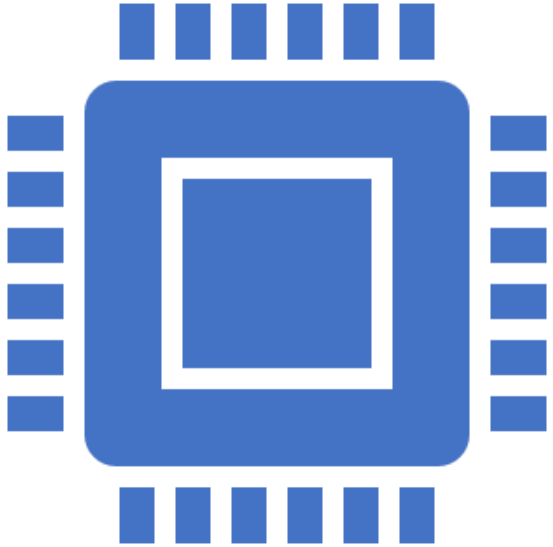


CuPy и Rapids

О чем поговорим?

- Библиотеки rapids
- CuPy и NumPy
- CuDF и Pandas
- Dask-cudf или CuDF с использованием Multi-GPU

Библиотеки rapids

1) cudf – похожая на pandas библиотека для Python, реализующая возможность работы с GPU DataFrames. В ее основе лежит Apache Arrow.
2) libcudf – это C/C++ CUDA библиотека, реализующая стандартные операции над DataFrames, являтся частью cudf.
3) cuml – библиотека, являющаяся аналогом scikit-learn, в которой реализованы многие модели машинного обучения на GPU.
4) cugraph – библиотека для работы с графами на GPU очень похожая на часто используемый аналог NetworkX, который реализован на CPU.
5) cusignal – библиотека для анализа и обработки сигналов, изображений.
6) cuspatial - обеспечивает значительное ускорение GPU для общих пространственных и пространственно-временных операций, таких как тесты точек в полигонах, расстояния между траекториями и кластеризация траекторий.
7) cuxfilter – фреймворк для web визуализации на GPU.
8) clx(cyber log accelerators) – библиотека для анализа в сфере кибербезопасности.
9) rmm(rapids memory manager) – отвечает за распределение, хранение и асинхронный доступ к данных на gpu.

CuPy

CuPy это открытая библиотека, которая очень похожа на NumPy, она также оперирует массивами. Часто чтобы переписать код с numpy на cupy достаточно просто изменить библиотеку:

cupy.ndarray : numpy.ndarray
cupy.dot : numpy.dot
cupy.sum : numpy.sum
….

Чем отличаются?

Об основных отличиях подробно можно посмотреть тут:
https://docs.cupy.dev/en/stable/reference/difference.html.

Но стоит отметить некоторые отличия:

1) Перевод из float в uint в numpy для отрицательных чисел сработает, а вот в cupy будет 0.

2) При генерировании случаных чисел через cupy.random.randn можно указать тип как float64, так и float32 (в numpy только float64).

3) Numpy не против в своих ufuncs использовать list, cupy ожидает только объекты своих типов.

По ссылке ниже есть список доступных методов в numpy и их «клоны» в cupy, если они есть. А также еще некоторые методы SciPy.
https://docs.cupy.dev/en/stable/reference/comparison.html

Немного примеров CuPy

import cupy as cp

x_gpu = cp.array([1, 2, 3]) – массив сразу будет размещен на GPU

```
np_array = np.random.random_sample(5000000)
cp_array = cp.random.random_sample(5000000)
```

```
%timeit np_array.mean()
```

```
2.4 ms ± 4.14 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
%timeit np_array.sum()
```

```
2.4 ms ± 11.7 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
%timeit -n 10 -r 3 cp_array.mean()
```

```
34.9 µs ± 6.22 µs per loop (mean ± std. dev. of 3 runs, 10 loops each)
```

```
%timeit -n 10 -r 3 cp_array.sum()
```

```
42.4 µs ± 18.6 µs per loop (mean ± std. dev. of 3 runs, 10 loops each)
```

Да, NumPy быстрый, 2.4 мс это ничто, но что если таких операций будет тысячи, миллионы?

