UCX-PY, RAPIDS AND DISTRIBUTED JOINS

Benjamin Zaitlen, Nikolay Sakharnykh, Dec 11, 2019

PYDATA

Contemporary Analytics

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

```
conda create -n ucx -c conda-forge -c rapidsai \
   cudatoolkit=<CUDA version> ucx-proc=*=gpu ucx ucx-py
 python=3.7
```

General purpose Python library for parallelism

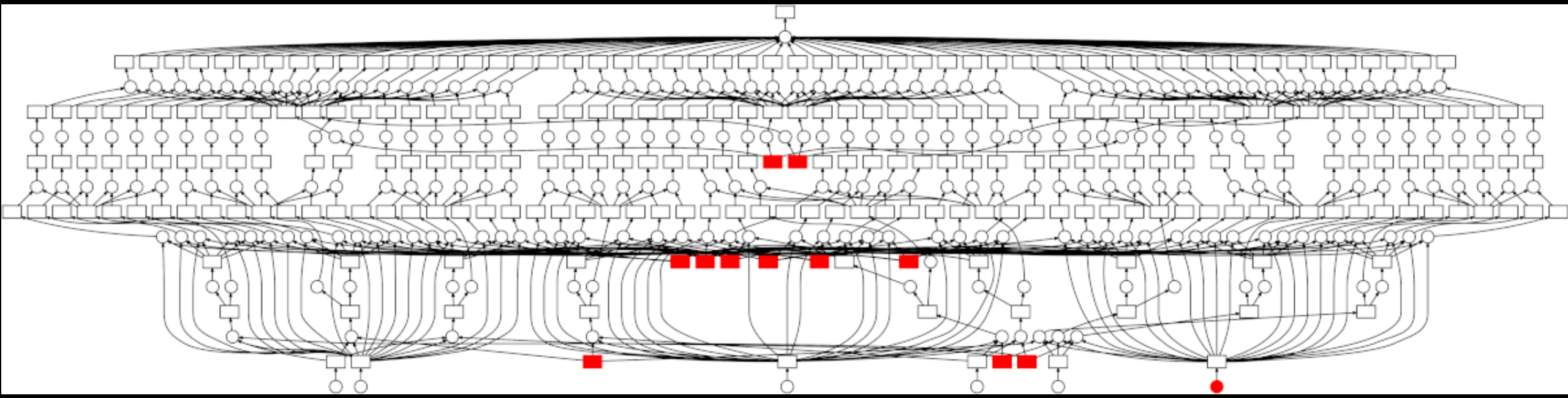
Scales existing libraries, like Numpy, Pandas, and Scikit-Learn

Flexible enough to build complex and custom systems

Accessible for beginners, secure and trusted for institutions

DASK SCHEDULERS THEN COMPUTE THEM

DASK EXECUTES TASK GRAPHS EFFICIENTLY OVER PARALLEL HARDWARE



Scikit-Learn cross-validated grid search over a pipeline

UCX-PY

Context

- Python
  - Multiple Threads
  - Multiple Processes
- Various CUDA Libraries
  - CuPy
    - CUDA Runtime API
  - cuDF
    - Mixed Runtime/Driver API
  - Numba
    - Driver API
- Env
  - Baremetal
  - Container
  - Cloud

- Python >=3.6
- Cython Bindings
  - C-Extensions for Python

PYTHON COMMS

Expected Python Usage

```
async def client_server(ep):
    # buffer -> __array_inteface __cuda_array_interface__
    msg = # allocate buffer
    await ep.send(msg)
```

```
async def client_server(ep):
    # buffer -> __array_inteface__cuda_array_interface__
    msg = # allocate buffer
    await ep.recv(msg)
```

UCX-PY EXAMPLE

Send/Recv with NumPy

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

    # increment array and send back
    arr += 1
    print("Sending incremented NumPy 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)
```

```
async def main():
    host = ucp.get_address(ifname='eth0')  # device
    ep = await ucp.create_endpoint(host, port)
    msg = np.zeros(n_bytes, dtype='u1') # data to send
    msg_size = np.array([msg.nbytes], dtype=np.uint64)

