CS294 Big Data System Course Project Report

Gemini: Boosting Spark Performance with GPU Accelerators

Guanhua Wang	Zhiyuan Lin	Ion Stoica
AMPLab	EECS	AMPLab
UC Berkeley	UC Berkeley	UC Berkeley

Abstract

Compared with MapReduce, Apache Spark is more suitable for iterative data processing. The main contribution of Spark is that it significantly reduces MapReduce disk I/O overhead by proposing in-memory data processing and sharing. However, Spark is running on CPUs. Compared with the CPU, a GPU has an order of magnitude higher computational power. To leverage GPU's computation power for Spark, we propose *Gemini*, an Apache Spark runtime system accelerated with GPUs. We have implemented several basic functions (e.g. logistic regression, etc) on GPUs within a cluster on Amazon Web Service EC2. The performance results show that *Gemini* outperforms CPU-based Spark by 5x to 20x speedup on various applications.

1. Introduction

MapReduce [1] is a successful paradigm for large-scale data processing in clusters. However, it does not perform well in iterative data processing like machine learning, deep learning methods. The main shortcoming is that, for each MapReduce cycle, the nodes need to write intermediate results back to the disk [2]. In MapReduce, the only way to share data across jobs is stable storage (e.g. a distributed file system). This high I/O overhead leads to long cluster running time for iterative data processing.

To deal with this high I/O overhead, Apache Spark [3] introduces Resilient Distributed Dataset (RDD) to keep intermediate results in memory for sharing instead of writing back to disk. With in-memory data processing, Spark significantly improves MapReduce performance for iterative data processing.

However, nowadays the performance of CPU has stagnated, graphic processor unit (GPU) achieves an order of magnitude gain in both system performance and price-cost. More precisely, GPU excels at high-parallel and throughput-oriented workloads (e.g. matrix multiplication, convolution, etc.). In addition, since many popular machine learning algorithms are running on graphic processor, processing data on GPU enable Spark to cooperate with machine learning, deep learning frameworks (e.g. Caffe, etc.).

Motived by these good properties introduced by GPU, we propose our project called *Gemini*. Basically, *Gemini* is trying to leverage the GPU inside each node within a cluster to boost up Spark performance. Specifically, *Gemini* is trying to leverage its thousands of cores within each GPU for parallel data processing. In the meanwhile, we regard CPU not only as a scheduler for assigning tasks to GPU, but also a communication agent with other nodes inside the cluster.

Gemini system framework is illustrated as Fig. 1. The workflow of *Gemini* can be

described in the following 3 steps. First, the TaskScheduler of Spark assigns tasks on each node within the cluster. On the CPU side of each node, it will generate function call to GPU. Second, we collect data from DataFrame/RDD and conduct memory allocation on GPU memory. Then we convert data into GPU Array and transfer data to GPU memory. Third, after data being processed on GPU, we transfer data back to RDD/DataFrame and emit FINISH signal to CPU. After that, the CPU can be assigned with new tasks to be processed.

