Big Data Management and Governance project work
A.A. 2021-2022

RAPIDS

Open GPU Data Science

Botti, Mirco (matr. 160220)
Pelacani, Francesco (matr. 161864)
Querzoli, Giulio (matr. 161654)

About RAPIDS

RAPIDS: Rapid Accelerate Platform Integrated for Data Science

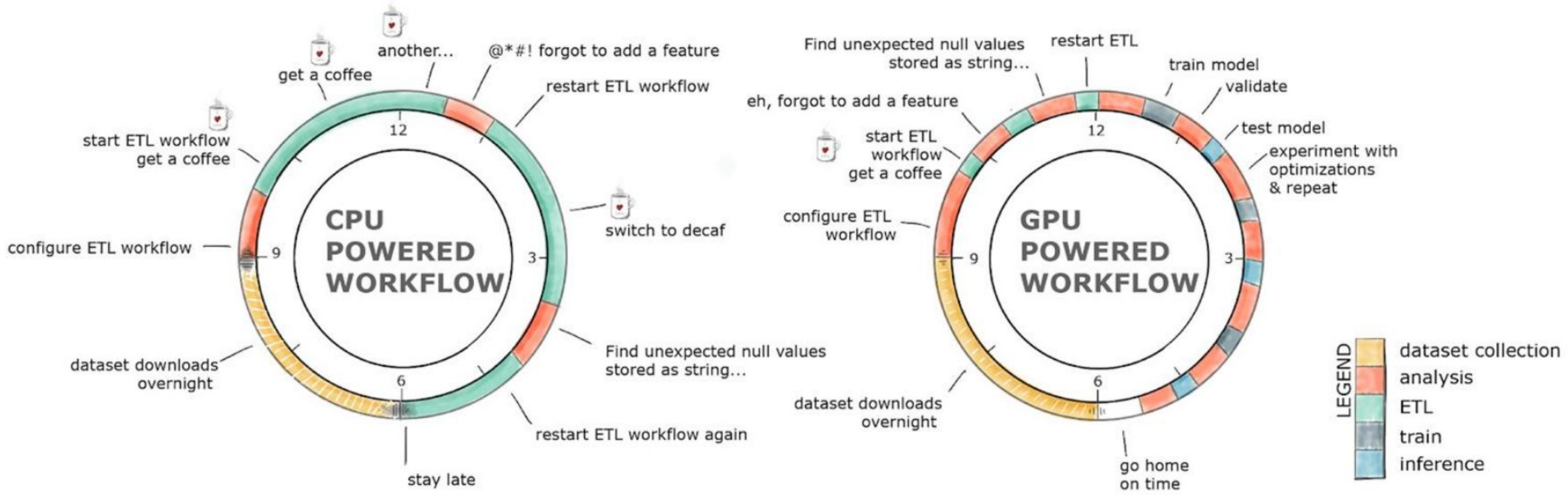
Introduction

RAPIDS is an open source software library and its APIs gives you the ability to execute end-to-end data science and analytics pipelines entirely on GPUs.

It’s licensed under Apache 2.0 and is incubated by **NVIDIA®** based on extensive hardware and data science experience. For this reason **RAPIDS** can utilize **NVIDIA** **CUDA®** primitives for low-level compute optimization, and exposes GPU parallelism and high-bandwidth memory speed through user-friendly Python interfaces.

RAPIDS also **focuses on common data preparation tasks for analytics and data science**.

