UCX-PYTHON: A FLEXIBLE COMMUNICATION LIBRARY FOR PYTHON APPLICATIONS

March 21, 2018

OUTLINE

Motivation and goals

Implementation choices

Features/API

Performance

Next steps

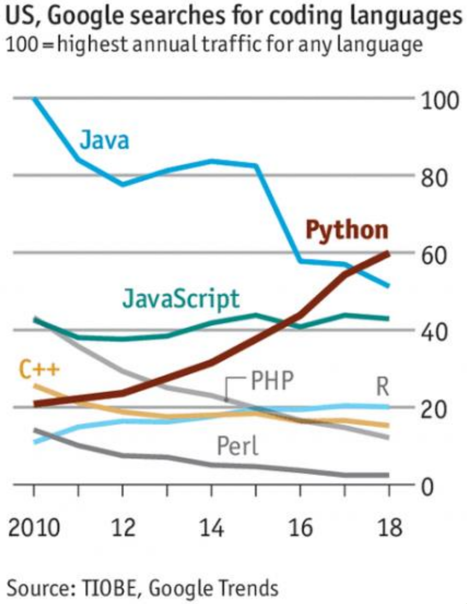
WHY PYTHON-BASED GPU COMMUNICATION?

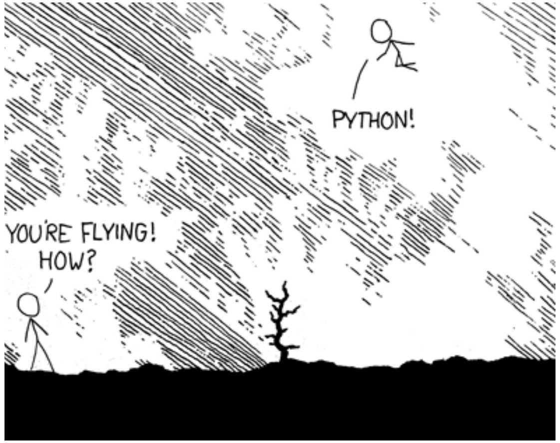
Python use growing

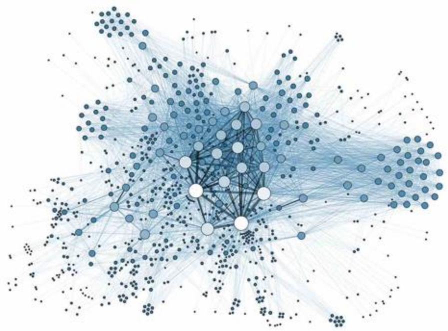
Extensive libraries

Python in Data science/HPC is growing

+ GPU usage and communication needs







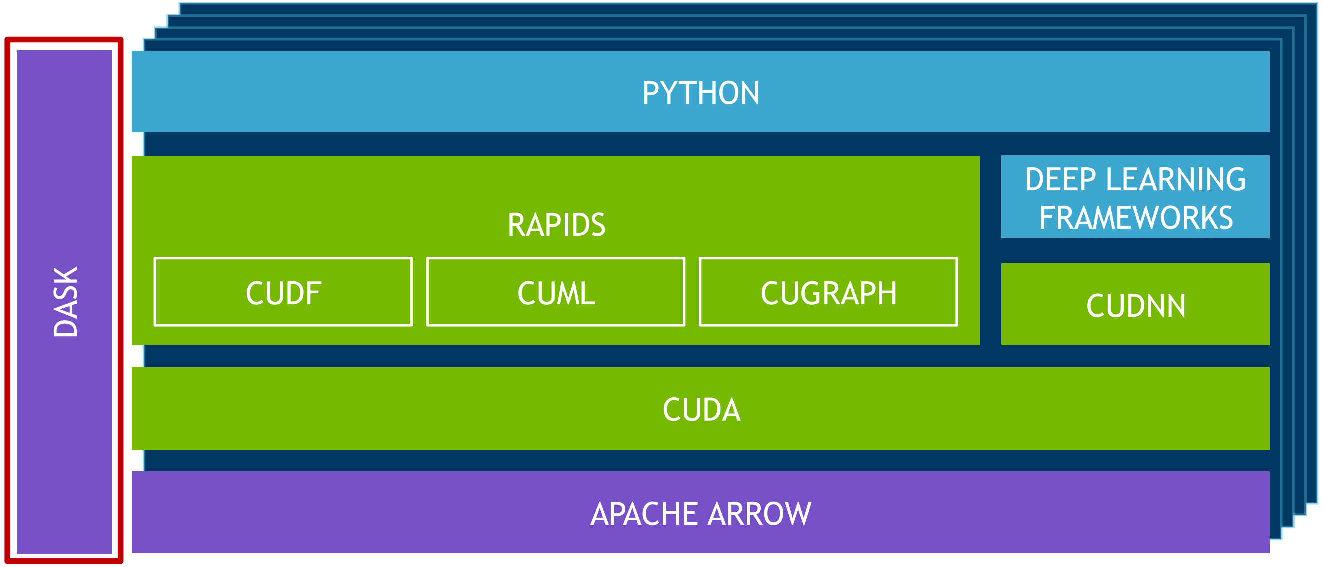
Numba

IMPACT ON DATA SCIENCE

RAPIDS uses dask-distributed for data distribution over python sockets

=> slows down all communication-bound components

Critical to enable dask with the ability to leverage IB, NVLINK

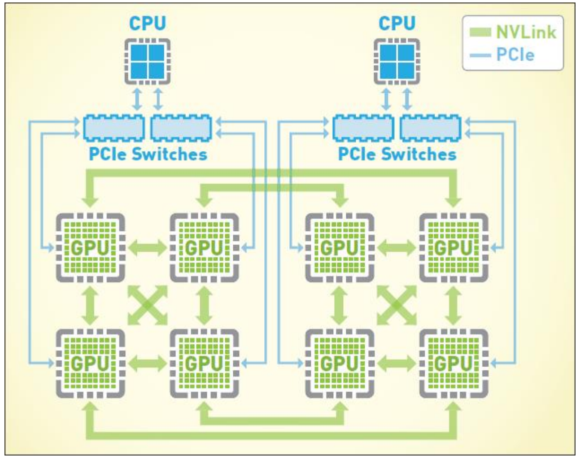


Courtesy RAPIDS Team

CURRENT COMMUNICATION DRAWBACKS

Existing python communication modules primarily rely on **sockets**

Low latency / high bandwidth critical for better system utilization of GPUs (eg: **NVLINK, IB**)



Frameworks that transfer GPU data between sites **make copies**

But CUDA-aware data movement is largely solved in HPC!

REQUIREMENTS AND RESTRICTIONS

Dask – popular framework facilitates scaling python workloads to many nodes

