Project Report

Adding Support for a GPU library within Apache Spark

CS 249 - Cloud Computing

Suket Sharma (504591609)
Salil Kanetkar (704557096)

1. Introduction

Spark is an open-source cluster computing framework. It provides an API in Java, Scala, R and Python which is centered on a data structure called RDD (Resilient Distributed Dataset). They are a collection of fault-tolerant objects which can be operated in parallel[1]. However, there is no in-built support for native libraries written in C/C++. GPU libraries are mostly all written as native code. While GPUs are only good for handling limited data types, they can run certain algorithms like sorting, much faster. In this project, we have tried to use Boost.Compute for sorting primitive data types within Spark and determined if there is any gain in speed in comparison to the in-built sorting available in Spark.

2. Boost.Compute

Boost.Compute is a GPU/parallel-computing library for C++ based on OpenCL. The core library is a thin C++ wrapper over the OpenCL API and provides access to compute devices, contexts, command queues and memory buffers[2].

We chose this library over Thrust, since OpenCL is supported by more GPUs and is not restricted to only Nvidia chips. This library made sorting on the GPU relatively straightforward and was similar to the STL syntax of C++. It required the OpenCL framework, Boost library, Boost.Compute library and a compatible GPU.

3. Spark

Our main job was to somehow send data being processed by Spark to the GPU and take it back. We had two alternatives to achieve this task. One way to communicate with a C++ program is to use the function pipe() defined for RDDs which would take each element of an RDD and send it to a given script's stdin. It would then then return a list of elements which was outputted to stdout by that script. The output is received as a RDD of strings in the spark code. The other alternative was to use Java Native Interface (JNI) which can be used to interface between Java and C++ code. Intuitively, we thought pipe() should be faster since it opens a direct stream of data from our Spark application to the native library. Hence, we did all our testing using the pipe() function.

However there were a few caveats with this approach[3],

- We could not provide any kind of metadata (say data type) to the native code, since pipe() would simply send only the elements of the RDD. We created different versions of native codes, which catered to different data types as a workaround.
- Sending data as a stream of bytes was the fastest way. However, we would have to encode using Base64 since Java could be using multiple encoding standard. Also, the endianness of the machines being used would also factor in.
- As a work around to the above we used LF (\n) as delimiters and sent our data as strings and obtained them back as strings as well.
- However, this caused an overhead of type conversion on both sides. We went ahead with this approach, since our main aim was to first determine if using GPUs within Spark was even viable.

4. <u>Overview of Our Approach</u>

The first thing we did was to coalesce JavaRDD objects. We did this because, rather than separately sorting each RDD partition in different GPU processes, followed by a merge, we found it faster to run just one GPU sort on the whole RDD content together.

We created randomized vectors of ints and floats which was fed to a parallelize() function and then pipe()'d to the forked C++ process. The C++ process, then copied the data onto the GPU's memory, where it was sorted using Radix or Merge and then finally returned to the Spark process. Boost.Compute does not have in-built support for memory buffers or objects so we had to use a workaround to test that.

5. <u>Sorting Objects</u>

Java Objects are sent by pipe() as a byte stream by using the Serializable Interface. It was difficult for us to parse the contents of this byte stream on the C++ side, and then recreate this object since we wanted a generic code, capable of reading any kind of object with any structure. We initially thought that we could first define a mapping from a Java class to a C++ class. We could then write this C++ class into a header file and have it included in the GPU code we were calling. This way we would have been able to parse and store the Java Object byte stream correctly. However, this eventually felt like reinventing the wheel and we considered using JNI for this task instead.

