ANACONDA

Building a Distributed GPU DataFrame with Python

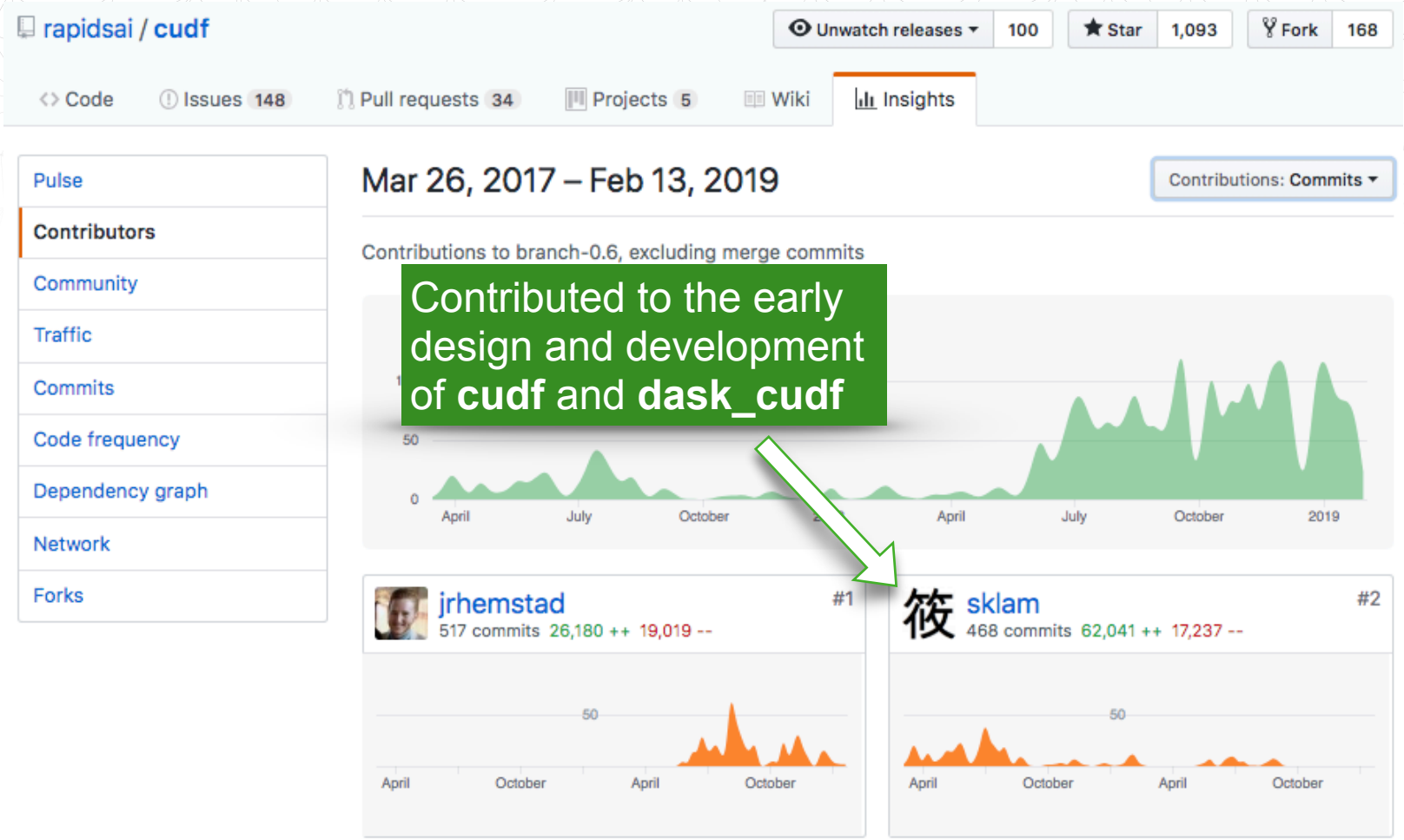
Siu kwan Lam

Agenda

- What's the GPU Open Analytics Initiative (GOAI)?
- The GPU DataFrame
- Scaling Out: Distributed GPU DataFrame

My Background

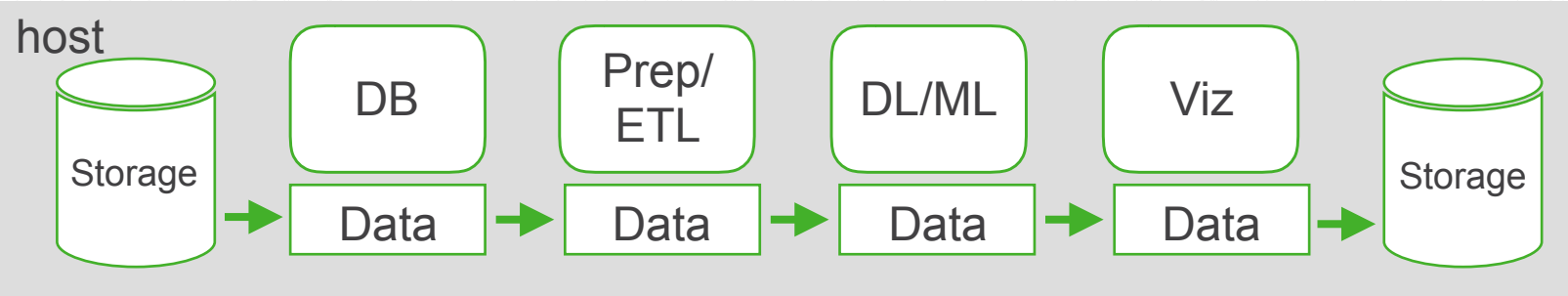
- Software Engineer at Anaconda
- Lead Developer of Numba
- 6+ years in Python + JIT + GPU


rapidsai / cudf
Unwatch releases 100
Star 1,093
Fork 168
Code
Issues 148
Pull requests 34
Projects 5
Wiki
Insights
Pulse
Contributors
Community
Traffic
Commits
Code frequency
Dependency graph
Network
Forks
Mar 26, 2017 – Feb 13, 2019
Contributions: Commits
Contributions to branch-0.6, excluding merge commits
Contributed to the early design and development of cudf and dask_cudf
50
0
April
July
October
April
July
October
2019
jrhemstad
#1
517 commits 26,180 ++ 19,019 --
50
April
October
April
October
筱 sklam
#2
468 commits 62,041 ++ 17,237 --
50
April
October
April
October

