GPU Acceleration of PySpark using RAPIDS AI

Abdallah Aguerzame, Benoit Pelletier and François Waeselynck
Atos, Grenoble, France
{abdallah.aguerzame, benoit.pelletier, francois.waeselynck}@atos.net

Keywords: CloudDBAppliance, BullSequana S, Scale-up, Spark, PySpark, GPU, Pandas UDFs, RAPIDS AI.

Abstract: RAPIDS AI is a promising open source project for accelerating Python end to end data science workloads. Our quest is to be able to integrate RAPIDS AI capabilities within PySpark and offload PySpark intensive tasks to GPUs to gain in performance.

1 INTRODUCTION

Within the scope of the CloudDBAppliance project, we investigate how Apache Spark™ can leverage a many cores and large memory platform, with a scale up approach in mind, as opposed to the commonly used scale out one. That is, rather than spreading a Spark cluster on many vanilla servers, the approach is to deploy it on a few BullSequana™ large servers with many cores (up to several hundreds) and large memory (up to several tera-byte).

The target hardware platform of CloudDBAppliance is a BullSequana S server, a highly scalable and flexible server, ranging from 2 to 32 processors (up to 896 cores and 1792 hardware threads), up to 32 GPUs and 48 TB RAM, and 64 TB NVRAM.

In previous work (Waeselynck, 2019), we inculcate NUMA awareness to Spark, that provides a smart and application transparent placement of executor processes. This paper in the other hand will be dedicated to expose a solution that leverages GPUs in PySpark workflow using the new Nvidia library RAPIDS AI (RAPIDS, 2019). Our initiative is inspired from a previous work done by an Nvidia team to accelerate UDFs (user defined functions) in PySpark with Numba and PyGDF (Kraus and Joshua Patterson, 2018). Our work however will be adapted with the new library recently launched by NVIDIA "RAPIDS AI".

This document is organised in the following order. We first start by introducing PySpark and explaining how it is possible to offload tasks to the GPU using Apache Arrow (Apache Arrow, 2019) and RAPIDS AI. Then we represent the two types (Scalar and Grouped Map) of Pandas UDF within PySpark that leverages Apache Arrow. After that, we illustrate two implementations examples, that show how to create within PySpark both a scalar Pandas UDF that computes on the GPU the product of 2 columns and a Grouped Map Pandas UDF that subtract on the GPU the mean from each value in the group. Finally, we show and discuss the results obtained from our experiments with RAPIDS AI and PySpark.

2 PySpark ACCELERATION WITH GPU

In 2018 Python was the most popular language in data science (KDnuggets, 2019), and year after year it gets more and more attraction by data scientists. PySpark is Spark's response to this trend: The python API for Spark, that enables Python programming on Spark.

In this paper we introduce and make use of new technologies that recently come to surface and allow to accelerate Python functions on GPUs, where the processing pipelines spans from Spark executors to Python workers and finally lands on GPUs, all of that without dramatically losing speed-ups in cost of serialisations, data conversions and data movement.

- Apache Arrow (Apache Arrow, 2019) is a framework to minimize data conversions and data serialisations when a data processing pipeline includes different computing frameworks.
- RAPIDS AI (RAPIDS, 2019) is a promising project recently announced by Nvidia. RAPIDS AI aims to speed-up data science end to end workflow; it contains APIs and libraries that allow for executing Python jobs on GPUs.

2.1 PySpark Execution Model

Spark runs Python programs differently than Scala programs: unlike Scala programs, Spark executors do not run Python programs directly – that operate on the data they hold, but delegate their execution to Python workers, that are separate processes. This architecture has some drawbacks, especially for data movements between Spark executors and Python executors due to data serialization/deserialization between those processes. The conversion of a Spark data frame to Pandas with the `DataFrame.toPandas()` method is quite inefficient: rows are collected from the Spark executor, serialized into Python's pickle format, then moved to the Python worker, after that deserialized (from pickle format) into a list of tuples, finally transformed to a Panda data frame. And the reverse operations are to do in the other way for results. This overload gives results far below to the execution of an equivalent Scala program. Developers commonly work around this problem by defining their UDFs (user defined functions) in Scala/Java, calling them from PySpark. Experiments have shown that the time spent in serializing and deserializing data often exceeds the compute time (Kraus and Joshua Patterson, 2018), thus, targeting GPU acceleration would not be a good option, because so much time will be lost in serializing, deserialization and copying data.