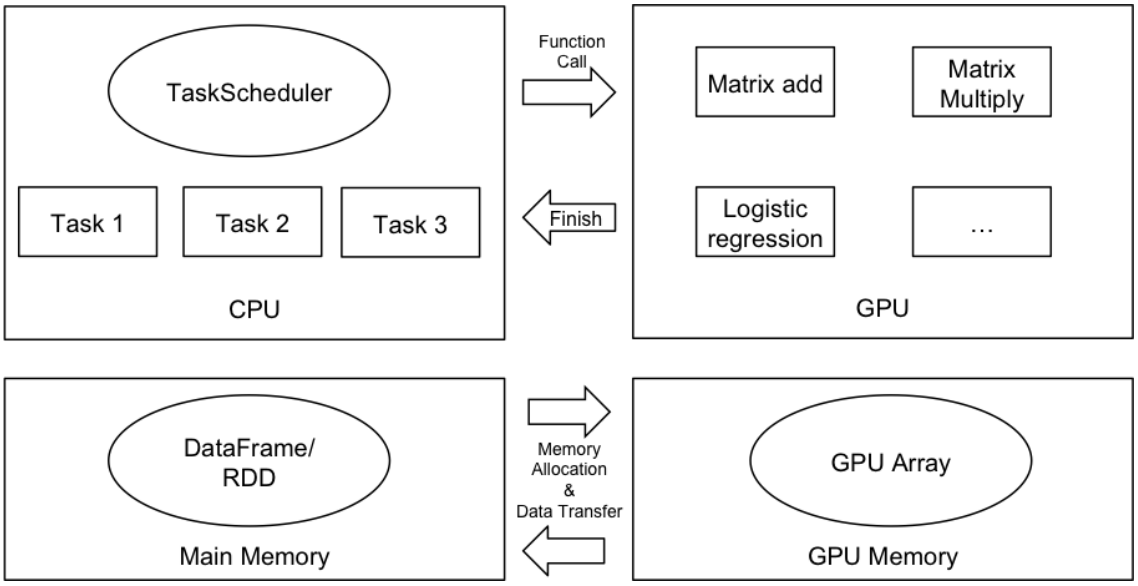

Figure 1: Framework of *Gemini* system.

In short, we made the following 3 main contributions.

- We enable data transfer between Spark's DataFrame/RDD and GPU memory.
- We implement several basic machine learning functions (matrix addition/ multiplication, logistic regression, etc.) on GPU, and integrate them with Spark functionalities, e.g., map and reduce.
- Instead of using single GPU, we enable parallel data processing among multiple GPUs in the cluster.

2. Related work

Previous works using GPU for data processing mainly falls into following two categories.

The first kind falls into single GPU-based data processing systems. Mars [4] is the first scalable GPU-based MapReduce system, through its scalability is limited for only single GPU and in-GPU-core tasks. MapCG [5] enables portable multicore MapReduce code to run on single GPU.

On the other hand, the pieces of art in the second category focuses on building GPU-based clusters for processing larger datasets. GPMR [6] proposes a novel and modified MapReduce system that leverages the power of GPU clusters. Sabne et al. [7] design a more efficient data delivery scheme for GPUs. It pipelines the CPU-GPU memory in the cluster, thus reduces the memory I/O overhead. Some other works like Hadoop+ [8] tries to balance the computing resources in a heterogeneous CPU-GPU cluster environment.

However, most of these stat-of-the-art works focus on designing GPU-based MapReduce systems without leveraging some new characteristics in Spark (e.g. RDD, Lineage, etc). *Gemini* is trying to build-up a GPU accelerator that leverages Spark's in memory data sharing, while boosting the speed of iterative data processing.

3. System Design of Gemini

Gemini is a framework designed for accelerating Spark running speed via harnessing computation power of GPUs within the cluster. As illustrated in Introduction section, there are mainly three components in *Gemini*'s system.

The first part is to enable data transfer between CPU memory and GPU memory. Recently Spark introduces a new data format called DataFrame in project Tungsten [10]. Spark now supports both RDD and DataFrame data formats. Given this, *Gemini* also enables data collection from both RDD and Data Frame.

The second component is adding machine learning functions into *Gemini*'s GPU accelerators. Basically, we implement matrix addition, matrix multiplication, logistic regression, k-means on GPU using CUDA library [11]. These functions are just wrapper functions of the corresponding Spark functions.

The third one is that, instead of using single GPU, we distribute data into multiple GPUs in the cluster. To achieve this, we need to have scheduling and data partition management.

We describe each component in detail in the following 3 subsections separately.

3.1 Data Transfer between CPU & GPU

Data Transfer mainly contains two parts. The first one is data transfer between RDD and GPU memory. Then second one is data transmission between DataFrame and GPU memory.

Transferring data between DataFrame and GPU memory is challenging, because we cannot directly convert elements in DataFrame to Scala array. The main reason for this issue is that, Scala array's type can only be basic data types, e.g., Int/Float, whereas the type of DataFrame wrapper is spark.sql.Row.

The data transfer between DataFrame and GPU includes the following five steps:

1. Collect data from DataFrame to SQL array (dataframe.collect())

2. Use getInt() to get values of elements in DataFrame and write these values to Scala array.

3. On the GPU side, we allocate memory for a GPU array.

4. Once we have a Scala array, we copy its elements to the GPU array.

5. After finishing data processing, we copy back the result to the host memory (CPU memory).

Data transmission between RDD and GPU memory is much easier, because elements in an RDD is compatible with Scala array. Here we do not need to bother collecting data to SQL array and copying data to Scala array.

1. First create/load some arrays (matrices).

2. Then use sc.parallelize() to create an RDD.

3. directly use rdd.map(gpuMemCpy())

4. When data processing complete, we transmit data back to CPU memory.

The function gpuMemCpy() copies elements in arrays/matrices to gpu array. After allocating memory for GPU arrays, we use JCuda functionality to transfer data between scala arrays to GPU memory directly, because the Scala array/matrix can be accessed by JCuda library.

