReduce-based offline batch processing system. Moreover, scant research attention has been paid to leverage GPUs for accelerating online parallel stream data processing. In this paper, we present the 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 executors (i.e., threads), workers (i.e., processes) and worker nodes (i.e., machines) in a cluster. A GPU has quite a different programming model and architecture, which pose 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 the 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 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. 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

Zhenhua Chen, Jielong Xu and Jian Tang are with the Department of Electrical Engineering and Computer Science, Syracuse University, Syracuse, NY, 13244. Email: {zchen03, jxu21, jtang02}@syr.edu. Both Kevin Kwiat and Charles Kamhoua are with the US Air Force Research Lab (AFRL), Rome, NY. The paper has been Approved for Public Release; Distribution Unlimited: 88ABW-2015-0763, Dated 02 Mar. 15. This research was supported in part by NSF grants #1218203 and #1443966. The information reported here does not reflect the position or the policy of the federal government.

978-1-4799-9926-2/15/$31.00 ©2015 IEEE

parallelization, data transfer and resource allocation automatically via a runtime system without user involvement.
3) G-Storm achieves high-throughput and low-overhead da-ta processing by buffering multiple data tuples and offloading them in a batch to GPUs for highly-parallel processing.

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 proposed architecture and methods can be extended to other parallel stream data processing systems, such as Yahoo!'s S4 [16], Google's MillWheel [1] and Microsoft's TimeStream [15], which have similar programming models and system architectures. Due to space limitation, we have to omit background introduction. The necessary background information can be found in the following references: Storm [17], [21], GPU programming [4] and JCUDA [11].

II. Design and Implementation of G-Storm

A. Overview

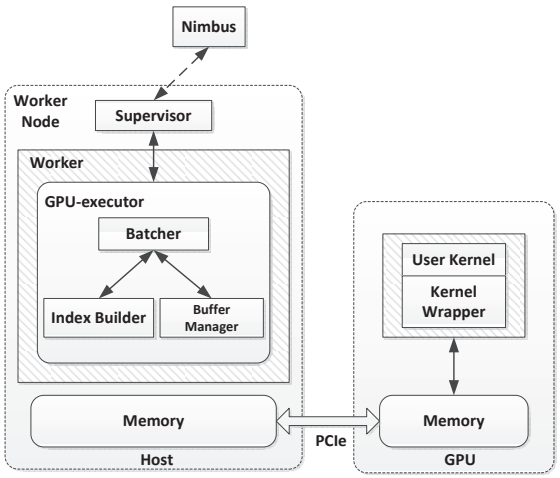


Fig. 1. The architecture of G-Storm

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. 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.

The overall design of G-Storm (Fig. 1) 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. There are 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.

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. In the original Storm, the master node controlling a cluster runs a daemon called *Nimbus*, which is responsible for distributing user code around the cluster, and assigning executors of running topologies to workers and worker nodes. *Supervisor* is a daemon that runs on each worker node that listens for any work assigned to it by Nimbus.

Next, we summarize the workflow of tuple processing in an 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 index information for batching and extracting tuples (See Section II-C3). 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.

B. G-Storm Programming Model

The G-Storm's programming model is very similar to Storm's programming model, where a user builds a Storm application (usually written in Java) 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, G-Storm's model incorporates the GPU programming model as well by using JCUDA [11] as a bridge to the GPU from Storm. 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.

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. Figure 2 shows the software stack of a GPU-executor.

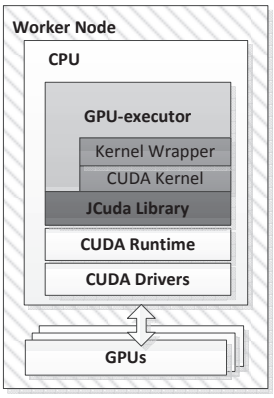


Fig. 2. Software stack of a GPU-executor

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. 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 basic operations (such as initialization and kernel launching) for a GPU-executor (See Section II-C1).

In addition, for a GPU-executor, a kernel and a kernel wrapper 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 II-C2) 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.

C. GPU-executor

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

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 the 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 [11].

2) Batcher: The *Batcher* is a key module of G-Storm. It 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:

- Calls the Buffer Manager to construct a tuple buffer of sufficient size in 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.

