Немного советов и рассказ про GPU

О чем поговорим

- Как упростить работу
- Неочевидные ошибки
- Собираем конфиг исходя из характеристик машины/кластера
- Spark on GPU

Как упростить работу

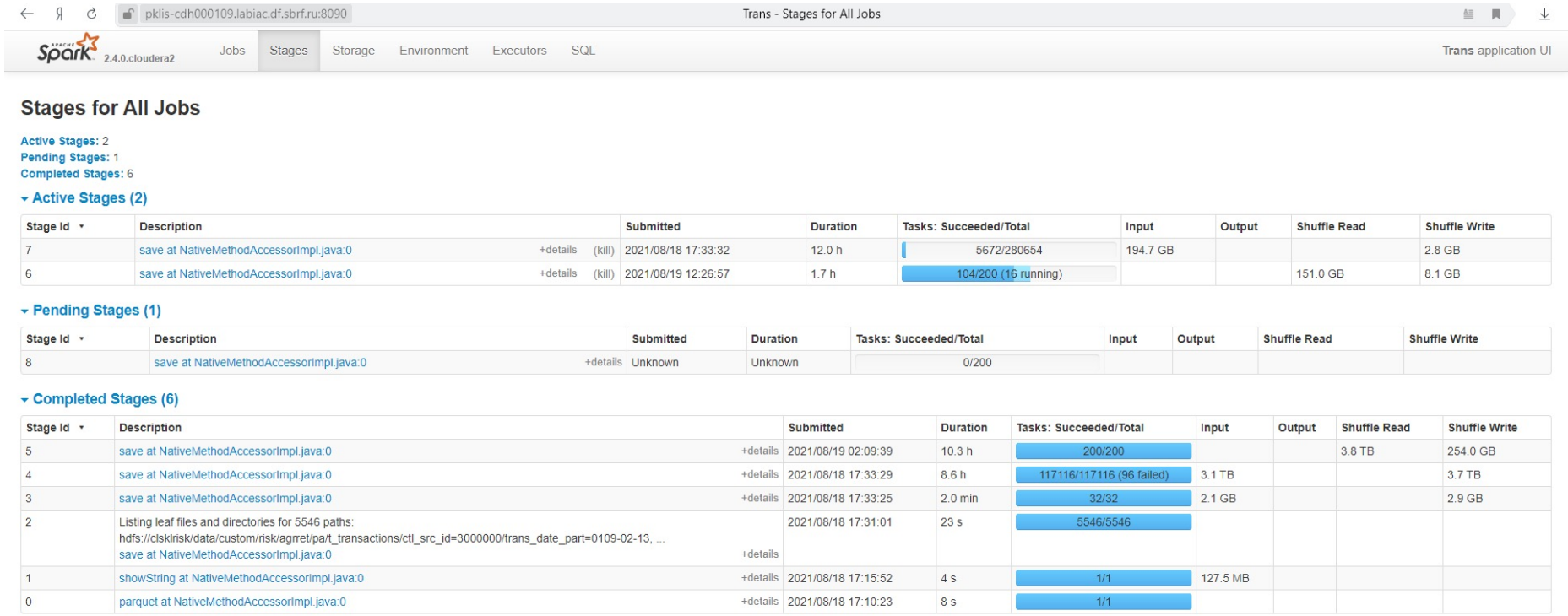
- Все начинается с конфига и многое зависит от него
  - Spark.python.worker.reuse конфиг интересный, но если у вас много операций, которые затратны по памяти, то высок риск, что java заругается и упадет;
  - Если при считывании таблицы, паркета у вас все падает по памяти, то есть конфиг, который использует hive serde для паркетов вместо встроенных алгоритмов. spark.sql.hive.convertMetastorepParquet нужно задать с параметром false. Будет больше job, дольше работать, но зато есть шанс посчитать.
  - Мы уже знающие и помним, что pyspark сериализует и десериализует объекты python. По умолчанию используется JavaSerializer, но намного быстрее KryoSerializer, поэтому spark.serializer лучше ставить с org.apache.spark.serializer.KryoSerializer + задать размер буфера spark.kryoserializer.buffer.max, если вдруг не хватит памяти (редкость), по умолчанию 64m, должно быть не больше 2048m.
  - Динамическая аллокация ресурсов spark.dynamicAllocation.enabled = True хорошая вещь, но работает на эвристиках и при резком росте ресурсов может упасть. В этом случае ручная настройка памяти драйвера, executor’а, размера максимального результата помогут. Если используете динамику, не забывайте про set('spark.shuffle.service.enabled', 'true’) для контроля и возврата неиспользуемых ресурсов.
  - Помните, что если задаете статические параметры ресурсов, то на них динамическая аллокация не распространяется.