Day in the life of a Data Scientist

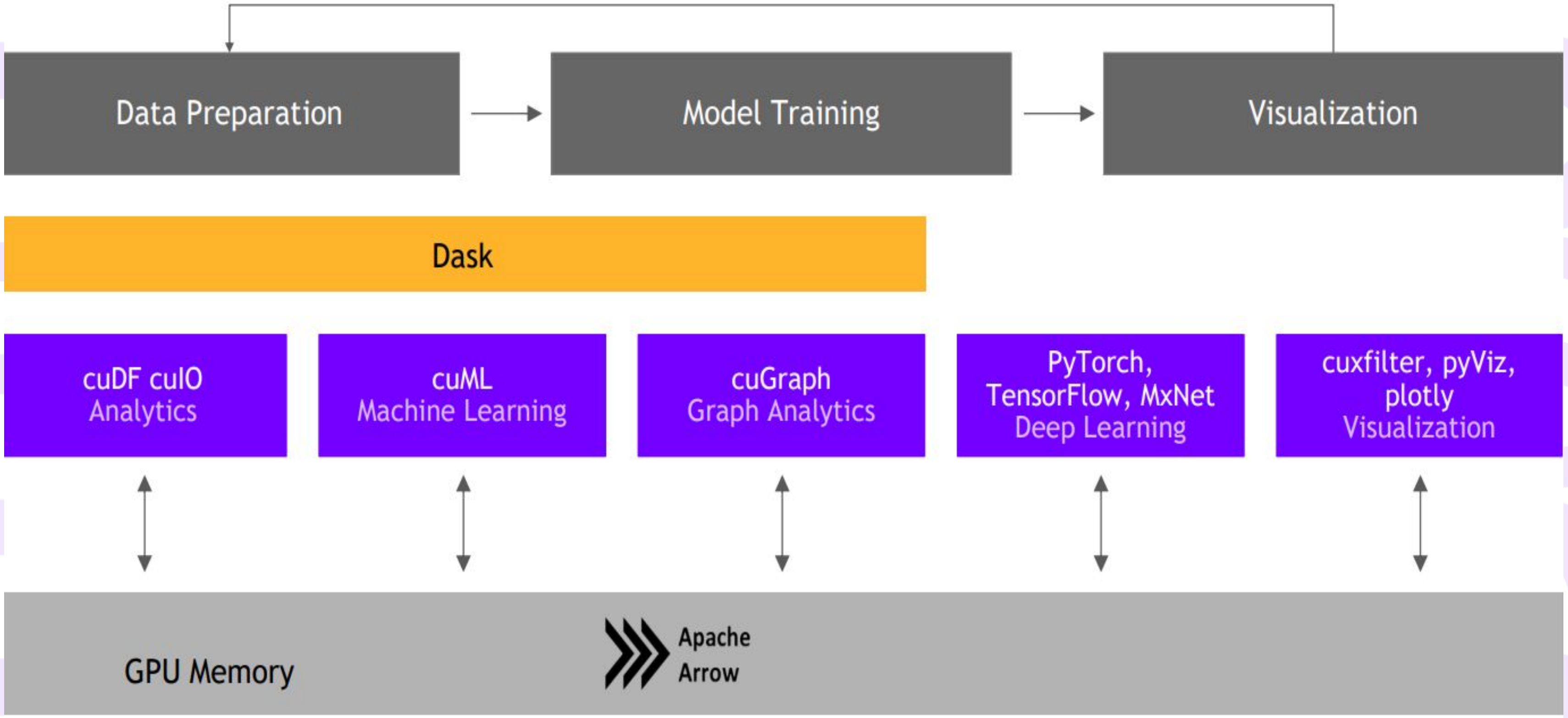


From Apache Arrow to today's Libraries

RAPIDS is based on the **Apache Arrow** columnar format, an in-memory data structure that delivers efficient and fast data interchange with the flexibility to support complex data models.

RAPIDS projects:

- **cuDF**, a pandas-like dataframe manipulation library;
- **cuML**, a collection of ML libraries that will provide GPU versions of algorithms available in scikit-learn;
- **cuGraph**, a NetworkX-like accelerated graph analytics library.

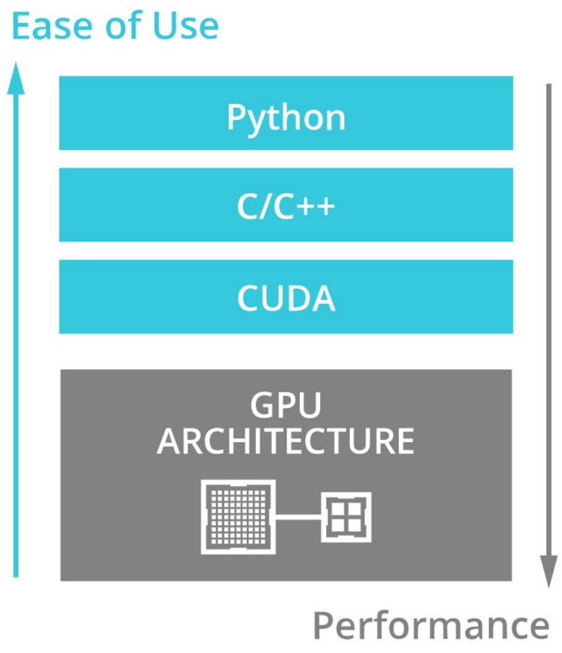


Data Preparation
Model Training
Visualization
Dask
cuDF cuIO
Analytics
cuML
Machine Learning
cuGraph
Graph Analytics
PyTorch,
TensorFlow, MxNet
Deep Learning
cuxfilter, pyViz,
plotly
Visualization
GPU Memory
Apache
Arrow

Visualization

The focus on Python allows **RAPIDS** to play well with most **data science visualization** libraries (plotly, pyViz).

For even greater performance, the developing team is continuously working towards deeper integration with these libraries since a native GPU in-memory data format provides high-performance, high-FPS data visualization capabilities, even with very large datasets.



cuDF & Dask-cuDF

What is cuDF?

cuDF is a Python GPU DataFrame library (built on the Apache Arrow columnar memory format) for loading, joining, aggregating, filtering, and otherwise manipulating tabular data using a DataFrame style API.

What is Dask-cuDF?

Dask-cuDF extends Dask to allow its DataFrame partitions to be processed by cuDF GPU DataFrames as opposed to Pandas DataFrames.

	Loads data larger than CPU memory	Scales to multiple CPUs	Uses GPU acceleration	Loads data larger than GPU memory	Scales to multiple GPUs
Pandas	✖	✖	✖	✖	✖
Dask Dataframe	✓	✓	✖	✖	✖
RAPIDS (cuDF)	-	-	✓	✖	✖
RAPIDS (cuDF) + Dask Dataframe	-	-	✓	✓	✓

When to use them?

If your workflow is fast enough on a single GPU or your data comfortably fits in memory on a single GPU, you would want to use cuDF; while **If you** want to distribute your workflow across multiple GPUs, **have more data than the amount that you can fit in memory on a single GPU**, or want to analyze data spread across many files at once, **you would want to use Dask-cuDF**.

What to use in our specific case?

Since performing even simple operation on our ~5GB dataset caused several time memory errors, we decided to work on it using **Dask-cuDF**.