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

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

EXAMPLE

Send/Recv with CuPy

```
async def send(ep):
    # recv buffer
    arr = cp.empty(n_bytes, dtype='u1')
    await ep.recv(arr)
    assert cp.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)
```

```
async def main():
    host = ucp.get_address(ifname='eth0')  # device
    ep = await ucp.create_endpoint(host, port)
    msg = cp.zeros(n_bytes, dtype='u1') # data to send
    msg_size = cp.array([msg.nbytes], dtype=np.uint64)

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

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

PUBLIC API

Consumable Interface

Closing EPs can be challenging

```python
def create_listener(callback_func, port=None, guarantee_msg_order=True):
    """Create and start a listener to accept incoming connections"""
    ...

class Listener:
    def __init__(self, backend):
        self._b = backend
        self._closed = False

    def closed(self):

    @property
    def port(self):

    def close(self):
```

```python
async def create_endpoint(ip_address, port, guarantee_msg_order=True):
    """Create a new endpoint to a server"""
    ...
class Endpoint:
    def abort(self):
        """Close the communication immediately and abruptly."""
    async def close(self):
        """Close the endpoint cleanly."""
    def close_after_n_recv(self, n, count_from_ep_creation=False):
        """Close the endpoint after `n` received messages."""
    async def send(self, buffer, nbytes=None):
        """Send `buffer` to connected peer."""
    async def recv(self, buffer, nbytes=None):
        """Receive from connected peer into `buffer`."""
    def ucx_info(self):
        """Return low-level UCX info about this endpoint as a string"""
    def cuda_support(self):
        """Return whether UCX is configured with CUDA support or not"""
    def get_ucp_worker(self):
    def get_ucp_endpoint(self):
```

INTERNALS

Endpoints

- Two Comms
- Main comm for msg passing
- Secondary comm for control
  - close
  - close remote listener (future work)
  - system info/memory consumption (future work)
- Secondary Comm can be included in application but we do this enough that ucx-py may undertake this work

```
async def create_endpoint(self, str ip_address, port,
                          guarantee_msg_order):
    # We create the Endpoint in four steps:
    #  1) Generate unique IDs to use as tags
    #  2) Exchange endpoint info such as tags
    #  3) Use the info to create the private part of an endpoint
    #  4) Create the public Endpoint based on _Endpoint
    msg_tag = hash(uuid.uuid4())
    ctrl_tag = hash(uuid.uuid4())
    peer_info = await exchange_peer_info(
        ...
        msg_tag=msg_tag,
        ctrl_tag=ctrl_tag,
    )
    ep = _Endpoint(
        msg_tag_send=peer_info['msg_tag'],
        msg_tag_recv=msg_tag,
        ...
         )
```

INTERNALS

Send/Recv

```
def tag_send(ucp_ep, buffer, nbytes, tag, pending_msg=None):
    cdef ucp_ep_h ep = <ucp_ep_h> PyLong_AsVoidPtr(ucp_ep)
    cdef void *data = PyLong_AsVoidPtr(get_buffer_data(buffer,
                                       check_writable=False))
    cdef ucs_status_ptr_t status = ucp_tag_send_nb(ep,
                                                   data,
                                                   nbytes,
                                                   ucp_dt_make_contig(1),
                                                   tag,
                                                   _send_callback)
    return create_future_from_comm_status(status, nbytes, pending_msg)
```

```
def tag_recv(ucp_worker, buffer, nbytes, tag, pending_msg=None):
    cdef ucp_woker_h worker = <ucp_worker_h> PyLong_AsVoidPtr(ucp_worker)
    cdef void *data = PyLong_AsVoidPtr(get_buffer_data(buffer,
                                       check_writable=True))
    cdef ucs_status_ptr_t status = ucp_tag_recv_nb(ep,
                                                   data,
                                                   nbytes,
                                                   ucp_dt_make_contig(1),
                                                   tag,
                                                   -1,
                                                   _tag_recv_callback)
    return create_future_from_comm_status(status, nbytes, pending_msg)
```

INTERNALS

Send/Recv