CuPy в данном случае примерно в 60 раз быстрее. 1 час или 60 часов, что выбрать? =)

А можно ли написать универсальный код для cupy и numpy?

Да, действительно можно! cp.get_array_module определит тип входного массива и отработает как с numpy, так и cupy. Но за универсальность обычно нужно платить скоростью.

```
def logistic_numpy(y):
    return 1 / (1 + np.exp(y))

def logistic_cupy(y):
    return 1 / (1 + cp.exp(y))
```

```
%time logistic_numpy(np_array)
```

```
CPU times: user 116 ms, sys: 28 ms, total: 144 ms
Wall time: 142 ms

array([0.46104531, 0.43251947, 0.35172739, ..., 0.38658712, 0.28411481,
       0.40694806])
```

```
%time logistic_cupy(cp_array)
```

```
CPU times: user 4 ms, sys: 0 ns, total: 4 ms
Wall time: 708 µs

array([0.49949608, 0.30466445, 0.40408563, ..., 0.40086763, 0.44655752,
       0.38417976])
```

```
def univers_logistic(y):
    yp = cp.get_array_module(y)
    return 1 / (1 + yp.exp(y))
```

```
%time univers_logistic(np_array)
```

```
CPU times: user 108 ms, sys: 36 ms, total: 144 ms
Wall time: 142 ms

array([0.46104531, 0.43251947, 0.35172739, ..., 0.38658712, 0.28411481,
       0.40694806])
```

```
%time univers_logistic(cp_array)
```

```
CPU times: user 0 ns, sys: 4 ms, total: 4 ms
Wall time: 904 µs

array([0.49949608, 0.30466445, 0.40408563, ..., 0.40086763, 0.44655752,
       0.38417976])
```

CuPy и GPU

CuPy умеет производить вычисления только на 1 видеокарте. По умолчанию использует видеокарту с id 0. Но что делать, если ваш сосед занял всю видеокарту, а вторая стоит без дела? Правильно, ~~выключать скрипт соседа~~ переключаться:

```
with cp.cuda.Device(0):
    gpu_0 = cp.array([1, 2, 3, 4, 5])
with cp.cuda.Device(1):
    gpu_1 = cp.array([1, 2, 3, 4, 5])

print(f'gpu_0 device - {gpu_0.device}, gpu_1 device - {gpu_1.device}')
```

```
gpu_0 device - <CUDA Device 0>, gpu_1 device - <CUDA Device 1>
```

Но есть очевидный нюанс: нельзя производить операции с массивами, расположенными на разных GPU. Выход: ~~выключаем все~~ переносим все на одну карту.

```
with cp.cuda.Device(0):
    gpu_1 = cp.asarray(gpu_1)
```

Собственные kernel

cupy. ElementwiseKernel – это @vectorize в numba, то есть kernel, определяющий поэлеметные операции. Для его написания нужны навыки написания кода на CUDA-C/C++, но что-то простое можно и без особых навыков написать.

cupy.ReductionKernel – это kernel, реализующий подход map-reduce. Он позволяет записать на том же CUDA-C/C++ map и reduce функции, а также еще map функцию, которая будет применена после reduce.

Звучит сложно, но внутри они могут оптимизировать код за счет магии с размерностями.