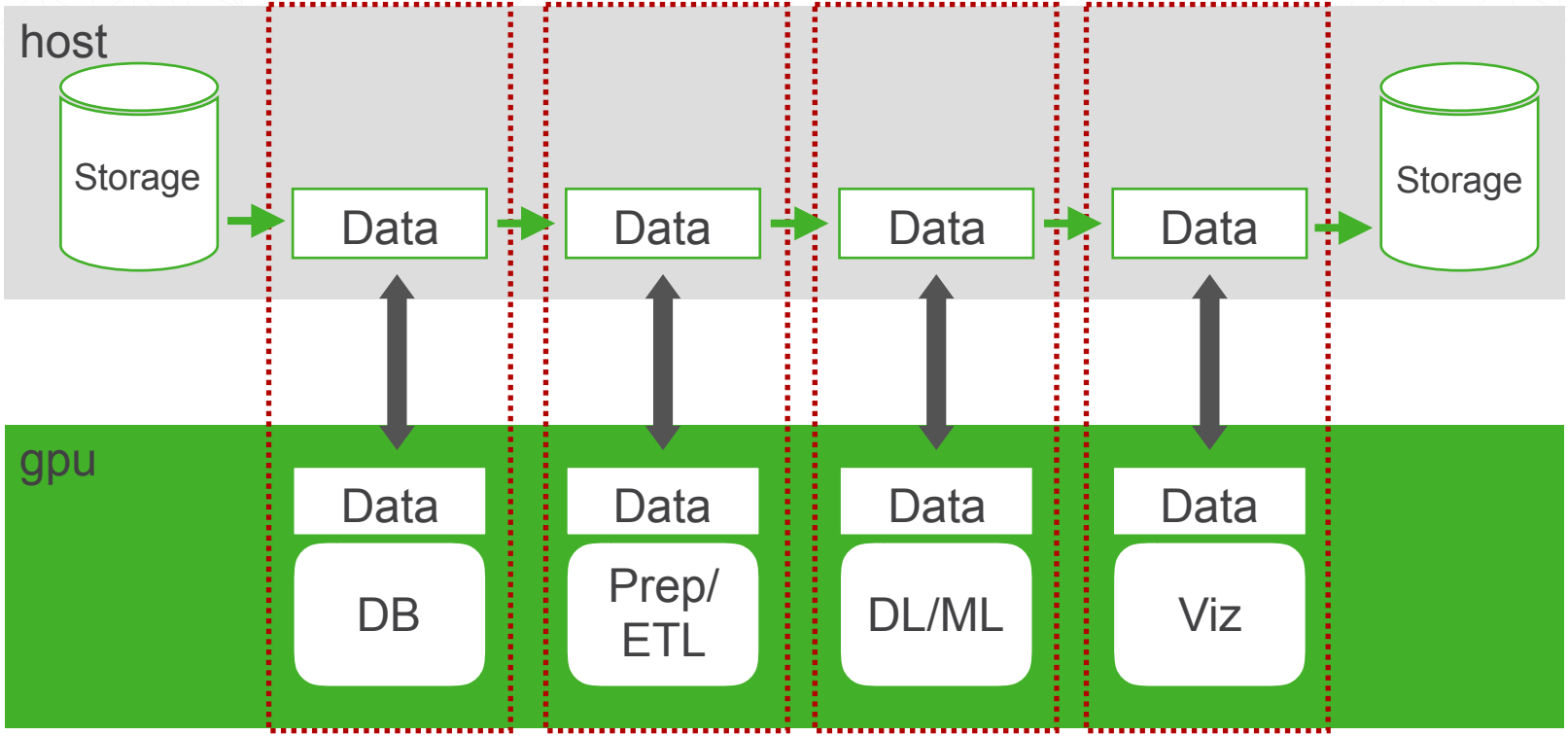
GOAI

“We want to make accelerated, end-to-end GPU analytics easy.”

