Twitter Language Specifier on Spark with GPU

James Jia, Qijing Huang, and Qifan Pu
Department of Electrical Engineering and Computer Science
University of California, Berkeley
{jamesjia, qijing.huang, qifan}@berkeley.edu

***Abstract*—As GPU-backed algorithms continue to dominate competitions in a variety of fields including machine learning, bioinformatics, and computer graphics, there has been much interest in the development of a computing platform that supports GPU-accelerated distributed systems. Apache Spark, although traditionally considered as a CPU-only distributed framework, has garnered interest within the academic community as an area to which GPU acceleration can be applied. However, the work regarding GPU acceleration on Spark has been concentrated on vanilla Spark, where the data is both received in large batches and is also outputted in large batches. However, there are many contexts in which data must be processed in real time or, at the very least, within a small time period. With this in consideration, we developed a prototype for GPU acceleration on Spark Streaming. We implemented a GPU-accelerated variant of the Twitter Language Classifier as a proof of concept, applying GPU acceleration using JCuda bindings [1] to both offline model training as well as real-time prediction. We also present our findings and analysis of its performance compared to the Spark CPU K-Means implementation for training and prediction, and pinpoint specific areas of interest for future improvement.**

***Index Terms*—Spark Streaming, GPU, K-Means, JCuda.**

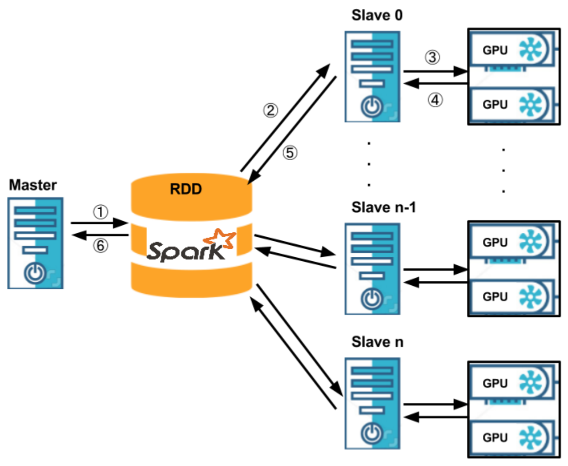


Fig. 1: Overall Architecture

I. Introduction

As the deployment of IoT devices continues to progress at a rapid pace, the ability to process real-time data and produce near-immediate results is becoming increasingly important. In many applications, the streams of data gathered are largely independent of one another, and thus the work can be efficiently parallelized across many nodes in a distributed system. With this in mind, many developers use Spark Streaming as their platform of choice to process numerous streams of data from these IoT devices with latencies as low as a second. Unlike traditional streaming databases, such as Aurora [2] and Borealis [3], which compile the streaming jobs into a collection of operators to run on separate machines, Spark Streaming uses a discretized streams (D-Streams) model [4]. In this model, records are chunked into small batches based on time intervals (from hundreds of milliseconds to seconds) and processing of each batch is transformed into a data-parallel job. Advantages of this model include:

- Batch and streaming applications can be written using a same set of APIs, which also means a streaming application can be written with advanced batch processing library, e.g., machine learning libraries (as we use latter in our project).
- The streaming system can reuse the fault tolerance mechanism of the underlying computing framework, e.g., Spark's lineage tracking.
- Batching of incoming stream greatly improve system throughput, while in-memory computation provide low latency of sub-seconds.

However, recent study on data analytics framework performance [5] suggests that Spark bottlenecks heavily on CPU. The discovery exposes a potential opportunity to use GPUs, which is considered as an ultimate resort to turn "any CPU-bound problem into a I/O bound problem". In addition, GPU accelerators are also suitable to exploit the inherently embarrassingly parallel nature of many streaming workloads.