Как упростить работу

- Партицирование
  - Смотрите на таблицы и попадайте в партиции! Фуллскан огромных таблиц занимает много времени, забивает память и в итоге ваши скрипты падают с ошибками.
  - Если сохраняете свои паркеты, то их тоже можно и нужно партицировать, когда данных много.

Как упростить работу

- Следите на shuffle
  - В большинстве случае Stages предполагают shuffle при переходе от одного к другому, подписывая об этом информацию.
  - Помните про то, какие операции приводят к полному shuffle, а какие делают это менее агрессивно.



Неочевидные ошибки

Они бывают у всех, но очень легко ошибиться тем, кто захочет писать sql-скрипты и сильно привык к oracle. Многое работает 1 в 1, но все же стоит смотреть документацию https://spark.apache.org/docs/latest/api/sql/ и быть очень внимательными при работе с датами:

- to_date работает криво, если писать дату начиная с дня месяца, надежнее to_date('2016-12-31', 'yyyy-MM-dd’) или просто date('2016-12-31’);
- date_trunc работает без проблем только для дат в формате string, либо datetime value;
- trunc кушает даты, но округляет только до года, квартала, месяца и недели;
- date_add добавляет только дни.

Собираем конфиг

Тут мы соберем свой конфиг, начнем с базовых вещей, которые не зависят от мощностей, а далее разберем, как сделать остальные кастомные настройки.

Таким образом план:

- Набираем те конфиги, которые лучше использовать сразу;
- Обсуждаем, как работает под капотом распределение ресурсов и постановка задач;
- Кастомизуем конфиг исходя из новых знаний;
- Обсуждаем, в каком направлении можно двигаться дальше.

Дефолтные конфиги

Что точно берем?

- 'spark.serializer’ – сериализация;
- 'spark.dynamicAllocation.enabled’ – динамическая аллокация;
- 'spark.shuffle.service.enabled’ – возврат ресурсов после динамики;
- 'spark.ui.port’ – порт UI, помним, что не все порты могут быть досутпны.

Получаем:

```
from pyspark.sql import SparkSession
from pyspark import SparkContext, SparkConf

conf = SparkConf().set('spark.ui.port', '4050’)\
                  .set('spark.app.name’, ‘My application’ )\
                  .set('spark.serializer', 'org.apache.spark.serializer.KryoSerializer’)\
                  .set('spark.dynamicAllocation.enabled', 'true’)\
                  .set('spark.shuffle.service.enabled', 'true')
```