2.2 Apache ARROW in Spark

Things have changed yet, as starting from version 2.3, Spark can leverage Apache Arrow technology. Apache Arrow (Kraus and Joshua Patterson, 2018) is a cross-language development platform for in-memory data. It specifies a standardized language-independent columnar memory format for flat and hierarchical data, organized for efficient analytic operations on modern hardware. It also provides computational libraries and zero-copy streaming messaging and interprocess communication. It enables execution engines to take advantage of the latest SIMD (Single instruction multiple data) operations included in modern processors (CPU, GPU)), for native vectorized optimization of analytical data processing. Columnar layout is optimized for data locality for better performance on CPUs and GPUs. The Arrow memory format supports **zero-copy reads** for lightning-fast data access without serialization overhead.

As of Spark 2.3, Apache Arrow is introduced as a supported dependency to offer increased performance with columnar data transfer. Once the data is in Arrow memory format, it can transit (possibly without moving) along the processing pipeline from a framework to the next without the need to multiple serialization/deserialization, e.g. from the Spark executor (a java process) to the GPU through the Python worker (a Python process).

2.3 RAPIDS AI

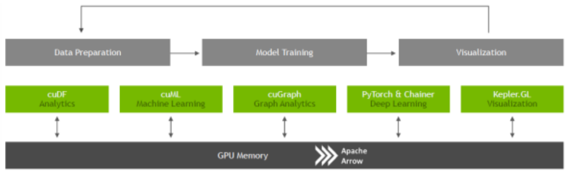


Figure 1: RAPIDS AI components (RAPIDS AI, 2019).

RAPIDS AI is a collection of open source software libraries and APIs recently launched by NVIDIA to execute end-to-end data science analytics pipelines entirely on GPUs. It relies on NVIDIA CUDA primitives for low-level compute optimization, but exposes GPU parallelism and high-bandwidth memory speed through user-friendly Python interfaces. RAPIDS AI also focuses on common data preparation tasks for analytics and data science. This includes a familiar DataFrame API that integrates with a variety of machine learning algorithms for end-to-end pipeline accelerations without paying typical serialization costs. RAPIDS AI also includes support for multi-nodes nodes, multi-GPU deployments, enabling vastly accelerated processing and training on much larger dataset sizes.

The RAPIDS AI cuDF API is a DataFrame manipulation library based on Apache Arrow that accelerates loading, filtering, and manipulation of data for model training and data preparation. The Python bindings of the core-accelerated CUDA DataFrame manipulation primitives mirror the Pandas interface for seamless onboarding of Pandas users. Previous efforts were provided by GoAI (GPU Open Analytics Initiative) project that initiated PyGDF (Python GPU DataFrame library): PyGDF is based on Apache Arrow data format, converts Pandas DataFrame to GPU DataFrame, and interfaces with CUDA using Numba, a compiler for Python arrays and numerical functions to speed up Python programs with high-performance functions. PyGDF is already integrated with cuDF, a more elaborated and complete library.

2.4 Bringing Data to the GPU

Our objective is to bring GPU capabilities to Pyspark framework, as shown in the figure 2. Thanks to the common playground using Apache Arrow, allowing python processes to work more efficiently on Spark, data movements are no or low burden with optimized data conversion and serialisation, with a columnar format suited for GPU consumption.

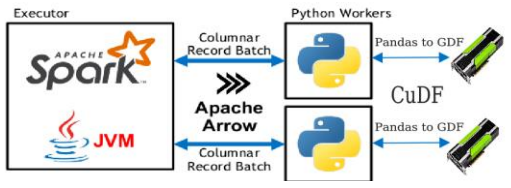


Figure 2: PySpark GPU acceleration model with cuDF.

3 PANDAS UDFs WITH GPU

In Spark we can create user defined functions (UDFs) that have a column-based format. This UDFs are used to create functions outside of the scope of Spark built-in functions, they can be defined in Scala/Java or Python and be called from PySpark. As explained in the previous chapter, Arrow format is used in Spark, so data can be transferred most efficiently between JVM and Python processes, then to the GPU. Using Arrow in spark is not automatic and requires some changes to configurations or code. As shown in figure 3, arrow optimization is needed when converting a Spark *DataFrame* to a Panda *DataFrame* using the call `toPandas()` and the way around when creating a Spark *DataFrame* from a Panda *DataFrame* with `creatDataFrame(pandas_df)`.

