UCX-Py

Peter Entschev (NVIDIA)
Matt Baker (ORNL)
Benjamin Zaitlen (NVIDIA)

November 30th, 2020

UCX-Py

What is it?

- "Pythonic" interface for UCX
- Easy to get started for Python developers
- Simple replacement for Python communications (e.g., sockets)

UCX-Py

Who is it for?

- Python developers
- No low-level communications, UCX or C knowledge required
- Data scientists benefit from it through applications such as Dask

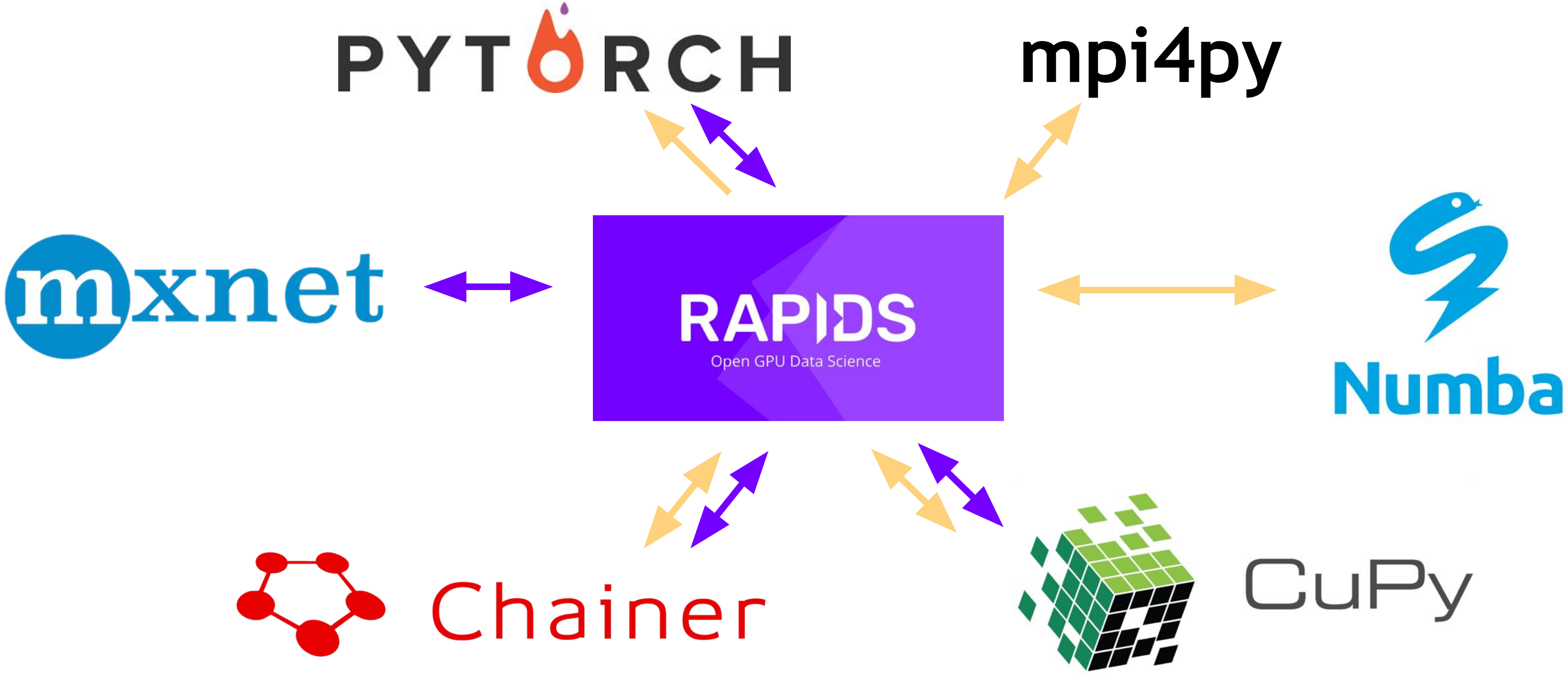
UCX-Py Primary Use Today

PyData — Contemporary Analytics

- PyData
  - Arrays
  - DataFrames
  - Machine Learning
  - Distributed Computing
- RAPIDS
  - GPU accelerated PyData