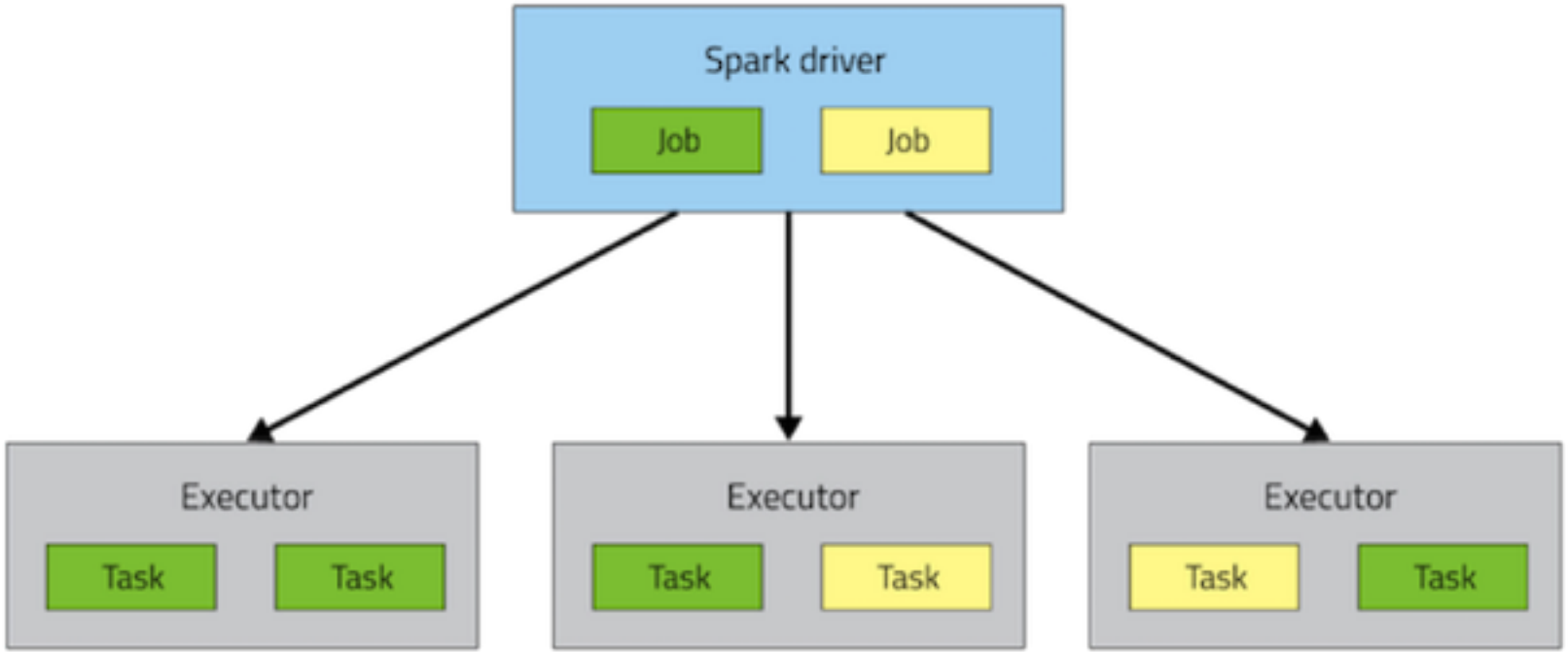
Дефолтные конфиги

- Сериализатор сначала использует небольшой объем памяти, если не хватает, то увеличивает его до максимума spark.kryoserializer.buffer.max. По умолчанию это 64m, максимум можно поставить 2048m. Попробуйте ставить 256m, далее уже от задачи.

```
conf = SparkConf().set('spark.ui.port', '4050')\
                  .set('spark.app.name', 'My application')
                  .set('spark.serializer', 'org.apache.spark.serializer.KryoSerializer')\
                  .set('spark.dynamicAllocation.enabled', 'true')\
                  .set('spark.shuffle.service.enabled', 'true')\
                  .set('spark.kryoserializer.buffer.max', '256m’)
```

И этого уже будет достаточно для решения огромного числа задач! Но что если мы хотим сами настроить, а может и улучшить?

Что под капотом



Драйвер определяет задания, говорит кому что делать, а executor выполняет выданную от driver работу.
Драйвер обеспечивает высокоуровневое управление рабочим процессом.