A. *Twitter Language Classifier*

The Twitter Language Classifier is an application that runs on top of the Spark Streaming framework. It collects a data set from Twitter's global stream of tweets, trains an offline K-Means model using that data, and then uses that model for predicting the language of a new stream of live tweets (Figure 2 shows the distribution of tweets' languages). There are two opportunities to apply GPU acceleration: in the offline model training as well as in real-time prediction. In our investigation, we chose to keep the number of features constant; we used Spark MLLib's TF-IDF implementation with a feature size of 1000.

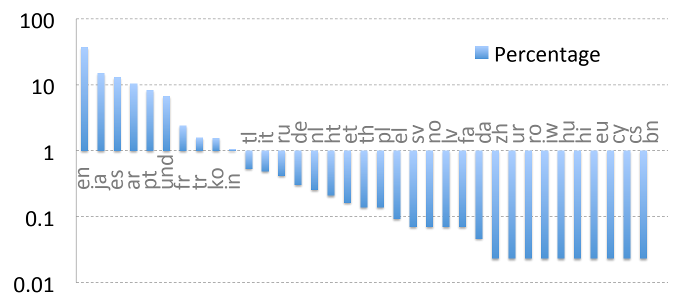


Fig. 2: Language distribution of 8000 tweets.

II. Systems Overview

A. Overall Architecture

Figure 1 shows an overview of the Spark architecture with GPUs. In our work, we configure our jobs to run in Spark client mode where the driver is launched in the master node that submits the application. A typical data flow within the architecture is described as follows: ① Initially, the master node creates the SparkContext and a new Resilient Distributed Dataset (RDD) for the target inputs. ② The driver maps the user-specified tasks to the slave nodes to run on different RDD partitions. ③ The slave node transfers data to GPU and launches the GPU program. ④ The GPU sends results back to the slave node. ⑤ The slave node returns output results to new RDD ⑥ The master node collects the results from the RDD.

B. Communication between CPU and GPU

We ran our experiments on the Amazon EC2 g2.2xlarge instances with Intel Xeon E5-2670 (Sandy Bridge) CPUs and NVIDIA GK104GL [GRID K520] GPUs. Each GPU is with 1,536 CUDA cores and 4GB of video memory. The CPUs and GPUs are connected through standard PCIe interface with gen1x16 configuration, which provides us up to 4GB/s bandwidth.

In order to run programs on GPU from Spark, we use an existing Java library for CUDA called JCuda [1] to bind CUDA driver and runtime APIs with our Spark code written in Scala.

III. Implementation Details

A. The K-Means Clustering Algorithm

Algorithm 1 describes the K-Means training and prediction algorithms we use for accelerating the Twitter Language Specifier Application. Lines 5, 6 of the TRAIN procedure and line 15 of the PREDICT procedure are ported to run on GPU kernels.

B. Training

The code flow for the training process is shown in c. After we gather the raw tweets from the online Twitter API. We use the featurizer from Spark machine learning library *mllib* to generate the feature vectors in the sparse matrix form. Since our Kmeans GPU code takes feature vectors in the dense matrix form, we wrote a formatter to transform the input in sparse matrix to a dense matrix.

Algorithm 1 K-Means Clustering Algorithm

```
1:  procedure TRAIN
2:      Input: Array of objects $o_1, o_2, ..., o_n$
3:      Output: Array of cluster centers $m_1, m_2, ..., m_k$
4:      Initialize the cluster centers $m_1, m_2, ..., m_k$ with the
    first k number of objects $o_1, o_2, ..., o_k$
5:      while $\frac{\#ofchangingobjects}{\#oftotalobjects} < \delta$ or $i < loopcount$ do
6:          for each object $o_i \in o_1, o_2, ..., o_n$ do
7:              mem($i$) = $find\_nearest\_cluster(o_i)$
8:              $compute\_delta(o_i)$
9:          end for
10:         for i from 0 to k do
11:             Replace $m_i$ with the mean of all of the samples
    for cluster i
12:         end for
13:     end while
14: end procedure
15: procedure PREDICT
16:     Input: Array of objects $o_1, o_2, ..., o_n$
17:     Input: Array of cluster centers $m_1, m_2, ..., m_k$
18:     Output: Array of objects' membership $mem$
19:     for each object $o_i \in o_1, o_2, ..., o_n$ do
20:         mem($i$) = $find\_nearest\_cluster(o_i)$
21:     end for
22: end procedure
```

