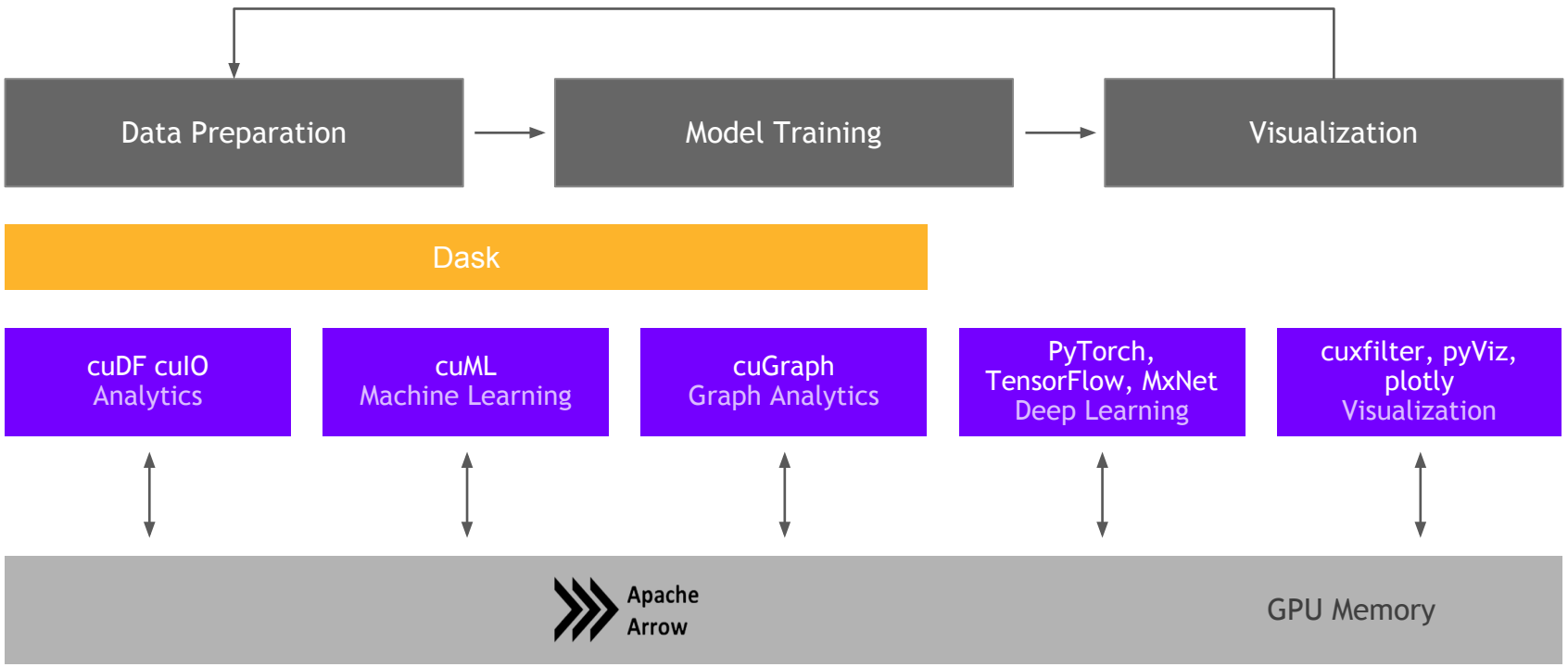
5th APAC HPC-AI
Intro To RAPIDS, Dask, And UCX-Py

Peter Entschev (NVIDIA)

June 17, 2022

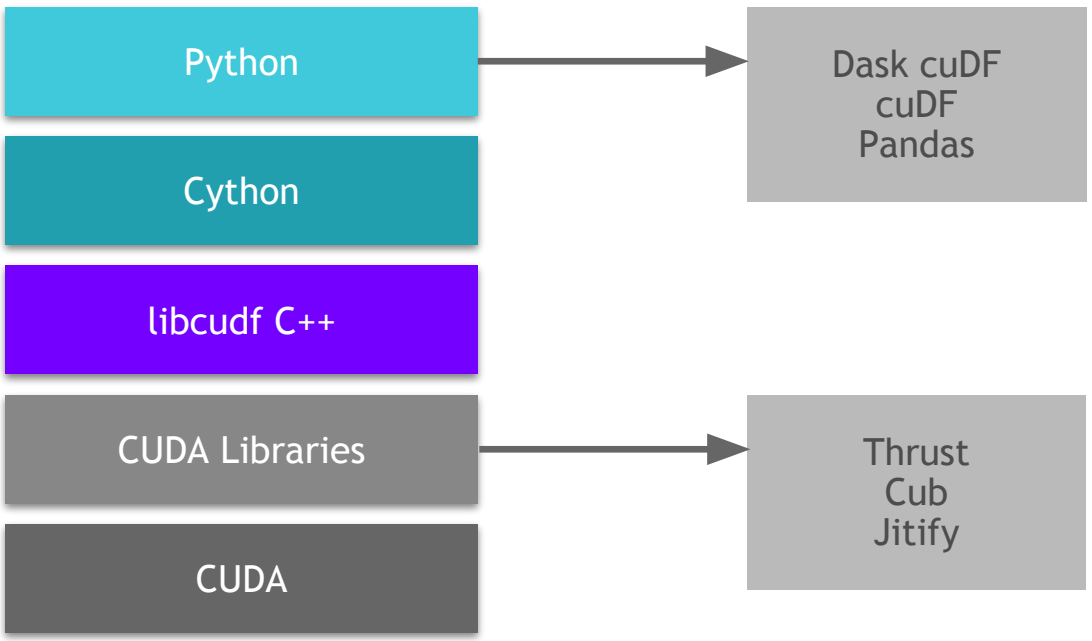
RAPIDS

End-to-End GPU Accelerated Data Science



What is cuDF?

Expandable platform for GPU data science

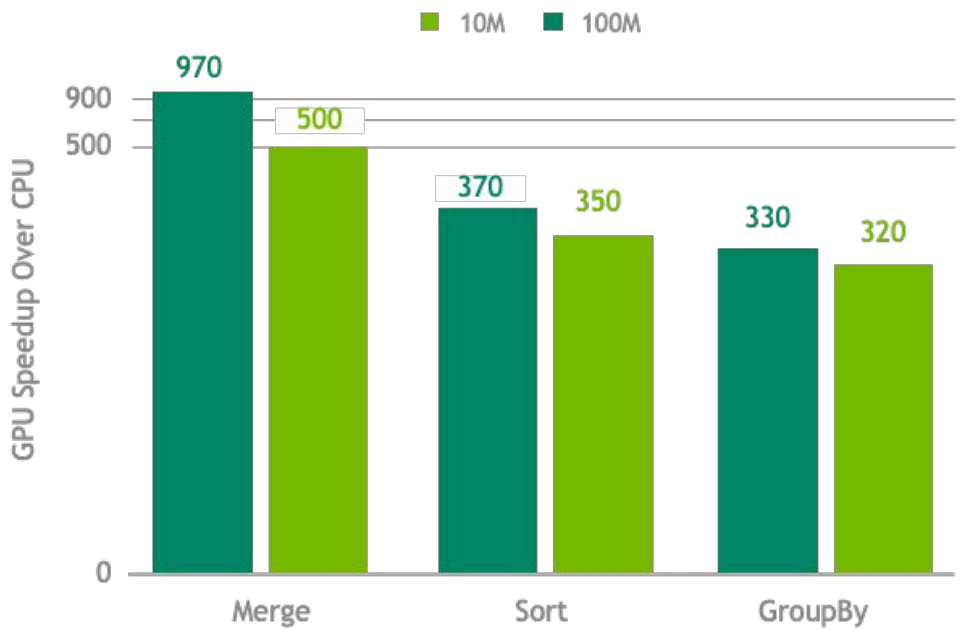


- Familiar pandas-like Python API
- Table (dataframe) and column types and algorithms
- High-performance C++ layer provides GPU-optimized CUDA kernels, data types, operations, and primitives
- CUDA/C++ is top level supported and used by many for integrating RAPIDS

Accelerated Pre-processing

A Familiar Experience for Data Engineers

RAPIDS provides a GPU DataFrame library with a pandas-like API while providing *significant* performance improvements.



Single GPU Speed-Ups vs pandas

GPU: NVIDIA Tesla V100 32GB on DGX-1
CPU: Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz

Comprehensive String Support

Backbone of ETL: Strings

Regular Expressions

Element-wise operations

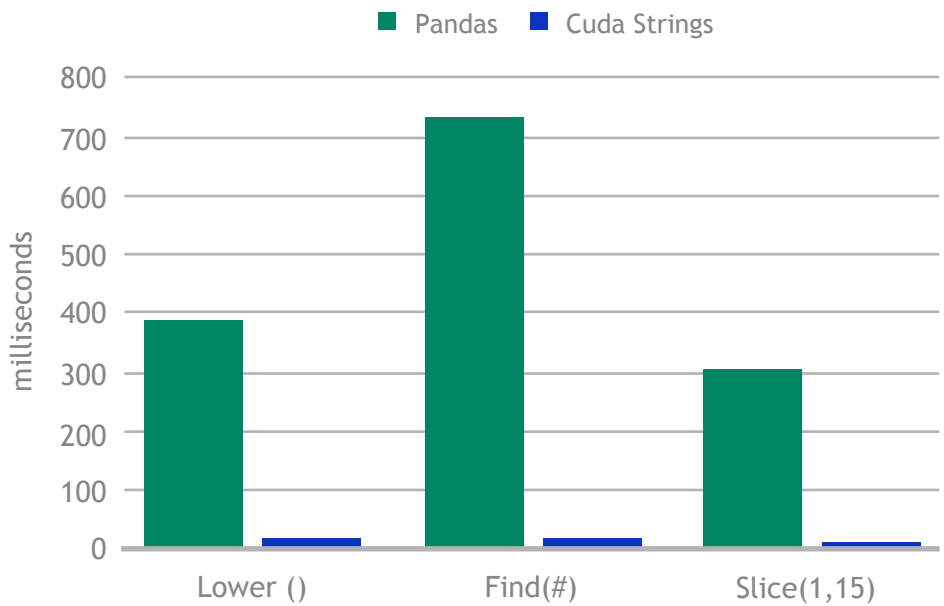
Split, Find, Extract, Cat, Typecasting, etc…

String GroupBys, Joins, Sorting, etc.