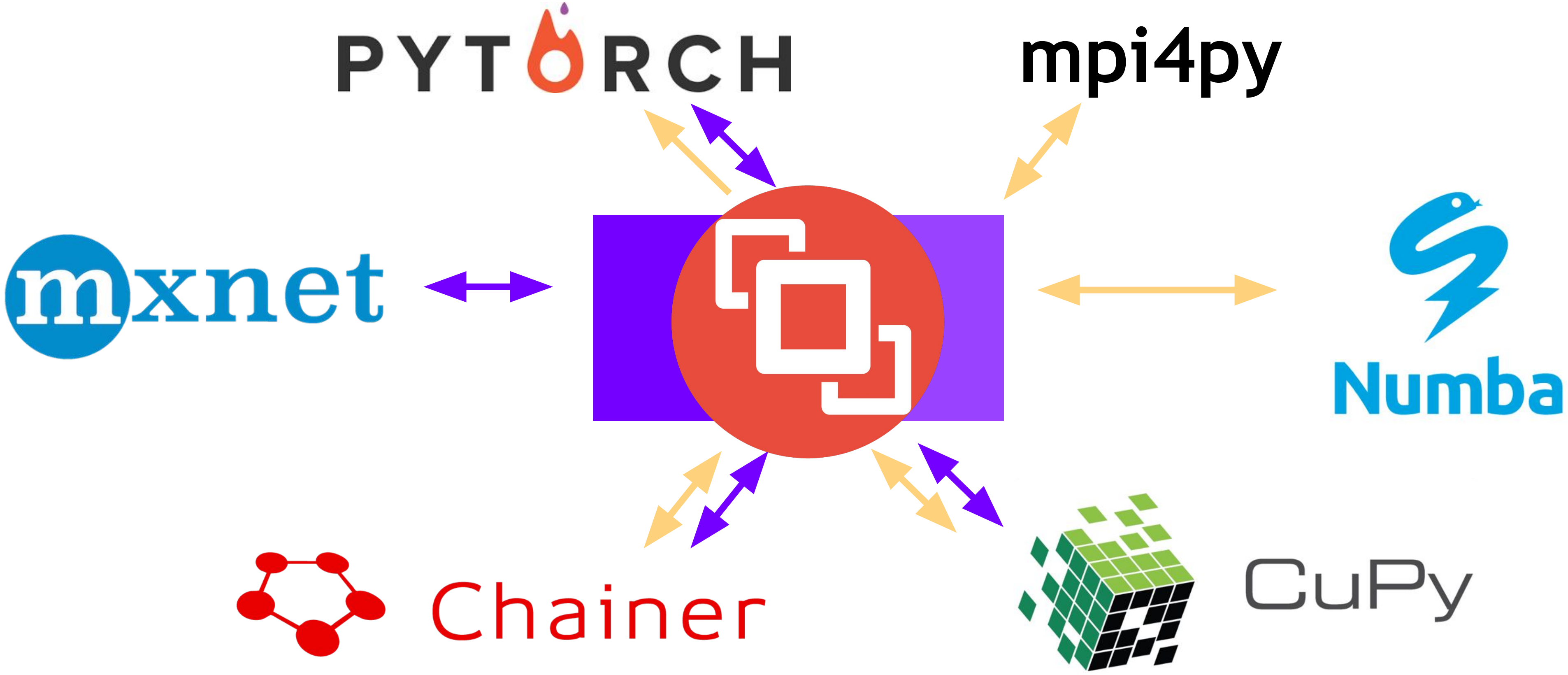
PyData Buffers

Community Interoperability with CUDA Array Interface and DLPack



PyData Buffers

Community Interoperability



Using UCX-Py

Installation

- Install from conda (with CUDA support)

```
conda create -n ucx -c conda-forge -c rapidsai \
cudatoolkit=<CUDA version> ucx-proc=*=gpu ucx ucx-py python=3.8
```

- Install from conda (without CUDA support)

```
conda create -n ucx -c conda-forge -c rapidsai \
cudatoolkit=<CUDA version> ucx-proc=*=cpu ucx ucx-py python=3.8
```

- Installing from source:

https://ucx-py.readthedocs.io/en/latest/install.html

Using UCX-Py

Supported Use Cases

- Officially supporting two use cases today:
    - Pure Python
    - RAPIDS / Dask-CUDA

Using UCX-Py

Python API

`ucp`
`ucp.create_listener`(callback_func[, port, …])
`ucp.create_endpoint`(ip_address, port[, …])
`ucp.get_address`([ifname])
`ucp.get_config`()
`ucp.get_ucp_worker`()
`ucp.get_ucx_version`()
`ucp.init`([options, env_takes_precedence, …])
`ucp.progress`()
`ucp.reset`()

`Endpoint`(endpoint, ctx, msg_tag_send, …)
`Endpoint.abort`()
`Endpoint.close`()
`Endpoint.closed`()
`Endpoint.close_after_n_recv`(n[, …])
`Endpoint.cuda_support`()
`Endpoint.get_ucp_endpoint`()
`Endpoint.get_ucp_worker`()
`Endpoint.recv`(buffer[, tag])
`Endpoint.send`(buffer[, tag])
`Endpoint.ucx_info`()
`Endpoint.uid`

`Listener`(backend)
`Listener.close`()
`Listener.closed`()
`Listener.port`

Using UCX-Py

Expected Python Usage

Server

```
async def server(ep):
    # buffer -> __array_inteface__ / __cuda_array_interface__
    msg = # allocate buffer
    await ep.send(msg)
```

Client

```
async def client(ep):
    # buffer -> __array_inteface__ / __cuda_array_interface__
    msg = # allocate buffer
    await ep.recv(msg)
```

Using UCX-Py

Send/Recv with CuPy

Server

```
async def send(ep):
    # recv buffer
    arr = cupy.empty(n_bytes, dtype='u1')
    await ep.recv(arr)
    assert (cupy.count_nonzero(arr) ==
            np.array(0, dtype=np.int64))
    print("Received CuPy array")

    # increment array and send back
    arr += 1
    print("Sending incremented CuPy array")
    await ep.send(arr)

    await ep.close()
    lf.close()

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

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

Client

```
async def main():
    host = ucp.get_address(ifname='eth0')  # device
    ep = await ucp.create_endpoint(host, port)
    msg = cupy.zeros(n_bytes, dtype='u1')  # data to send