- Permits use of cuda-based python objects
- Allows workers to be added and removed dynamically
- communication backed built around coroutines (more later)

Why not use mpi4py then?

Dimension	mpi4py
CUDA-Aware?	No - Makes GPU<->CPU copies
Dynamic scaling?	No - Imposes MPI restrictions
Coroutine support?	No known support

GOALS

Provide a flexible communication library that:

1. Supports CPU/GPU buffers over a range of message types

    - raw bytes, host objects/memoryview, cupy objects, numba objects

2. Supports Dynamic connection capability

3. Supports pythonesque programming using Futures, Coroutines, etc (if needed)

4. Provides close to native performance from python world

How? – Cython, UCX

OUTLINE

Motivation and goals

Implementation choices

Features/API

Performance

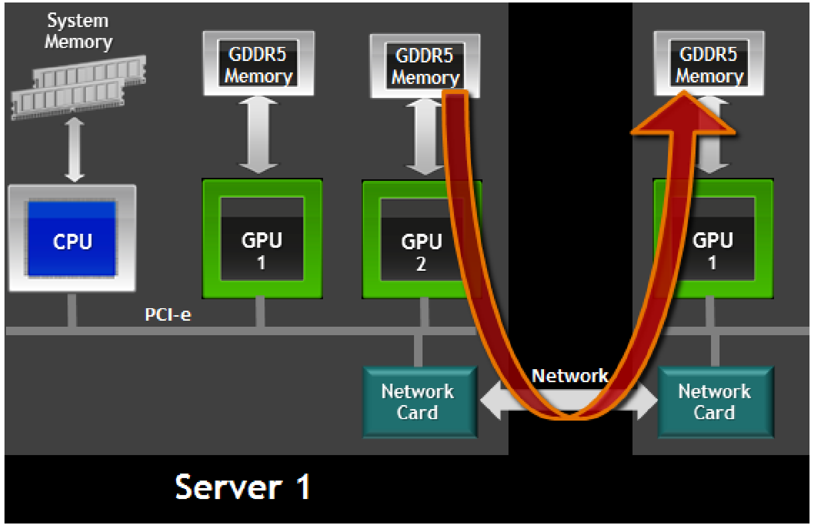
Next steps

WHY UCX?

Popular unified communication library used for MPI/PGAS implementations such as OpenMPI, MPICH, OSHMEM, etc

Exposes API for:

- Client-server based connection establishment
- Point-to-point, RMA, atomics capabilities
- Tag matching
- Callbacks on communication events
- Blocking/Polling progress
- Cuda-Aware Point-to-point communication



C library!