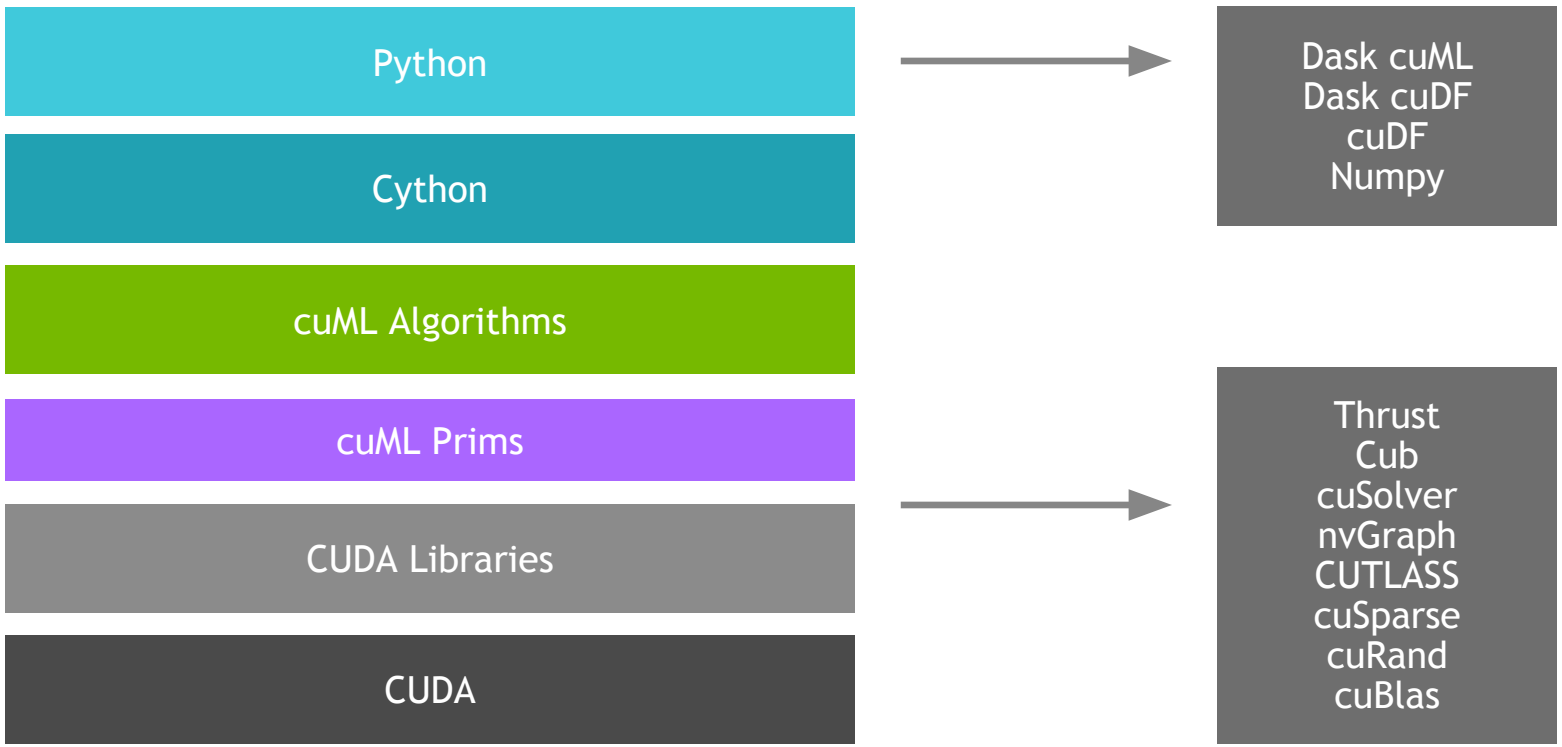
Categorical columns fully on GPU

NLP Preprocessors

Tokenizers, Normalizers, Edit Distance, Porter Stemmer, etc.



ML Technology Stack

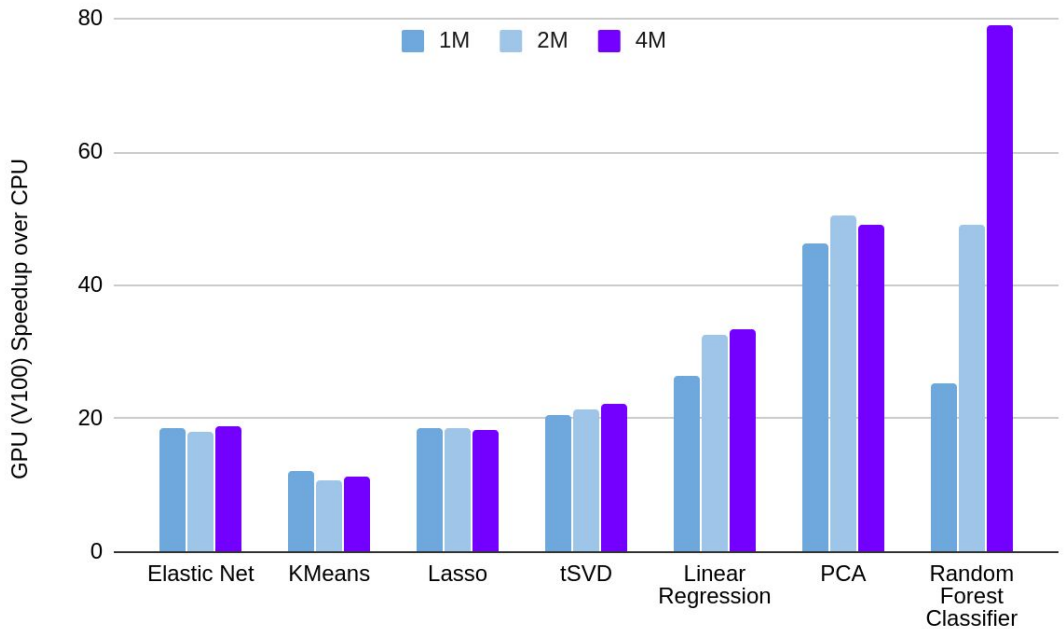


Accelerated Machine Learning

GPU Power With the Feel of scikit-learn

RAPIDS provides a GPU ML library with a scikit-learn API while providing *significant* performance improvements.

26 GPU-Accelerated Algorithms & Growing



1x V100 vs. 2x 20 Core CPUs (DGX-1, RAPIDS 0.15)

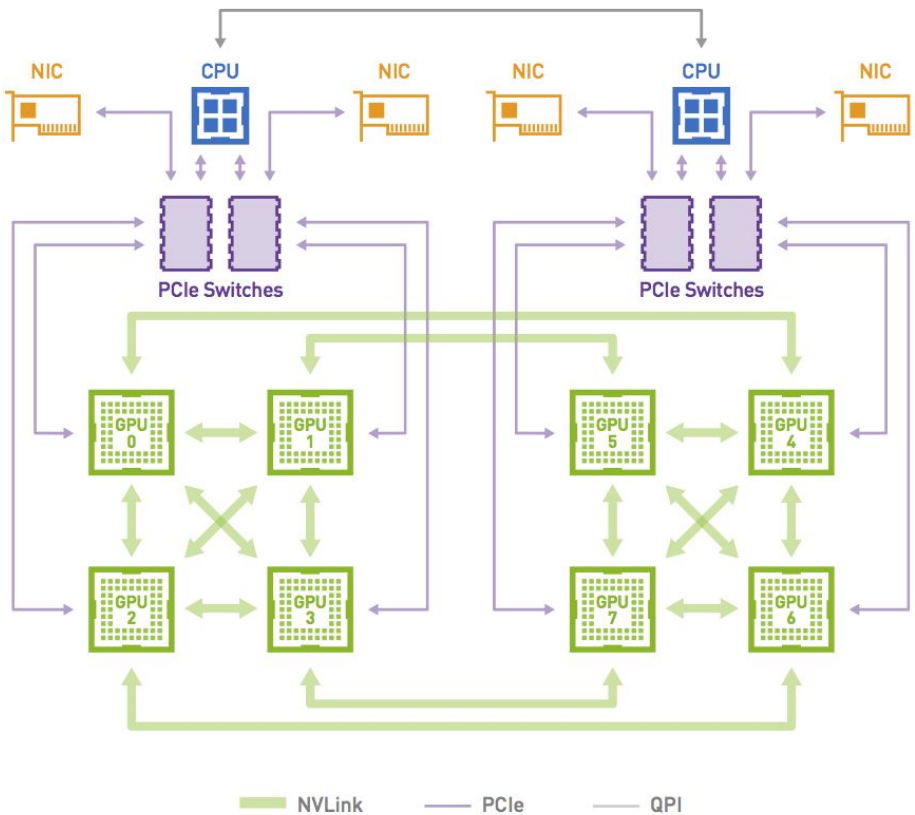
Build End-to-End Data Science Applications

Leverage RAPIDS Core Libraries to Build Custom Solutions

	Description	Similar To	Problem Domain	Maturity	Performance	Example User	API Docs
cuDF	Dataframes & ETL	pandas	Data Preparation			Walmart	Read the Docs
Apache Spark 3.0 Plugin	ETL	Apache Spark	Data Preparation			CLOUDERA	Read the Docs
Dask-SQL	ANSI SQL	SQL	Data Preparation				Read the Docs
cuML	Machine Learning	scikit-learn	Model Training			Capital One	Read the Docs
cuGraph	Graph Analytics	NetworkX	Model Training			VISA	Read the Docs
XGBoost	GBMs	XGBoost	Model training			Scotiabank	Read the Docs
RAPIDSViz	Large-Scale Visualization	Bokeh, DataShader, HoloViews	Visualization			plotly	Read the Docs