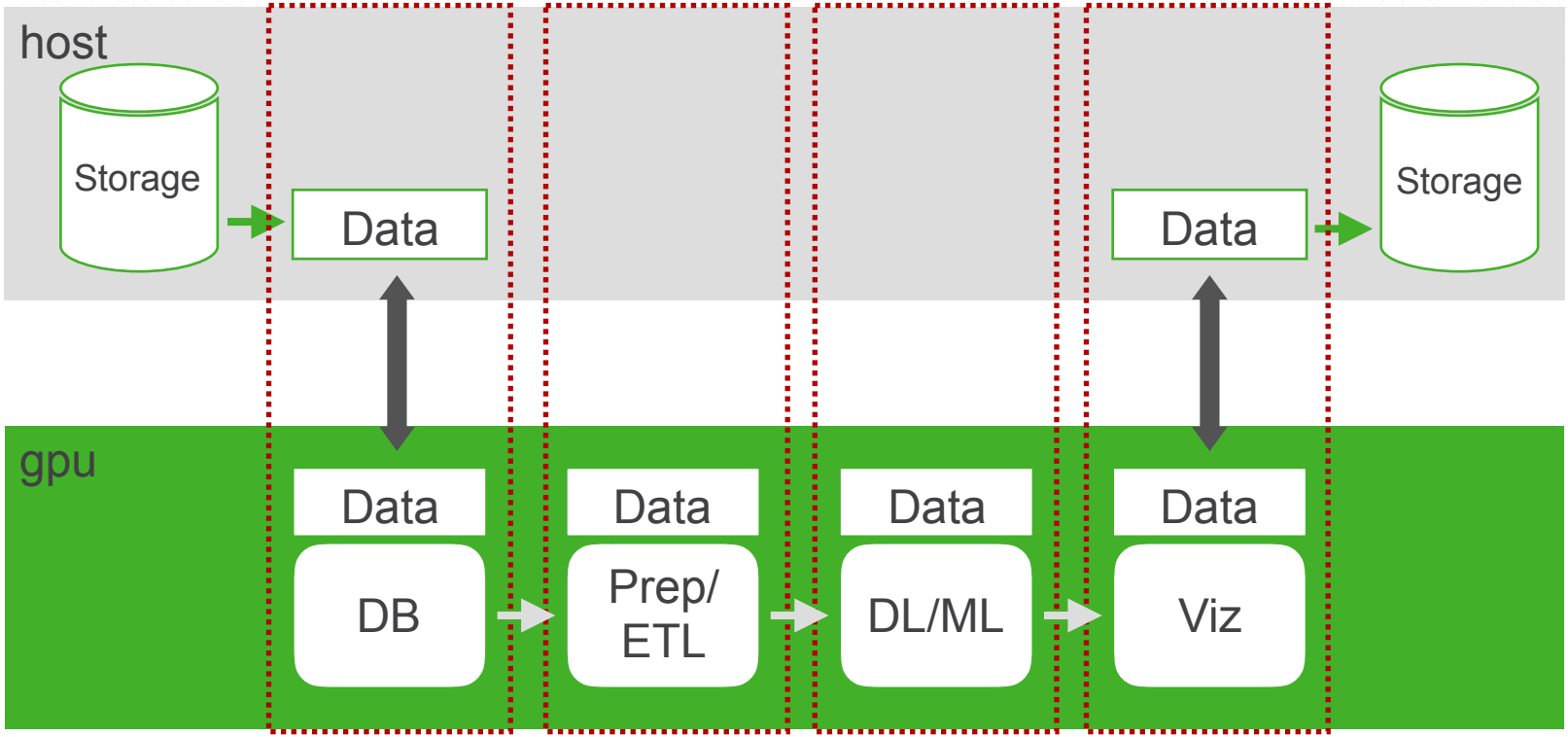
Analytic Pipeline Before GPU-ization



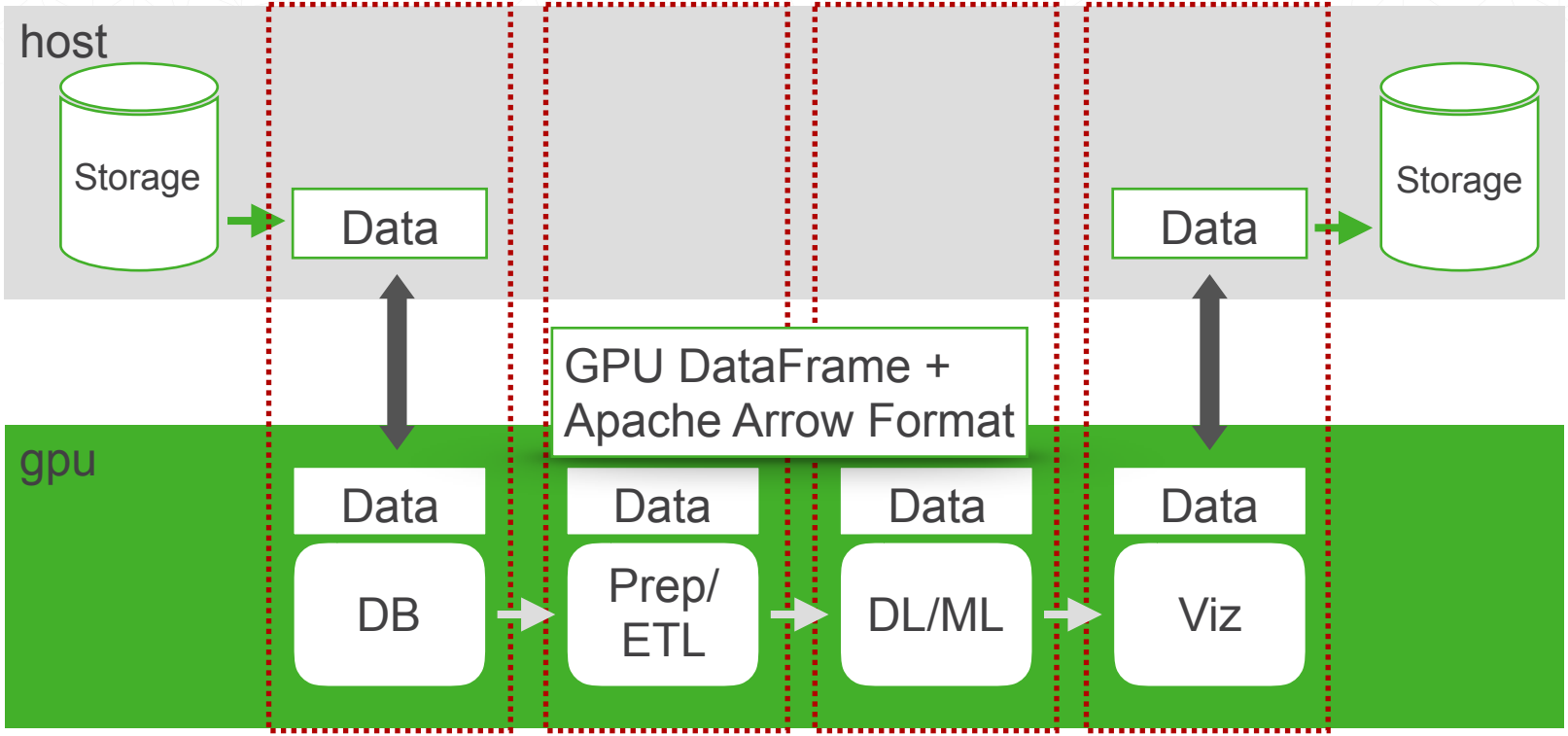
GPU-ization & Silos



Solution: Share GPU Data



Solution: Share GPU DataFrame



Other Needs

- GPU-aware distributed computing frameworks
- Out-of-core capability in task scheduling
  - GPU memory is limited
- Smart GPU data transfer
  - PCI-express, NVLINK topology aware