Algorithm 2 $find\_nearest\_cluster()$

```
1:  Input: Object $o$
2:  Input: Array of cluster centers $m_1, m_2, ..., m_k$
3:  Output: Membership of object $mem$
4:  Initialize $\Delta \rightarrow \infty$
5:  for each cluster center $m_i \in m_1, m_2, ..., m_k$ do
6:      tmp = euclidean distance between $o$ and $m_1$
7:      if tmp < $\Delta$ then
8:          $\Delta$ = tmp
9:          $mem$ = i
10:     end if
11: end for
```

The dense matrix representation of features is then fed into our actual training code circled by the dotted line in Figure 3. The modules highlighted in yellow are the Spark code we wrote in Scala. We first create an RDD of Kmeans input objects, then call *mapPartitions()* to launch Spark workers to run parallel tasks on the RDD partitions. The mapped GPU host code on the Spark workers is written with the JCuda library. The input to the GPU kernels includes the input objects, cluster centers and the objects' membership. The modules highlighted in red are the Kmeans kernel code written in CUDA. Since our work emphasizes on the Spark framework with GPU, we directly sourced a high-performance Kmeans

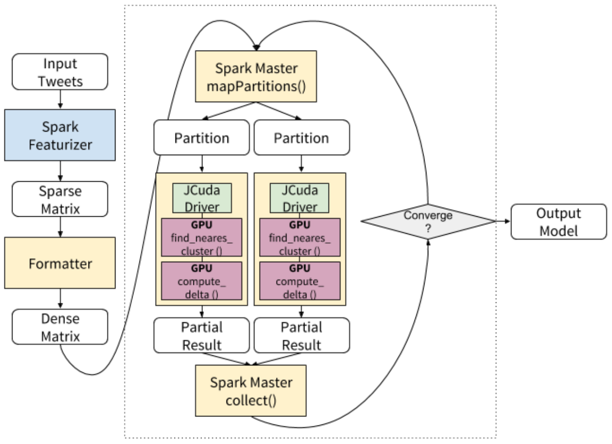


Fig. 3: Flowchart for Training

CUDA implementation online [6]. A *collect()* operation is performed at the end of each iteration to send the results to the Spark driver node. The new cluster centers are calculated on the driver and get redistributed to the workers in the next iteration. The loop will terminate once the change to the objects' membership is below a user specified threshold or the loop bound is reached.

C. Prediction

For each RDD in the minibatch of tweets from the discretized stream, we aggregate the data within each partition into a N by 1000 matrix, where N is the number of tweets in that partition and 1000 is the number of features per tweet. We batch tweets on a per-partition basis in order to amortize the overhead of instantiating the GPU context across each tweet. We send this data matrix along with the serialized model matrix to the GPU kernel, which will then compute the nearest cluster for each tweet, outputting the predicted result into a new RDD.

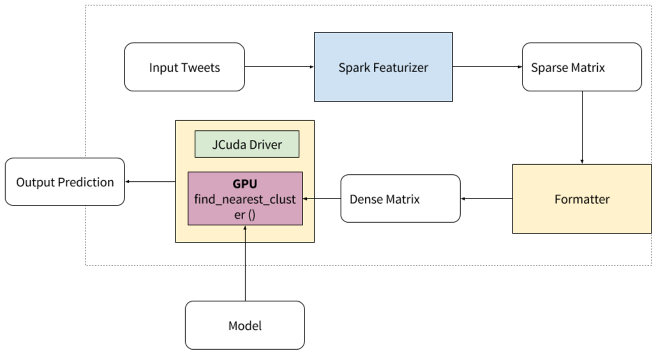


Fig. 4: Flowchart for Prediction

IV. Performance Evaluation

In this section, we present a performance evaluation of our Spark-GPU framework for both the offline training and online prediction.