Hardware Layout

DGX-1

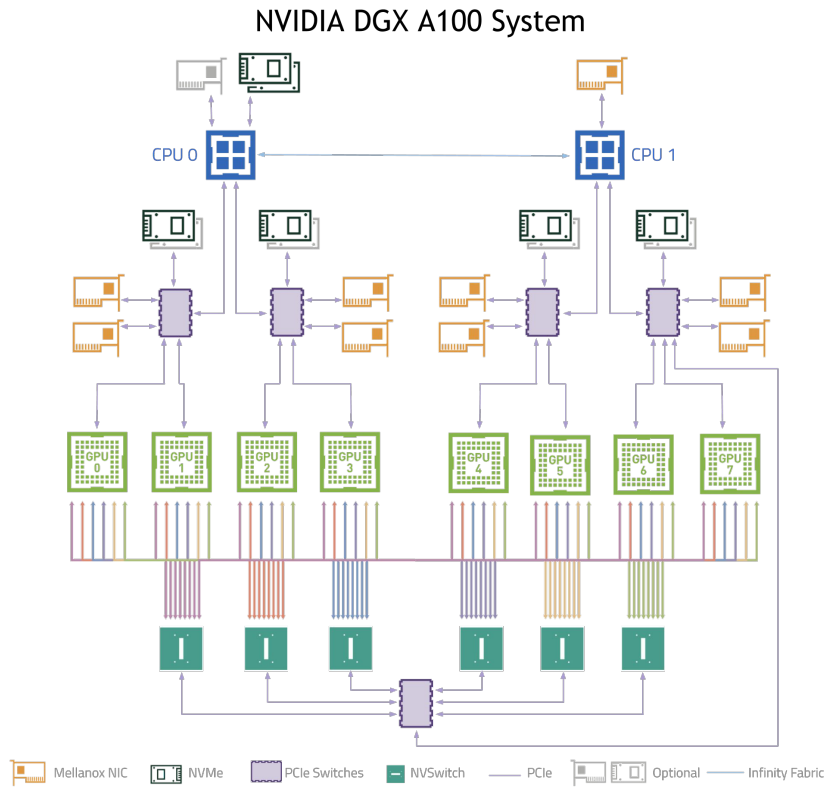


Why GPUs for Data Science?

Numerous hardware advantages

- Thousands of cores with up to ~20 TeraFlops of general purpose compute performance
- Up to 1.5 TB/s of memory bandwidth
- Hardware interconnects for up to 600 GB/s bidirectional GPU <--> GPU bandwidth
- Can scale up to 16x GPUs in a single node

Almost never run out of compute relative to memory bandwidth!



dask

Python library for parallel computing

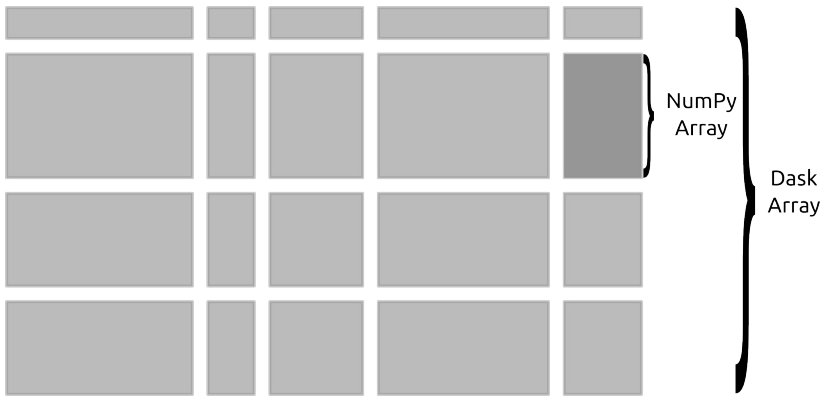
1. Scales Numpy, Pandas, and scikit-Learn
2. General Purpose Computing Framework

Easy for beginners,
Secure and trusted for institutions

Dask Accelerates the Existing Python Ecosystem

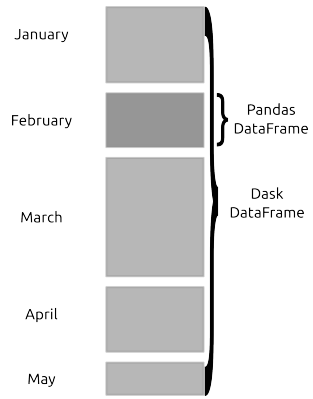
Built alongside the current community

Numpy



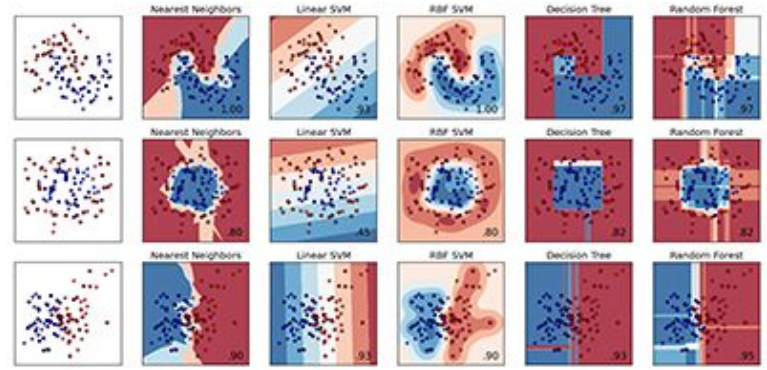

```
import numpy as np

x = np.ones((1000, 1000))
x + x.T - x.mean(axis=0)
```

Pandas




```
import pandas as pd

df = pd.read_csv("file.csv")
df.groupby("x").y.mean()
```

Scikit-Learn

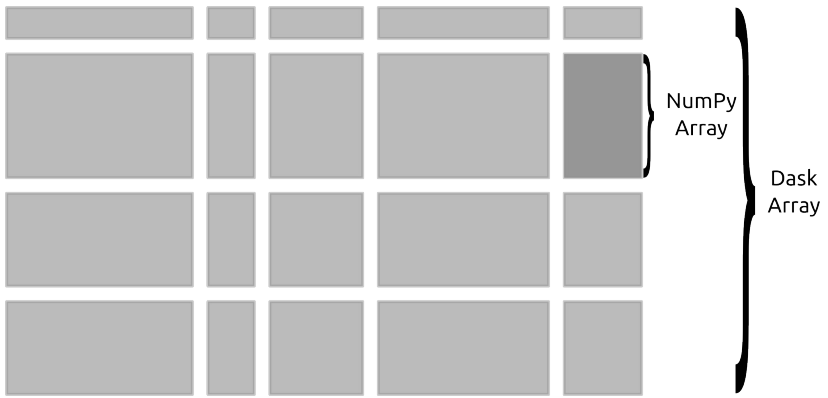


```
from scikit_learn.linear_model \
    import LogisticRegression

lr = LogisticRegression()
lr.fit(data, labels)
```

Dask Accelerates the Existing Python Ecosystem

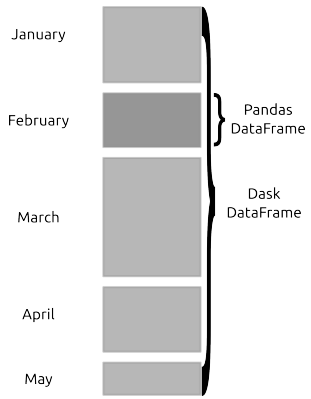
Built alongside the current community

Numpy



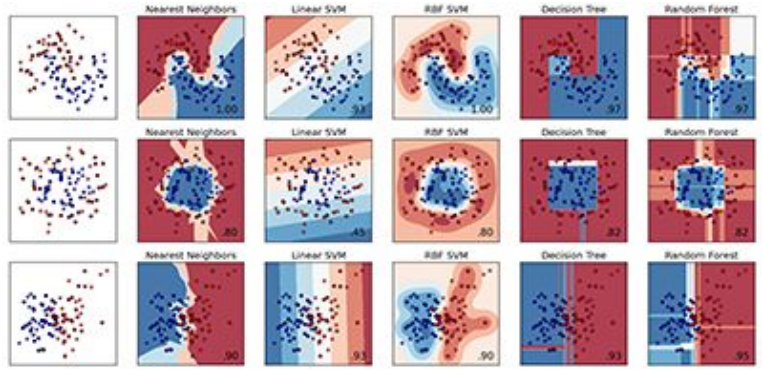

```
import dask.array as da

x = da.ones((10000, 10000))
x + x.T - x.mean(axis=0)
```

Pandas




```
import dask.dataframe as dd

df = dd.read_csv("s3://*.csv")
df.groupby("x").y.mean()
```