3.2 Wrapper Functions using CUDA Library

In current version of Gemini, we mainly implement four functions using CUDA library on GPU. More precisely, we implement Matrix Add, Matrix Multiplication, Logistic Regression and K-means. Basically, we write CUDA code to implement some basic machine learning functions.

The major contribution we have made is that we modified JCuda library to make it compatible with Spark tasks, which must be serializable. Serialization is the process of converting an object into a stream of bytes in order to store the object or transmit it to memory, a database, or a file. In Java (Scala), serializability of a class is enabled by the class implementing the java.io.Serializable interface.

Considering Spark will complain “task not serializable” if we use standard JCuda library. We modify the JCuda source code with implementing java.io.Serializable in some basic classes, including JCuda.Pointer. This modification enables running CUDA functionalities within Spark tasks.

3.3 Extend from Single GPU to multiple GPUs

For distributed data processing on multiple GPUs, we still leverage Spark’s DAGScheduler and TaskScheduler to achieve data partition, processing, and sharing.

The whole workflow of Gemini’s distributed cluster is illustrated as follows. First, after job has been submitted to Spark Cluster, the DAGScheduler buildup processing stages and then using TaskScheduler to generate tasks of current stage. Second, after each node gets the task assigned to itself, on the CPU side, it will generate a function call to GPU. Third, after GPU receives the function call, it enables data copy from RDD/DataFrame to GPU memory. Fourth, after GPU finished data processing, it will write back the processed data to CPU memory, and then send back a FINISH signal to CPU side. After that, this node can ask Scheduler to assign new tasks to it.

4. Implementation

This section mainly describes the implementation details about *Gemini*, including hardware platform, experiment datasets.

4.1 Hardware Platform

We run our benchmark evaluation on Amazon Web Service using EC2 instance. More precisely, we use g2.2xlarge instance to build a 5 nodes *Gemini* cluster (with 4 slave nodes and 1 master node). Each g2.2xlarge node is backed by a NVIDIA GRID GPU (Kepler GK104).

For Spark, we use m3.xlarge instance to implement a 5 nodes cluster (with 4 slave nodes, 1 master node). Each m3.xlarge instance is equipped with an Intel Xeon E5-2670 (Sandy Bridge) Processor running at 2.6 GHz (4 cores) and 15 GB main memory, with 2*40 GB SSD.

4.2 Experimental Data Sets

For testing data transfer overhead, we randomly generate several float matrices with different size (1000*1000, 2000*2000, 3000*3000, 4000*4000, 5000*5000, 8000*8000) and then distribute matrices among nodes within a cluster.

For basic functions like matrix multiplication, addition, the data is also from the randomly generated matrices mentioned above.

One thing needs to be mentioned is that, currently all the data we use are a bit smaller than the GPU memory size. It is because when the data set is larger than GPU memory size, the Gemini system will crush with Out-Of-Memory problem. It is possible to achieve data set as large as host memory size by implementing data pipeline between GPU and CPU. And we remain it as one of our future work.

5. Performance Evaluation

In this section, we mainly discuss about *Gemini*'s performance on several differnt functions (e.g. logistic regression, matrix multiplication, etc.). We compare execution time of *Gemini* with Spark's original CPU-based cluster.

5.1 Data Transfer Overhead

First, we evaluate the data transfer overhead between Spark's RDD and GPU Memory. As shown in Fig. 2, we test the data transfer time cost with varied data size ranging from 16 MB upto 1.1 GB. The corresponding data transferring time for several 1000*1000 matrix (roughly 16MB) is roughly 800ms. And with data size around 64 MB, the corresponding data transfer time is less than 1.7s. When the data size scales from 256 MB up to 1.1 GB, the data transferring time is not negligible. With 400MB to 1.1 GB, the corresponding data transfer time is 8.4s and 14.3s respectively.