    # send message
    print("Send Original CuPy array")
    await ep.send(msg)  # send the real message

    # recv response
    print("Receive Incremented NumPy arrays")
    resp = cupy.empty_like(msg)
    await ep.recv(resp)  # receive the echo
    await ep.close()
    cupy.testing.assert_array_equal(msg + 1, resp)
```

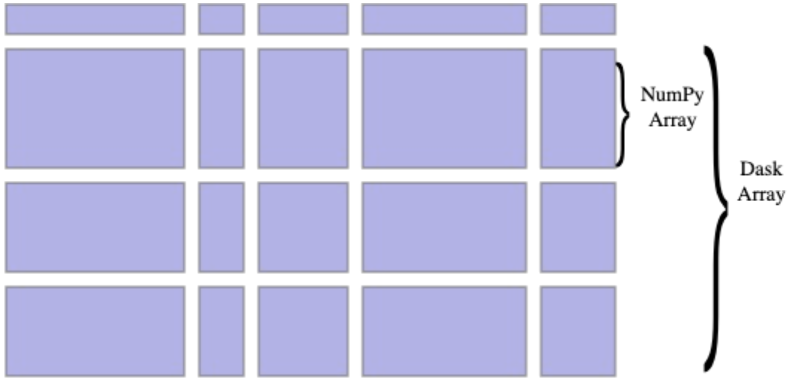
Dask

- General purpose Python library for parallelism
- Scales existing libraries, like NumPy, Pandas, and Scikit-Learn
- Flexible enough to build complex and custom systems
- Accessible for beginners, secure and trusted for institutions

Dask

Accelerates Existing Python Ecosystem

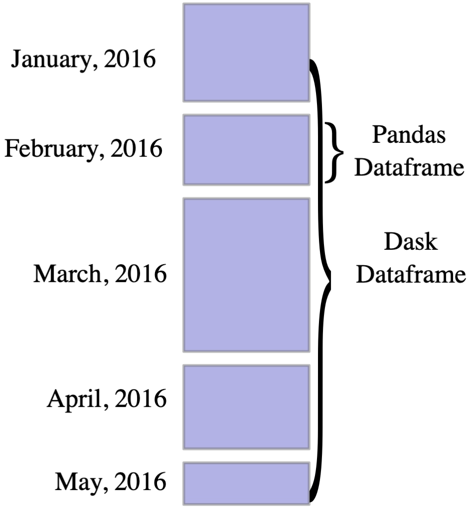
NumPy




```
import dask.array as da

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

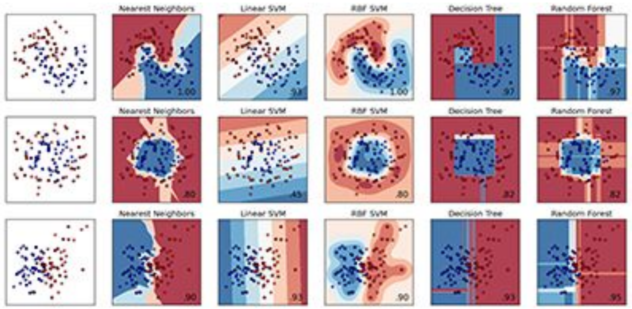
Pandas




```
import dask.dataframe as dd

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

Scikit-Learn

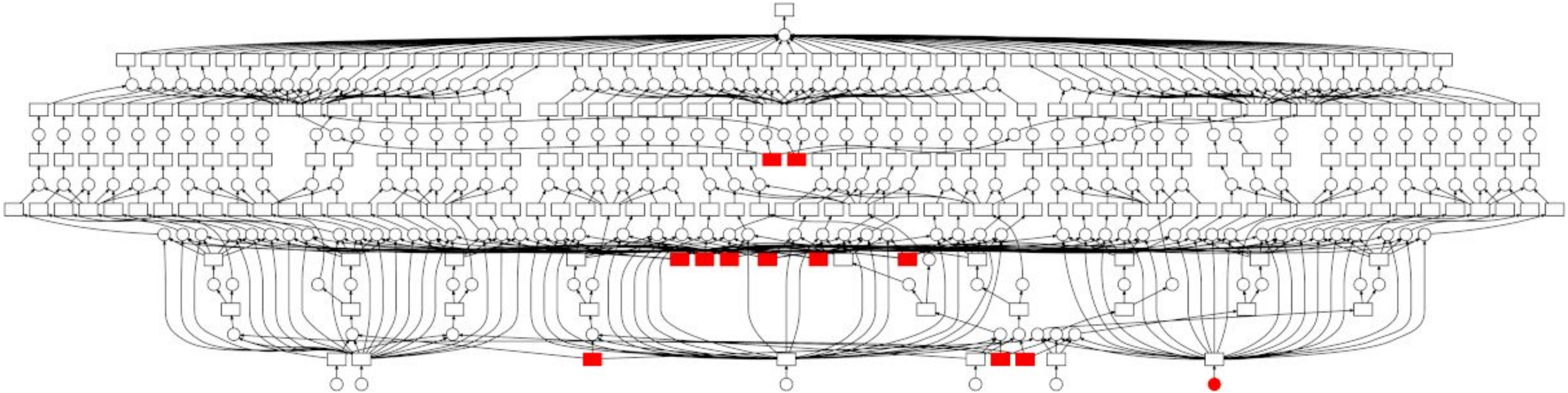


```
from dask_ml.linear_model \
    import LogisticRegression

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

Dask

Efficient Task Graph Execution on Parallel Hardware



Scikit-Learn cross-validated grid search over a pipeline

Using UCX-Py

RAPIDS / Dask-CUDA

Setup

```
from dask.distributed import Client
from dask_cuda import LocalCUDACluster
from dask_cuda.initialize import initialize

# ON/OFF settings for various transports
enable_tcp_over_ucx = True
enable_infiniband = False
enable_nvlink = False

cluster = LocalCUDACluster(
    interface="enp1s0f0",  # Ethernet interface
    protocol="ucx",
    enable_tcp_over_ucx=enable_tcp_over_ucx,
    enable_infiniband=enable_infiniband,
    enable_nvlink=enable_nvlink,
)
client = Cluster(client)
```

Drop-in replacement for TCP

Analysis

```
d1 = dask_cudf.from_cudf(..., npartitions=10)

d2 = dask_cudf.from_cudf(..., npartitions=10)

res = d1.merge(d2, how='inner', on=['id'])

res_sorted = res.sort_values(by='id')

res_sorted = res_sorted.persist()
```

Changes since UCF 2019

Releases 0.11 through 0.17

- Substantial increase in documentation
- API improvements and extensions
- Support for user-specified tags
- EndpointReuse – May be deprecated after cudaIpcOpenMemHandle fix
- Separating Python API from Cython backend
- Many Cython improvements reducing Python overhead
- Reduced hardcoded UCX variables – rely more on UCX defaults