A. Training

We gathered the execution time information of the training phase for Twitter Language Specifier on both CPU and GPU. Figure 5, Figure 6 and Figure 7 show execution time comparisons between works running on CPU and GPU for various input sizes (2MB, 18MB, and 189 MB). The x-axis in the figures represents the number of Kmeans clusters. The y-axis represents the execution time of running the Kmeans training code for 20 iterations. We use the original Twitter Language Specifier code running on CPUs as a reference to compare our results to. The original implementation calls the Kmeans clustering function from the Spark $mllib$ library. The results for the original Spark implementation and our GPU implementation are shown in blue and red respectively in these figures. The results show that, the speedup of the GPU implementation increases with the number of clusters and the size of inputs. As each GPU thread runs the $find\_nearest\_cluster()$ function for one object, the increased number of cluster centers will increase the amount of computation. The increased size of input will increase the number of GPU threads. Thus, increasing the number of GPU threads to fill the GPU pipeline and the computation amount of each thread is crucial to achieve peak performance on the GPU.

Noted from Figure 6 and 7, we were not able to generate results for 1000 clusters on the original Spark implementation with CPU as the runs took too long. After we investigated into it, it turned out the original Spark Kmeans algorithm runs a $Kmeans++$ algorithm serially to initialize clusters centers. The initialization occupies most of its execution time during the training phase for a large number clusters. Given the Kmeans algorithms used for the two implementations are different, it is hard to make a conclusion if running tasks on GPU is more efficient. We thus decided to write another implementation for Algorithm 1 on Spark with CPU.

The results are shown in Figure 8 - Figure 10. Larger size of inputs with more cluster centers tend to perform better on GPU as the problem shifts towards computation-bound. For a training set with 189MB input data to train a Kmeans model of 1000 clusters for 20 iterations, running on GPUs speed the process up by 17.9x.