However, Boost.Compute does not support sorting objects yet. We cannot define comparator function for vectors of objects either, so we decided to use another way to achieve object sorting. We defined a mapping from Objects to Sort Key, where the Sort key could be any Integer or Float Value. Say for Example, Employee Object to Employee Salary. We would then send this

Sort Key to the GPU for sorting. Based on the result of the sorting, we would then re-arrange the objects as well. This can be understood from the following diagram:

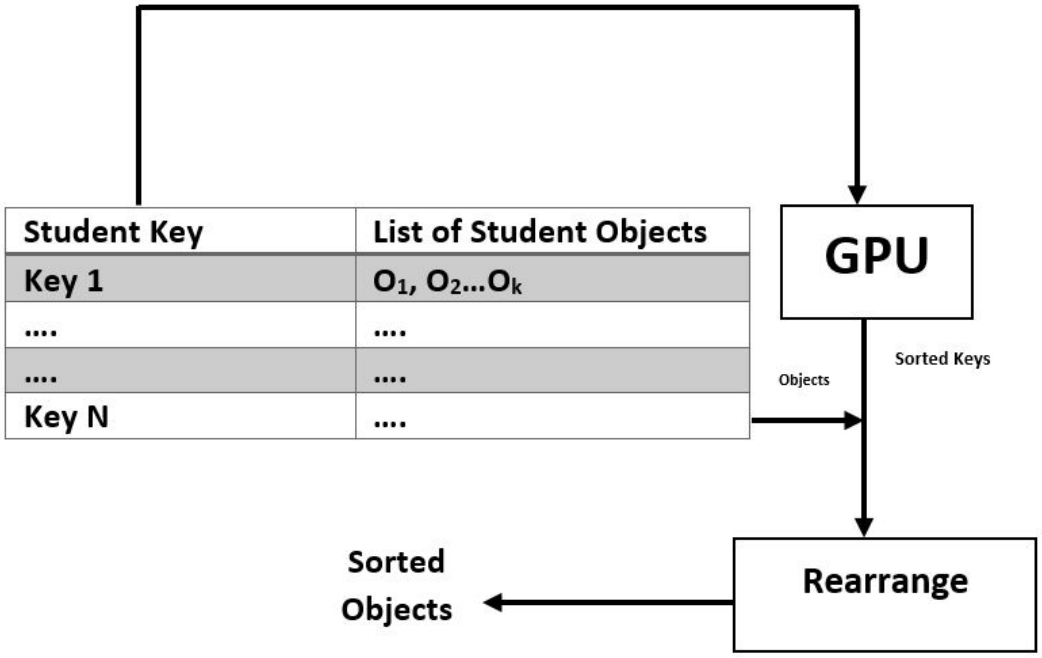

Figure 1. Sorting of Student Objects