```
cdef void _send_callback(void *request, ucs_status_t status):
    cdef ucp_request *req = <ucp_request*> request
    if req.future == NULL:
        # This callback function was called before ucp_tag_send_nb()
returned
    cdef object future = <object> req.future
    if asyncio.get_event_loop().is_closed():
        pass
    elif status == UCS_ERR_CANCELED:
    elif status != UCS_OK:
    else:
        future.set_result(True)
```

```
cdef create_future_from_comm_status(ucs_status_ptr_t status,
                                    int64_t expected_receive,pending_msg):
    """Create Asyncio future of the message to be sent/received
```

INTERNALS

Blocking/Non-blocking

```
def _blocking_progress_mode(self, event_loop):
      """Bind an asyncio reader to a UCX epoll file descripter"""
      assert self.blocking_progress_mode is True

      def _fd_reader_callback():
          cdef ucs_status_t status
          self.progress()
          while True:
              status = ucp_worker_arm(self.worker)
              if status == UCS_ERR_BUSY:
                  self.progress()
              else:
                  break
          assert_ucs_status(status)
       event_loop.add_reader(self.epoll_fd, _fd_reader_callback)
```

```
def _non_blocking_progress_mode(self, event_loop):
    """Creates a task that keeps calling self.progress()"""

def progress(self):
    while ucp_worker_progress(self.worker) != 0:
          pass
```

BUFFERS

Community Interoperability

```
def get_buffer_data(buffer, check_writable=False):
    """
    Returns data pointer of the buffer. Raising ValueError if the buffer
    is read only and check_writable=True is set.
    """
    iface = None
    if hasattr(buffer, "__cuda_array_interface__"):
        iface = buffer.__cuda_array_interface__
    elif hasattr(buffer, "__array_interface__"):
        iface = buffer.__array_interface__