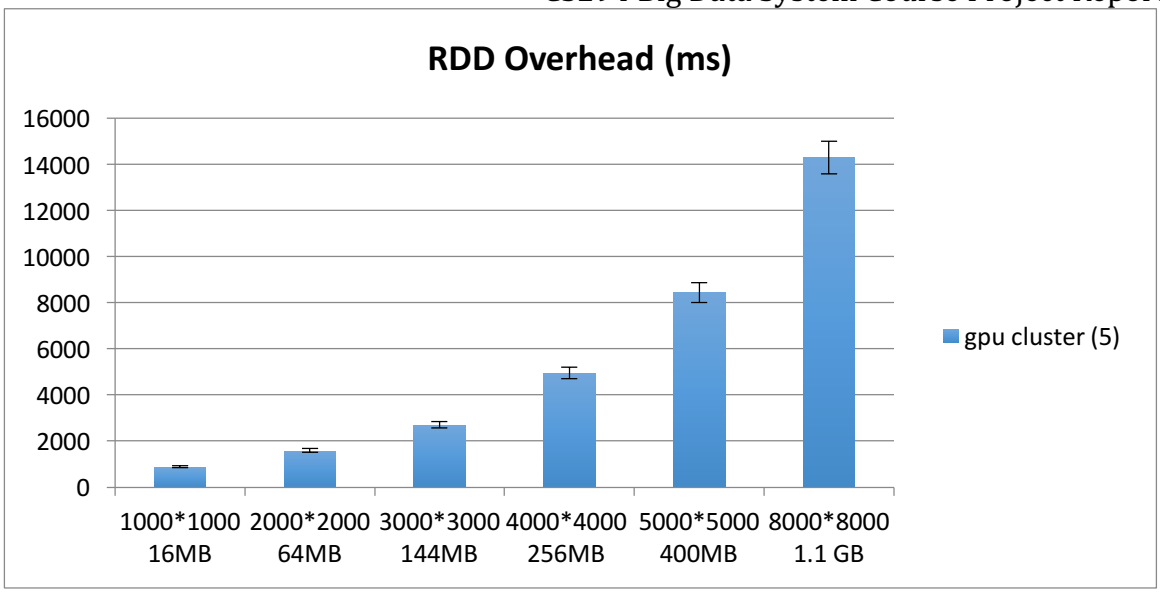

Figure 2: Data transfer overhead between RDD and GPU memory.

Next we evaluate the data transfer overhead between DataFrame and GPU memory. Compared with RDD & GPU memory communication, the time cost data transfer between DataFrame and GPU memory is much higher. The detailed execution time for data transfer between DataFrame and GPU is depicted in Fig. 3. As shown in Fig. 3, for DataFrame data collection, 16 MB data transmission among nodes in the cluster takes 8s, which is roughly the same amount of time for 400 MB data transfer between the RDD & GPU. And this data transfer overhead grows linearly when the data size increases. When collecting data with size of 400 MB and 1.1 GB, the corresponding data transfer time is 219s and 401s. This data collection overhead is very high. And this overhead may sometimes even cancel out the GPU processing speedup gains. Overall, compared with data collection between RDD & GPU, data transfer between DataFrame & GPU memory is roughly 10x to 20x slower. Finding a smarter way to transfer data between DataFrame and GPUs could be part of future directions of this project.

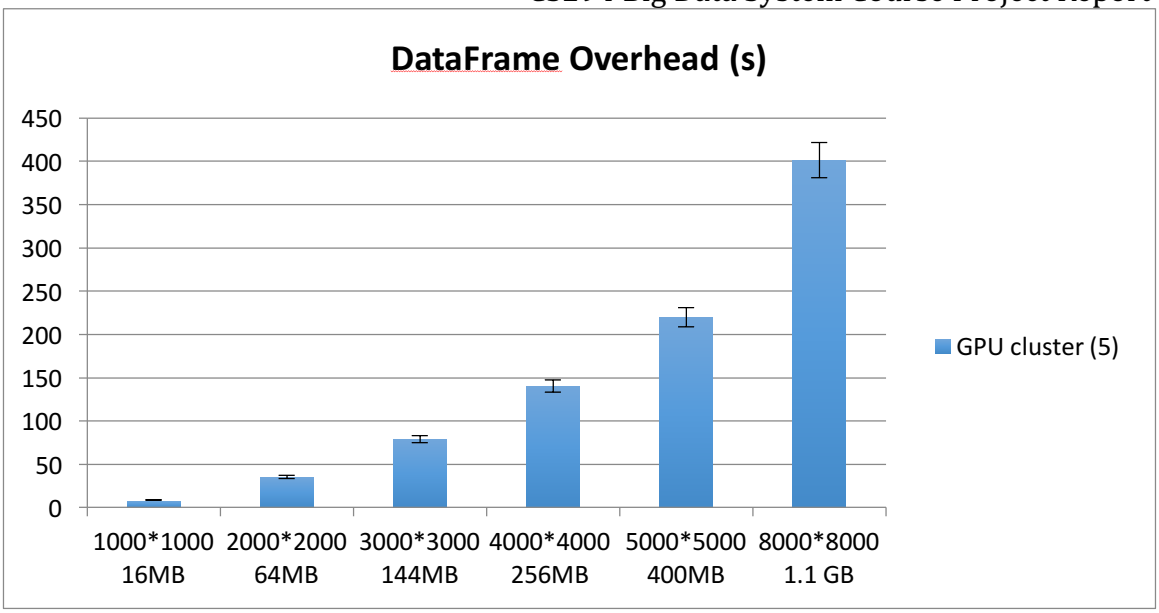

Figure 3: Data transfer overhead between DataFrame & GPU memory.