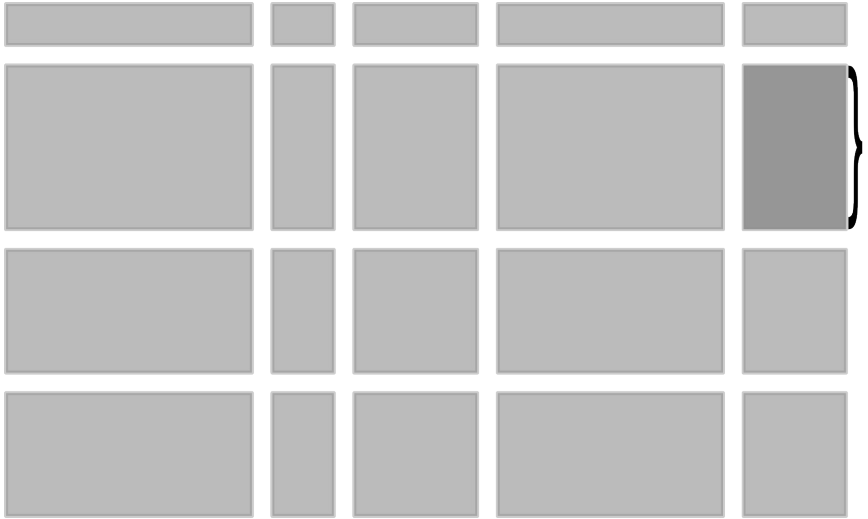
Scikit-Learn



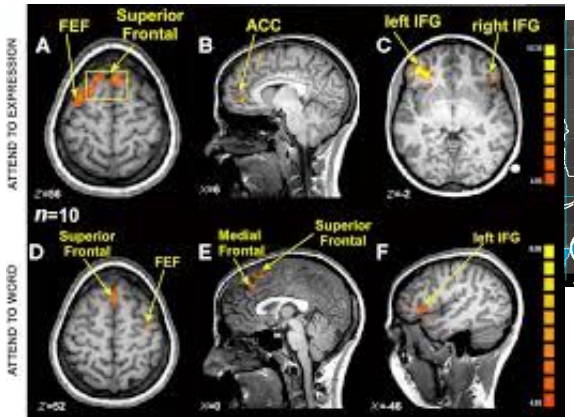
```
from dask_ml.linear_model \
    import LogisticRegression

lr = LogisticRegression()
lr.fit(data, labels)
```

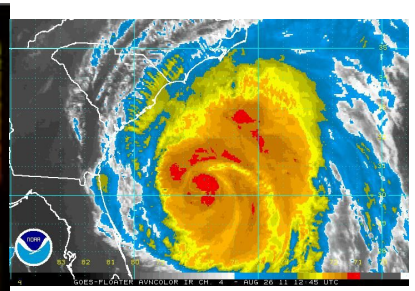
NumPy + Dask: Scalable array computing

Supporting scientific and gridded computations




```
>>> import dask.array as da
>>> x = da.from_hdf5("...")
>>> (x + x.T) - x.mean(axis=0)
```



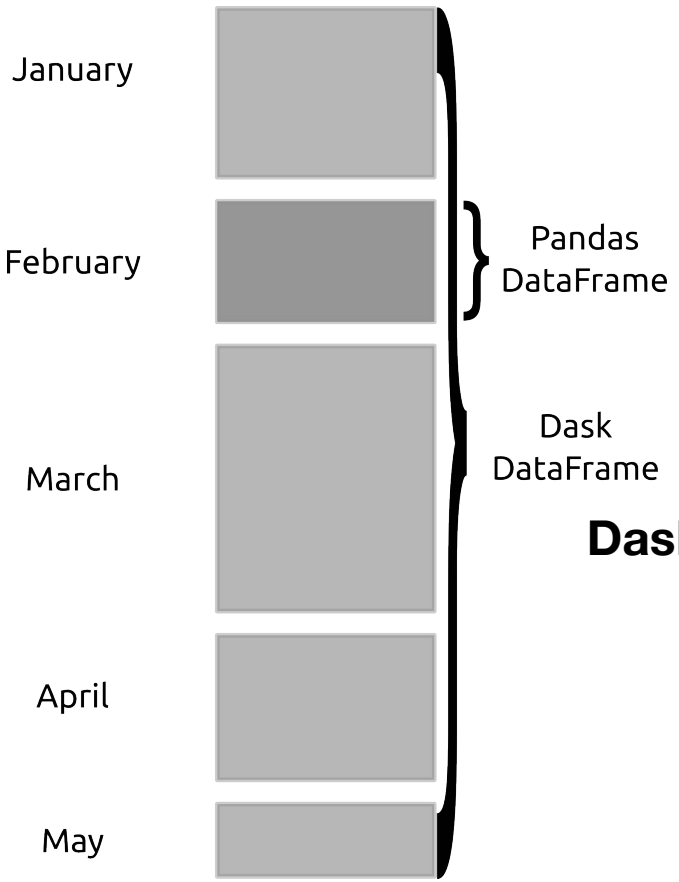


Non tabular big data is common
But is underserved by most enterprise tooling

Dask applies equally well to non-traditional data problems, enabling scientific and industrial problems.

Pandas + Dask: Scalable Dataframe

Supporting traditional tabular workflows on large clusters




```
>>> import dask.dataframe as dd
>>> df = dd.read_csv("data/*.csv")
>>> df.groupby("name").balance.mean()
```

Dask extends existing Python libraries

enables an easy transition for users

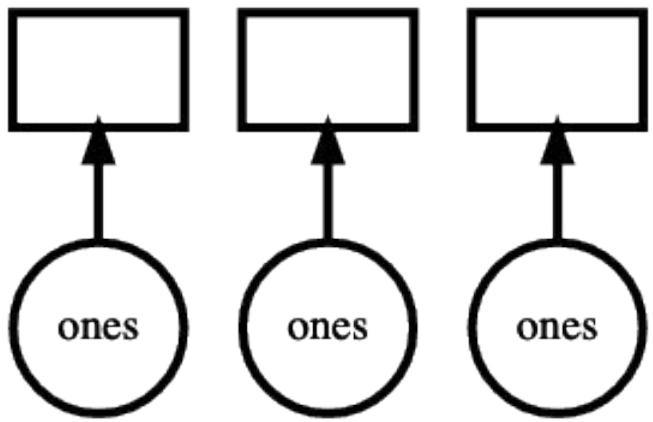
leverages existing work, rather than reinvents wheels

Parallel Algorithms

Dask parallelizes common Python operations

Dask APIs Produce Task Graphs

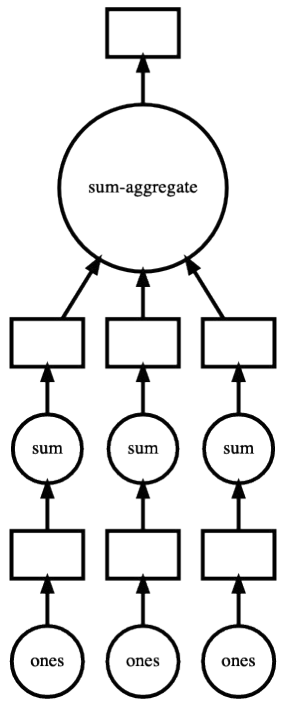
Create a 1D array of ones




```
import dask.array as da

x = da.ones(15, chunks=(5,))
```

Dask APIs Produce Task Graphs

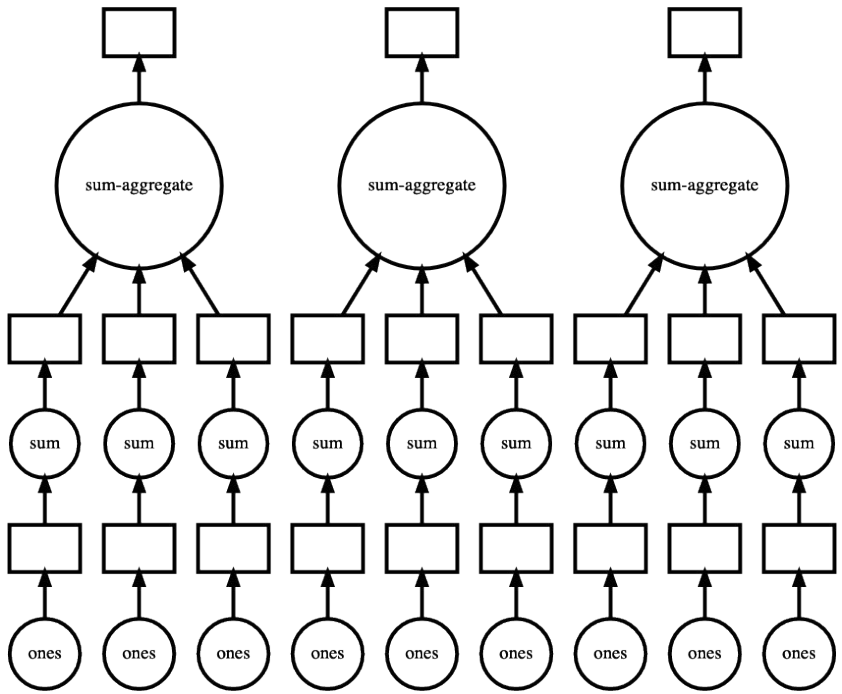
Sum that array




```
import dask.array as da

x = da.ones(15, chunks=(5,))
x.sum()
```

Dask APIs Produce Task Graphs

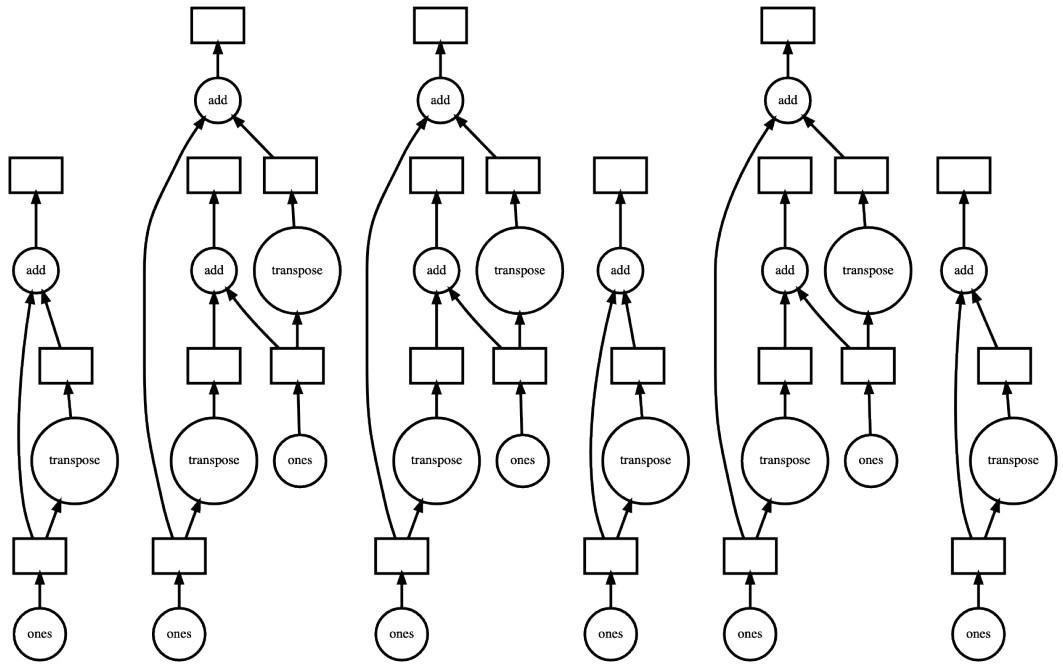
Sum a 2D arrays of ones




```
import dask.array as da

x = da.ones((15, 15), chunks=(5, 5))
x.sum(axis=0)
```