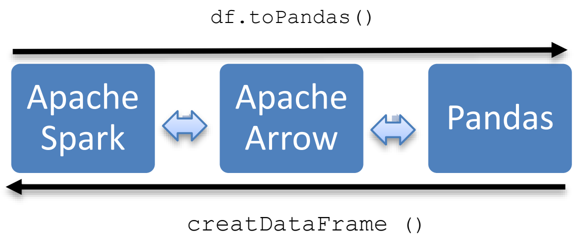


Figure 3: Data conversion with Apache Arrow.

To insure that Arrow format is used when executing these calls, we need to set the spark configuration ‘spark.sql.execution.arrow.enabled” to ‘true’, this is disabled by default in Spark.

In PySpark, we recognize two types of UDFs, Scalar Pandas UDFs and Group Map Pandas UDFs.

3.1 Scalar Pandas UDFs

Scalar Pandas (Spark SQL, 2019) UDFs are used for vectorizing scalar operations. They can be used with functions such as `select` and `withColumn`. They require a `pandas.series` as input and a `pandas.series` as output. Both the input and the output must be of the same length. Internally when executing a Pandas UDF, Spark will split its columns into batches and calls the function for each batch as a subset of the data, then concatenating the results together. In this regard cuDF API call “`cudf.Series()`” can be used inside the UDF to transform the input Python *DataFrame* to a GPU *DataFrame*, so operations will be carried out on the GPU.

```
def gpu_scalar_pandas_udf_example(spark):
    import pandas as pd
  import cudf
    from pyspark.sql.functions import col,
    pandas_udf
    from pyspark.sql.types import LongType
# Declare the function and create the UDF
def gpu_multiply_func(a, b):
# Create GPU Data Frame using cuML API
    gdf_a = cudf.Series(a)
    gdf_b = cudf.Series(b)
    gdf_rslt = gdf_a * gdf_b
    return gdf_rslt.to_pandas()

multiply_gpu =
pandas_udf(gpu_multiply_func,return
Type=LongType())
```

The above code is an example of a scalar user defined function UDF that takes as input two columns and gives the computed product as a result. We create a Spark *DataFrame* from a Pandas *DataFrame* and applied a UDF function to its column.

```
x = pd.Series([1, 2, 3])
df =
spark.createDataFrame(pd.DataFrame(x
, columns=["x"]))
```

The variable `‘x’` is a Pandas series that we transform to `‘df’` which is a Spark *DataFrame*. Internally Arrow format will be implicitly applied as we activate it in our Spark session configuration `“spark.conf.set("spark.sql.execution.arrow.enabled", "true")”`.

Therefore, in the code above `‘gdf_a’` and `‘gdf_b’` will be executed as GPU *DataFrame* and the multiplication operator will be a CUDA primitive provided by the cuDF API and will be executed on GPU. Finally we convert back the result to a Pandas *DataFrame* `‘return gdf_rslt.to_pandas()’`

because the specified type in the return value of the UDF is `'LongType()'` which is a Pandas type.

```
  df.select(multiply_gpu(col("x"),
col("x"))).show()
    # +-------------------+
    # |gpu_multiply_func(x, x)|
    # +-------------------+
    # |                  1|
    # |                  4|
    # |                  9|
    # +-------------------+
```

As a result, we have the multiplication of the column of a spark *DataFrame* by itself.

3.2 Grouped Map Pandas UDFs

Grouped map Pandas[7] UDFs are used by the `groupBy().apply()` function, it has a three steps execution paradigm "split-apply-combine". It first splits the data into groups by using `DataFrame.groupBy` then apply a function on each group and finally combine the results into a new *DataFrame*. The input and output data are both `pandas.DataFrame`. The output DataFrame can be any length.

In order to use `groupBy().apply()` we need to designate a Python function that defines the computation for each group, and a *StructType* object or a string that defines the schema of the output *DataFrame*. Each column of the returned `pandas.DataFrame` should be labelled according to the specified output schema. Either they match the field names in the defined output schema if specified as strings or they match the field data types by position if not strings.