Based on our analysis, the most time consuming part is DataFrame.collect() processing. It is the only API that we can use to collect data from DataFrame. In Section 6, we have propose one potential optimization for reducing the data transfer time between DataFrame and GPU memory.

5.2 Benchmarks for Basic Machine Learning Functions

In this section, we mainly test four functions we implemented for Gemini with CUDA library on GPU. The four functions are Matrix Add, Matrix Multiplication, Logistic Regression and K-means. We compare the execution time of original CPU-based Spark and our GPU-base Gemini system.

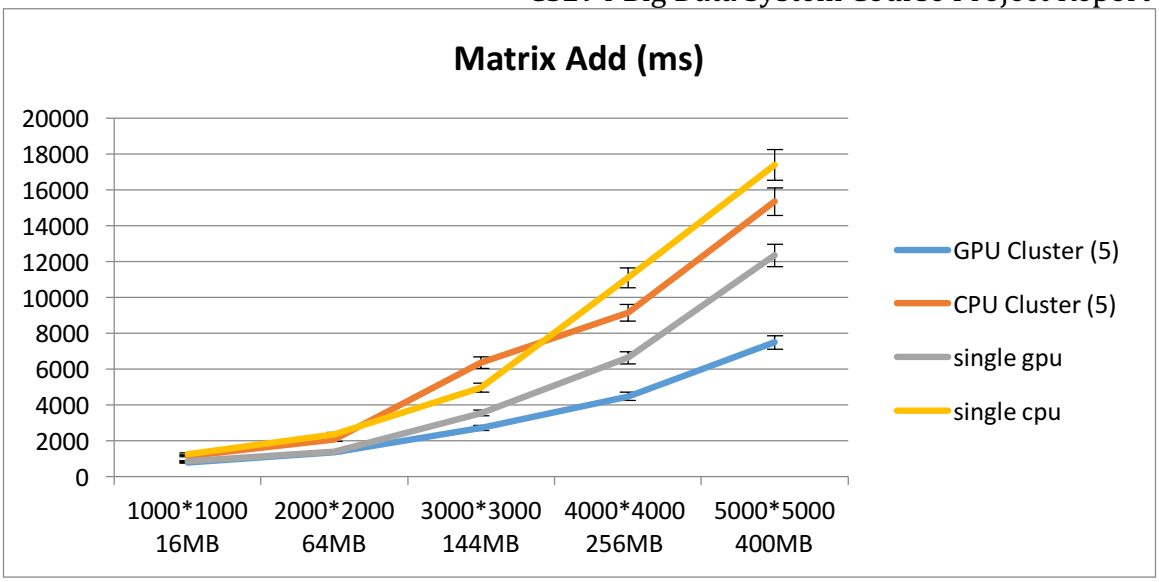

Figure 4: Benchmark of matrix add with varied data size.

We first measure the execution time of Matrix Add on Spark and Gemini. And we test it with single slave (i.e. single cpu/gpu) and multiple slaves (i.e. GPU/CPU Cluster) scenarios. As mentioned before, the cluster has 4 slave nodes and one master node, whereas single slave scenario only has 1 slave node and 1 master node.

As shown in Fig. 4, we run 20 times of matrix add in each time execution, and we also include the data transfer time between RDD and GPU. The data processing time is shown in Fig. 4. It is noted that, considering about data transfer time mentioned in Fig.2, the real computation time (i.e. matrix add) cost little in Gemini's running time. If we remove the memory copy time, the real computation time for Gemini's GPU nodes is little. For example, a single GPU's real computation (i.e. matrix add) time is ranging from 200ms to 3s with varied data size, which is only 1/4 to1/3 overall execution time. However, if we run more complicated algorithms like logistic regression, the real computation time will be more than half of the overall execution time.

Since the add function is simple, there is only 0.5x to 1x speedup compared with CPU-based counterpart. We can see more performance gain in logistic regression and k-means functions.

Another point worth mentioning is that, in Fig.4, for data size of 144MB, CPU cluster runs longer than single CPU. It is because that networking communication overhead is larger than the parallel computation acceleration using cluster.