6. Results

1. Integers

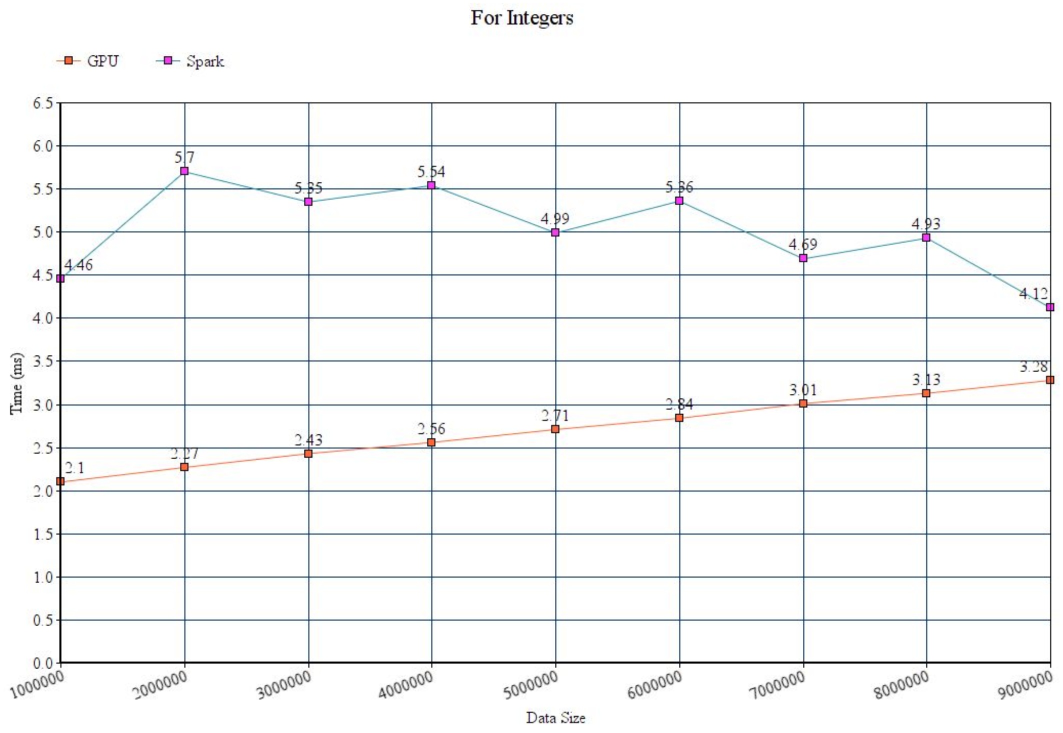

Figure 2. Result of Sorting Integers

- Execution times on Spark were not very stable.
- Our guess is that this was because of varying CPU loads when we are testing.
- However, GPU sorting was clearly always faster than using the sorting primitives available in Spark.

2. <u>Floats</u>

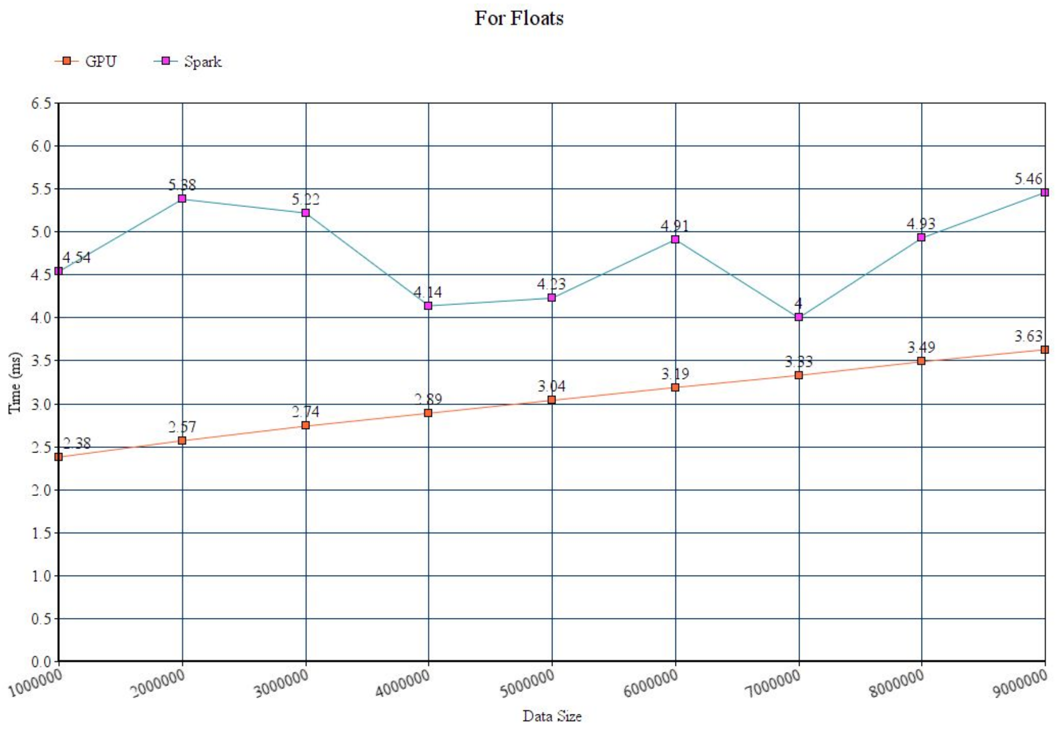

Figure 3. Result of Sorting Floats

- Very similar to results obtained for integers.
- GPUs can help sort floating values much faster. This includes time taken for sending and retrieving data from the GPU.