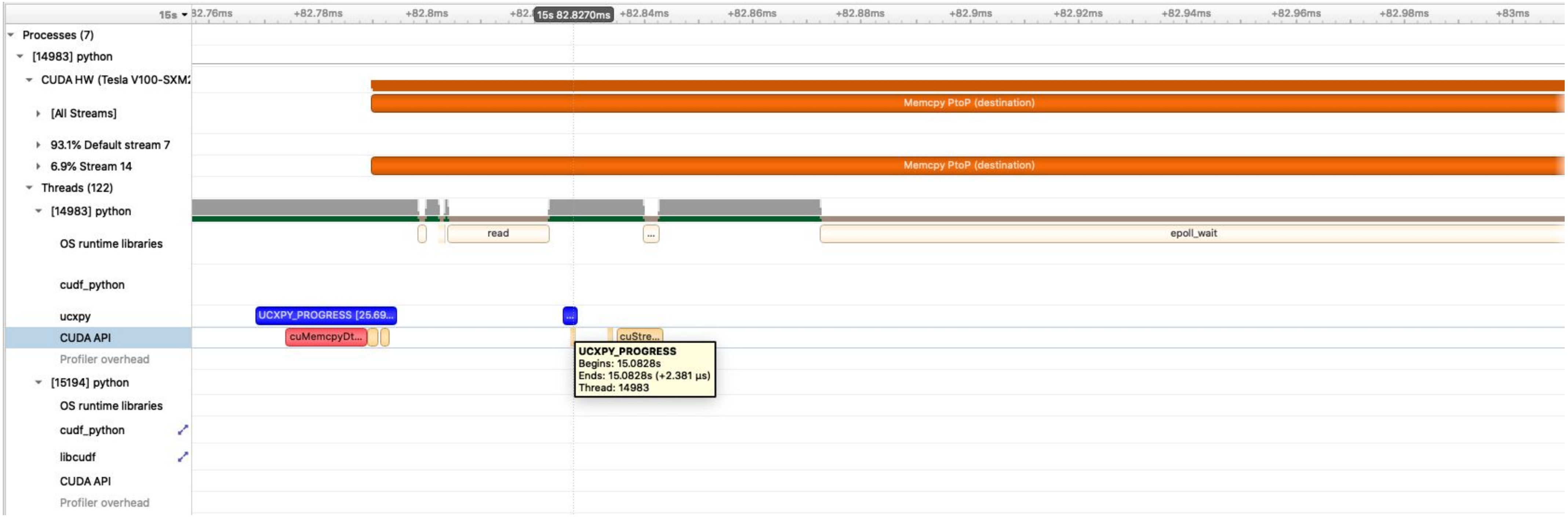
Changes since UCF 2019

Releases 0.11 through 0.17 (cont.)

- Many bug fixes, e.g., segfaults, deadlocks and cleanup issues
- Better error handling
- Support for endpoint error handler
- Logger formatting similar to UCX's
- NVTX annotations
- More benchmarks
- Multi-GPU and multi-node tests