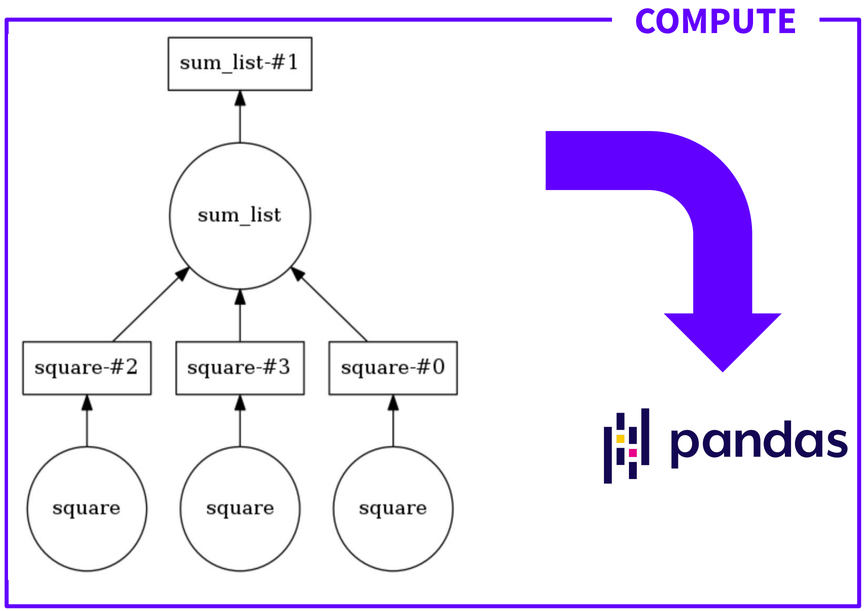
Dask-cuDF behaviour

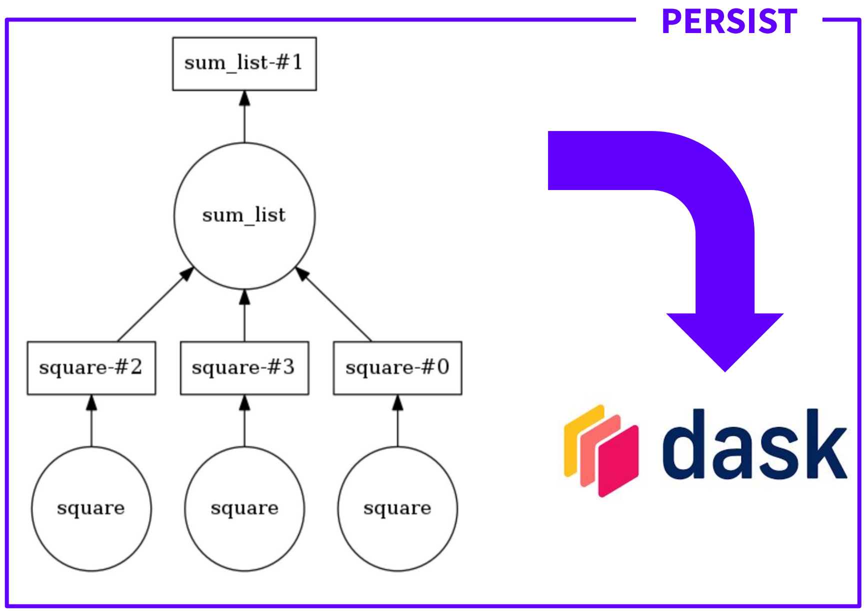
Dask operations are **lazy**. Instead of being executed at that moment, most operations are added to a task graph and the actual evaluation is delayed until the result is needed.

If we want to force the execution of operations we can call `persist` on a Dask collection and it will fully compute it (or actively computes it in the background), persisting the result into memory.

When using distributed systems, we may want to wait until `persist` is finished before beginning any downstream operations. We can enforce this by using `wait`. Wrapping an operation with `wait` will ensure it doesn’t begin executing until all necessary upstream operations have finished.

Dask-cuDF Key Feature





Map partition

Row-level parallelism operations should be executed with map_partition, because each partition is distributed on different GPUs of the cluster.

Map_partition applies the specified function on every partitions.

It’s possible to set either block size or partitions’ number in the configuration. According to the documentation, to obtain better performance, we decided to specify a 256 MB blocksize.

Our study case

To create a data preparation pipeline with which analize and work on a massive dataset in order **to obtain as output 3 clean DataFrames** (in which every column has the correct data type):

1. **Customer**: with customer's data.
2. **Utilities**: utility's data with a reference to the customer for each utility.
3. **Invoices**: invoice's data with a reference to the utility for each invoice.

First Step - Setup

“Put the pen down and think.”

We first studied the dataset (a sub-sample of it) and its columns, in order to familiarize with the data contained in it.

We noticed that:

1. Very big data set, memory-fitting problems
2. Some columns have a different data type with respect to data contained
3. Huge number of columns, some seemed duplicated
4. Some columns seem to be derived from others
5. Hardware and software requirements

Second Step - Tools' preparation

After having become acquainted with the functions present in the **Dask-cuDF/cuDF libraries documentation** (most of which follow the syntax of Pandas) we first implemented the class of functions contained in the `base.py` file following the given template.

We paid a lot of attention about the parameters of the functions and we implemented them using the appropriate syntax.

We also added some extra functions that helped us through the code development.

Major Problem encountered during first stages

During these very first stages, since Dask-cuDF is currently under-developed, we noticed that it didn’t meet our needs in term of implemented utility functions to perform operations over the dataset.

In particular, we struggled during the date and string to numeric conversion with dask-cuDF.