    if iface is not None:
        data_ptr, data_readonly = iface['data']
    else:
        mview = memoryview(buffer)
        data_ptr = _data_from_memoryview(mview)
          data_readonly = mview.readonly
```

__array__interface__:

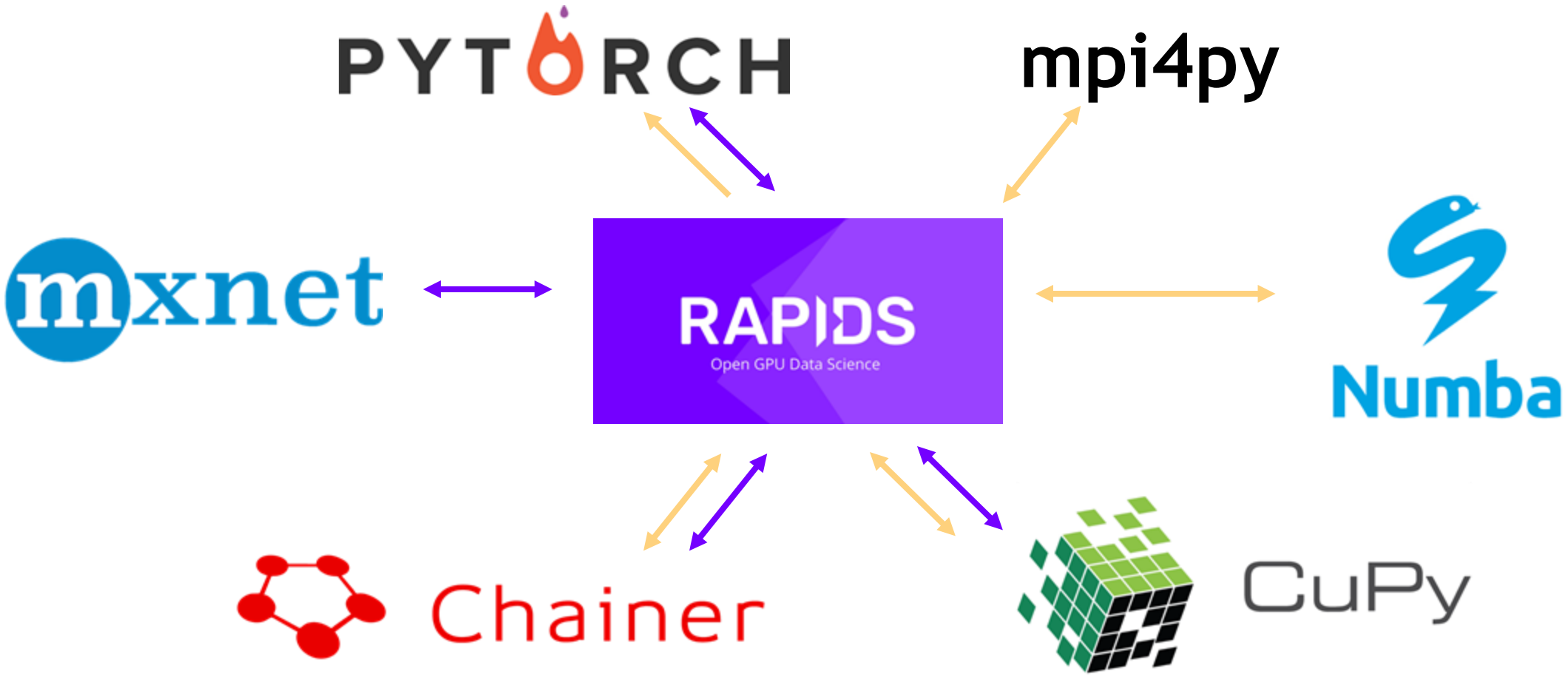
- shape
- typestr
- version
- strides
- data (pointer to *host* memory)

__cuda_array_interface__:

- shape
- typestr
- version
- strides
- data (pointer to *device* memory)

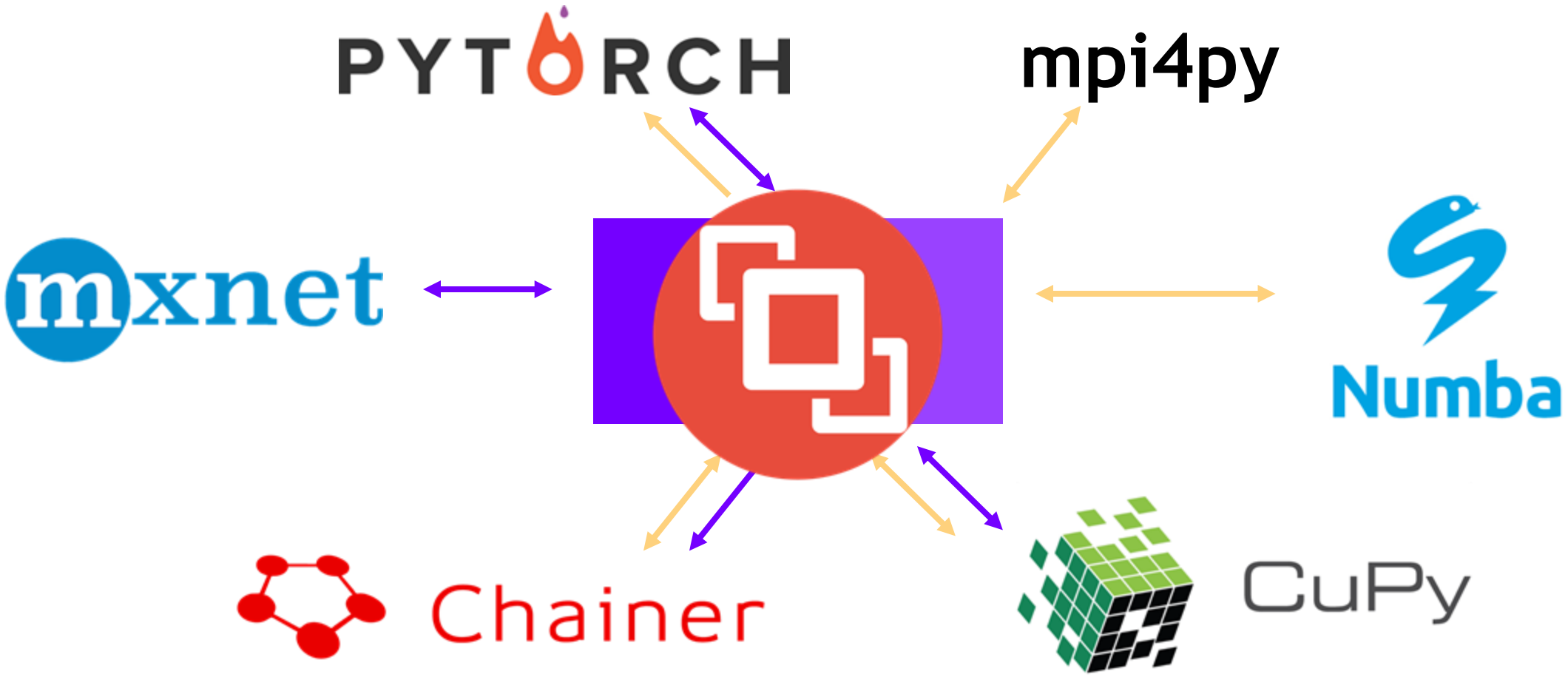
BUFFERS

Community Interoperability



BUFFERS

Community Interoperability



MISC

Configuration

- Configuration is possible via environment variables or UCX-Py's initializer

- Environment variables:

```
UCX_TLS=tcp,sockcm,cuda_copy UCX_SOCKADDR_TLS_PRIORITY=sockcm python app.py
```

- UCX-Py initializer:

```
import ucp
ucp.init(options={"TLS": "tcp,sockcm,cuda_copy",
                  "SOCKADDR_TLS_PRIORITY": "sockcm", ...},
         ...)
```

- Challenging to get right!
  - How do I use NVLINK? IB? NVLINK+IB ?

RAPIDS EXAMPLE

dask-cuda

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

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


# initialize client with the same settings as workers
initialize(
    create_cuda_context=True,
    enable_tcp_over_ucx=enable_tcp_over_ucx,
    enable_infiniband=enable_infiniband,
    enable_nvlink=enable_nvlink,
)

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

Drop-in replacement for TCP

```python
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()
```

MISC

Topological Distance

- Topological Distance allows us to find devices of specific type closest to a certain device (e.g., a CUDA-capable GPU)
- Requires hwloc
- Written in C with Python bindings

MISC

Topological Distance

```
>>> from ucp._libs.topological_distance import TopologicalDistance
>>> td = TopologicalDistance()
>>> td.get_cuda_distances_from_device_index(0, "network")
[{'distance': 2, 'name': 'ib0'}, {'distance': 4, 'name': 'enp1s0f0'},
 {'distance': 4, 'name': 'enp1s0f1'}, {'distance': 4, 'name': 'ib1'},
 {'distance': 4, 'name': 'ib2'}, {'distance': 4, 'name': 'ib3'}]
>>> td.get_cuda_distances_from_device_index(0, "openfabrics")
[{'distance': 2, 'name': 'mlx5_0'}, {'distance': 4, 'name': 'mlx5_1'},
 {'distance': 4, 'name': 'mlx5_2'}, {'distance': 4, 'name': 'mlx5_3'}]
```

BENCHMARKS

GPU-GPU w/ CuPy

TCP

```
$ python local_cuda.py -p tcp -d 1,2 --size 40000
Roundtrip benchmark
--------------------------
Size        | 40000*40000
Chunk-size  | 128 MiB
Ignore-size | 1.05 MB
Protocol    | tcp
Device(s)   | 1,2
npartitions | 100
==========================
Total time  | 29.72 s
==========================
(w1,w2)     | 25% 50% 75% (total nbytes)
--------------------------
(01,02)     | 107.76 MB/s 108.44 MB/s 108.94 MB/s (3.20 GB)
(02,01)     | 107.64 MB/s 108.16 MB/s 108.96 MB/s (3.20 GB)
```

~215 MB/s

CUDA IPC