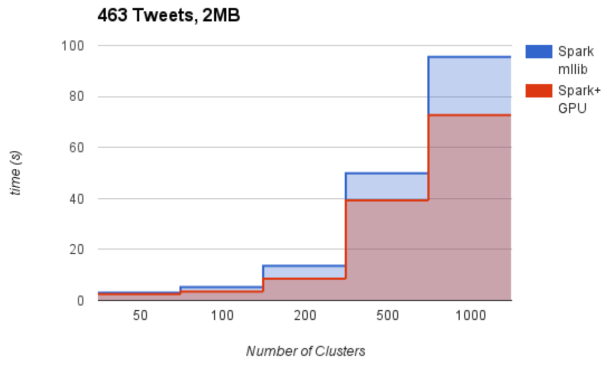


Fig. 5: Training Time for 2MB Data

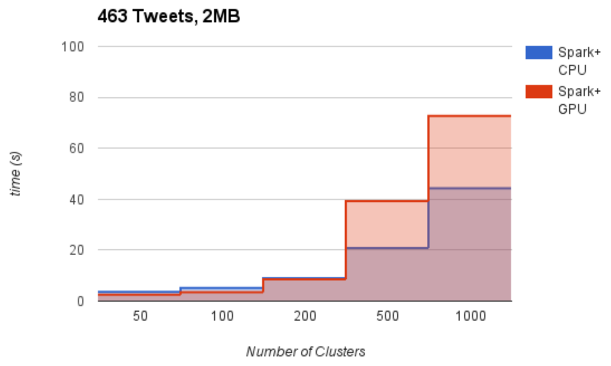


Fig. 8: Training Time for 2MB Data

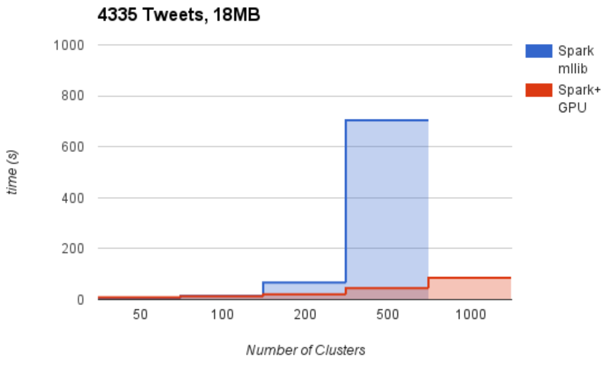


Fig. 6: Training Time for 18MB Data

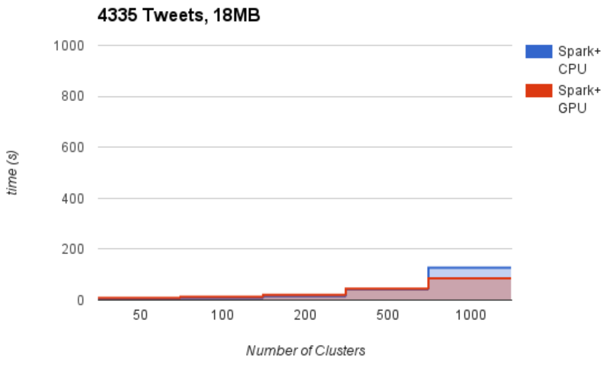


Fig. 9: Training Time for 18MB Data

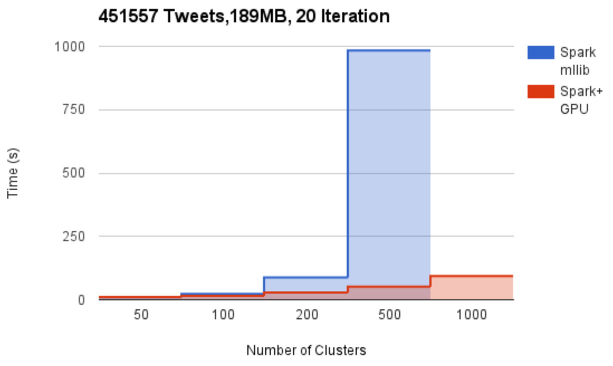


Fig. 7: Training Time for 189MB Data

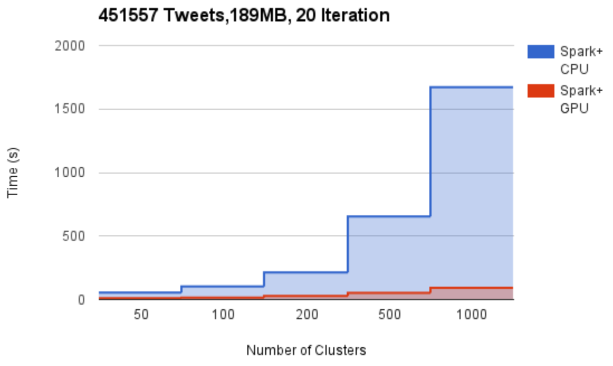


Fig. 10: Training Time for 189MB Data

Considering that the cloud service is charged based on time, the execution time actually represents the real costs of training a model for Big Data analytics. One GPU instance g2.2xlarge on Amazon EC2 costs \$0.65 per hour. A comparable CPU instance c3.2xlarge with 2 more CPU cores and 100 GB more SSD space costs \$0.60 per hour. For computation intensive applications that fits in the 60GB SSD of the GPU instance, using the GPU instances is likely to be more cost-effective as the speedup can be much higher than the 10% price difference. Another metric to measure the efficiency of the system is energy per bit. The average power of Intel Xeon E5-2670 is 115W while the max power of NVIDIA GRID K520 is 225W. Assuming they all consume the power as stated above, the GPU is more energy efficient only when it achieves more than 2x speedup for an application without considering the power consumed by the its host CPU.