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 Figure 3. The first portion of the index buffer stores the total

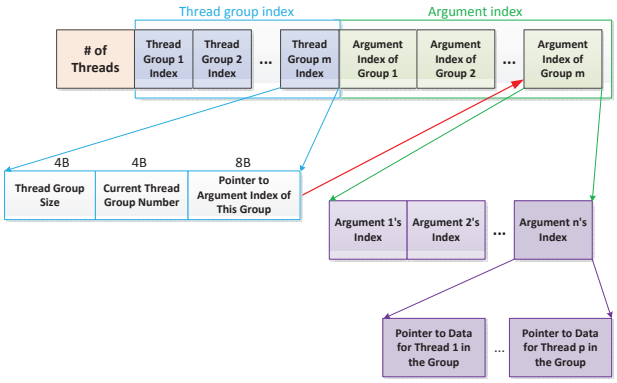


Fig. 3. Index buffer

number of threads to be used during user kernel execution. The next 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. Lastly, 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 tuples of any batching size.

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.

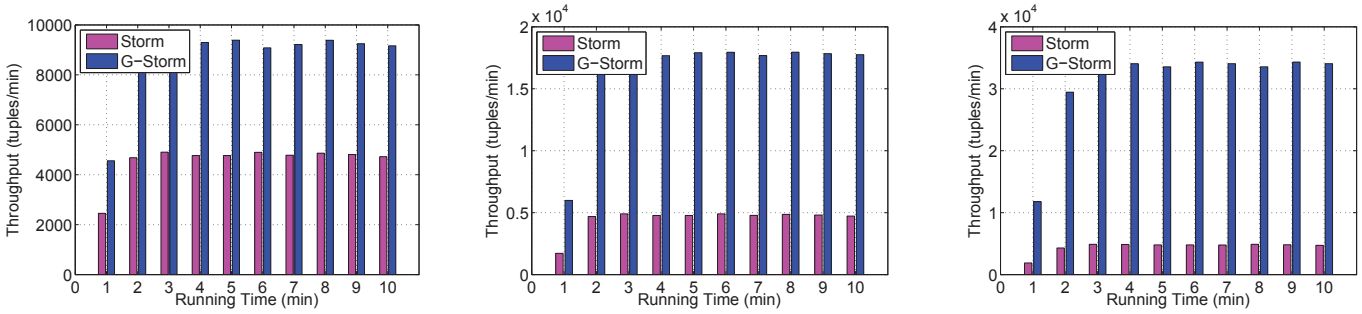
III. Performance Evaluation

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 [18]), JCUDA [11] and CUDA 6.5 [4], and installed it on top of Ubuntu 12.04. We performed real experiments with the NVIDIA Quadro K5200 GPU.

We evaluated the performance of G-Storm in terms of throughput. 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.

We compared G-Storm with the original Storm that runs only on CPUs by conducting extensive experiments using two typical online data processing applications (topologies), namely, *Continuous Query* and *Matrix Multiplication* (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.



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

Fig. 4. Throughput on the continuous query topology

A. 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 [2]. 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 (e.g. names and SSNs), and we randomly generated queries to search the table to find 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 Figure 4. The running time for each experiment is 10 minutes. In the first experiment, we set G-Storm's batching size to 128. From Figure 4(a), 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 1.94x improvement. Over the entire running time, G-Storm consistently outperforms Storm and achieves an average of 1.9x improvement on throughput.

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 Figure 4(b), we can see that on average, G-Storm outperforms Storm by 3.7 times in terms of throughput. In the third experiment, as shown in Figure 4(c) we further increased the batching size to 512. We observed on average the batching size increase leads to 7x improvement on throughput over Storm.

From these results, we observe that 1) G-Storm consistently outperforms Storm in terms of throughput. 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 cause latency 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 Figure 5. 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 Figure 4. Moreover, we can see that when G-Storm stabilizes, the average tuple processing time is generally below 0.9s, which is acceptable for online data processing.

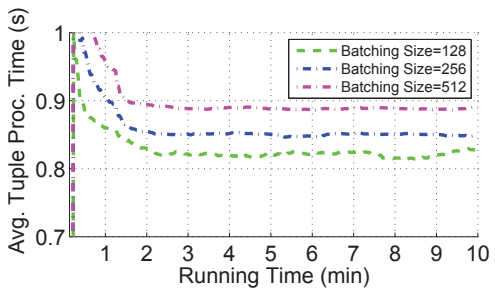


Fig. 5. Average tuple processing time

B. 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 $9K$ randomly generated 80×80 double-precision matrices. To feed the matrices into the topology, we used an open-source tool called LogStash [14] 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. As shown in Figure 6, we can see that 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 2.4x improvement. On average, G-Storm offers a 2.3x improvement over Storm on throughput.

This application demonstrates the robustness of G-Storm in that it can work with different applications by adjusting the batching size. In these applications, the batching size can be 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.