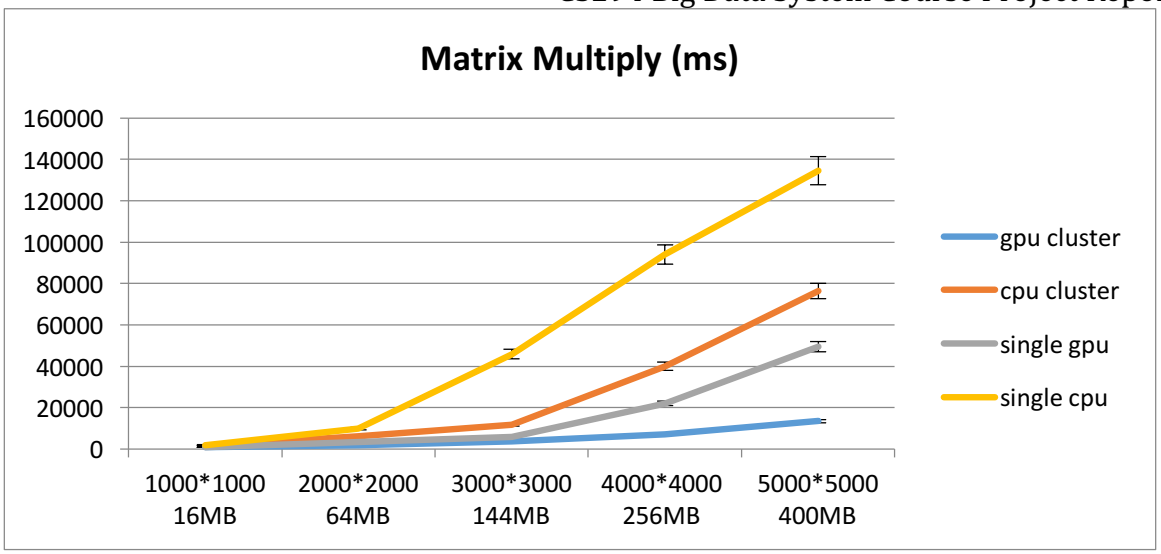

Figure 5: Benchmark of matrix multiplication with varied data size.

Here we evaluate the basic function, Matrix Multiplication both Gemini and Spark. We also run 20 times of matrix multiplication on all four scenarios. The GPU-based execution time also includes memory copy time. As shown in Fig.5, with more complicated functions like multiplication, Gemini achieves roughly 5x to 10x speedup compared with CPU-based Spark in both single slave and multiple slaves scenarios.

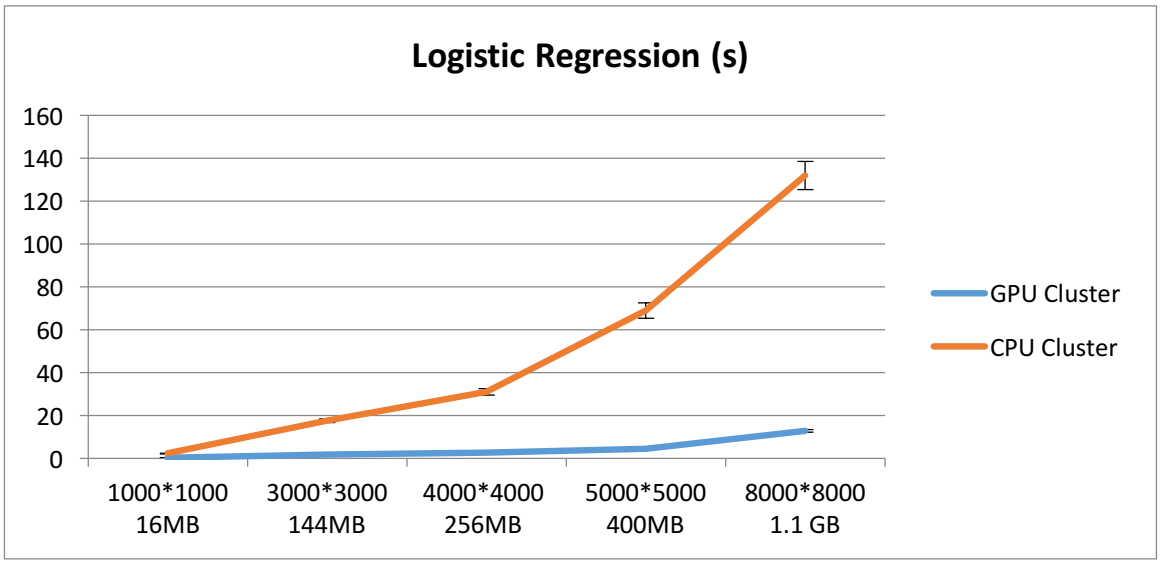

Figure 6: Benchmark of logistic regression with varied data size.