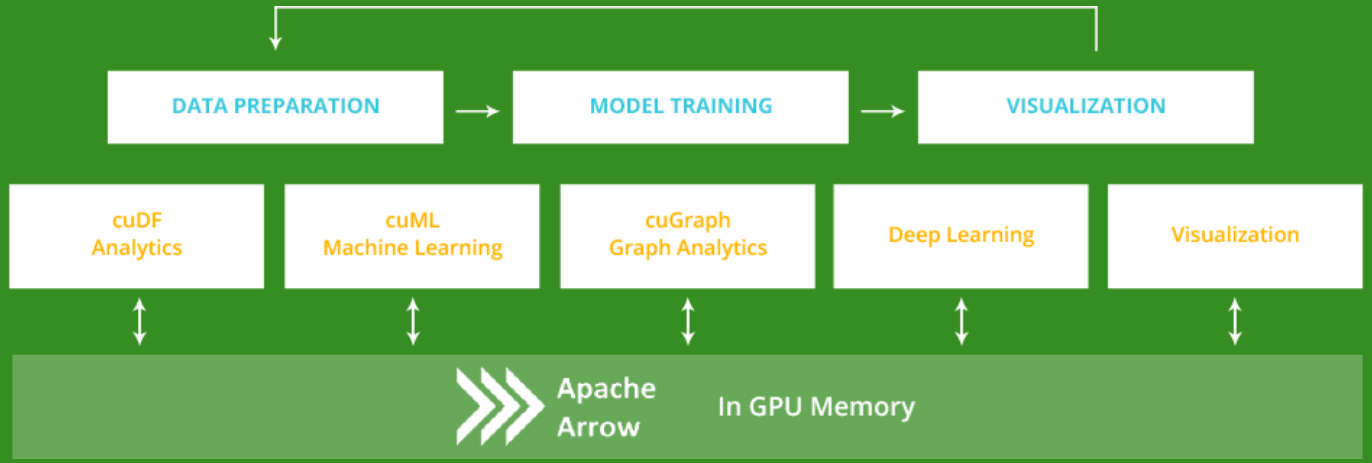
A Brief GOAI History: 2017

- 2017:
  - Prototyped **pygdf**, a **pandas**-like GPU DataFrame
  - Prototyped **dask_gdf,** a distributed **pygdf**
  - Ingest data from **MapD** into **PyGDF** and do ETL
  - Export feature matrices to **H2O GLM**

A Brief GOAI History: 2018

- 2018
  - Nvidia created **RAPIDS**
    - **pygdf** evolved into **cudf**
    - **dask_gdf** evolved into **dask_cudf**

RAPIDS


DATA PREPARATION
MODEL TRAINING
VISUALIZATION
cuDF
Analytics
cuML
Machine Learning
cuGraph
Graph Analytics
Deep Learning
Visualization
Apache
Arrow
In GPU Memory

GPU DataFrame

What's a DataFrame?

- table of data
- column ~= array
- python: pandas
- important ops:
  - read_csv
  - groupby
  - join

```
In [1]: import pandas as pd

In [2]: df = pd.DataFrame()

In [3]: df['kinds'] = ['fruits', 'vegetables', 'meat']
        df
```

Out[3]:

	kinds
0	fruits
1	vegetables
2	meat

```
In [4]: df['count'] = [10, 5, 4]
        df
```

Out[4]:

	kinds	count
0	fruits	10
1	vegetables	5
2	meat	4

GPU DataFrame Design

Start Simple

- Single machine
- Single GPU
- Single process
- In-GPU memory

GPU DataFrame Design

Start Simple

- Single machine
- Single GPU
- Single process
- In-GPU memory

Easy to Learn

- Mirror the **Pandas** API

GPU DataFrame Design

Start Simple

- Single machine
- Single GPU
- Single process
- In-GPU memory

Easy to learn

- Mirror the **Pandas** API

Build on Opensource

- **Apache Arrow** data format
- **Numba** for GPU Python JIT

Main DataTypes are Numbers

- GPU very good with numbers
- Basic: **int{1, 8, 16, 32, 64}**, **float{16, 32, 64}**
- **Datetime** encoded in int64 + unit [s, ms, us, ns]
- Strings encoded as **Categoricals**
- Bitmask for **Missing data**

Use Arrow

- Use Arrow format to share data
  - in-memory
  - columnar
  - no-copy
  - open-standard
    - spark, hadoop, parquet, impala, etc..

Using Numba

- Use Numba for GPU functions and GPU arrays
- JIT user-defined functions as GPU kernels
- Interactive CUDA programming in notebooks

Numba CPU Demonstration: 100x speedup in 3 lines

In [*]:

```
import numpy as np
import math


def cnd(d):
    "Cumulative normal distribution"
    A1 = 0.31938153
    A2 = -0.356563782
    A3 = 1.781477937
    A4 = -1.821255978
    A5 = 1.330274429
    RSQRT2PI = 0.39894228040143267793994605993438
    K = 1.0 / (1.0 + 0.2316419 * math.fabs(d))
    ret_val = (RSQRT2PI * math.exp(-0.5 * d * d) *
               (K * (A1 + K * (A2 + K * (A3 + K * (A4 + K * A5))))))
    if d > 0:
        ret_val = 1.0 - ret_val
    return ret_val


def cnd_array(arr):
    out = np.zeros_like(arr)
    for i in range(arr.size):
        out[i] = cnd(arr[i])
    return out

# Time the execution
arr = np.random.random(100000)
%timeit cnd_array(arr)
```

Simple Groupby

Most of the time, users do not need to know about the GPU.