PYTHON BINDING APPROACHES

Three main considerations:

- SWIG, CFFI, Cython

Problems with SWIG, CFFI

- Works well for small examples but not for *C libraries*
- UCX definitions of structures isn't consolidated
- Tedious to populate interface file / python script by hand

CYTHON

Call C functions and structures from cython code (.pyx)

Expose classes, functions from python which can use C underneath

```
ucx_echo.py
...

ep=ucp.get_endpoint()
ep.send_obj(…)
…
```

```
ucp_py.pyx

cdef class ucp_endpoint:
    def send_obj(…):
        ucp_py_send_nb(…)
```

```
ucp_py_ucp_fxns.c

struct ctx *ucp_py_send_nb()
{
    ucp_tag_send_nb(…)
}
```

Defined in UCX-PY module

Defined in UCX C library

UCX-PY STACK

UCX-PY (.py)
OBJECT META-DATA EXTRACTION/ UCX C-WRAPPERS (.pyx)
RESOURCE MANAGEMENT/CALLBACK HANDLING/UCX CALLS(.c)
UCX C LIBRARY

OUTLINE

Motivation and goals

Implementation choices

Features/API

Performance

Next steps

COROUTINES

Co-operative concurrent functions

Preempted when

- read/write from disk
- perform communication
- sleep, etc

Scheduler/event loop manages execution of all coroutines

Single thread utilization increases

```
def zzz(i):
    print("start", i)
    time.sleep(2)
    print("finish", i)

def main():
    zzz(1)
    zzz(2)

main()

Ouput:

start 1     # t = 0
finish 1    # t = 2
start 2     # t = 2 + △
finish 2    # t = 4 + △
```

```
async def zzz(i):
    print("start" , i)
    await asyncio.sleep(2)
    print("finish“, i)

f = asyncio.create_task

async def main():
    task1 = f(zzz(1))
    task2 = f(zzz(2))
    await task1
    await task2

asyncio.run(main())

start 1     # t = 0
start 2     # t = 0 + △
finish 1    # t = 2
finish 2    # t = 2 + △
```

UCX-PY CONNECTION ESTABLISHMENT API

Dynamic connection establishment

.start_listener(accept_cb, port, is_coroutine) : Server creates listener

.get_endpoint (ip, port) : client connects

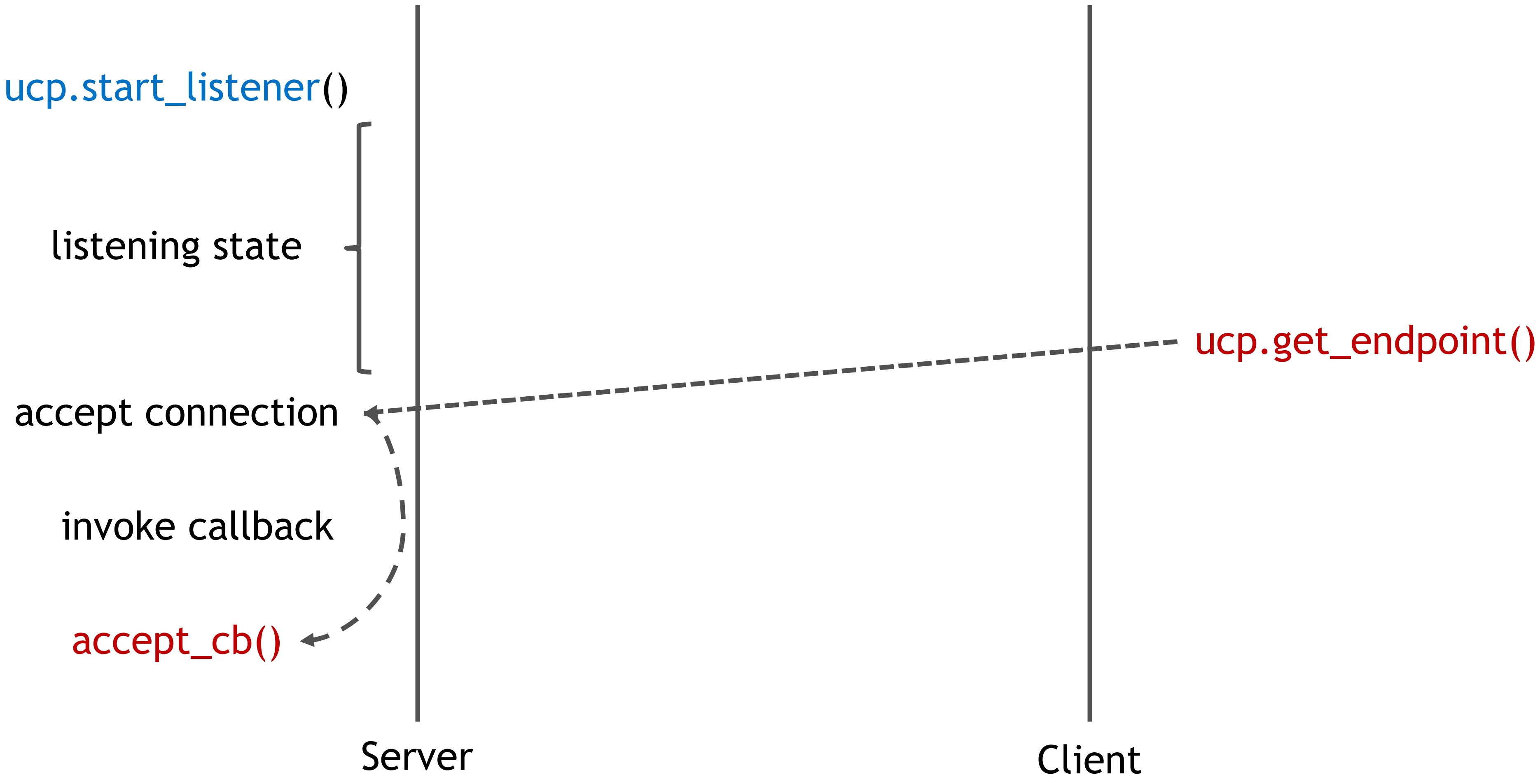
Multiple listeners allowed, multiple endpoints to server allowed

```
async def accept_cb(ep, …):
        …
        await ep.send_obj()
        …
        await ep.recv_obj()
        …

ucp.start_listener(accept_cb, port,
is_coroutine=True)
```

```
async def talk_to_client():
        ep = ucp.get_endpoint(ip, port)
        …
        await ep.recv_obj()
        …
        await ep.send_obj()
        …
```