Figure 11 shows a typical breakdown of execution time for running one iteration of Kmeans algorithm. The breakdown charts show the overheads take up significant amount of time. Only when the benefit of running tasks on GPU outweighs the overheads, it is worthwhile to use GPU to accelerate tasks.

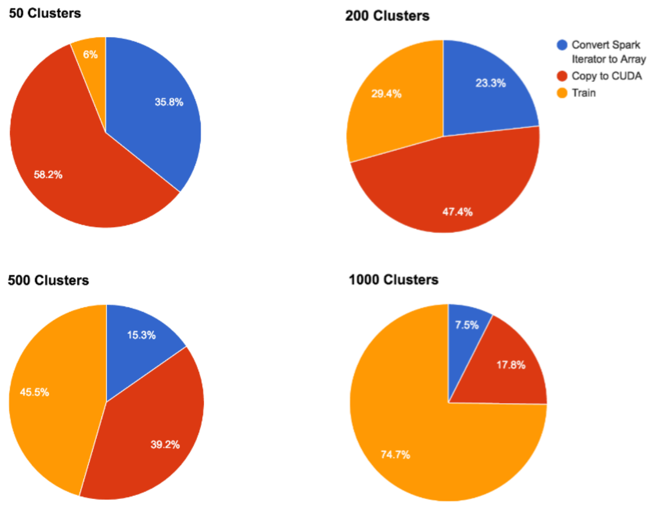


Fig. 11: Execution Time Breakdown

B. Prediction

Figure 12 shows that, even as we increase the number of clusters, running prediction using the CPU achieves a better performance compared to running K-Means prediction on the GPU. Thus, we need to profile the GPU prediction time breakdown in order to pinpoint the bottleneck.

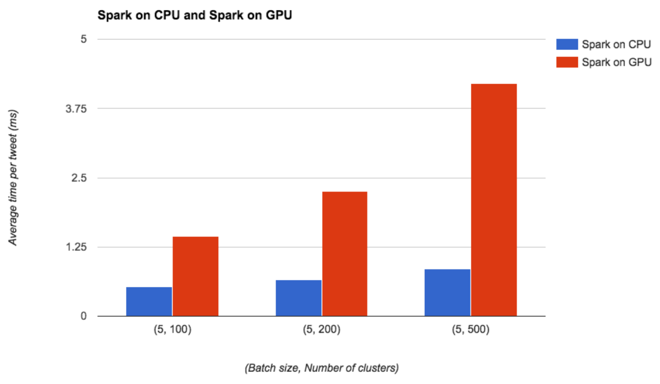


Fig. 12: Prediction Time

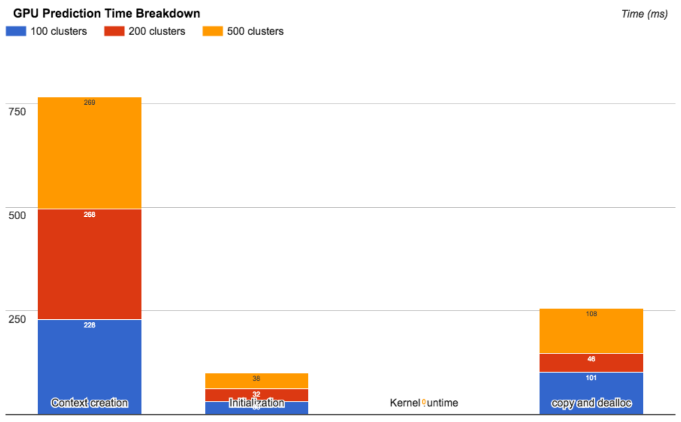


Fig. 13: Prediction Time Breakdown

From the Figure 13, we see that the majority of the processing time is spent initializing the GPU context. However, as we increase the number of clusters in the K-Means model, the percentage of time spent copying the prediction results from the GPU back to the CPU and memory deallocation time also increase. The time spent actually performing flops is negligible. This highlights the importance of designing a system that eliminates the need of reinitializing the GPU context for each streaming microbatch. Unfortunately, the JCuda CUDA context object is not serializable by default, and the JCuda binding does not provide an interface to access the context object's fields, so writing a custom serializer is also out of the question.