Dask APIs Produce Task Graphs

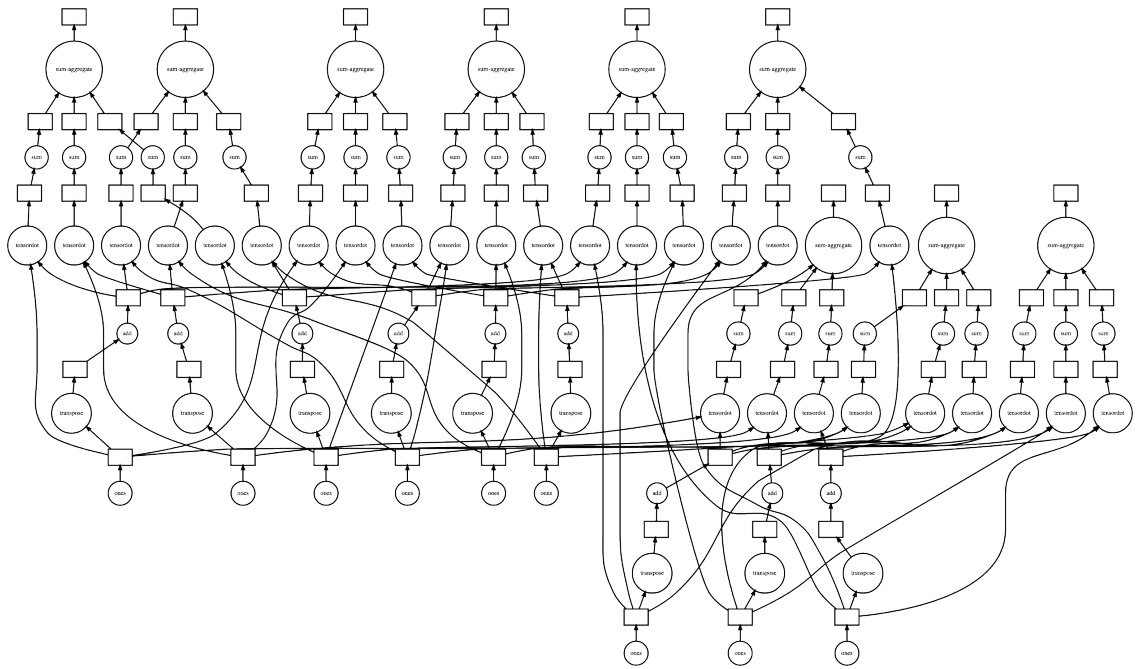
Add array to its transpose




```
import dask.array as da

x = da.ones((15, 15), chunks=(5, 5))
x + x.T
```

Dask APIs Produce Task Graphs

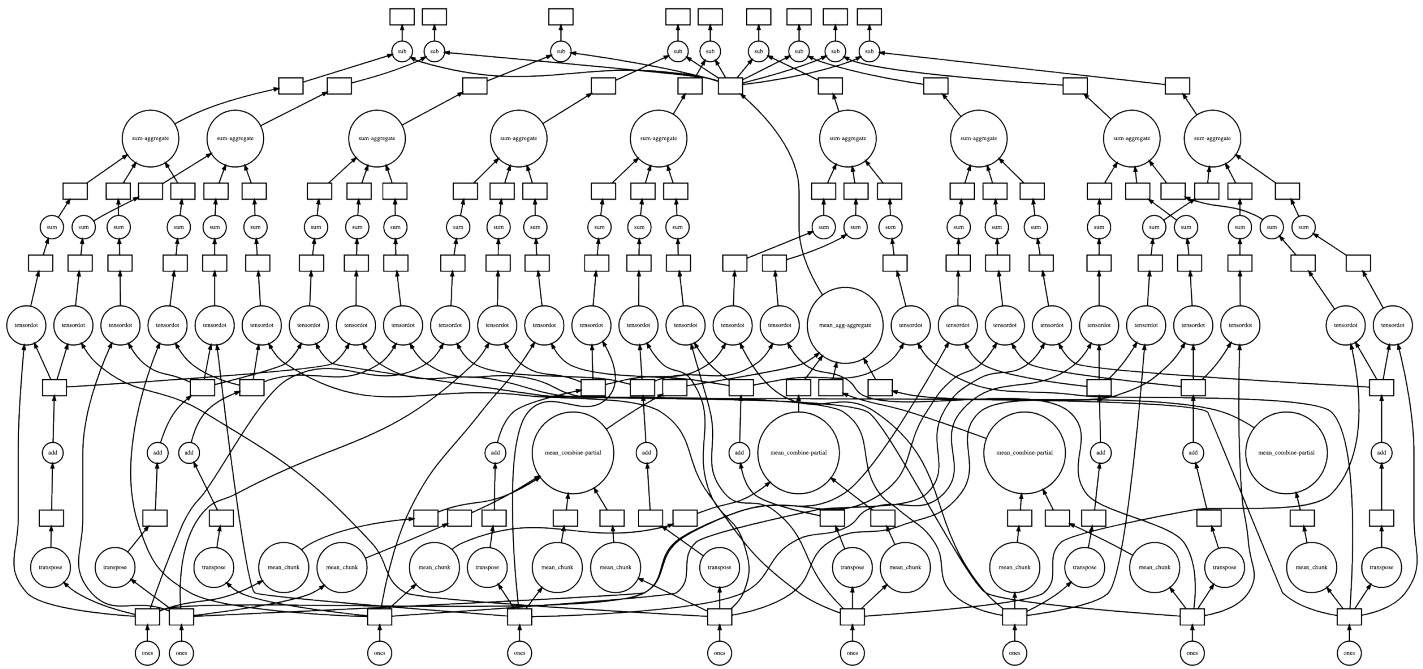
Add in a matrix multiply



```
import dask.array as da

x = da.ones((15, 15), chunks=(5, 5))
x.dot(x.T + 1)
```

Dask APIs Produce Task Graphs

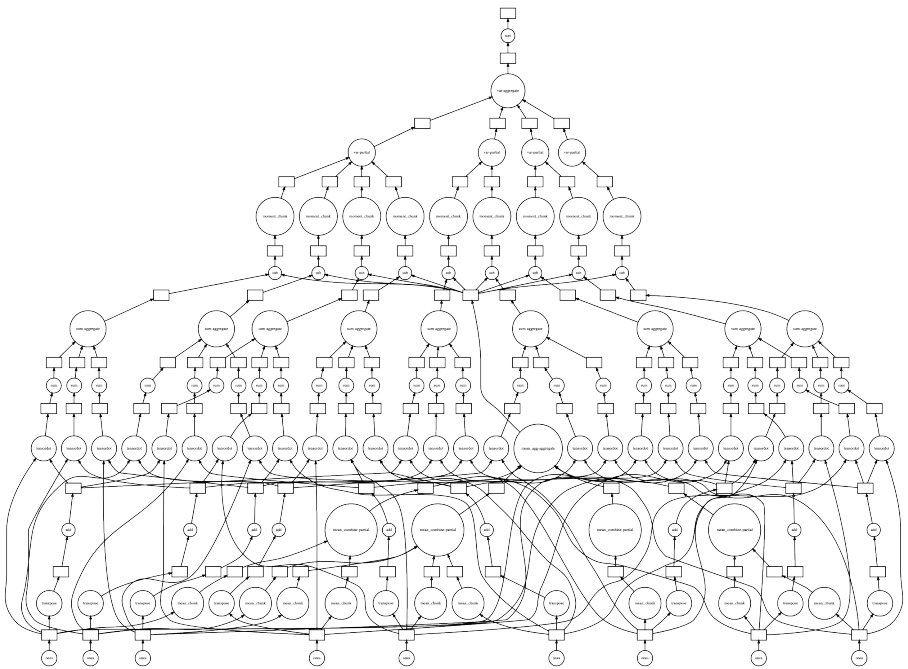
Add in a matrix multiply and a mean




```
import dask.array as da

x = da.ones((15, 15), chunks=(5, 5))
x.dot(x.T + 1) - x.mean(axis=0)
```