On the first part of evaluation, we test basic functions like matrix add and multiplication. We can figure out that, running more complicated functions on Gemini will have more speedup gain. As mentioned above, add function only has 0.5x to 1x gain,

whereas multiplication has 5x to 10x speedup gain. Then we evaluate Gemini performance via running two simple machine learning functions, namely, Logistic Regression and K-means.

As shown in Fig. 6, there is a significant speedup of *Gemini* (i.e. GPU Cluster) execution time. On average, the acceleration gain of Gemini compared with CPU-based Spark is roughly 10x to 20x on average. This performance result also verifies the argument that, in order to mitigate the memory copy overhead, we need to run more complicated functions on GPUs to achieve high speedup gain.

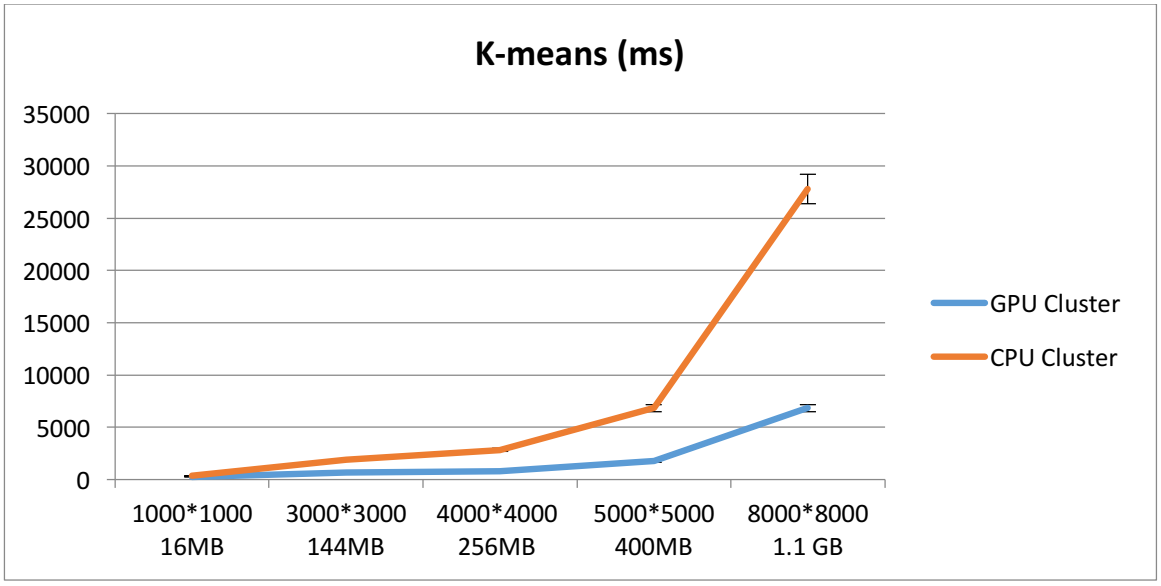

Figure 7: Benchmark of K-means implementation with varied data size.

Another function we test is K-Means algorithm. As shown in Fig.7, Gemini (i.e. GPU-Cluster) still outperform Spark (CPU-Cluster) on varied data size. Overall, Gemini has 3x to 5x speedup compared with Spark. Therefore, we may conclude that, Gemini's acceleration gain depends on different applications.

6. On-going Optimizations

Based on our evaluation results, the single thread data copy from host memory (CPU memory) to device memory (GPU memory) contributes to more than half of the execution time. Especially in simple functions like Matrix Add, Matrix Multiplication, this memory copy dominates *Gemini*'s execution time. The essential problem here is, currently we use single thread to handle partitions one by one. As shown in Fig. 8, the system has to wait until everything in the previous partition finished. And then begin the next round partition execution. It is a huge waste of time.