Что под капотом

- Каждому исполнителю (executor) в приложении Spark выделяется одинаковое фиксированное количество ядер и одинаковый фиксированный размер памяти;
- Один executor может выполнять несколько задач параллельно, если у него более одного ядра в распоряжении;
- Application master, является невыполняющим контейнером (non-executor container) и обладает специальной возможностью запрашивать контейнеры у YARN, в процессе своей работы также потребляет ресурсы, которые необходимо учитывать. Это наш driver.
- Выделение исполнителям слишком большого объема памяти обычно приводит к чрезмерным задержкам в процессе сборки мусора (garbage collection). Примерно 64 ГБ – оптимальный верхний предел для одного исполнителя.
- У HDFS-клиента возникают проблемы при большом количестве одновременно выполняющихся потоков. Как максимум пять задач на исполнитель позволяют использовать всю пропускную способность записи, поэтому необходимо, чтобы количество ядер на исполнитель было меньше этого значения.
- Применение «маленьких» исполнителей (например, с одним ядром и объемом памяти, достаточным для запуска только одной задачи) лишает преимуществ от выполнения нескольких задач на одной JVM;
- Collect и прочие вещи отправляют данные на драйвер.

Выделяем ресурсы

Пусть у нас есть одна тачка со следующими параметрами:

- 16 ядер
- 128 Gb оперативной памяти

Важно! Мы не выделяем 100% ресурсов для контейнеров YARN, потому что на каждом узле необходимо зарезервировать некоторые ресурсы для работы операционной системы и демонов Hadoop!

Словарь:

- spark.executor.instances – сколько executor’ов у нас будет
- spark.executor.cores - сколько ядер у одного исполнителя
- spark.driver.cores – количество ядер драйвера
- spark.driver.memory – объем памяти драйвера
- spark.executor.memory - объем памяти исполнителя

Ну что, настраиваем?

Выделяем ресурсы

1 ядро и немного памяти надо оставить для для драйвера, а все остальное отдаем исполнителям, но сделать 1 исполнителя с 15 ядрами плохо, вспоминаем предыдущие слайды.

Лучше всего начать с такого:

- 1 ядро на spark.driver.cores
- 3 исполнителя spark.executor.instances
- 5 ядер у каждого исполнителя spark.executor.cores
- Память делим честно 32 Гб каждому исполнителю spark.executor.memory и 32 отдадим драйверу spark.driver.memory, но на самом деле драйверу будет много, если вы не будете отдавать кучу данных ему.

А если кластер и 10 тачек? Ну тогда конфигурируем одну машину, количество исполнителей умножаем на 9 (1 тачку не трогаем), немного можем увеличить число драйверов.

Итог

```
conf = SparkConf().set('spark.ui.port', '4050')\
                  .set('spark.app.name', 'My application')
                  .set('spark.serializer', 'org.apache.spark.serializer.KryoSerializer')\
                  .set('spark.dynamicAllocation.enabled', 'true')\
                  .set('spark.shuffle.service.enabled', 'true')\
                  .set('spark.kryoserializer.buffer.max', '256m')\
                  .set('spark.executor.memory', '32G')\
                  .set('spark.executor.instances', '3')\
                  .set('spark.executor.cores', '5')\
                  .set('spark.driver.cores', '1')\
                  .set('spark.driver.memory', '32G')
```