V. Discussion

A. JCuda Overhead

To mitigate the problem of instantiating new GPU contexts for each task, we can create a service on each worker that is loaded up during the initialization of Spark. Upon instantiation, this service would create the GPU context only once, and will act as the liaison between the worker's CPU JVM and the CUDA kernel. It would receive RMI function calls from the CPU JVM, and send the serialized result back to the worker's CPU JVM. It could also keep track of the memory allocation on the GPU, keeping a hash table of unique identifiers to the pointers to GPU memory, effectively keeping the data "in-memory" as in the Spark paradigm.

HeteroSpark[7] is a Spark-GPU framework bearing a resemblance to what we described here but without the memory tracking features.

However, compared to our current system where the GPU jobs are instantiated and run under full control of JCuda worker threads,a system like HeteroSpark is less fault tolerant. If the GPU service is detached to Spark worker threads, then the threads have no way of distinguishing system errors from system delays. Also, since the GPU kernel functions need to be known ahead of time, modifying or adding new GPU kernel functions require the service to restart. Thus, a lightweight serializable JCuda library may be necessary for a truly high-performance, fault-tolerant Spark-GPU framework.

B. Proper Size of Workload

For training, we see that bigger the training data is, faster our GPU implementation is compared to out-of-box Spark. This is because our GPU code has successfully turn kmeans training into a I/O bound problem. In addition, the overhead of initializing GPU context is constant. Similarly, we enjoy a larger workload when it comes to prediction, as long as the data fits into GPU memory.

C. Peak Performance on GPU

With the help of performance profiling tools, we found that our GPU training code, when running on the 189MB dataset, uses 7 Gflops out of 2448 Gflops, which is the marked floating point performance of NVIDIA GRID K520. The 189MB dataset contains 451,557 objects which are transferred into equal number of running threads on GPU. Assuming each instruction takes an average of 22 cycles to run, and the GPU runs at 797 MHz, the amount of computation can reach 451,557 x 797MHz / 22 cycles = 16,358Gops, which is enough for filling the GPU pipeline. One factor that might contribute to the low utilization is that the use of share memory on GPUs is disabled. We were not able to use the share memory because the input size is too large. Since the latency of shared memory can be 100x lower that the global memory, most of the time might be spent on waiting for data to arrive. Since our focus is not on improving the Kmeans algorithm on GPU, we did not spend too much time pushing the performance to its limit. The good news is that there is a huge space for improvement moving forward.