Major Problem encountered during first stages

Despite the use of map_partition, we encountered Dask-cudF’s metadata problems and/or functions not implemented.

Also, the documentation relative to Dask-cuDF is very poor, at the time of writing.

For these reasons we decided to move to **Dask** for the pre-processing phase.

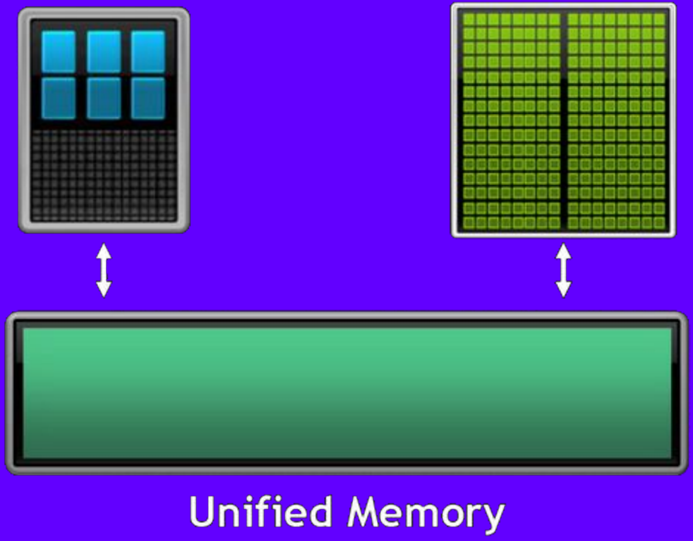
Dask uses a collection of pandas DataFrame that rely on CPUs instead of GPUs.

RAPIDS
dask
RAPIDS

Third Step / Dask-cuDF deep-dive

Using `LocalCUDACluster` we create a one dask-worker per GPU on the local machine.

Then, we set up a GPU cluster. With our `client` set up, Dask-cuDF computation will be distributed the cluster.



Data Preparation Pipeline (1)

Ingestion & Discovery

- Load dataset from source .csv and convert to .parquet
- Locate missing values and "hot" cols
- Locate outliers in numerical columns

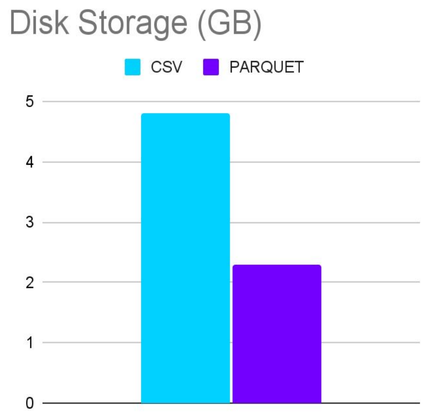
Validation

- Check data range (Age, Sex, …)
- Check column uniqueness (repeated data cols and gas_average_cost)
- Find data-mismatched data types

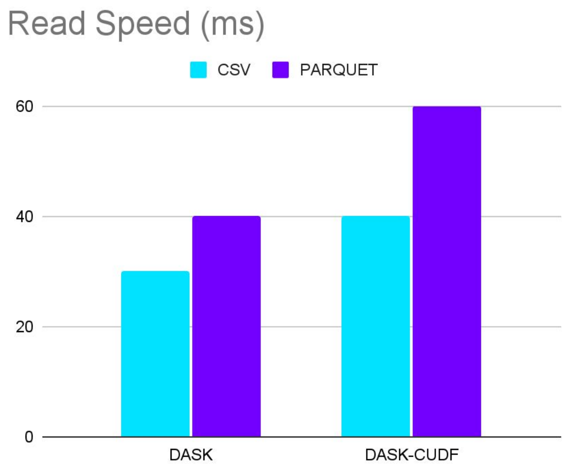
Structuring

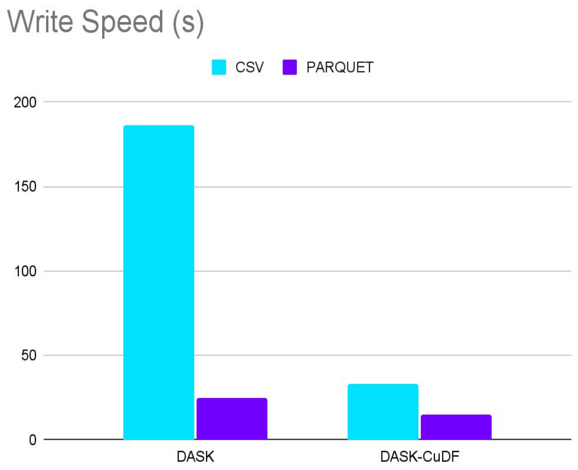
- Change column data types
- Delete (duplicates), split (emission date, howmuch_pay) and merge (F1, F2, F3)
- Pivot (supply_type x sex, cosumption x year)

Apache Parquet



lower is better





Enrichment

Calculated columns

- **light_consumption** as sum **Fx_kWh**
- **average_gas** and **light_cost** as ratio **consumption** over **amount** increasing precision