Changes since UCF 2019

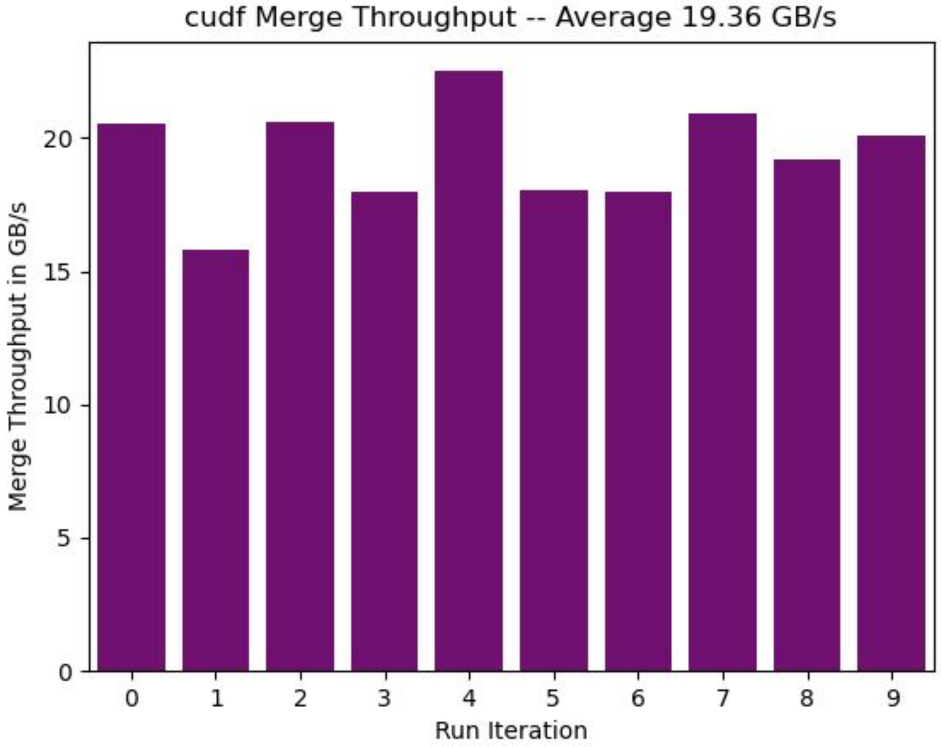
NVTX Annotations

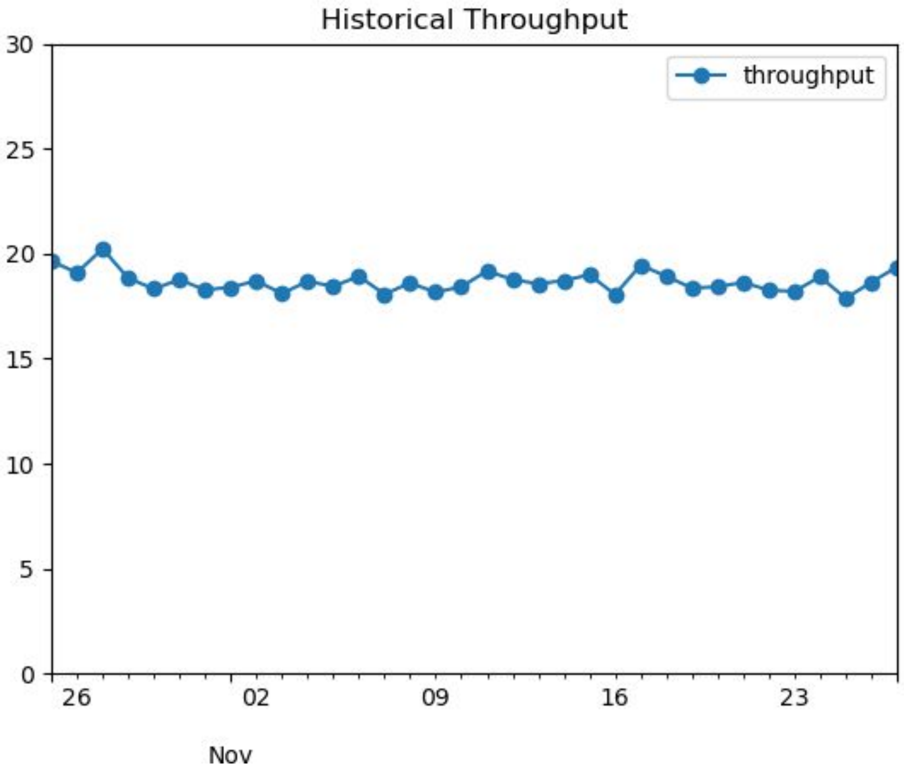


NVIDIA

Changes since UCF 2019

Testing and Benchmarking





Benchmarks

GPU-GPU Communications with cuDF

TCP

```
$ python dask_cuda/benchmarks/local_cudf_merge.py -p tcp  -d
1,2 -c 100_000_000
Merge benchmark
-------------------------------
backend        | dask
merge type     | gpu
rows-per-chunk | 100000000
protocol       | tcp
device(s)      | 1,2
rmm-pool       | True
frac-match     | 0.3
data-processed | 6.40 GB
===============================
Wall-clock     | Throughput
-------------------------------
17.07 s        | 374.98 MB/s
17.40 s        | 367.81 MB/s
17.27 s        | 370.62 MB/s
===============================
(w1,w2)     | 25% 50% 75% (total nbytes)
-------------------------------
(00,01)     | 273.70 MB/s 311.72 MB/s 401.11 MB/s (18.60 GB)
(01,00)     | 323.58 MB/s 355.55 MB/s 447.96 MB/s (18.60 GB)
```

~400 MB/s

CUDA IPC

```
$ python dask_cuda/benchmarks/local_cudf_merge.py -p ucx -d
1,2 -c 100_000_000
Merge benchmark
-------------------------------
backend        | dask
merge type     | gpu
rows-per-chunk | 100000000
protocol       | ucx
device(s)      | 1,2
rmm-pool       | True
frac-match     | 0.3
tcp            | True
ib             | True
nvlink         | True
data-processed | 6.40 GB
===============================
Wall-clock     | Throughput
-------------------------------
541.41 ms      | 11.82 GB/s
523.50 ms      | 12.23 GB/s
473.35 ms      | 13.52 GB/s
===============================
(w1,w2)     | 25% 50% 75% (total nbytes)
-------------------------------
(01,02)     | 33.17 GB/s 38.50 GB/s 40.26 GB/s (16.20 GB)
(02,01)     | 35.17 GB/s 39.39 GB/s 39.88 GB/s (16.20 GB)
```

~40 GB/s

Benchmarks

GPU-GPU Communications with cuDF

TCP

```
$ python dask_cuda/benchmarks/local_cudf_merge.py -p tcp  -d
1,2 -c 100_000_000
Merge benchmark
-------------------------------
backend        | dask
merge type     | gpu
rows-per-chunk | 100000000
protocol       | tcp
device(s)      | 1,2
rmm-pool       | True
frac-match     | 0.3
data-processed | 6.40 GB
===============================
Wall-clock     | Throughput
-------------------------------
17.07 s        | 374.98 MB/s
17.40 s        | 367.81 MB/s
17.27 s        | 370.62 MB/s
===============================
(w1,w2)     | 25% 50% 75% (total nbytes)
-------------------------------
(00,01)     | 273.70 MB/s 311.72 MB/s 401.11 MB/s (18.60 GB)
(01,00)     | 323.58 MB/s 355.55 MB/s 447.96 MB/s (18.60 GB)
```

~400 MB/s

InfiniBand

```
$ python dask_cuda/benchmarks/local_cudf_merge.py -p ucx -d
1,2 -c 100_000_000 --disable-nvlink --ucx-net-devices=auto
Merge benchmark
-------------------------------
backend        | dask
merge type     | gpu
rows-per-chunk | 100000000
protocol       | ucx
device(s)      | 1,2
rmm-pool       | True
frac-match     | 0.3
tcp            | True
ib             | True
nvlink         | False
data-processed | 6.40 GB
===============================
Wall-clock     | Throughput
-------------------------------
680.44 ms      | 9.41 GB/s
951.01 ms      | 6.73 GB/s
678.32 ms      | 9.44 GB/s
===============================
(w1,w2)     | 25% 50% 75% (total nbytes)
-------------------------------
(01,02)     | 8.48 GB/s 8.83 GB/s 9.04 GB/s (13.80 GB)
(02,01)     | 8.44 GB/s 8.53 GB/s 8.78 GB/s (13.80 GB)
```