UCX-PY CONNECTION ESTABLISHMENT



UCX-PY DATA MOVEMENT API

Send data (on endpoint)

.send_*() : raw bytes, host objects (numpy), cuda objects (cupy, numba)

Receive data (on endpoint)

.recv_obj() : pass an object as argument where data is received

.recv_future() 'blind' : no input; returns received object; *low performance*

```
async def accept_cb(ep, …):
    …
    await ep.send_obj(cupy.array([42]))
    …
```

```
async def talk_to_client():
    ep = ucp.get_endpoint(ip, port)
    …
    rr = await ep.recv_future()
    msg = ucp.get_obj_from_msg(rr)
    …
```

UCX-PY DATA MOVEMENT SEMANTICS

Send/Recv operations are non-blocking by default

Issue of the operation returns a future

Calling await on the future or calling future.result() blocks until completion

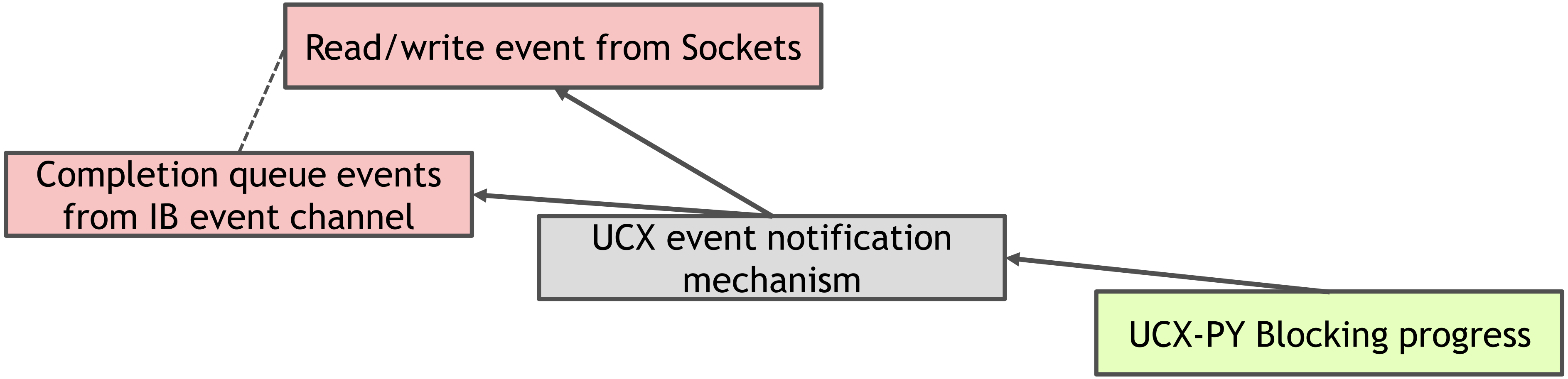
Caveat - Limited number of object types tested

memoryview, numpy, cupy, and numba

UNDER THE HOOD

Layer	UCX Calls
Connection management	ucp_{listener/ep}_create
Issuing data movement	ucp_tag_{send/recv/probe}_nb
Request progress	ucp_worker_{arm/signal/progress}