@cupu.fuse()

Этот декоратор позволяет создавать Elementwise и Reduction kernels намного проще. Но в документации предупреждают, что очень сложные вещи декорировать не получится.

```
@cp.fuse()
def l2norm_dec(x):
    return cp.sqrt(cp.sum(cp.power(x, 2), axis=1))

l2norm_dec(x)
```

```
array([ 5.477226 , 15.9687195], dtype=float32)
```

```
%time cp.sqrt(cp.sum(cp.power(x, 2), axis=1))
```

```
CPU times: user 4 ms, sys: 0 ns, total: 4 ms
Wall time: 598 µs

array([ 5.477226 , 15.9687195], dtype=float32)
```

```
%time l2norm_dec(x)
```

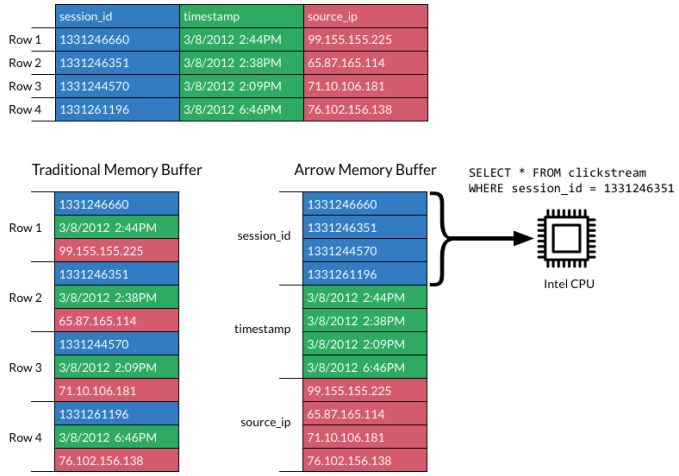
```
CPU times: user 0 ns, sys: 0 ns, total: 0 ns
Wall time: 288 µs

array([ 5.477226 , 15.9687195], dtype=float32)
```

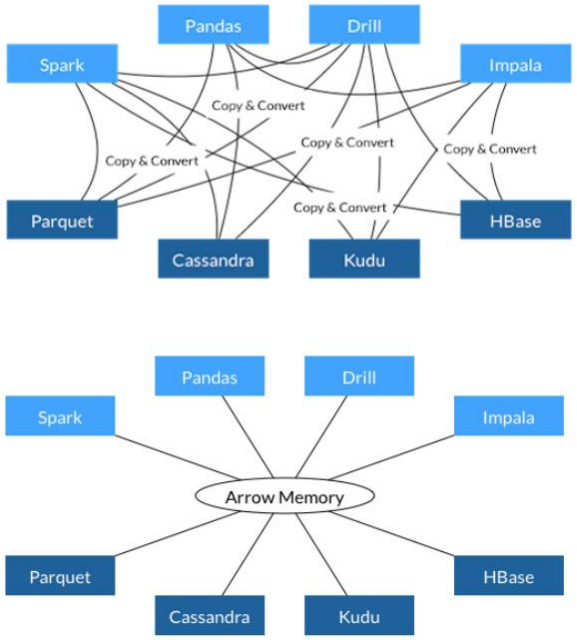
Ускорение в 2 раза благодаря одному декоратору – неплохо.

CuDF

Apache Arrow

Apache Arrow-это платформа разработки программного обеспечения для создания высокопроизводительных приложений, обрабатывающих и передающих большие данные. Она предназначен как для повышения производительности аналитических алгоритмов, так и для повышения эффективности переноса данных из одной системы или языка программирования в другую.
Иначе говоря, в основе всего лежит столбцово-ориентированный формат хранения данных в памяти, который позволяет снять накладные расходы при передаче данных от одного приложения к другому.