Dask APIs Produce Task Graphs

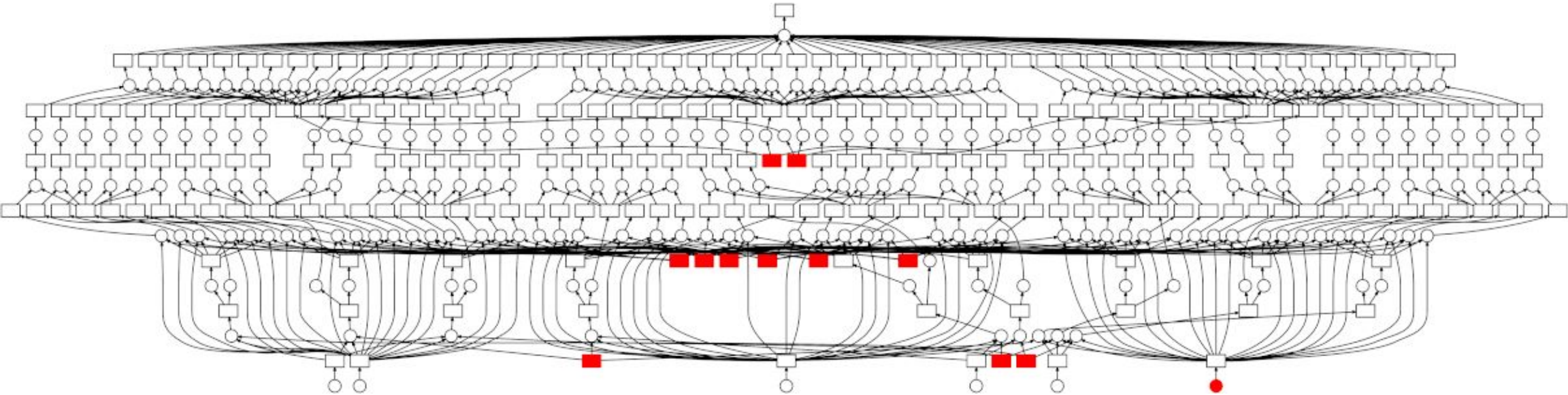
Add in a matrix multiply, a mean and a standard deviation



```
import dask.array as da

x = da.ones((15, 15), chunks=(5, 5))
(x.dot(x.T + 1) - x.mean(axis=0)).std()
```

Dask Schedules and Compute Task Graphs

Execute task graphs efficiently over parallel hardware

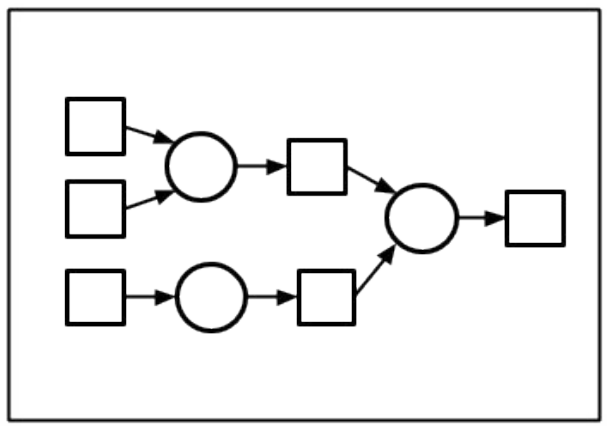


Scikit-Learn cross-validated grid search over a pipeline

Dask APIs Build Graphs, Schedulers Execute

Collections
(create task graphs)
→
Task Graph
→
Schedulers
(execute task graphs)

- Dask Array
- Dask DataFrame
- Dask Bag
- Dask Delayed
- Futures



- Single-machine
  (threads, processes, synchronous)
- Distributed

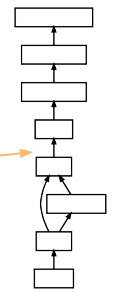
A variety of frontend APIs

A variety of backend Execution systems

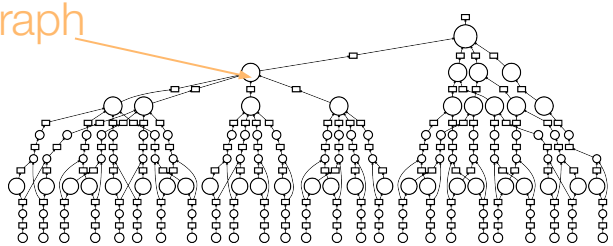
Client

```
x = da.ones((100, 100), chunks=(20, 20))
x += 1
y = x + x.T
z = y.sum()
z.compute()
```

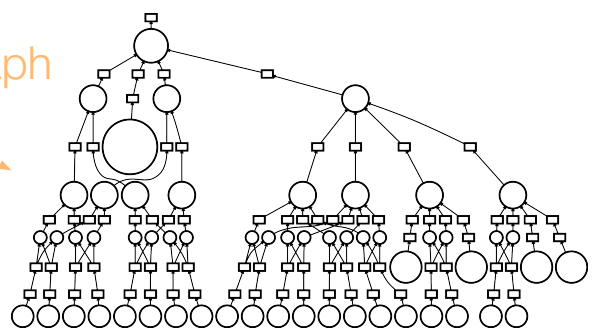
High Level graph



Low level graph



Optimized graph



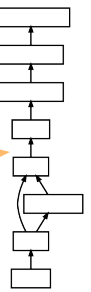
Scheduler

Workers

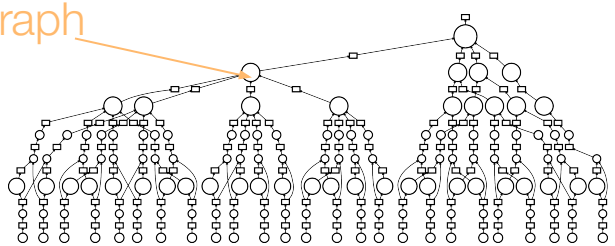
Client

```
x = da.ones((100, 100), chunks=(20, 20))
x += 1
y = x + x.T
z = y.sum()
z.compute()
```

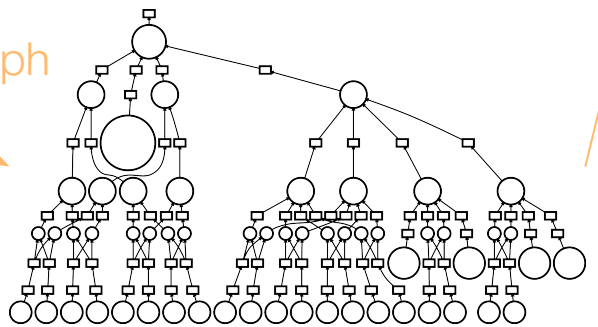
High Level graph



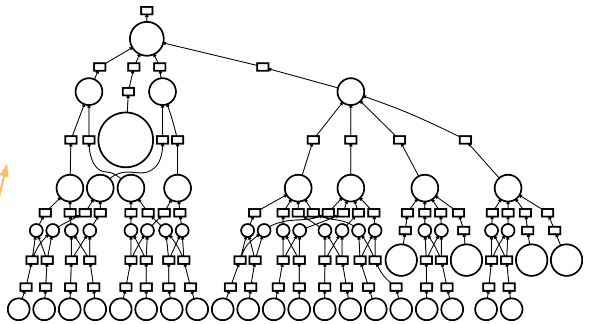
Low level graph



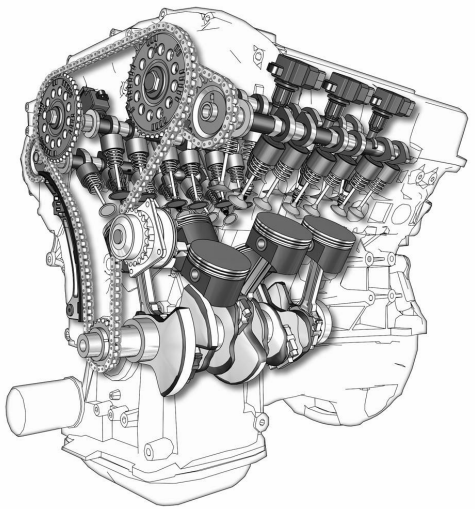
Optimized graph



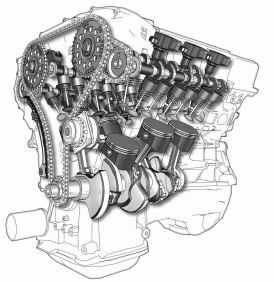
Scheduler

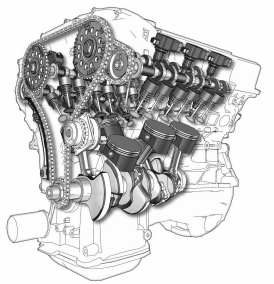


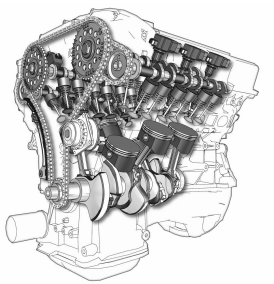
Internal State Machine



Workers



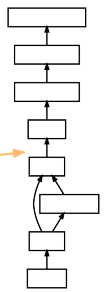




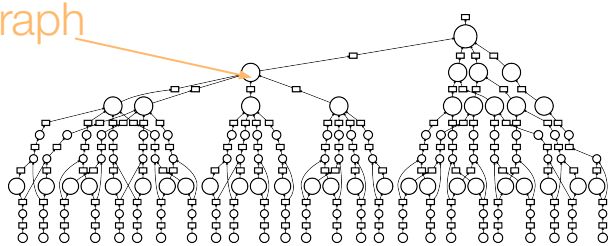
Client

```
x = da.ones((100, 100), chunks=(20, 20))
x += 1
y = x + x.T
z = y.sum()
z.compute()
```