Куда двигаться дальше

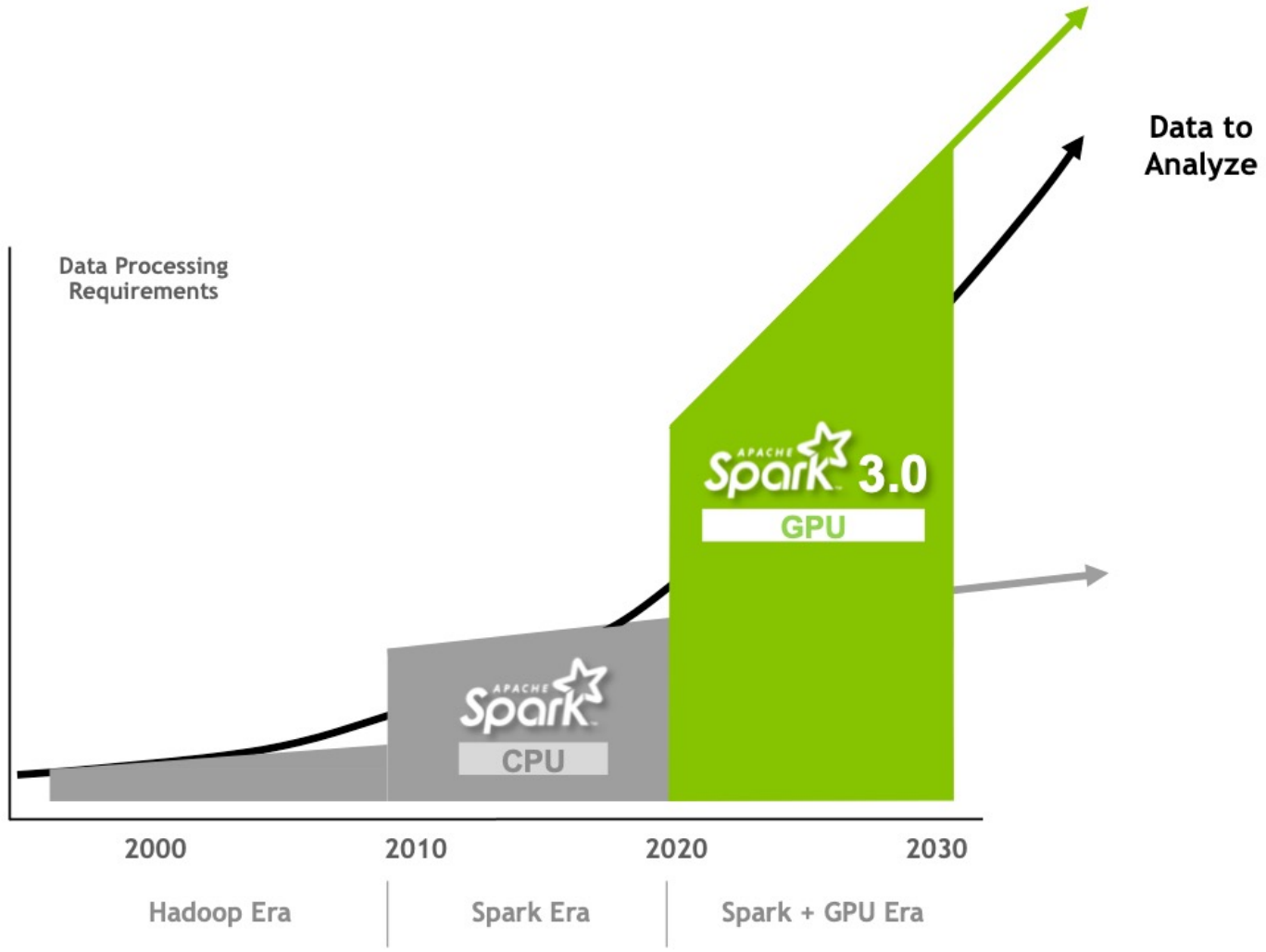
Чаще всего хватит и просто динамической аллокации, но она работает на эвристиках и бывает падает при резком увеличении требований:

- Корректируйте размер памяти драйвера и исполнителя
- Корректируйте размер буфера сериализатора
- spark.driver.maxResultSize – фиксируйте сразу максимальный объем оперативной памяти для результатов драйвера
- spark.default.parallelism – корректируйте число партиций

И иные вещи, связанные с python workers и так далее.

Spark on GPU

- Все развивается, сначала был Hadoop, потом Spark. Начиная с версии 3.0 Spark неплохо начал общаться с GPU.