Primary key

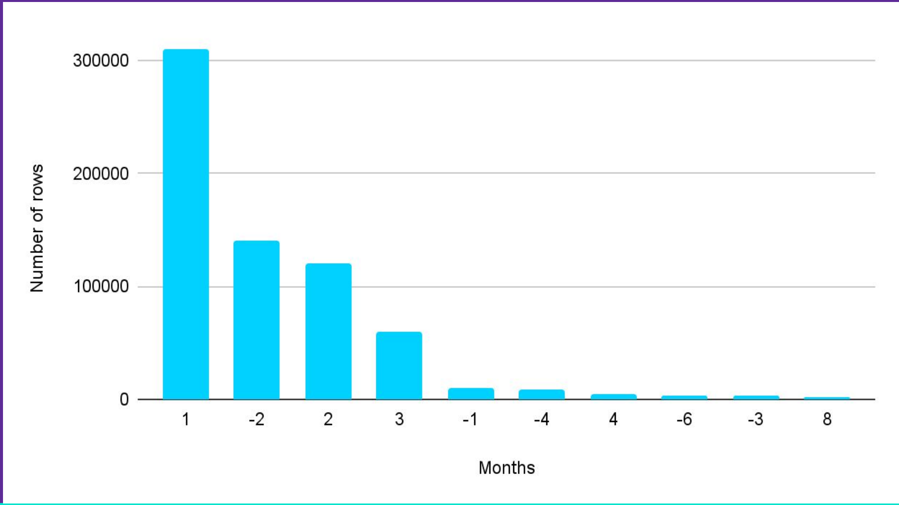
- **customer_code** and **user_code** for the utility table
- **bill_id** and **emission_date** for invoice

Encoding

- **sex**
- **supply_type**
- **bill_type**
- **billing_frequency** only with **map()**

Billing frequency Analysis

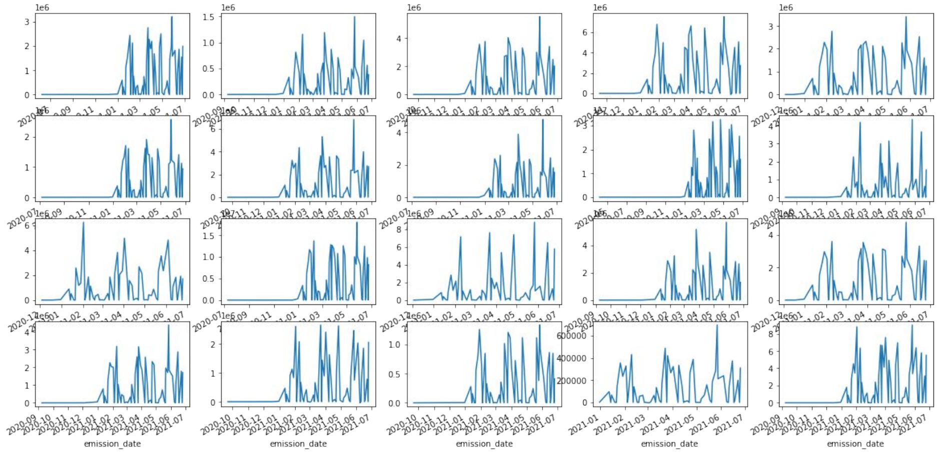
```
# Map billing frequency with real numbers
num = {
    'bimester': 2,
    'quarterly': 4,
    'monthly' : 1,
}
base['b_freq'] = base['billing_frequency'].map(num)

# Difference between end and start in month
date_diff = ((base['gas_end_date'] - base['gas_start_date'])
date_diff = date_diff / np.timedelta64(1, 'M')).round()
val_count = (base['b_freq'] - date_diff).value_counts().compute()

val_count[(val_count > 1000) & (val_count < 1000000)].plot(kind='bar')
```



Enrichment

OPEN DATA links cities with regions and provinces

GEOSPATIAL analysis → consumes place related



Data Preparation Pipeline (2)

Enrichment

- Calculated columns
- Primary key setup for the 3 new tables
- Scale column values into a certain range
- Encode categorical data (one-hot encoding and label encoding)

Filtering

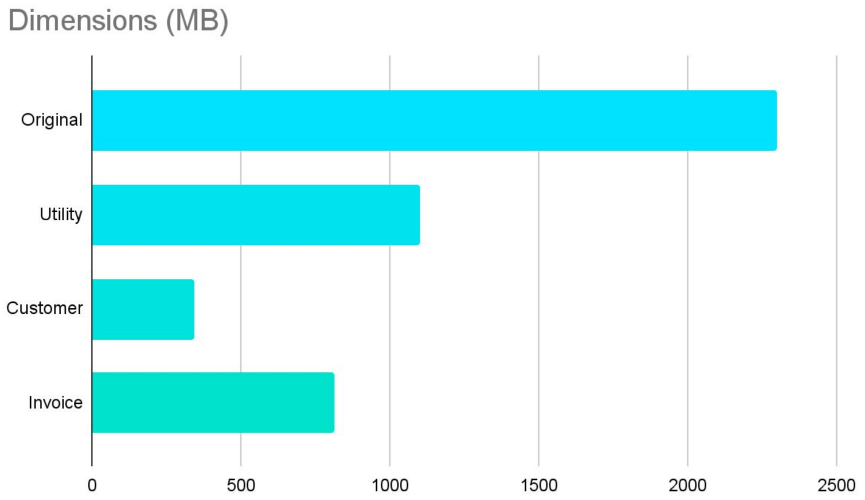
- New **utility**, **customer** and **invoice** tables from subset of rows

Cleaning

- Unit of measurement removal
- ITA to ENG
- String stripping
- CAPS LOCK removal
- **age** null value to PIVA

Final Results & Comments

Results



We obtained 3 cleaned datasets saved in parquet format:

1. **Utility dataset**
2. **Customer dataset**
3. **Invoice dataset**

Appendix I - Memory

```
MemoryError: std::bad_alloc: CUDA error at: /opt/conda/envs/rapids/include/rmm/mr/device/cuda_memory_resource.hpp
```

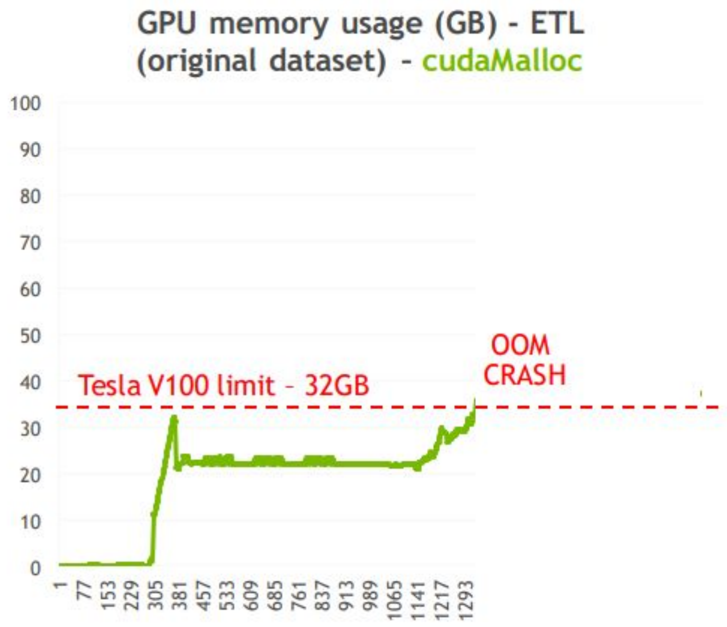
MemoryError occurred too many times

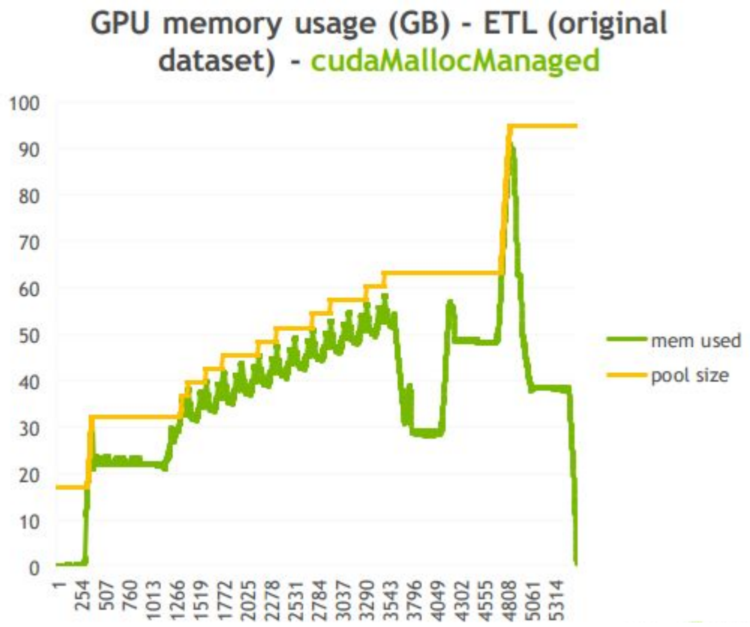
16GB of memory → 7GB already taken

OFFLOADING always require to free memory explicitly

MANAGED MEMORY → easier memory management

Appendix I - Memory





Appendix I - Memory

Memory bottleneck

GPU utilization doesn't exceed 20%

GPUs are useful in **ALU**-bounded applications

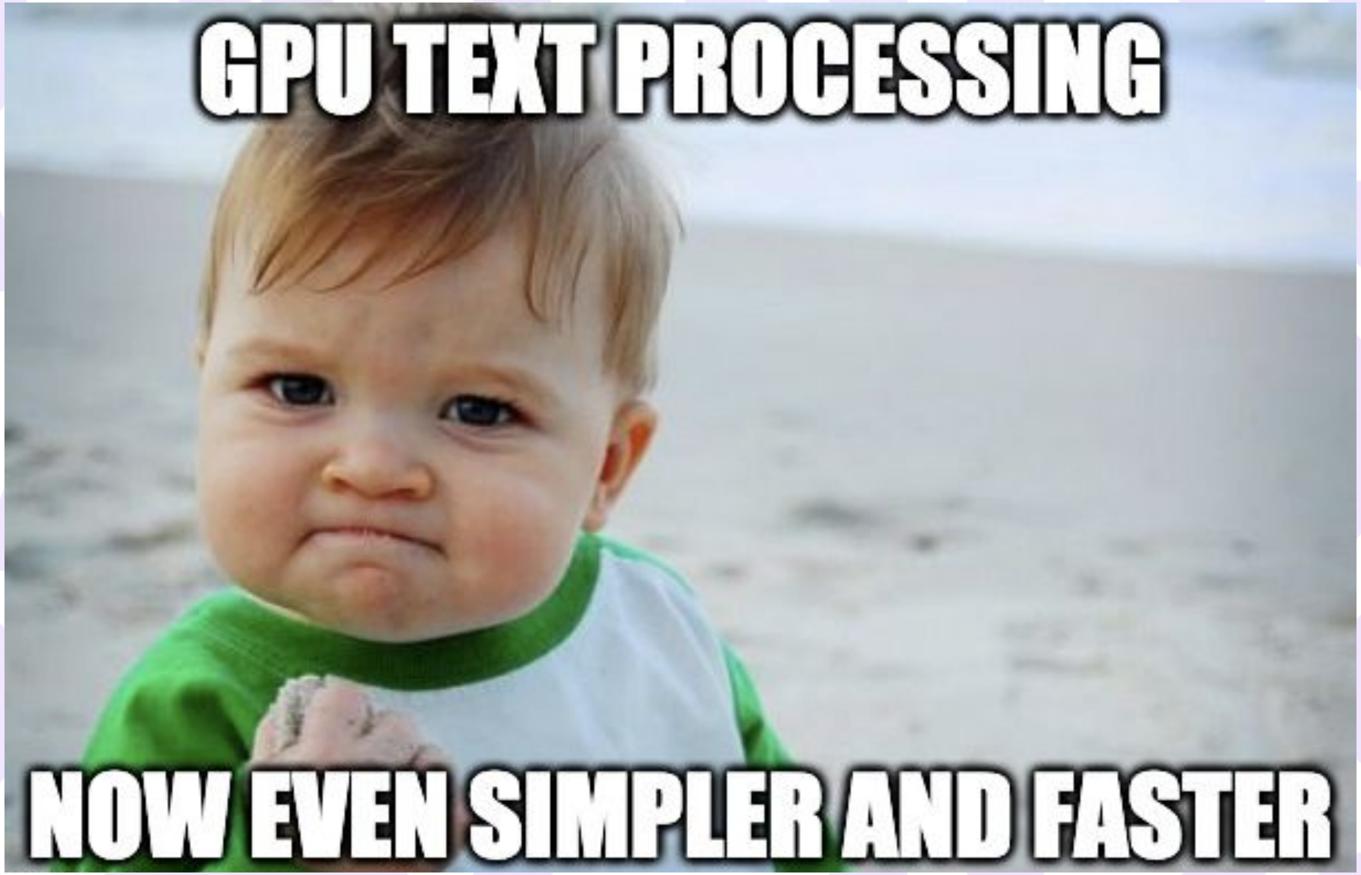
This is not the case!

Appendix II - String

Strings’ variable length break **LOCKSTEP**

BUT fully-featured string and regular expression processing

cuDF works well also with **WEAK-SCALING**


GPU TEXT PROCESSING
NOW EVEN SIMPLER AND FASTER

Appendix III - Mask

Several cuDF’s functions create a variety of problems