https://arrow.apache.org/overview/

CuDF и Pandas

Работа с cuDF почти ничего не отличается от нашего любимого pandas

```
%time df = pd.read_parquet('./data/covid.gzip')
```

```
CPU times: user 3.75 s, sys: 1.38 s, total: 5.12 s
Wall time: 2.21 s
```

```
%time male_df = df.loc[df['sex']=='Male']
```

```
CPU times: user 732 ms, sys: 128 ms, total: 860 ms
Wall time: 858 ms
```

```
%time gdf = cudf.read_parquet('./data/covid.gzip')
```

```
CPU times: user 124 ms, sys: 148 ms, total: 272 ms
Wall time: 271 ms
```

```
%time male_gdf = gdf.loc[gdf['sex']=='Male']
```

```
CPU times: user 8 ms, sys: 24 ms, total: 32 ms
Wall time: 32.8 ms
```

```
gdf
```

	current_status	sex	age_group	Race and ethnicity (combined)	hosp_yn	icu_yn	death_yn	medcond_yn
0	Laboratory-confirmed case	Male	10 - 19 Years	Black, Non-Hispanic	No	Unknown	No	No
1	Laboratory-confirmed case	Male	10 - 19 Years	Black, Non-Hispanic	No	No	No	No
2	Laboratory-confirmed case	Male	10 - 19 Years	Black, Non-Hispanic	No	No	No	No
3	Laboratory-confirmed case	Male	10 - 19 Years	Black, Non-Hispanic	Missing	Missing	No	Missing
4	Laboratory-confirmed case	Male	10 - 19 Years	Black, Non-Hispanic	No	No	No	Yes
...	...	...	...	...	...	...	...	...
8405074	Probable Case	Missing	30 - 39 Years	Unknown	No	Unknown	No	Missing
8405075	Laboratory-confirmed case	Missing	30 - 39 Years	Unknown	Missing	Missing	Missing	Missing
8405076	Laboratory-confirmed case	Missing	30 - 39 Years	Unknown	Missing	Missing	Missing	Missing

В чем отличия CuDF?

1) Нельзя писать udf-функции для столбцов типа str или category
2) В Pandas писали .apply(), теперь applymap()
3) Можно писать собственные udf не только через классический python, но и через numba
4) Applymap применяется только к 1 одному столбцу, для нескольких столбцов используется apply_rows
5) Под капотом apache arrow и работает на GPU
6) Работает с cupy

То есть отличия есть, функционал заметно уже, чем в pandas, но зато на GPU и для многих задач вполне подойдет.
Часто просто импорта библиотеки cudf и замены pd на cudf будет достаточно.

Dask-CuDF

Что такое dask-cudf?

Это расширение библиотеки dask для работы с CuDF (cupy) также, как dask работае с DataFrame из pandas. То есть он разбивает данные на партиции, размещая партиции на разных GPU. После применяет необходимые трансформации параллельно.

Опять же многое знакомо, но есть нюансы. Например, теперь все наши вычисления – это граф, как в spark. Соответственно, пока мы явно не попросим нам вывести результат или что-то вернуть, то вычисляться ничего не будет.

Нужно создать кластер и можно выбрать GPU (можно не на всю котлету заходить на сервер)

```
from dask_cuda import LocalCUDACluster
from dask.distributed import Client
import dask.dataframe as dd
import dask_cudf
from cuml.dask.common import utils as dask_utils
```

```
import subprocess
cmd = "hostname --all-ip-addresses"
process = subprocess.Popen(cmd.split(), stdout=subprocess.PIPE)
output, error = process.communicate()
IPADDR = str(output.decode()).split()[0]
cluster = LocalCUDACluster(ip=IPADDR, CUDA_VISIBLE_DEVICES=[0, 1, 2, 3])
client = Client(cluster)
client
```

«Волшебные методы»

.compute – заставит посчитать весь граф
Выполнили compute = сложили данные на одну видеокарту и тип данных уже cudf
.persist – посчитается весь граф и результат будет в памяти, а не в локальном процессе, то есть данные еще не отданы, но формально готовы. Результаты хранятся в распределенной памяти.

Когда использовать CuDF, а когда dask-cudf

Везде подход один. Если ваши данные влезают на 1 GPU, то не надо жадничать, остальные GPU оставьте другим. Часто получите только снижение производительности или ее незначительный прирост.