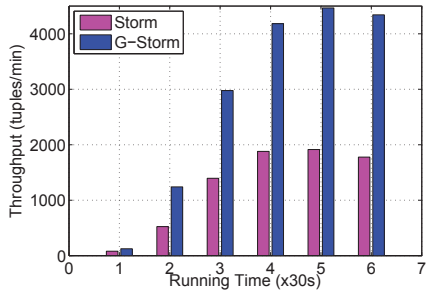


Fig. 6. Throughput of matrix multiplication application (Batching Size = 16)

IV. Related Work

Mars [9] 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 [10] 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 [6] is a similar system, which, however, was developed particularly for AMD GPUs. Efficient methods were introduced in [3], [12] to leverage shared memory on a GPU for further speedup. The authors of [19] introduced the development of a MapReduce-based system on a GPU cluster and presented several optimization mechanisms. Other related research efforts include MATE-CG [13] and Pamar [20].

G-Storm was built based on Storm for general online stream data processing, with the addition of GPU acceleration, which has a quite different programming model and architecture.

V. Conclusions

In this paper, we presented the 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 applications 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. We implemented G-Storm based on Storm 0.9.2 with JCUDA, CUDA 6.5, and the NVIDIA K5200 GPU. We tested it with two different applications, including continuous query and matrix multiplication. It has been shown by extensive experimental results that compared to Storm, G-Storm achieves over 7x improvement on throughput on continuous query while maintaining reasonable average tuple processing time. It also leads to 2.3x throughput improvements on the matrix multiplication application.

References

[1] T. Akidau, A. Balikov, K. Bejiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, S. Whittle, MillWheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 2013, pp. 1033–1044.

[2] P. Bakkum, K. Skadron, Accelerating SQL database operations on a GPU with CUDA. *3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU '10)*. pp. 94–103.

[3] L. Chen and G. Agrawal, Optimizing MapReduce for GPUs with effective shared memory usage, *ACM HPDC'2012*, pp. 199–210.

[4] CUDA C Programming Guide, *http://docs.nvidia.com/cuda/cuda-c-programming-guide/*

[5] J. Dean and S. Ghemawat, MapReduce: simplified data processing on large clusters, *Proceedings of USENIX OSDI'2004*.

[6] M. Elteiry, H. Liny, W. Fengy, and T. Scogland, StreamMR: an optimized MapReduce framework for AMD GPUs, *Proceedings of IEEE ICPADS'2011*, pp. 364–371.

[7] GeForce Kepler GK100 basic specs leaked, *http://fudzilla.com/26459-geforce-kepler-gk110-basic-specs-leaked*

[8] Apache Hadoop, *http://hadoop.apache.org/*

[9] B. He, W. Fang, Q. Luo, N. K. Govindaraju and T. Wang, Mars: A MapReduce framework with graphics processors, *Proceedings of ACM PACT'2008*, pp. 260–269.

[10] C. Hong, D. Chen, W. Chen, W. Zheng and H. Lin, MapCG: writing parallel program portable between CPU and GPU, *Proceedings of ACM PACT'2010*, pp. 217–226.

[11] Java bindings for the CUDA runtime and driver API, *http://www.jcuda.org/jcuda/JCuda.html*

[12] F. Ji and X. Ma, Using shared memory to accelerate MapReduce on graphics processing units, *IEEE IPDPS'2011*, pp. 805–816.

[13] W. Jiang and G. Agrawal, MATE-CG: a MapReduce-like framework for accelerating data-intensive computations on heterogeneous clusters, *Proceesings of IEEE IPDPS'2011*, pp. 644–655.

[14] Logstash - Open Source Log Management, *http://logstash.net/*

[15] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang, Timestream: reliable stream computation in the cloud, *Proceedings of EuroSys'2013*.

[16] S4, *http://incubator.apache.org/s4/*

[17] Storm, *http://storm-project.net/*

[18] Storm 0.9.2 on Apache Software Foundation, *https://storm.apache.org/2014/06/25/storm092-released.html*

[19] J. A. Stuart and J. D. Owens, Multi-GPU MapReduce on GPU clusters, *Proceesings of IEEE IPDPS'2011*, pp. 1068–1079.

[20] Y. S. Tan, B-S. Lee B. He and R. H.Campbell, A Map-Reduce based framework for heterogeneous processing element cluster environments, *Proceedings of IEEE/ACM CCGrid'2012*, pp. 57–64.

[21] J. Xu, Z. Chen, J. Tang, S. Su, T-Storm: Traffic-Aware Online Scheduling in Storm, *Proceedings of IEEE/ICDCS 2014*.