Action under the hood → **map_partition()**

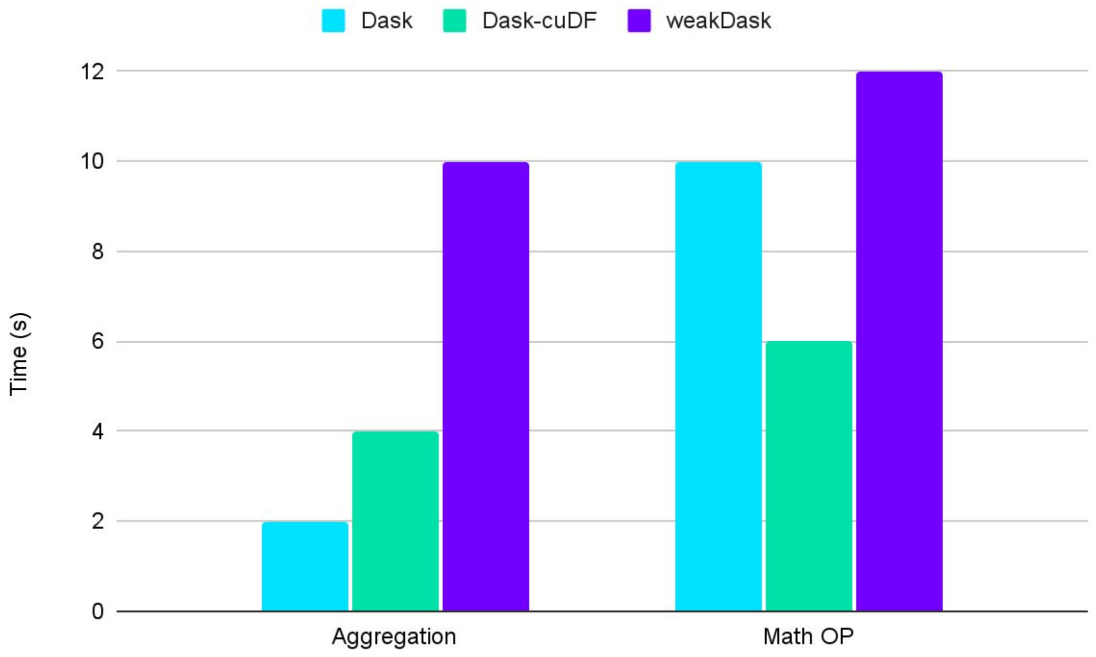
mask() does not work properly in cuDF

Programming model downside

```python
any_na = base[['F1_kWh', 'F2_kWh', 'F3_kWh']].isna().sum(axis=1) >= 1

def mean_sub(df):

    any_na = df[['F1_kWh', 'F2_kWh', 'F3_kWh']].isna().sum(axis=1) >= 1

    D = df[['F1_kWh', 'F2_kWh', 'F3_kWh']][any_na].isna().sum(axis=1)
    N = df['light_consumption'][any_na] - df[['F1_kWh', 'F2_kWh', 'F3_kWh']][any_na].fillna(0).sum(axis=1)

    replace = df['F1_kWh'].isna()
    df['F1_kWh'][replace] = N[replace] / D[replace]
    replace = df['F2_kWh'].isna()
    df['F2_kWh'][replace] = N[replace] / D[replace]
    replace = df['F3_kWh'].isna()
    df['F3_kWh'][replace] = N[replace] / D[replace]

    return df

columns = ['F1_kWh', 'F2_kWh', 'F3_kWh','light_consumption']
base[columns ] = base.df.map_partitions(mean_sub, meta=base.df)[columns]
base[columns ][any_na][columns ].compute()
```