~9 GB/s

UCX-Py Layers

- Currently divided in 3 layers:
  - Python: what the user sees
  - Cython: where Python interacts with C
  - C:
    - Utils that are considerably cleaner to write than with Cython
    - Code that interacts with other libraries (e.g., hwloc)

UCX-Py Data Type

Array

- `Array` class implemented in Cython
- Emulates NumPy's `ndarray` class
- Compliant with protocols
  - `__array_interface__`
  - `__cuda_array_interface__`
- Easy to interface with PyData libraries implementing protocols
  - NumPy, CuPy, Numba, etc.
- Efficient to interface with PyData and UCX

UCX-Py Use Cases

RAPIDS TPCx-BB

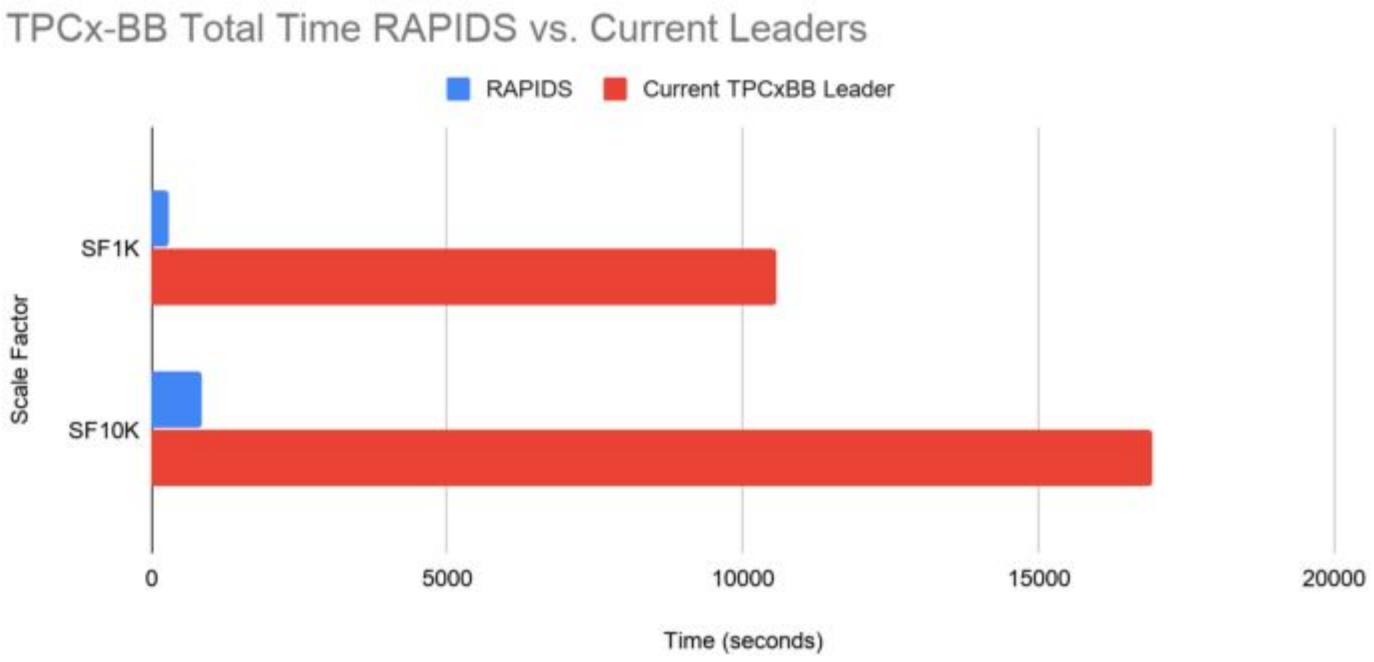
- Public benchmark of 30 big data analytics queries
- Represents real-world ETL and ML workflows
- Benchmarked by RAPIDS team on 1TB and 10TB scales