But, we still provide a way…(next slide)

Grouping hierarchically then applying the `sum` function to grouped data.

```
[21]: print(df.groupby(['agg_col1', 'agg_col2']).sum())
```

```
      agg_col1 agg_col2 sum_a sum_b sum_c
0            0        0    73    60    73
1            0        1    27    30    27
2            1        0    54    60    54
3            1        1    36    40    36
```

Grouping and applying statistical functions to specific columns, using `agg`.

```
[22]: print(df.groupby('agg_col1').agg({'a':'max', 'b':'mean', 'c':'sum'}))
```

```
      agg_col1 mean_b sum_c max_a
0            0      9   100    19
1            1     10    90    18
```

Advanced Groupby

Numba **compiles** this python function as a **CUDA device function** that operates on each group

```
from cudf import DataFrame
from numba import cuda
import numpy as np

df = DataFrame()
df['key'] = [0, 0, 1, 1, 2, 2, 2]
df['val'] = [0, 1, 2, 3, 4, 5, 6]
groups = df.groupby(['key'], method='cudf')

# Define a function to apply to each group
def mult_add(key, val, out1, out2):
    for i in range(cuda.threadIdx.x, len(key), cuda.blockDim.x):
        out1[i] = key[i] * val[i]
        out2[i] = key[i] + val[i]

result = groups.apply_grouped(mult_add,
                              incols=['key', 'val'],
                              outcols={'out1': np.int32,
                                       'out2': np.int32},
                              # threads per block
                              tpb=8)

print(result)
```

Output:

```
   key  val out1 out2
0    0    0    0    0
1    0    1    0    1
2    1    2    2    3
3    1    3    3    4
4    2    4    8    6
5    2    5   10    7
6    2    6   12    8
```