3. Objects

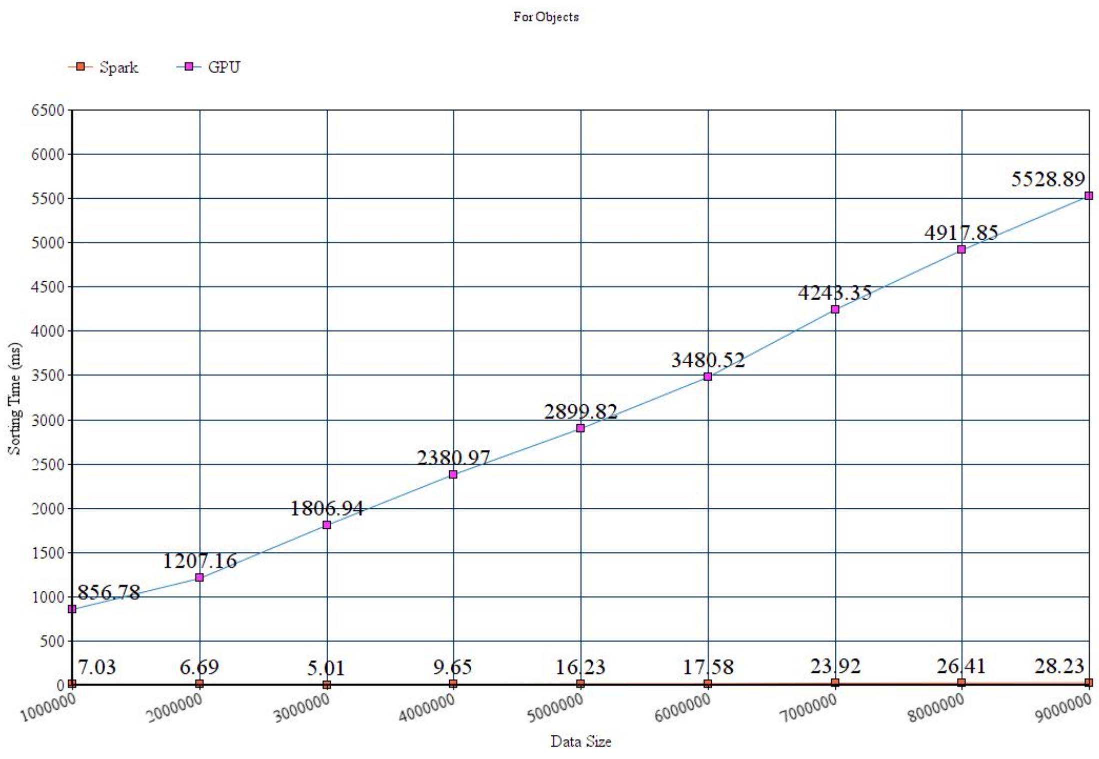

Figure 4. Result of Sorting Student Objects

- There were two major overheads in sorting objects - Creating a hash table and converting a list to JavaRDD.
- This results in extra O(N) memory for the hash table and two linear passes through the entire hash table.
- Lesson Learnt - Speed-Up in sorting ints and floats is not sufficient to offset overhead in hash creation and re-arranging.

7. Platforms Used

- Intel Core i5 - 2.7Ghz with 2 cores - 8GB
- Intel Iris Graphics 6000 - 1.5GB
- OpenCL 1.1 and Boost.Compute 1.7

8. Future Work

- Parsing Java Object Streams and reconstructing CPP objects could be faster.
- It would be easier to just let the Boost.Compute library take over sorting of all <Integer> and <Float> objects.
- Kyle Lutz has mentioned[4] that he intends to add support for sorting buffers soon.

- Once that happens, shifting sorting of strings etc to the GPU would make more sense.

9. References

[1] https://en.wikipedia.org/wiki/Apache_Spark
[2] https://github.com/boostorg/compute
[3] https://groups.google.com/forum/#!topic/spark-users/cPWREWuFoxc
[4] http://kylelutz.blogspot.com/2014/03/custom-opencl-functions-in-c-with.html