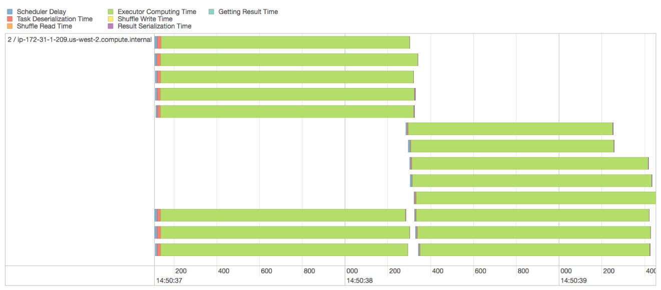
D. Portability

Porting existing Spark code written in Scala to our current framework requires a fair amount of modifications. Though the Spark worker code is inherently parallel to each other, the parallelism within the worker code is not exposed. It is up to the users to parallelize the worker code or refactor the work distributions among the workers with GPUs. For example, when there is little parallelism in the worker code, the use of map() might no longer be feasible, mapPartitions() can be a better choice in such scenarios. A perfectly integrated Spark-GPU system should be able to auto partition the workload between CPUs and GPUs with few code modifications. However, the different programming languages and running platforms remain to be barriers to the Spark integration with other hardware platforms such as GPUs and FPGAs. SparkCL [8] proposed a framework to use Java OpenCL bindings and map the OpenCL kernels to various hardware platforms. The use of OpenCL as the intermediate language makes it possible to run the same code on various platforms.

Another systematic approach with better usability would be directly adding features to the JVM compiler to compile Java bytecode to various hardware compatible machine code based on runtime statistics.

E. Spark Load Balancing

Spark's emphasis on its ease of use also comes at the detriment of being hard to optimize. While producing our benchmarks for training, we noticed that Spark was preferentially repeatedly assigning tasks to the same executor, rather than assigning those tasks to idle executors on other nodes. Although this makes sense from the perspective of data locality, this type of task scheduling performs extremely poorly considering that there is only one GPU on each executor. Strangely enough, Spark was even assigning more tasks to that executor than the number of cores running on that executor. For example, when we set the number of partitions to be 16, two times the number of cores on each executor, Spark still assigned all 16 tasks to the same executor.



It was only when we changed the Spark configuration to be more lenient towards assigning non-local tasks that we ended up with better load balancing - this was a configuration parameter (spark.locality.wait) that we had to tune by hand for our particular context in order to achieve the performance detailed in the previous section.
However, even this assumes that each partition of data is distributed roughly evenly. Although repartioning the data to achieve equal-sized data partitions is a valid strategy in traditional batch applications, the overhead of doing so can be too high for latency-sensitive applications. Thus, writing application level code that achieves proper load balancing in Spark is an open problem that can be the topic for future exploration.

VI. Conclusion

Although the overhead is significant to run Spark tasks on GPU using JCuda library, the idea of introducing new hardware platforms to cloud computing is still promising. Our training results shows that deploying on GPUs resulted in up to 17.9x speedup compared to running the same Kmeans algorithm on CPUs for the Twitter Language Classifier, albeit with the notable overheads. Spark Streaming applications are only feasible on GPU when the workload is computation heavy with enough parallelism.

The existing future work to improve the performance includes:

1) Investigate various Kmeans clustering algorithms
2) Keep the GPU context/device pointers for different iterations to reduce overhead
3) Run Spark code on both CPU and GPU

4) Schedule the Spark work based on the available GPU resources

Acknowledgment

The authors would like to thank Professor Jim Demmel for his advice.

References

[1] Java bindings for cuda. [Online]. Available: http://www.jcuda.org/

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: A new model and architecture for data stream management," *The VLDB Journal*, vol. 12, no. 2, pp. 120–139, Aug. 2003. [Online]. Available: http://dx.doi.org/10.1007/s00778-003-0095-z

[3] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina *et al.*, "The design of the borealis stream processing engine."

[4] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," 2012.

[5] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making sense of performance in data analytics frameworks," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, May 2015, pp. 293–307. [Online]. Available: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/ousterhout

[6] Parallel k-means data clustering. [Online]. Available: https://github.com/serban/kmeans

[7] P. Li, Y. Luo, N. Zhang, and Y. Cao, "Heterospark: A heterogeneous cpu/gpu spark platform for machine learning algorithms," in *Networking, Architecture and Storage (NAS), 2015 IEEE International Conference on*, Aug 2015, pp. 347–348.

[8] O. Segal, P. Colangelo, N. Nasiri, Z. Qian, and M. Margala, "Sparkcl: A unified programming framework for accelerators on heterogeneous clusters," *CoRR*, vol. abs/1505.01120, 2015. [Online]. Available: http://arxiv.org/abs/1505.01120