10x speedup by changing import from pandas to cudf
CPU: Core i7 | GPU: Titan XP

In [1]:
```
from pandas import DataFrame
import numpy as np

df = DataFrame()

nelem = 10**8
df['key1'] = np.random.randint(0, 100, nelem)
df['key2'] = np.random.randint(0, 3, nelem)
df['val'] = np.random.random(nelem)

%timeit df.groupby(['key1', 'key2']).sum()
```

```
10.3 s ± 31 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

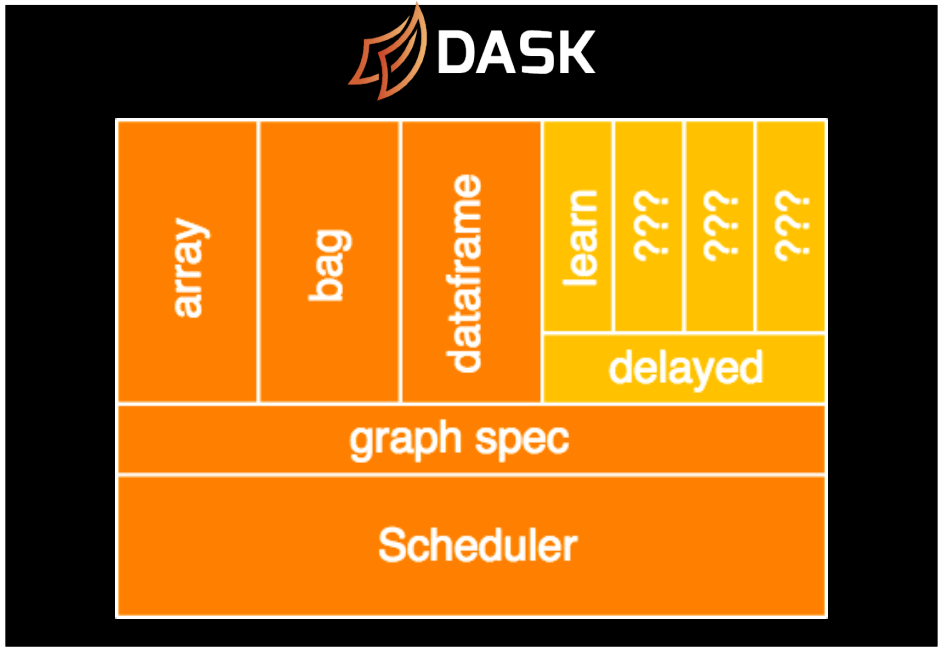
Scale Out

The Need to Scaling Out

- GPU RAM max: 32GB
- Out-of-core: dataset larger than RAM
- Multi-GPU
- Multi-nodes

How? Dask

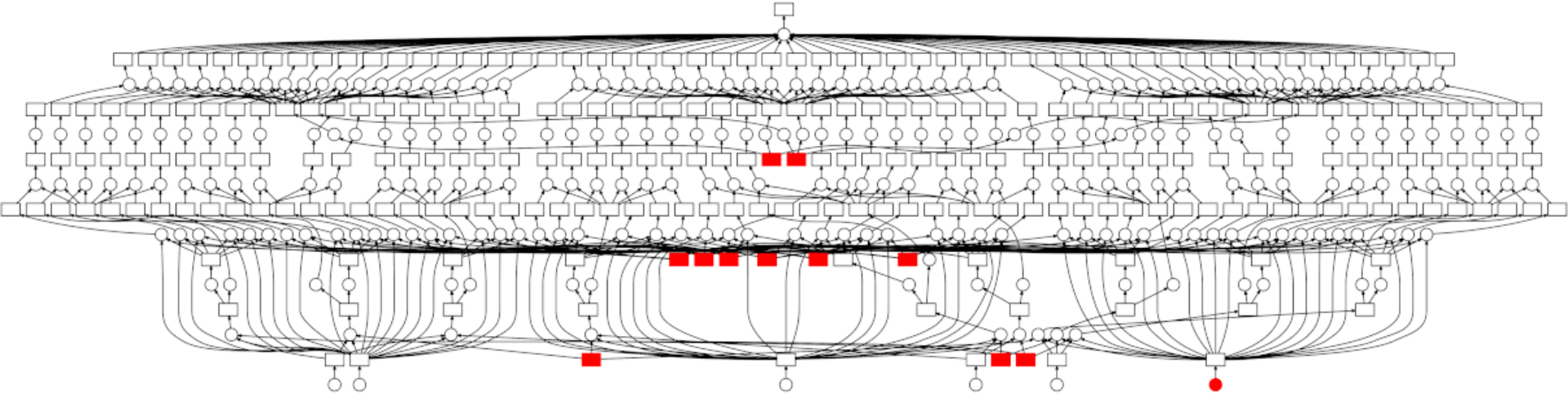
- Pure-python
- Flexible library for parallelism
- Block algorithms
- Dynamic scheduling
- Multi-scheduler
    - threads, processes, distributed



Dask Task Scheduling

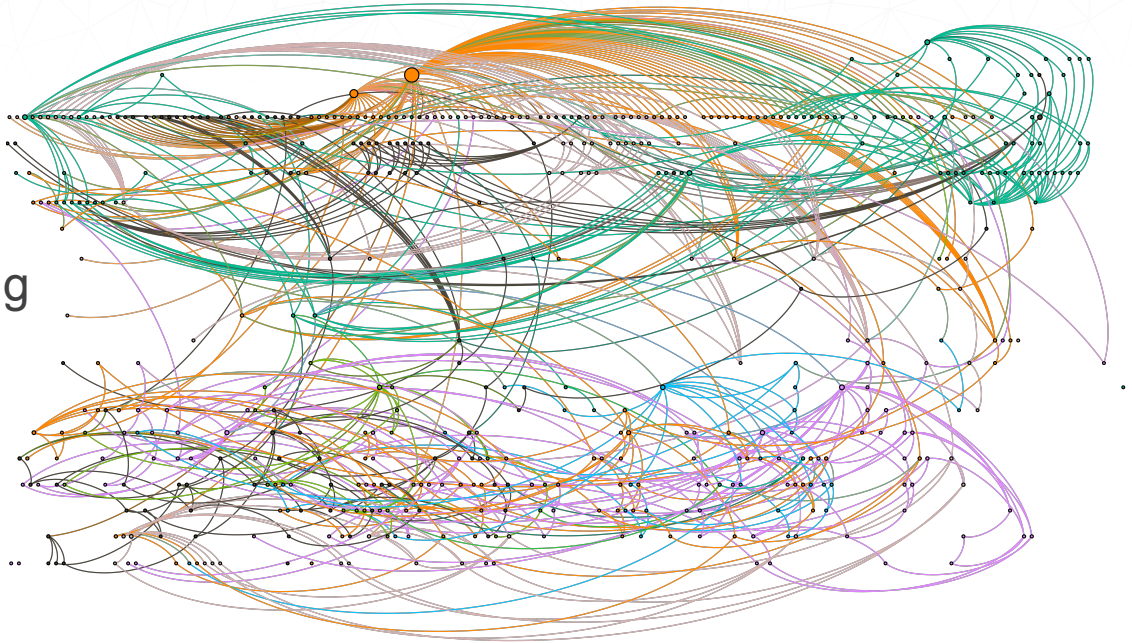
Optimizes for:

- Data locality, Communication, Parallelism



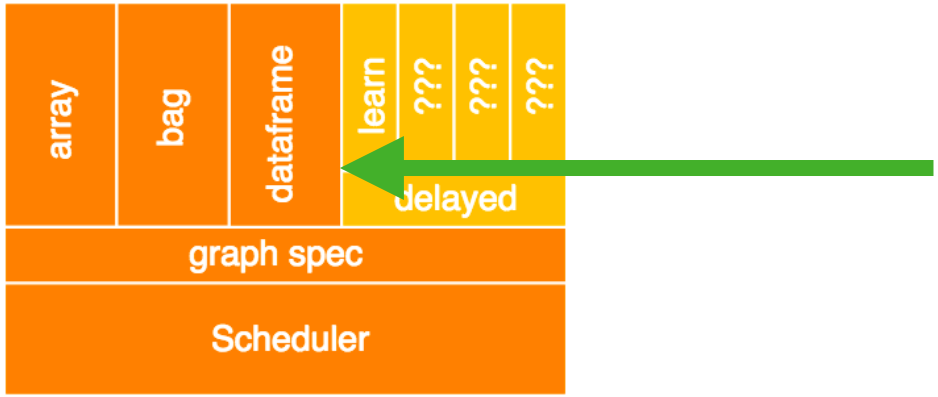
Dask can handle complicated task graphs

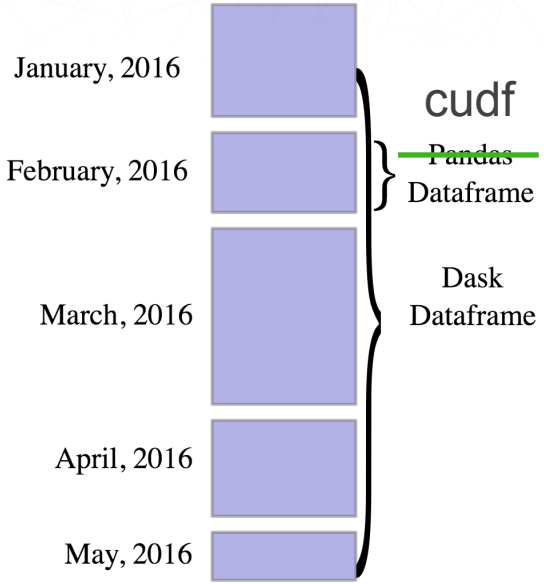
dask graph from a credit modeling system of a large retail bank



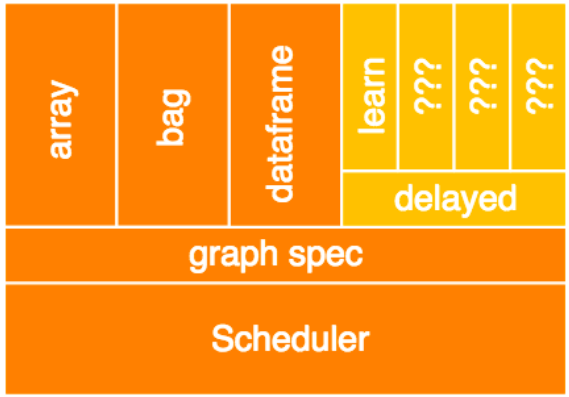
reference: http://blog.dask.org/2018/02/09/credit-models-with-dask

Model after dask.dataframe





Using dask.delayed for custom operations