UCX depends on event notification to avoid the main thread from constantly polling



OUTLINE

Motivation and goals

Implementation choices

Features/API

Performance

Next steps

EXPERIMENTAL TESTBED

Hardware includes 2 Nodes:

Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz

Tesla V100-SXM2 (CUDA 9.2.88, driver version 410.48)

ConnectX-4 Mellanox HCAs (OFED-internal-4.0-1.0.1)

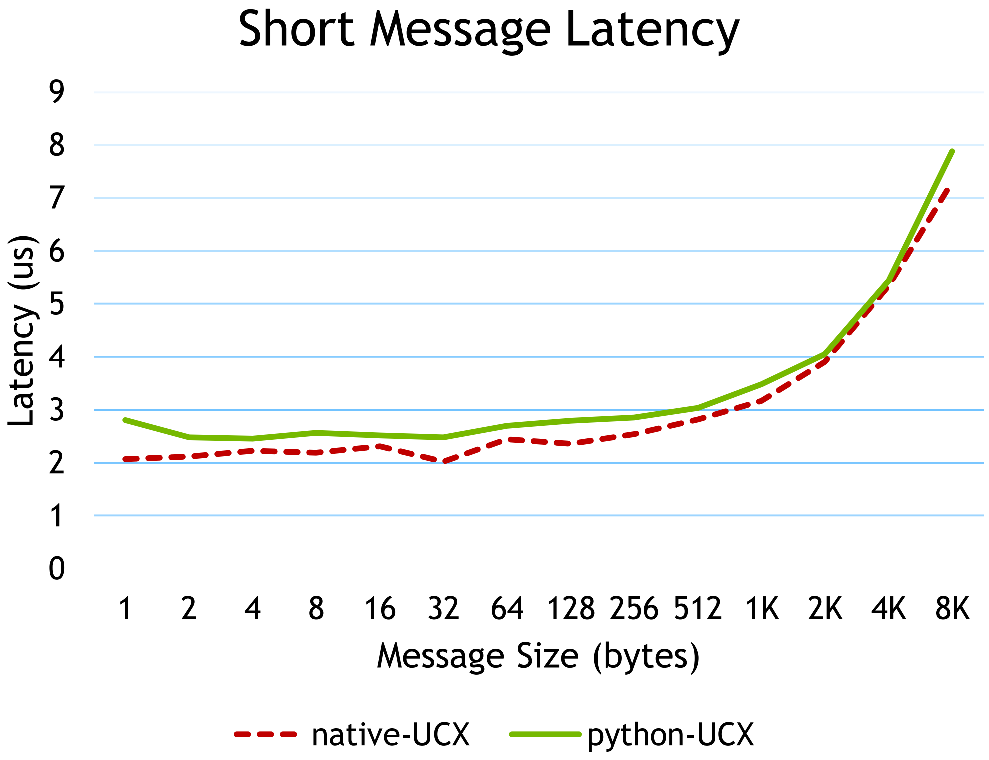
Software:

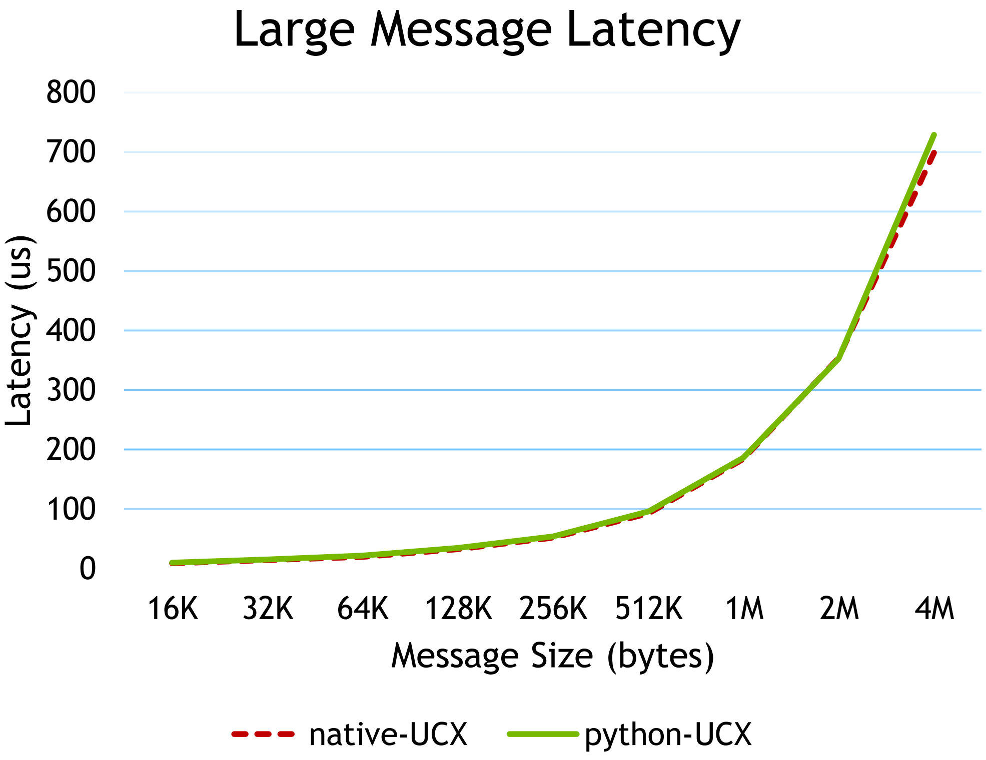
UCX 1.5, Python 3.7.1

Case	UCX progress mode	Python functions
Latency bound	polling	regular
Bandwidth bound	Blocking (event notification based)	coroutines