Seemingly data for each group must be converted to GPU *DataFrame* using "`cudf.DataFrame.from_pandas` ()" from cuDF API. Therefore, the execution will be carried out by the GPU.

```
from pyspark.sql.functions import
pandas_udf, PandasUDFType
import cudf
df = spark.createDataFrame(
   [(1, 1.0), (1, 2.0), (2, 3.0), (2,
   5.0), (2, 10.0)],("id", "v"))

@pandas_udf("id long, v double",
PandasUDFType.GROUPED_MAP)
def subtract_mean(pdf):
   # pdf is a pandas.DataFrame
   gdf = cudf.DataFrame.from_pandas(pdf)
   v = gdf.v
   return gdf.assign(v=v -
   v.mean()).to_pandas()

df.groupby("id").apply(subtract_mean).sho
w()
```

In this example, we show how to implement a Grouped Map user defined function that subtracts the mean from each value in the group. We start by creating a Spark *DataFrame* `'df'` then we create a UDF where the inputs matches the Spark *DataFrame* schema `(id,v)` and a returned value as `PandasUDFType.GROUPED_MAP` Pandas type.

Inside the the UDF we convert the pdf to a gdf (GPU *DataFrame*) so the execution of the mean will be carried out by the GPU. Seemingly the returned result is a Pandas *DataFrame* so the result can be shown as bellow.

```
+---+----+
| id|   v|
+---+----+
|  1|-0.5|
|  1| 0.5|
|  2|-3.0|
|  2|-1.0|
|  2| 4.0|
+---+----+
```

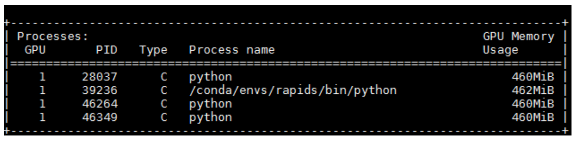
4 EXPERIMENTS

In our work, we use a docker container containing all the needed libraries. RAPIDS AI proposes different docker containers, we use the latest image tag "cuda9.2-runtime-ubuntu16.04" that comes with RAPIDS AI 0.6, CUDA 9.2 and a Jupyter notebook. We add to it Spark 2.4.3 that supports Apache Arrow Format.

We run our code in a Jupyter notebook. It's based on the Grouped Map Pandas UDF example we show in the section 3. It contains two functions, one that uses RAPIDS AI API to calculate the mean and subtracted it from each value in the group, and another function that does the same thing but uses PySpark native API instead.

The system under test comprises of a BullSequana S800 server with 8x12 cores processors and 4 terabytes (TB) RAM memory, that is a total of 96 cores and 192 hardware threads, as hyperthreading is activated. The machine is coupled with 4 Tesla V100 GPUs with 16 Gb of memory each.

Our PySpark application is run with a parallelism of 40 tasks - where each task is run by a thread, launched in local mod with 40 GB Heap size. System metrics are gathered by means of a *sar* command.



```
+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    1     28037      C   python                                        460MiB |
|    1     39236      C   /conda/envs/rapids/bin/python                 462MiB |
|    1     46264      C   python                                        460MiB |
|    1     46349      C   python                                        460MiB |
+-----------------------------------------------------------------------------+
```

Figure 4: Screenshot of nvidia-smi output.

Only one GPU is used during our tests. Figure 4 shows a screenshot of the *nvidia-smi* command output, that shows the GPU is used while executing the first function that implements RAPIDS AI API.

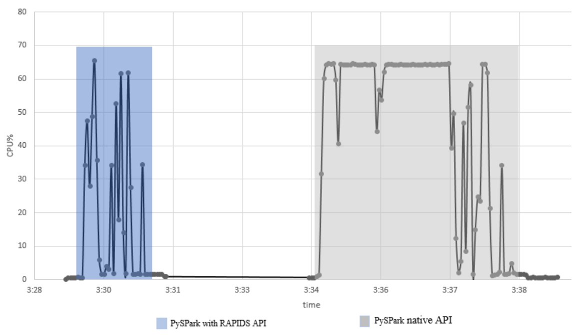


Figure 5: CPU Load for the Grouped Map UDF example.

The input data is generated locally using Spark. Figure 5 illustrates the CPU load while executing our application of a grouped map Pandas UDF over a vector of 800 million elements.