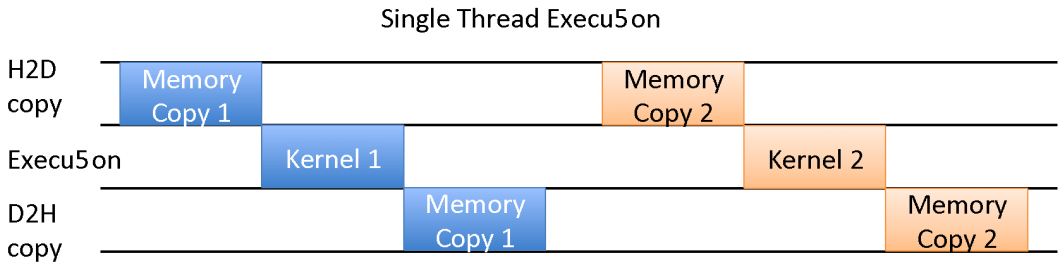

Figure 8: Single Thread data processing on GPU

Instead of waiting for the previous partition to finish, we can pipeline the data processing on GPU. Specifically, we can create a thread pool and use **cuda stream** to hide the cost of CudaMemCpy from CPU to GPU. As shown in Fig. 9, each thread is assigned with a unique CudaSteam. In this case, multiple partitions of data in multiple threads can be executed in pipeline. And this pipeline execution will significantly reduce the MemCopy latency.

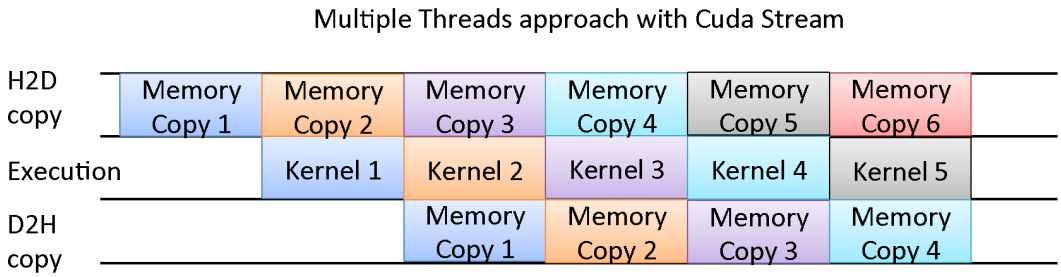

Figure 9: Thread Pool Approach with Cuda Stream Pipeline

In future work, we will try to enable multiple thread allocation on each node first. The main challenge here is to enable coordinate among different thread on one node. There are some issues like data dependency, thread management, etc. By implementing thread pool scheme, we believe it can significantly reduce *Gemini*'s data transfer overhead.

7. Conclusion

This project proposes *Gemini*, a Spark-based, runtime system using GPUs for acceleration. We implement mainly three components. First, we enable data communication between RDD/DataFrame and GPU memory. Second, we rewrite some basic functions (e.g. logistic regression) on GPU using CUDA/JCUDA library. Third, we distribute data processing with multiple GPUs within the cluster. Based on our evaluation results, compared with CPU-based Spark, Gemini can achieve 5x to 20x speedup, depending on different applications. We believe designing multi-threads

pipeline processing using CudaStream will further boost up *Gemini*'s performance.

Reference

[1] Jeffrey Dean, Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", in Proc. of USENIX OSDI 2004.
[2] Apache Hadoop, https://hadoop.apache.org/
[3] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica, "Resilient Distributed Datasets: a fault-tolerant abstraction for in-memory cluster computing", in Proc. of USENIX NSDI'12.
[4] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang, "Mars: a MapReduce framework on graphics processors", in Proc. of ACM PACT 2008.
[5] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng, and Haibo Lin, "MapCG: writing parallel program portable between CPU and GPU", In Proc. Of ACM PACT 2010.
[6] Jeff Stuart, John Owens, "Multi-GPU MapReduce on GPU Clusters", In Proc. of IEEE IPDPS 2011.
[7] Amit Sabne, Putt Sakdhnagool, Rudolf Eigenmann, "Scaling Large-Data Computations on Multi-GPU Accelerators", In Proc. of ACM ICS 2013.
[8] Wenting He, Huimin Cui, Binbin Lu, Jiacheng Zhao, Shengmei Li, Gong Ruan, Jingling Xue, Xiaobing Feng, Wensen Yang, and Youliang Yan, "Hadoop+: Modeling and Evaluating the Heterogeneity for MapReduce Applications in Heterogeneous Clusters", In Proc. of ACM ICS 2015.
[9] Caffe Deep Learning Framework, http://caffe.berkeleyvision.org/
[10] Project Tungsten, https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html
[11] NVIDIA CUDA Parallel Computing Platform and programming model, http://www.nvidia.com/object/cuda_home_new.html