HOST MEMORY LATENCY

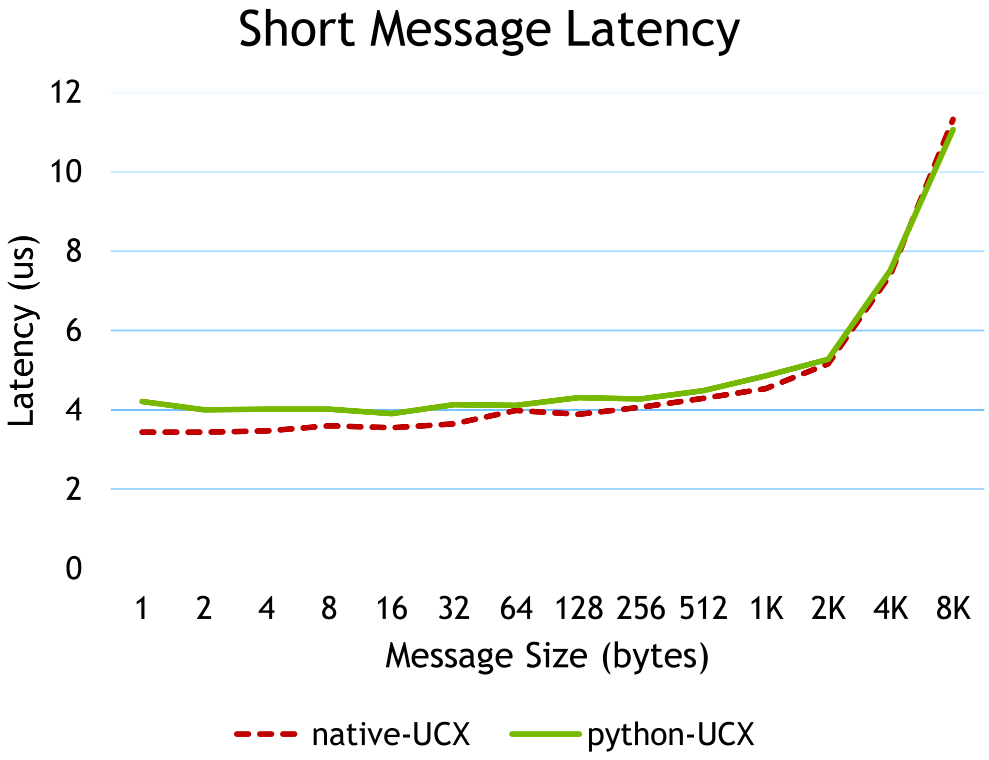
Latency-bound host transfers

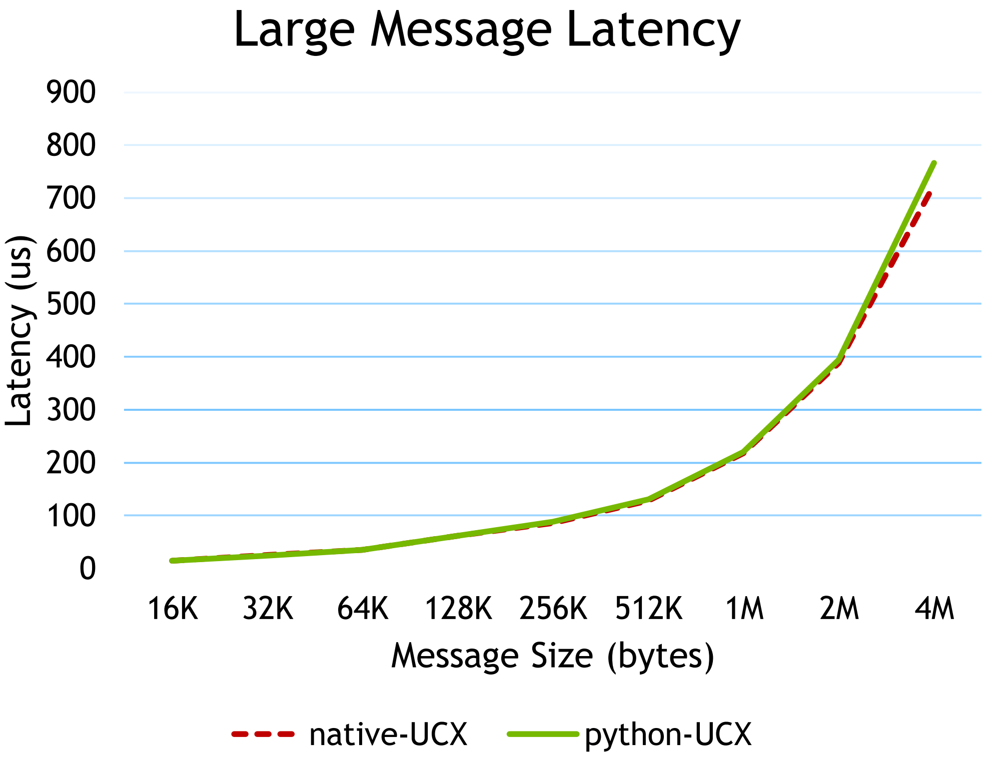




DEVICE MEMORY LATENCY

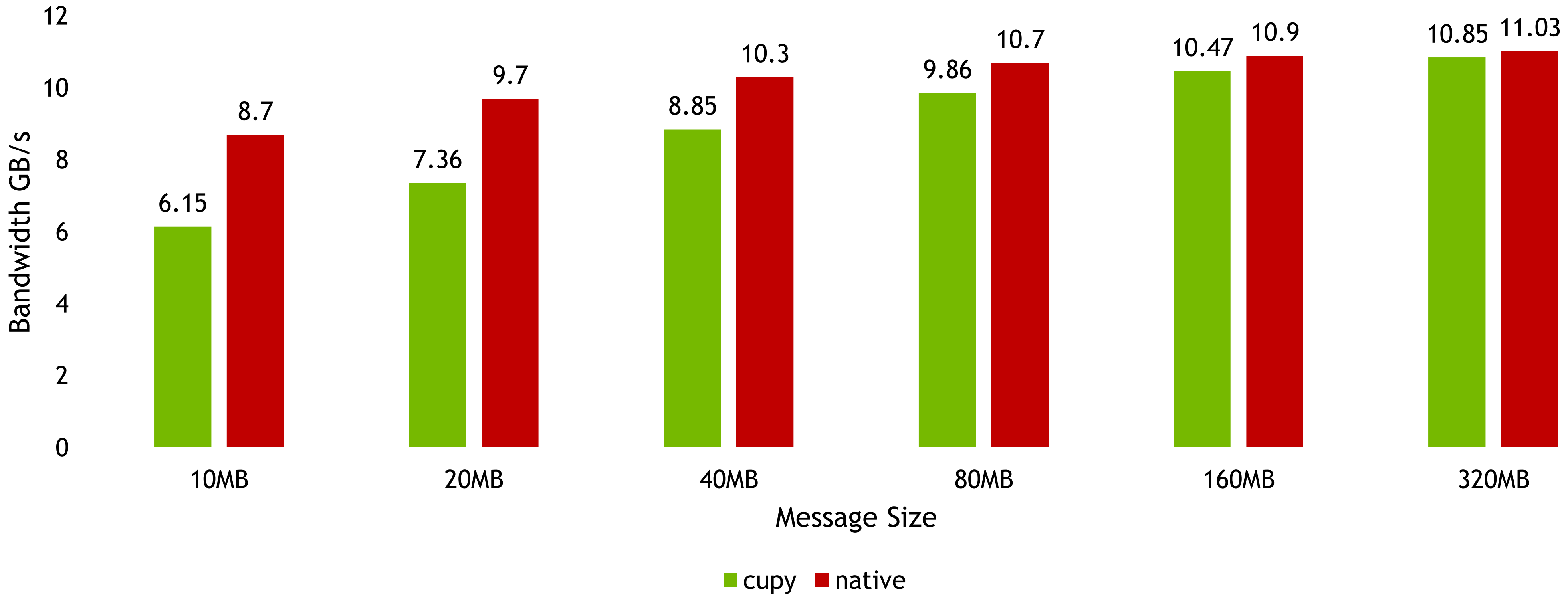
Latency-bound device transfers





DEVICE MEMORY BANDWIDTH

Bandwidth-bound transfers (cupy)



OUTLINE

Motivation and goals

Implementation choices

Features/API

Performance

Next steps

NEXT STEPS

Performance validate dask-distributed over UCX-PY with dask-cuda workloads

Objects that have mixed physical backing (CPU and GPU)

Adding blocking support to NVLINK based UCT

Non-contiguous data transfers

Integration into dask-distributed underway (https://github.com/TomAugspurger/distributed/commits/ucx+data-handling)