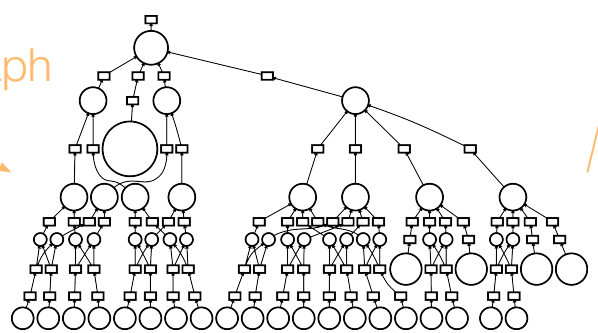
High Level graph



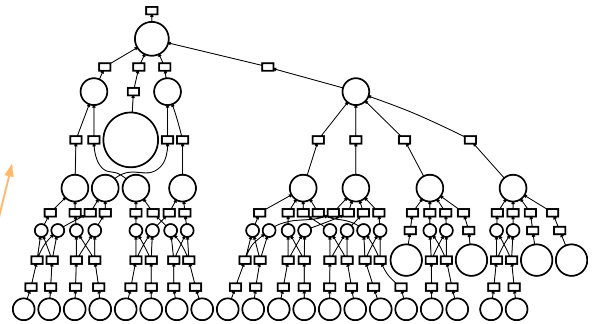
Low level graph



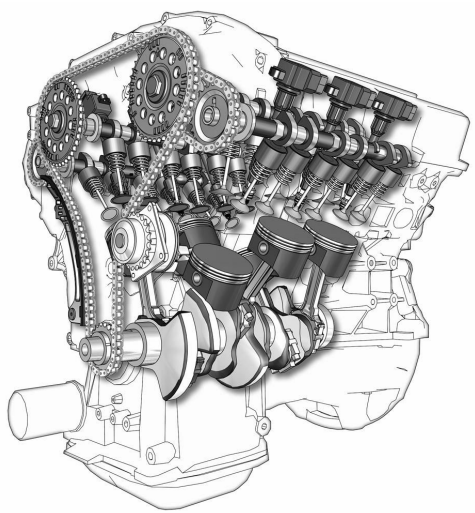
Optimized graph



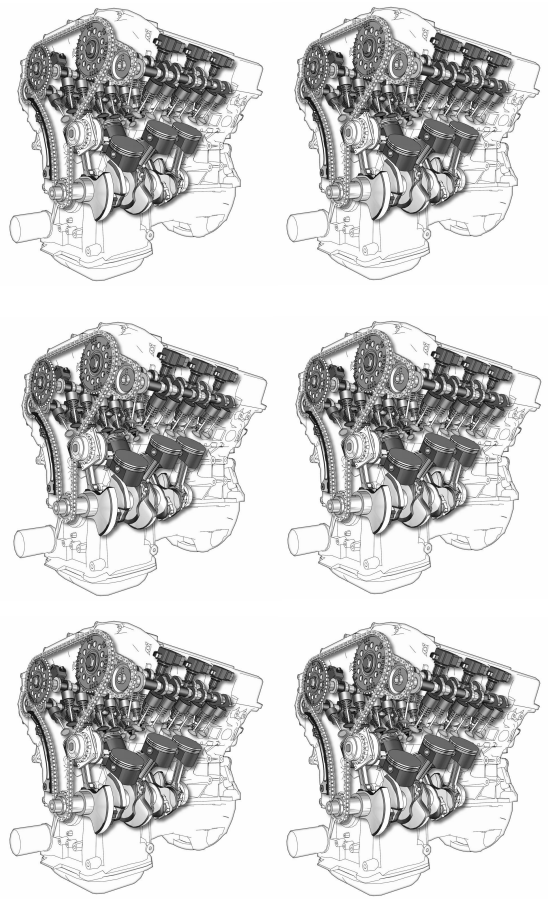
Scheduler



Internal State Machine



Workers



UCX-Py
Introduction

- Python interface for UCX
- Provides sync and async APIs
- Simple replacement for Python communications (e.g., sockets)
- Targeted at library and framework developers
- No low-level communications, UCX or C knowledge required
- Made available to users (e.g., data scientists) via frameworks such as Dask

Using UCX-Py

Send/Receive with CuPy

Server

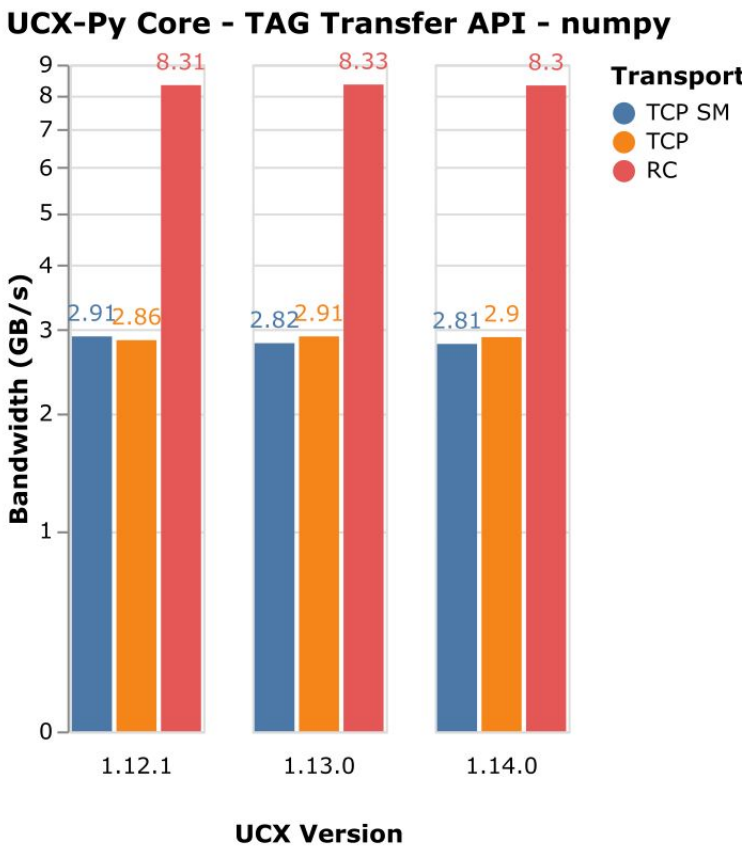
```
async def server(ep):
    # allocate buffer and receive tag message
    arr = cupy.empty(10000, dtype='u1')
    await ep.recv(arr)

    # send data back via tag message
    await ep.send(arr)

    await ep.close()
    lf.close()

async def main():
    global lf
    lf = ucp.create_listener(server, port=12345)

    while not lf.closed():
        await asyncio.sleep(0.1)
```

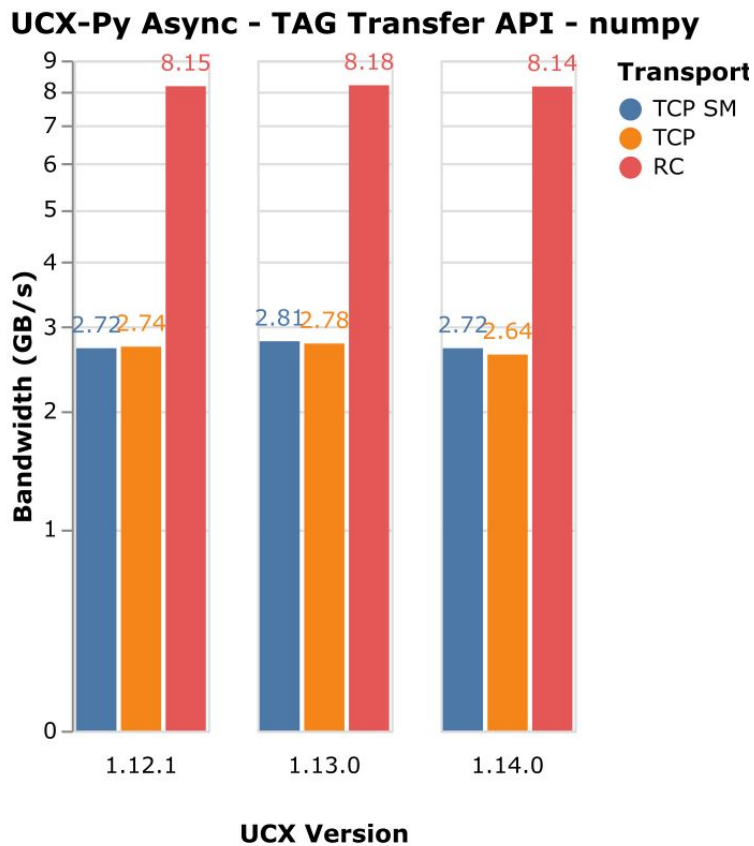
Client

```
async def client():
    host = ucp.get_address(ifname='eth0')  # get address for eth0
    ep = await ucp.create_endpoint(host, port=12345)

    msg = cupy.zeros(10000, dtype='u1')  # data to send
    await ep.send(msg)  # send tag message

    # receive tag response
    resp = cupy.empty_like(msg)
    await ep.recv(resp)  # receive the echo
    cupy.testing.assert_array_equal(msg, resp)
    await ep.close()
```