UCX-Py Use Cases

RAPIDS TPCx-BB

UCX-Py Use Cases

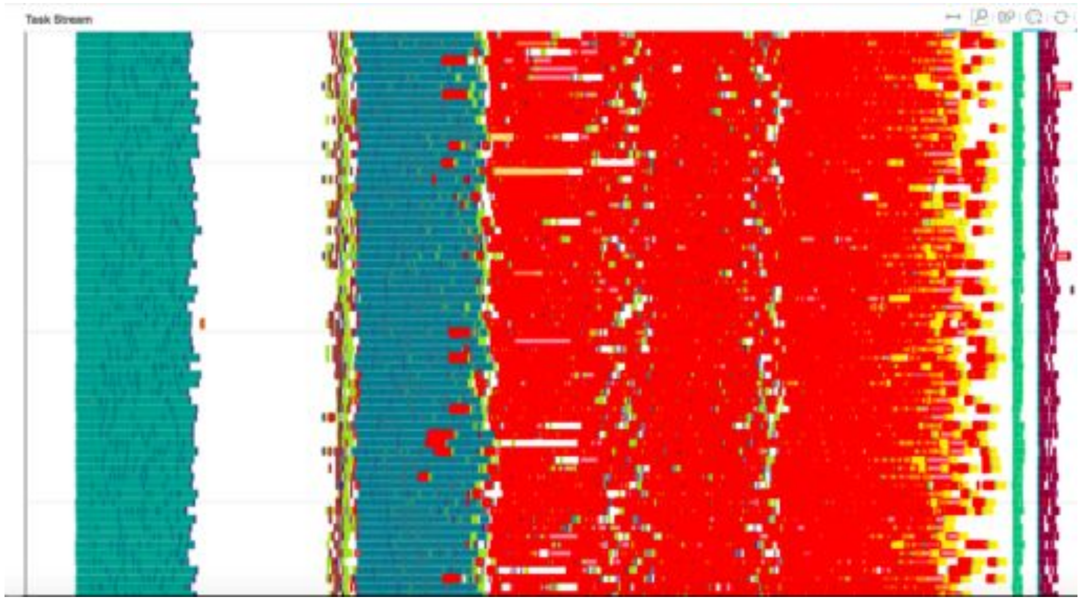
RAPIDS TPCx-BB



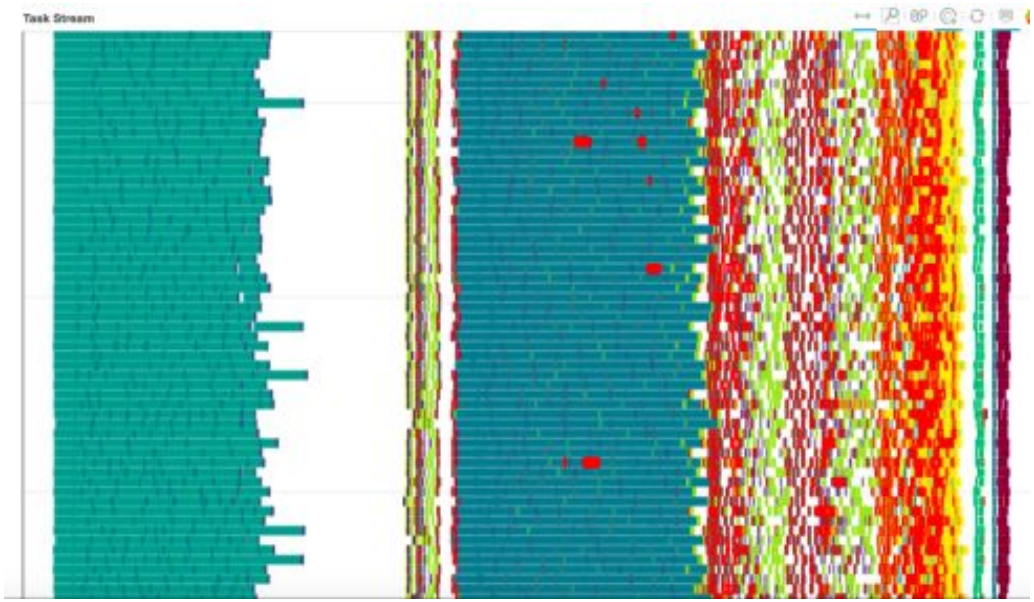
As of June 2020

UCX-Py Use Cases

RAPIDS TPCx-BB



Dask task stream with Python sockets
(Red is communication)

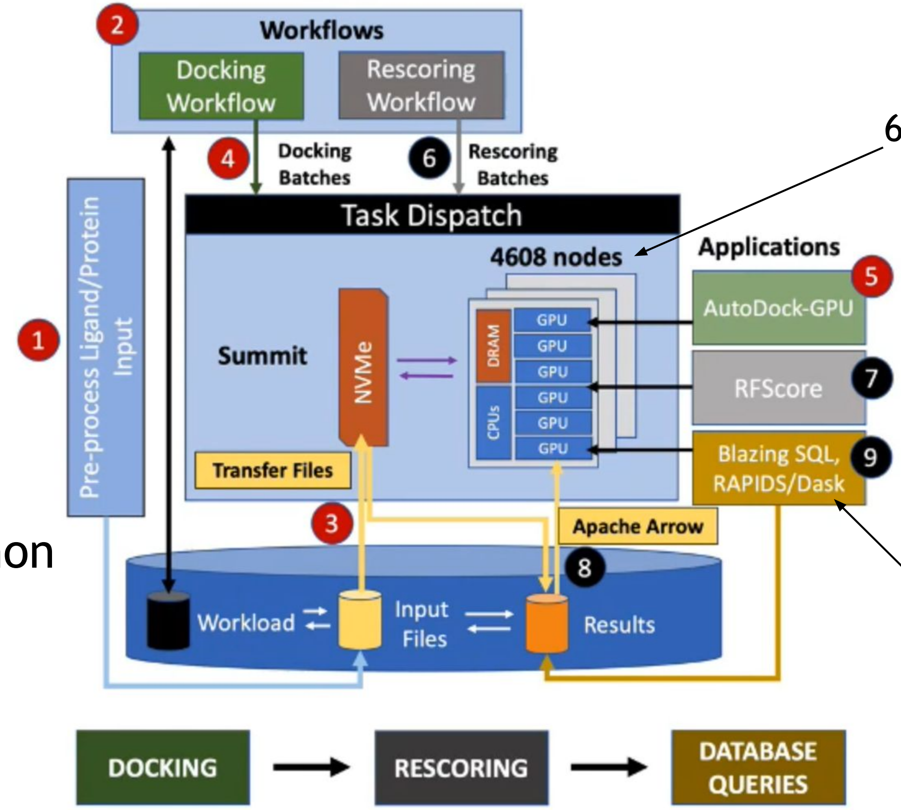


Dask task stream with UCX
(Red is communication)

UCX-Py Use Cases

Summit

- Research for COVID-19
- Structure-based drug discovery
- Total 52x single-node speedup
  - 42 CPU cores per node
  - 6 GPUs per node
- UCX responsible for ~2x speedup vs Python sockets (pure TCP)