Current implementation (https://github.com/Akshay-Venkatesh/ucx/tree/topic/py-bind)

Push to UCX project underway (https://github.com/openucx/ucx/pull/3165)

SUMMARY

UCX-PY is a flexible communication library

Provides python developers a way to leverage high-speed interconnects like IB

Can support pythonesque way of overlap communication with other coroutines

Or can be non-overlapped like in traditional HPC

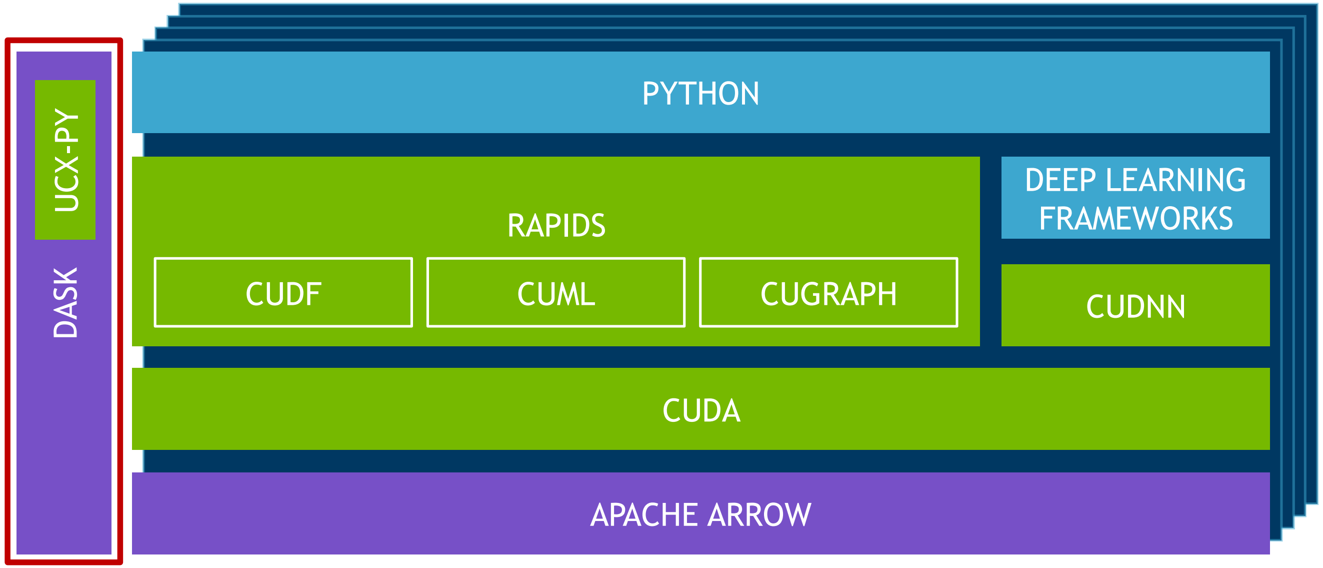
Can support data-movement of objects residing on CPU memory or on GPU memory

- users needn’t explicitly copy GPU<->CPU

UCX-PY is close to native performance for major use case range

BIG PICTURE

UCX-PY will serve as a high-performance communication module for dask



NVIDIA