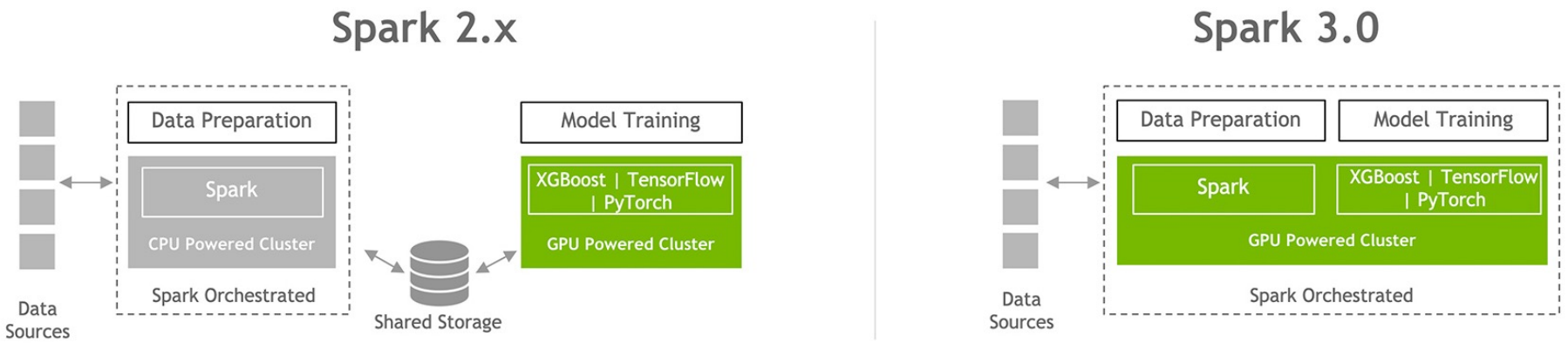
https://developer.nvidia.com/blog/optimizing-and-improving-spark-3-0-performance-with-gpus/

Spark on GPU

Все новшества решают какие-то проблемы. Spark решил проблему I/O у Hadoop, добавив обработку в памяти, но все развивается и появляется новая проблема: приложений слишком много, вычислений также много, хочется считать все быстрее.

NVIDIA посидели с сообществом Apache Spark и выпустили Spark 3.0 вместе с RAPIDS Accelerator for Spark. Что получили:

- Ускорение времени обработки данных и обучения моделей;
- Единая инфраструктура может быть использована под Spark и остальные DL/ML фреймворки;
- Можно купить меньше серверов – снизим затраты на инфраструктуру (стоимость GPU не в счет).