```python
# Second, do groupby internally for each partition.
@delayed
def _groupby(df, by, method):
    grouped = df.groupby(by=by, method=method)
    return grouped
```

(actual code from dask_cudf)

cudf.DataFrame

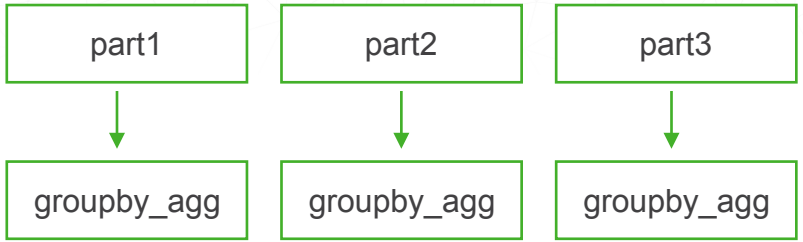
Groupby + Aggregation


```
@dask.delayed
def groupby_agg(df, key):
    return df.groupby(key).sum()

@dask.delayed
def combine(*dfs):
    return concat(dfs)

firstlevel = [groupby_agg(p, "keys")
              for p in [part1, part2, part3]]
secondlevel = combine(*firstlevel)
thirdlevel = groupby_agg(secondlevel, "keys")
```

Groupby + Aggregation

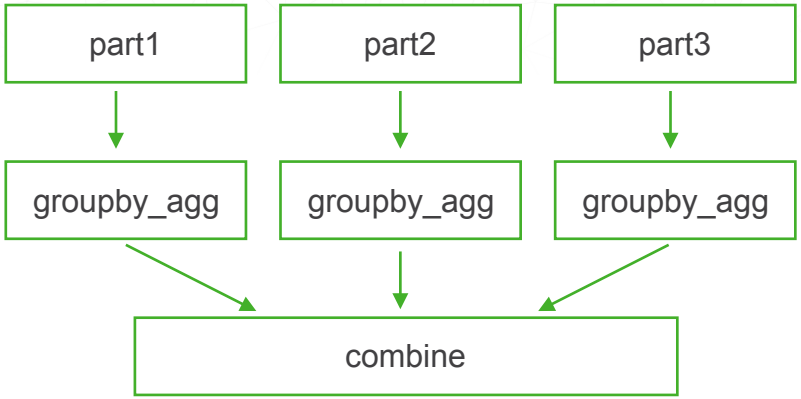



```
@dask.delayed
def groupby_agg(df, key):
    return df.groupby(key).sum()

@dask.delayed
def combine(*dfs):
    return concat(dfs)

firstlevel = [groupby_agg(p, "keys")
              for p in [part1, part2, part3]]
secondlevel = combine(*firstlevel)
thirdlevel = groupby_agg(secondlevel, "keys")
```

Groupby + Aggregation

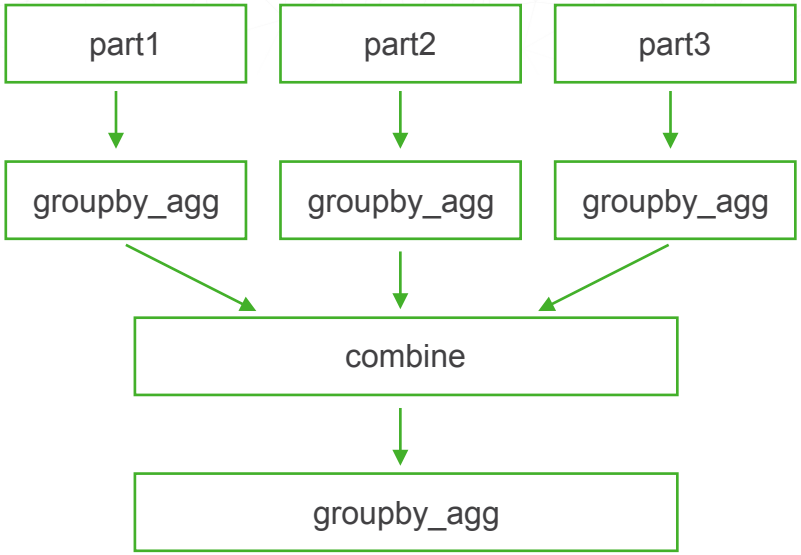



```
@dask.delayed
def groupby_agg(df, key):
    return df.groupby(key).sum()