PERFORMANCE

lower is better



Final Considerations

GPUs → **EMBARRASSINGLY PARALLEL** workload

Aggregation function works better on **DASK**

Strong-scaling operation works well on **CUDF**

Evaluate the tradeoff!

The End

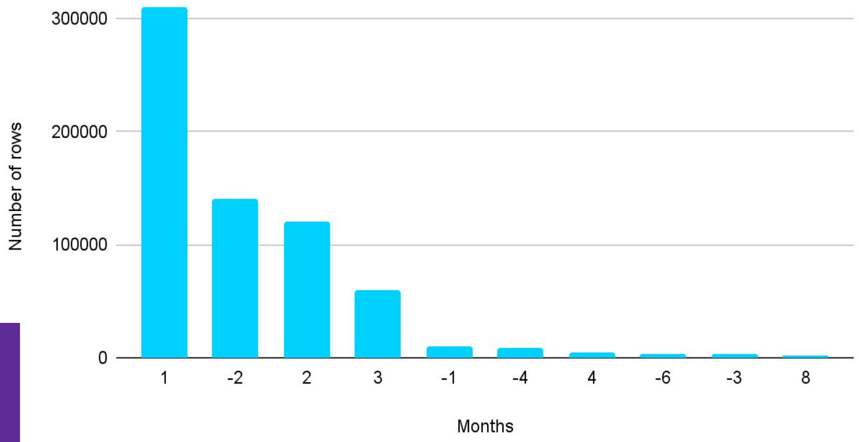
Thanks for your attention

References

1. RAPIDS Documentation
2. Dask Documentation
3. What is the difference between Dask and RAPIDS?
4. RAPIDS AI Conf 02-07-2019 presentation
5. RAPIDS: the platform inside and out - Joshua Patterson

Billing frequency Analysis

```
# Map billing frequency with real numbers
num = {
    'bimester': 2,
    'quarterly': 4,
    'monthly' : 1,
}
base['b_freq'] = base['billing_frequency'].map(num)

# Difference between end and start in month
date_diff = ((base['gas_end_date'] - base['gas_start_date'])
date_diff = date_diff / np.timedelta64(1, 'M')).round()
val_count = (base['b_freq'] - date_diff).value_counts().compute()

val_count[(val_count > 1000) & (val_count < 1000000)].plot(kind='bar')
```



Notes

We noticed that:

- Rows with nan dates all have a bill_type = ‘change note correction’
- Some records are from people had age <18, we casted them at 18
- We couldn’t populate missing anagraphic data using data from other records referred to the same person
- We executed a strip passage over all text attributes, to uniform them
- We removed unit measures from number attributes
- Some columns are duplicated (all dates and gas_average_cost)

PERFORMANCE

Prima di dire i risultati direi il set up della macchina:
1 NVIDIA T4 da 16 GB vs 24 CPU con 187 GB di RAM
Config bonus (LIMIT_config) potrebbe essere 4 cpu con 1 thread e 16 gb di ram (ocio che che forte comunque)

- Verifica quanti light_consumption non coincidono per ogni tipo di supply type: base[(base['light_consumption'] != (base['F1_kWh'] + base['F2_kWh'] + base['F3_kWh']))]['supply_type'].value_counts().compute() → DASK 12s, DASK_CUDF 6s, LIMIT_config 10.2 s
- Restituisce, in base alla regione, il consumo totale di luce e gas: base[['Region', 'light_consumption', 'gas_consumption']].groupby(['Region']).sum().compute()
  base.groupby(['Region'], sum)[['light_consumption', 'gas_consumption']].compute()
  Le 2 funzioni sopra fanno la stessa cosa, la prima impiega 2 secondi la seconda più di 15 min (possiamo anche dire 2 ordini di grandezza in più)
- Mappa in modo differente sex e poi ne conta i valori (fatto in modo puramente dimostrativo): base['sex'].map({'M': 'male', 'F': 'female', 'P': 'PIVA'}).value_counts().compute()→ DASK 1.88 s, DASK_CUDF 4.2 s, LIMIT_config 9.5 s
- Per ogni regione, per anno e mese di emission date fa vedere il consumo della luce e gas: base[['Region', 'MONTH_em', 'YEAR_em', 'light_consumption', 'gas_consumption']].groupby(['Region', 'YEAR_em', 'MONTH_em']).sum()[['light_consumption', 'gas_consumption']].compute() → DASK 6.26 s, DASK_CUDF 8.7 s, LIMIT_config 9s

CONFIGURATION

Setup of the environment used:
Nr. 1 NVIDIA T4 with 16 GB RAM
Nr. 24 CPU
187 GB RAM

Bonus Config (weakDask):
4 CPU with 16 GB RAM