The blue area of the graph shows CPU load while executing the first section of the code that uses RAPIDS AI API and the gray area shows CPU load when executing the second section that uses native PySpark API.

We clearly distinguish that the CPU load is less important in the blue area, because tasks are offloaded to GPU when using RAPIDS AI API, whereas tasks are fully computed by the CPUs where native PySpark API is used.

Figure 6 demonstrates how much speed up we get when using RAPIDS AI API with PySpark. We can go as high as 3 time faster with RAPIDS AI when using input data of 800 million elements. However, we can only benefit of speed up when there is enough data, so offloading tasks to GPU will not be significantly affected by the overhead of data transfer between CPU and GPU. Our experiments show that when using RAPIDS AI we get a speed up for input datasets of more than 500 million elements, whereas we get a slow down below.

The results obtained here are bounded by the fact that our code is executed on a Jupyter notebook from a Docker container. We presume that more speed up can be achieved if the Jupyter notebook is bypassed. Speed up can also be more significant if executing more complex operations over larger datasets, so GPU power can be fully exploited and data transfer overhead between GPU and CPU will be negligible.

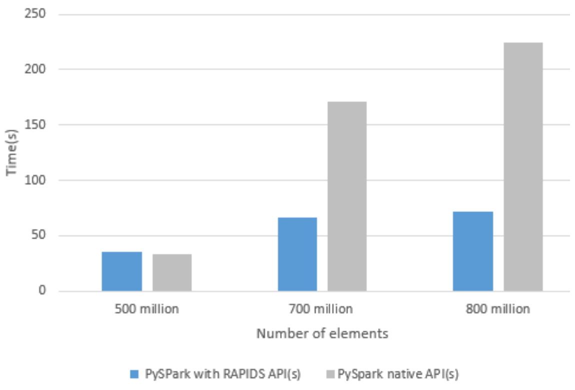


Figure 6: Grouped map panda UDF example with different data size.

5 CONCLUSION

Big Data analytic workloads have become more and more demanding in terms of computational power. Thus, CPU-only systems can no longer handle efficiently the task. Especially that Moore’s law is now coming to an end. So, hardware acceleration using GPU becomes a key feature in modern computational systems. Spark, as the most used framework in Big Data, is a target for scaling up its working environment to the GPUs.

RAPIDS AI might be a young project but it’s growing rapidly profiting from NVIDIA’s backup. RAPIDS AI cuML API contains a handful of machine learning algorithms, such as K-Means, NN, SGD and others… which makes a big asset for data scientist and data engineers. There is also a graph analytics API cuGraph containing a collection of graph analytics that process data found in a GPU *DataFrame*. All this makes RAPIDS AI an auspicious project that is well suited for bringing GPU capabilities to Apache Spark.

Our work is a verification process to make sure that the connection between RAPIDS AI and Spark is possible, and delegating Spark tasks to GPU is within our reach. This however will in return open for us other possibilities to make Apache Spark capable of leveraging High-end servers such as BullSequana S coupled to GPU. Thus enabling to process large amount of data within a single system or few systems in less time than if using larger systems with CPUs only. Further work can be carried out to establish a comparison test between Spark ML lib and RAPIDS

AI cuML with Spark to elaborate on how much acceleration we can get and provide some insights on the benefits of using GPUs.

ACKNOWLEDGEMENTS

The CloudDBAppliance project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 732051.

REFERENCES

Waeselynck, F. and Pelletier, B. (2019). "A NUMA Aware SparkTM on Many-cores and Large Memory Servers". In *Proceedings of the 9th International Conference on Cloud Computing and Services Science - Volume 1: ADITCA*, ISBN 978-989-758-365-0, pages 648-653. DOI: 10.5220/0007905506480653

RAPIDS. https://rapids.ai.

Keith Kraus and Joshua Patterson. "GPU-accelerating UDFs in PySpark with Numba and PyGDF". *AnacondaCon 2018*.

Apache Arrow, http://arrow.apache.org.

KDnuggets. https://www.kdnuggets.com/2018/05/poll-tools-analytics-data-science-machine-learning-results.html.

RAPIDS AI, *FOSDEM'19*, NVIDIA.

Spark SQL. https://spark.apache.org/docs/latest/sql-pyspark-pandas-with-arrow.html.