@dask.delayed
def combine(*dfs):
    return concat(dfs)

firstlevel = [groupby_agg(p, "keys")
              for p in [part1, part2, part3]]
secondlevel = combine(*firstlevel)
thirdlevel = groupby_agg(secondlevel, "keys")
```

Groupby + Aggregation

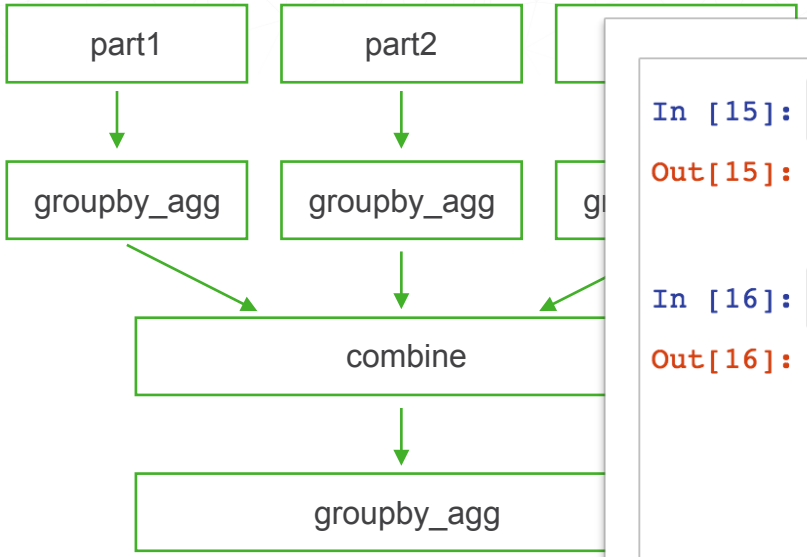



```
@dask.delayed
def groupby_agg(df, key):
    return df.groupby(key).sum()

@dask.delayed
def combine(*dfs):
    return concat(dfs)

firstlevel = [groupby_agg(p, "keys")
              for p in [part1, part2, part3]]
secondlevel = combine(*firstlevel)
thirdlevel = groupby_agg(secondlevel, "keys")
```

Groupby + Aggregation




```
In [15]: thirdlevel

Out[15]: Delayed('groupby_agg-22cf0898-c8f8-4dd1-8957-026933
         20a30a')

In [16]: thirdlevel.compute()

Out[16]:
```

	vals
keys	
0	7.890715
1	12.159657
2	9.954078
3	10.302435

Problems Solved by Using Dask

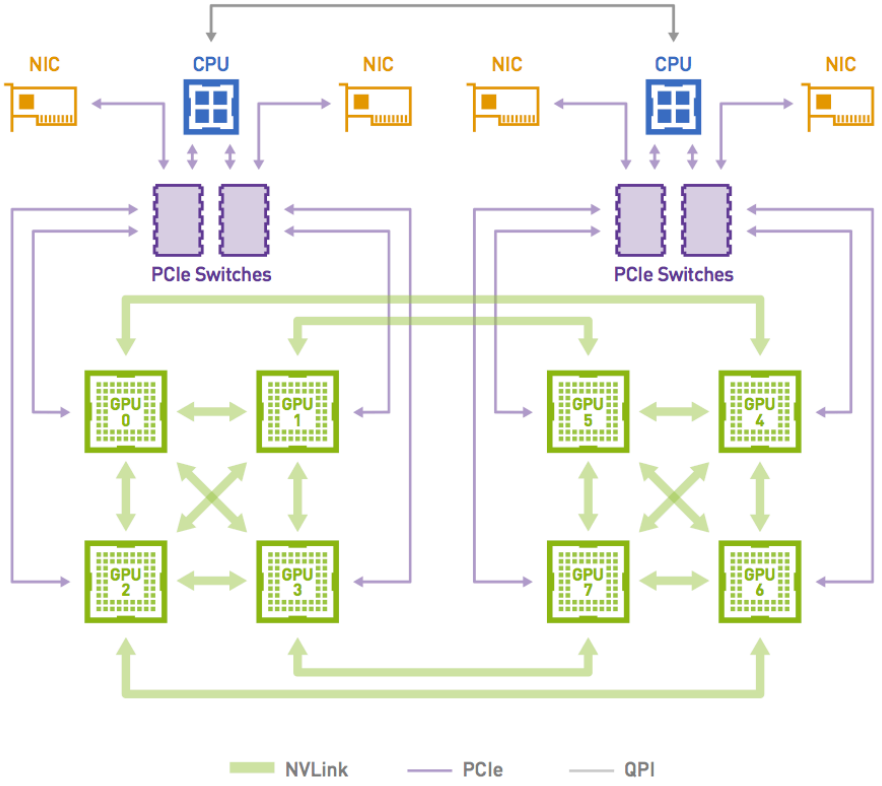
- distributed, out-of-core scheduling
- spill-to-disk
- easy scale out
- clustering
    - kubernetes, yarn, HPC

GPU-to-GPU communication

- Dask defaults to use TCP + serialization (pickle)
- GPU has more efficient ways:
  - CUDA IPC
  - Inter-node RDMA

GPU Peer Access

CUDA IPC peer access requires
- 2 GPUs are connected by NVLINK
- 2 GPUs are in the same PCIe domain



(image from: https://devblogs.nvidia.com/dgx-1-fastest-deep-learning-system/)

Inter-node Transfer

- Now:
  - Copy to host
  - Send via normal network
- WIP:
  - UCX
    - GPUDirect
    - RDMA over InfiniBand

Conclusions

Key Decisions

- Focus on the Data-Scientist experience
  - Mirror Pandas API
  - Don’t let performance drive API design
  - Interactive GPU JIT with Numba
- Leverage existing OSS ecosystem
  - Arrow, Dask, Numba

Ongoing + Future works

- Integrate UCX into Python
- Better GPU memory manager (RMM)
- Common Python interface to:
  - the CUDA runtime
  - share Array data

Summary

- We talked about…
  - GOAI
  - GPU DataFrame
  - Scale out to Distributed GPU DataFrame
- I hope you learn a new technique to scale-out your GPU workload using Dask.

Questions?

goai:
http://gpuopenanalytics.com
rapids:
https://rapids.ai
source code:
https://github.com/rapidsai
https://github.com/dask
https://github.com/numba

Extras

Setup Dask Cluster

- Manual:
  - `dask-scheduler`
  - `dask-worker <scheduler ip+port> –nprocs=<nprocs>`
- Other cluster support for kubernetes, yarn, HPC
- Support scale-up, scale-down

Setup Dask GPU Cluster

- Manually setup to have 1 GPU per worker
- So, a 8 GPU server will have 8 workers
- Each worker will use a different GPU as default