UCX-Py Performance

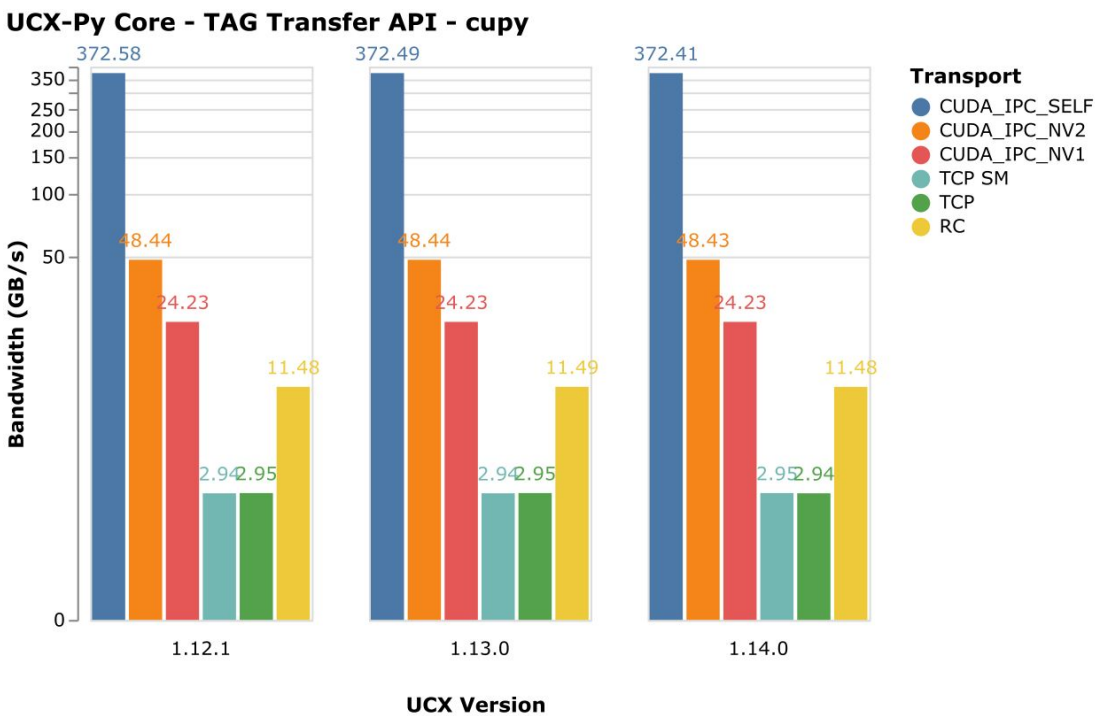
NumPy - Host Memory

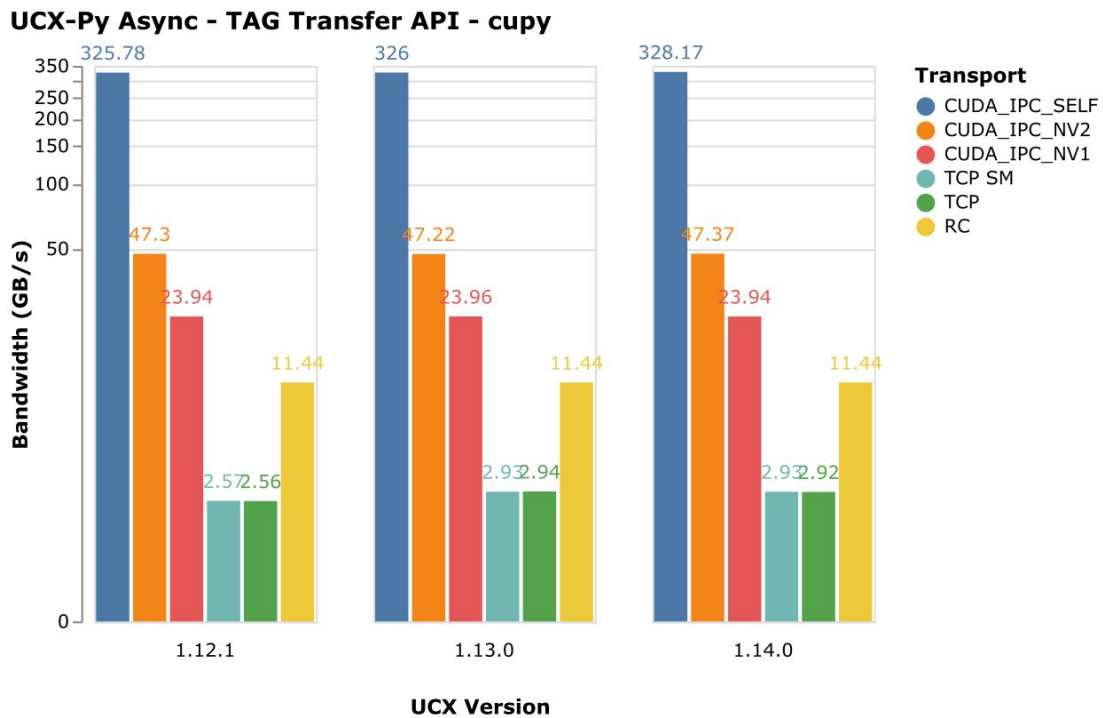




UCX-Py Performance

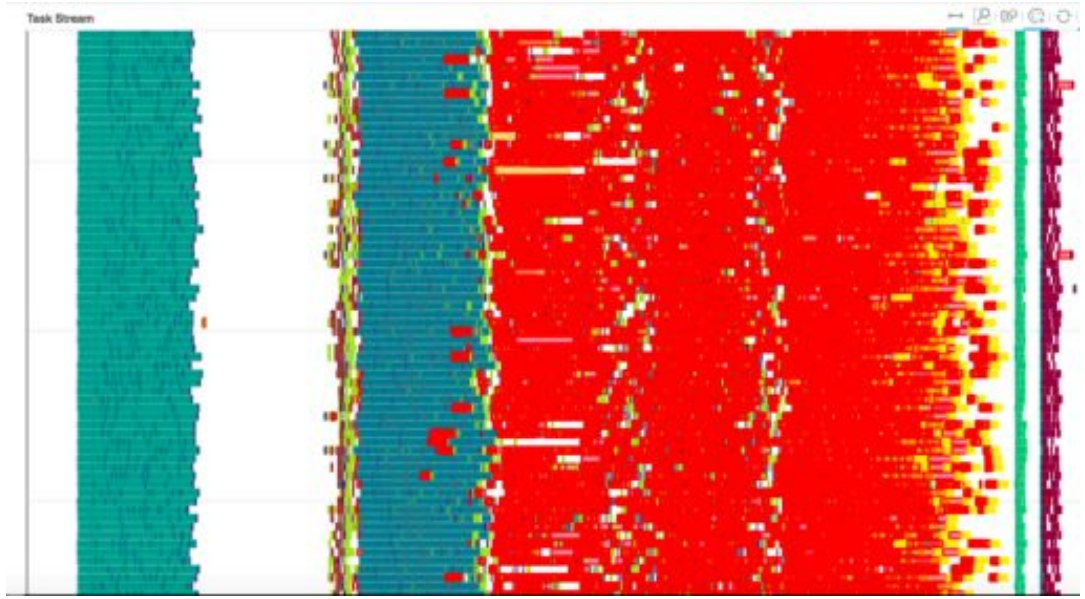
CuPy - CUDA Memory



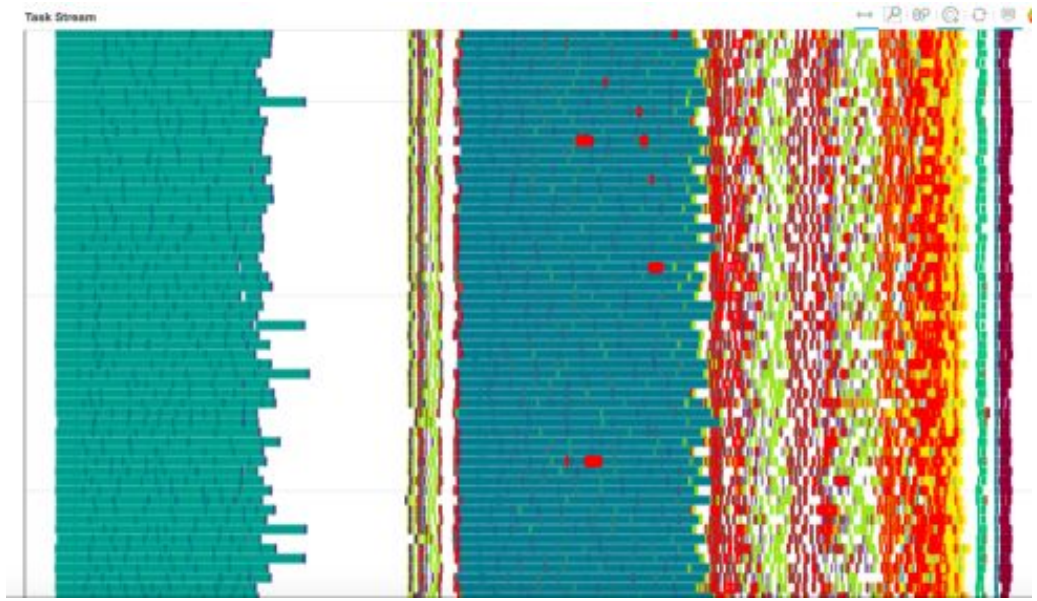


UCX-Py in Dask

RAPIDS GPU-BDB



Dask task stream with Python sockets
(Red is communication)



Dask task stream with UCX-Py
(Red is communication)

UCX-Py

Active Messages

- Provide more familiar interface for Python users
- Avoid unnecessary tag-matching overhead
- Metadata (size, memory type) can be packed into the message
- User can register allocator for received messages
- Zero-copy conversion possible for allocators supporting
  - `__array_interface__`
  - `__cuda_array_interface__`
  - Python buffer protocol

UCX-Py — Progress 2021

Tag Messages Example

Server

```
async def server(ep):
    # buffer -> __array_inteface__ / __cuda_array_interface__
    msg = numpy.zeros(10000, dtype='f8')

    # send tag message
    await ep.send(msg)
```

Client

```
async def client(ep):
    # buffer -> __array_inteface__ / __cuda_array_interface__
    tag_msg = numpy.empty(10000, dtype='f8')  # Must match sender

    # receive tag message into tag_msg
    await ep.recv(tag_msg)
```

UCX-Py — Progress 2021

Active Messages Example

Server

```
async def server(ep):
    # buffer -> __array_inteface__ / __cuda_array_interface__
    msg = numpy.zeros(10000, dtype='f8')

    # send tag message
    await ep.send(msg)

    # send active message
    await ep.am_send(msg)
```

Client

```
async def client(ep):
    # buffer -> __array_inteface__ / __cuda_array_interface__
    tag_msg = numpy.empty(10000, dtype='f8')  # Must match sender

    # receive tag message into tag_msg
    await ep.recv(tag_msg)

    ucp.register_am_allocator(
        lambda n: numpy.empty(n, dtype='u1'), 'host'
    )
    ucp.register_am_allocator(
        lambda n: cupy.empty(n, dtype='u1'), 'cuda'
    )

    # receive active message, no pre-allocation or prior knowledge
    # of size/memory type required
    am_msg = await ep.am_recv()
```

Live Tutorial


THANK YOU
Peter Entschev (NVIDIA), pentschev@nvidia.com
NVIDIA