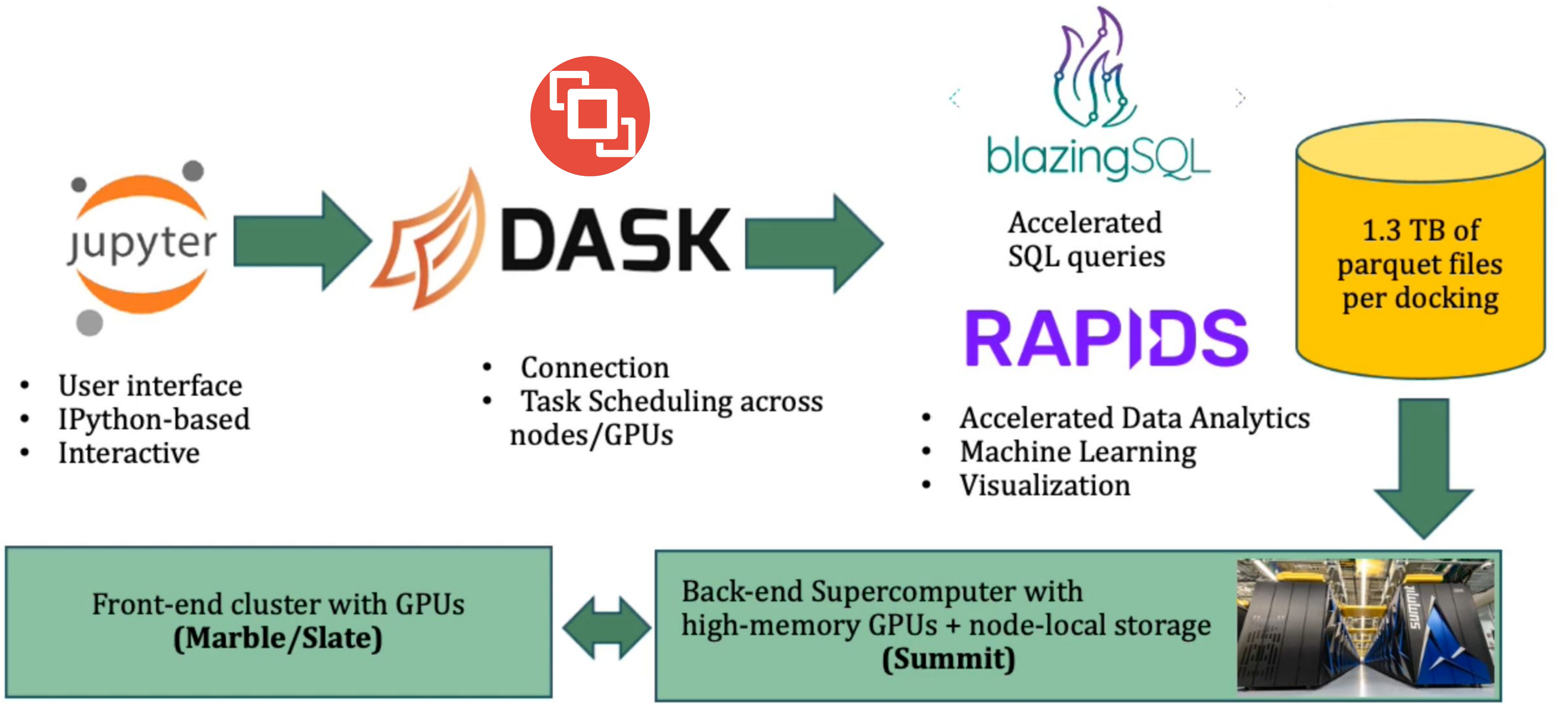
6 GPUs per node
=
27648 NVIDIA V100 GPUs

UCX-Py

Source:
Glaser, Jens, et al. "High-Throughput Virtual Laboratory for Drug Discovery Using Massive Datasets", SC20

UCX-Py Use Cases

Summit



Source:
Glaser, Jens, et al. "High-Throughput Virtual Laboratory for Drug Discovery Using Massive Datasets", SC20

UCX-Py Challenges

UCX-Py:

- Support for UCX conda packages with IB
- Conda packages supporting every transport
- Multi-threading support
- Shared memory support (double check, probably fine in 1.9.0)
- Further latency reduction

UCX:

- CUDA UVM support

Upstreaming UCX-Py to OpenUCX

- In the process of completely separating API and Cython backend
- Our plan was to do it in 2020, but got delayed to early 2021
- Current plan is to upstream in 6 phases:
  - C backend
  - Backend (Cython) API
  - Backend (Cython) code
  - Frontend (Python) API
  - Frontend (Python) code
  - Python packaging

UCX-Py and the Future of HPC

- Future compute clusters will get more complicated
- Future use cases will also get more sophisticated and more demanding
- Urgent need for a way to explore run time design space in a rapid way that is accessible to new programmers
- While performance is critical, some performance may be traded for a more accessible code base that is friendlier to explore new design patterns and runtime systems

Experiences based on Summit + Covid-19

- Run times such as Dask + UCX pushed to bleeding edge
    - Lots of bleeding
- Urgent nature of Covid work meant that performance portability and application readiness were critical
    - From zero to docking and scoring 1.6 billion compounds in months, not years
    - Software written to leverage UCX scales well on many different hardware configurations
    - Efficient data transfer and scheduling, whether running on a DGX box or a full Summit run
    - How can we do better?

Experiences based on Summit + Covid-19

- UCX + IO in task graph
  - How to arrange IO as a part of the compute graph?
  - How to do light reprocessing and data format changes in line with UCX?

References

- UCX-Py documentation
  - https://ucx-py.readthedocs.io/
- TPCx-BB Medium blogpost
  - https://medium.com/rapids-ai/no-more-waiting-interactive-big-data-now-32f7b903cf41
- Glaser, Jens, et al. "High-Throughput Virtual Laboratory for Drug Discovery Using Massive Datasets", SC20
  - https://sc20.supercomputing.org/presentation/?id=cov103&sess=sess388
  - https://www.youtube.com/watch?v=c2nwJlgUwrk

THANK YOU

Peter Entschev (NVIDIA), pentschev@nvidia.com
Matt Baker (ORNL), bakermb@ornl.gov
Benjamin Zaitlen (NVIDIA), bzaitlen@nvidia.com