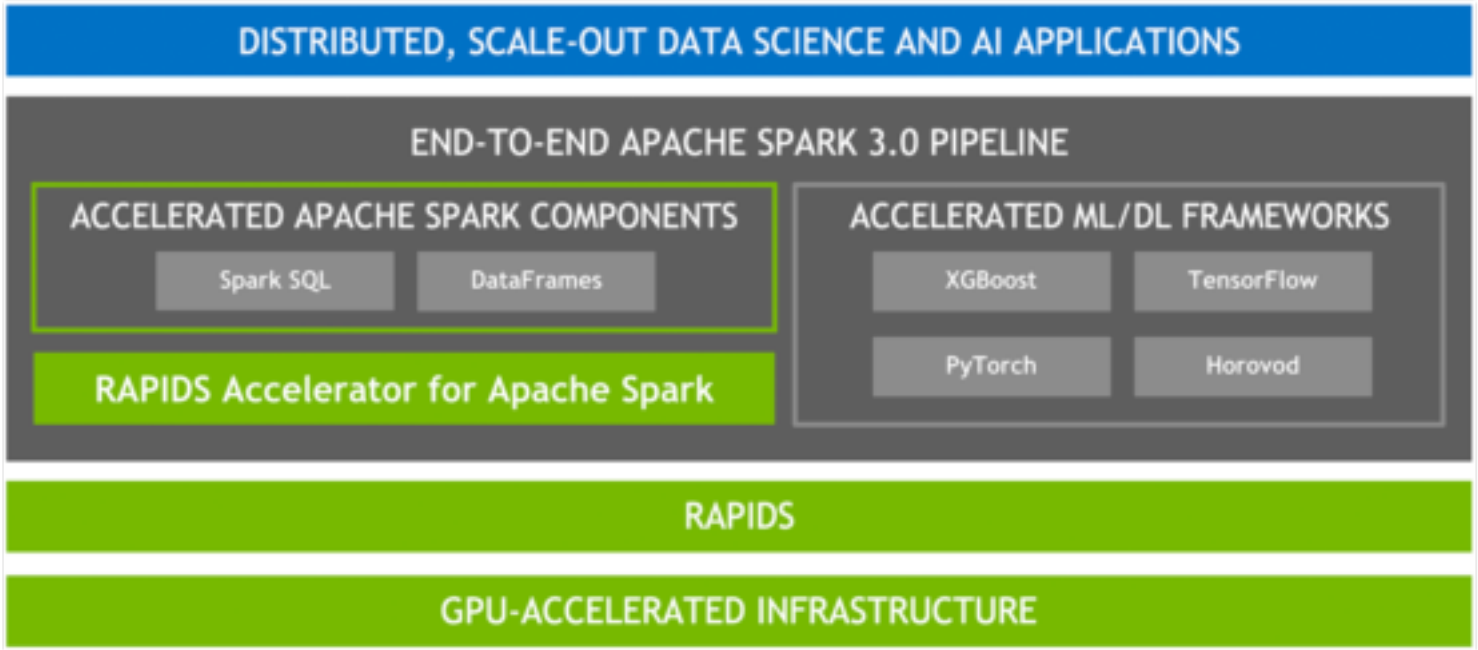


https://www.nvidia.com/ru-ru/deep-learning-ai/solutions/data-science/apache-spark-3/

RAPIDS Accelerator for Apache Spark

Про Rapids многие слышали и даже работали с ним.
Теперь Rapids подружился со Spark и у нас есть построенный поверх NVIDIA CUDA и UCX интересный RAPIDS Accelerator for Apache Spark, который позволяет выполнять SQL-операции и работать с DataFrame при помощи GPU без изменения кода + оптимизированы операции shuffle.

А дальше можно использовать на том же сервере наши любимые pytorch и прочее, строя нейроночки.



https://developer.nvidia.com/blog/optimizing-and-improving-spark-3-0-performance-with-gpus/

RAPIDS Accelerator for Apache Spark

SQL и DataFrame

Обработка идет не по строкам, а по столбцам, это более дружелюбная структура для GPU. Catalyst был изменен и при анализе задач, планов запросов он знает про RAPIDS, использую его возможности. То есть теперь мы анализируем задачу с точки зрения возможности применения GPU.
А что если памяти не хватит? Spark умный, где не хватит, перейдет на CPU.

Shuffle

Операции Spark, которые сортируют, группируют или объединяют данные по значению, должны перемещать данные между нодами при создании нового DataFrame из существующего между этапами. Shuffle включает disk I/O, сериализацию данных и network I/O. Новая реализация RAPIDS accelerator shuffle использует UCX для оптимизации передачи данных на GPU, сохраняя как можно больше данных на GPU. Он находит самый быстрый путь для перемещения данных между узлами, используя лучшие из доступных аппаратных ресурсов, включая обход центрального процессора для передачи данных с графического процессора на графический процессор внутри и между узлами.

RAPIDS Accelerator for Apache Spark

Accelerator-aware scheduling

Планирование задачи с учетом наличия ускорителя в виде GPU.
В рамках крупной инициативы Spark по лучшей унификации DL и обработки данных в Spark графические процессоры теперь являются планируемым ресурсом в Apache Spark 3.0. Это позволяет Spark планировать исполнителей с заданным количеством графических процессоров, и можно указать, сколько графических процессоров требуется для каждой задачи. Spark передает эти запросы ресурсов базовому менеджеру кластера: Kubernetes, YARN или Standalone.

Spark и numba

В курсе по GPU мы начинали с numba и разбирались, как с ее помощью писать kernel functions на GPU. Хорошая новость для тех, кто хочет писать кастомные штуки, а не использовать то, что написали за вас – функции, которые разогнаны в numba, могут использовать в Spark.
Что нужно кроме знаний по numba, чтобы запустить функции на GPU:

- ~~Загуглить;~~
- Написать функцию через @vectorize или @cuda.jit;
- Написать функцию-обертку для вызова (определить сетку вычислений, то есть количество блоков и потоков на блок);
- Применить к RDD через mapPartitions, а если несколько GPU, то через mapPartitionWithIndex.

https://on-demand.gputechconf.com/gtc/2016/presentation/s6413-stanley-seibert-apache-spark-python.pdf