```
$  python local_cuda.py -p ucx -d 1,2 --size 40000
Roundtrip benchmark
--------------------------
Size        | 40000*40000
Chunk-size  | 128 MiB
Ignore-size | 1.05 MB
Protocol    | ucx
Device(s)   | 1,2
npartitions | 100
==========================
Total time  | 1.01 s
==========================
(w1,w2)     | 25% 50% 75% (total nbytes)
--------------------------
(01,02)     | 12.87 GB/s 13.08 GB/s 13.58 GB/s (3.20 GB)
(02,01)     | 12.26 GB/s 12.80 GB/s 13.44 GB/s (3.20 GB)
```

~25 GB/s

FUTURE WORK

- UCX Documentation
- cuDF Improvements
  - Unnecessary Copies
  - Managed Allocations
- Dask Improvements
  - Unnecessary Copies
  - Possible Graph improvements
- UCX Improvements
  - TCP
  - IPC Caching Bug/Feature

DISTRIBUTED JOINS USING CUDF + UCX

RELATIONAL JOIN

Customer ID	Customer Name
202	Jensen
201	Cyril
203	Hao

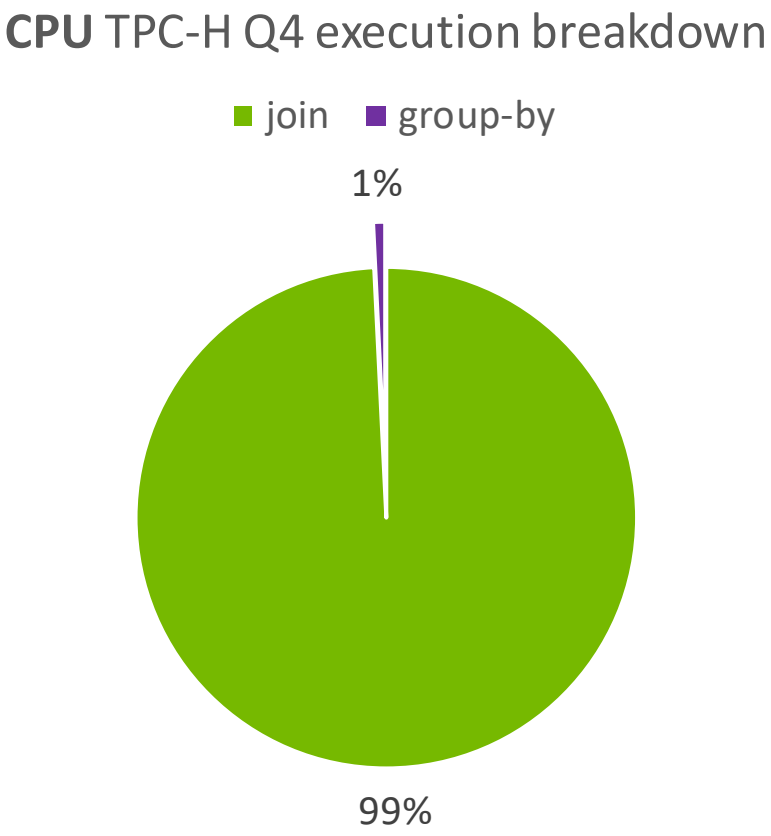
$\bowtie$

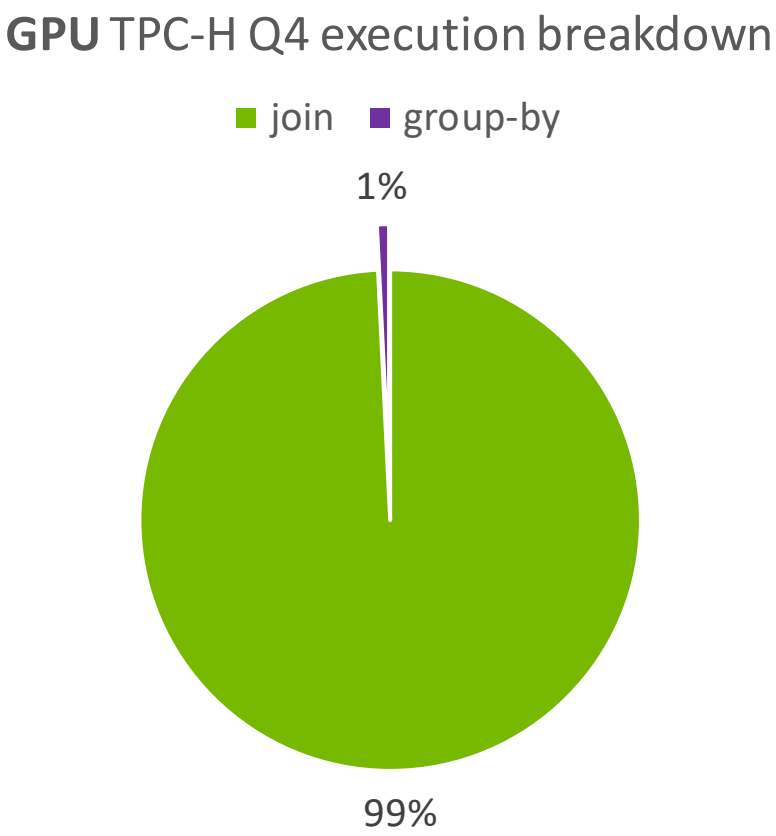
Customer ID	Item
201	Banana
203	Yogurt
202	Yogurt
203	Bread

$=$

Customer ID	Customer Name	Item
202	Jensen	Yogurt
201	Cyril	Banana
203	Hao	Yogurt
203	Hao	Bread

JOINS & E2E PERFORMANCE





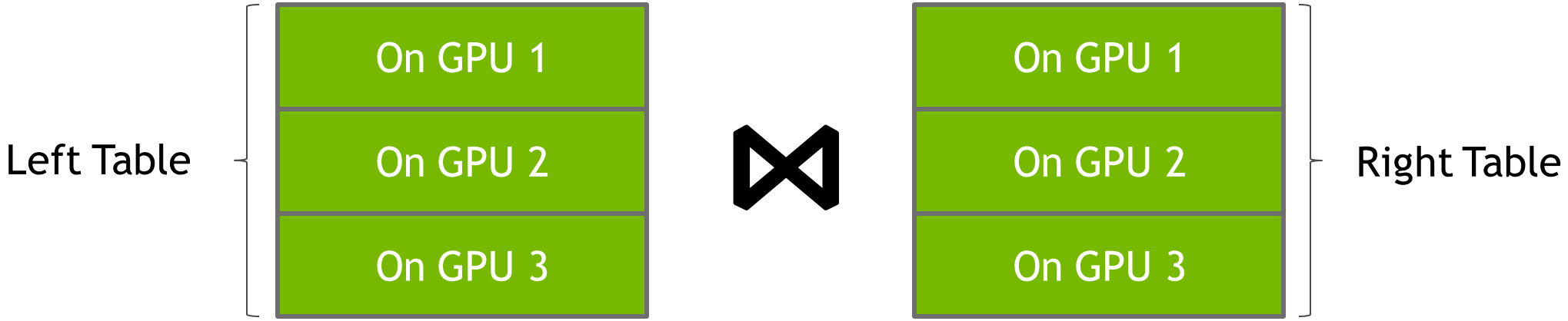
18/22 TPC-H Queries involve **Joins** and are the longest running ones[1]

[1] c.f. recently published TPC-H results at http://www.tpc.org/tpch/results/tpch_last_ten_results.asp

REPARTITIONED JOIN

Assume the table is already distributed

Suppose we have 3 GPUs



REPARTITIONED JOIN

Repartition rows by hashing

One table on a single GPU

REPARTITIONED JOIN

Local join on GPU i

REPARTITIONED JOIN

Overview

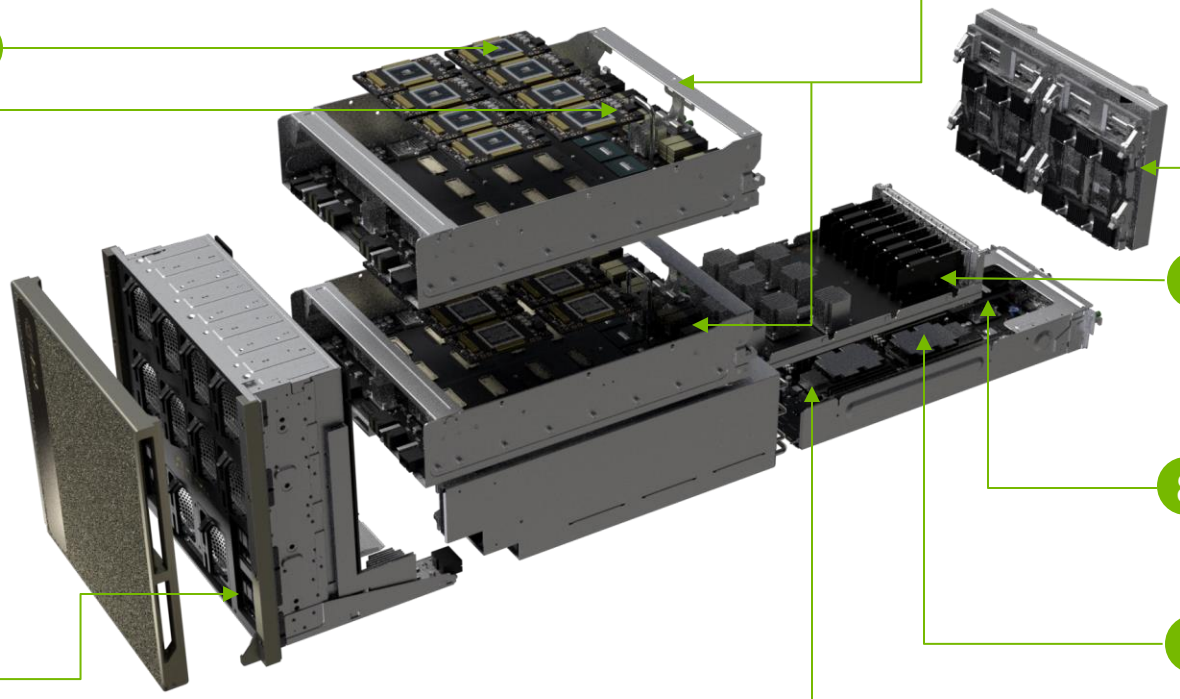
- Assume the table is already distributed.
- Step 1: Each GPU **hash** the table into N partitions. (N = number of GPUs).
- Step 2: **All-to-all communication**. Send partition i to GPU i. Receive N partitions.
- Step 3: Merge received partitions and perform **local join**.

ALTERNATIVE STRATEGIES

- Broadcast join (replicated join): when one of the table is very small, it makes sense to broadcast the whole table.
- Partitioned join: send hash tables instead of input tables, presented in GTC 2019 [1].

[1]: https://on-demand-gtc.gputechconf.com/gtcnew/sessionview.php?sessionName=s9557-effective%2c+scalable+multi-gpu+joins

NVIDIA DGX-2 OVERVIEW



1 **NVIDIA Tesla V100 32GB**

2 **Two GPU Boards**
8 V100 32GB GPUs per board
6 NVSwitches per board
512GB Total HBM2 Memory
interconnected by
Plane Card

3 **Twelve NVSwitches**
2.4 TB/sec bi-section
bandwidth

4 **Eight EDR Infiniband/100 GigE**
1600 Gb/sec Total
Bi-directional Bandwidth

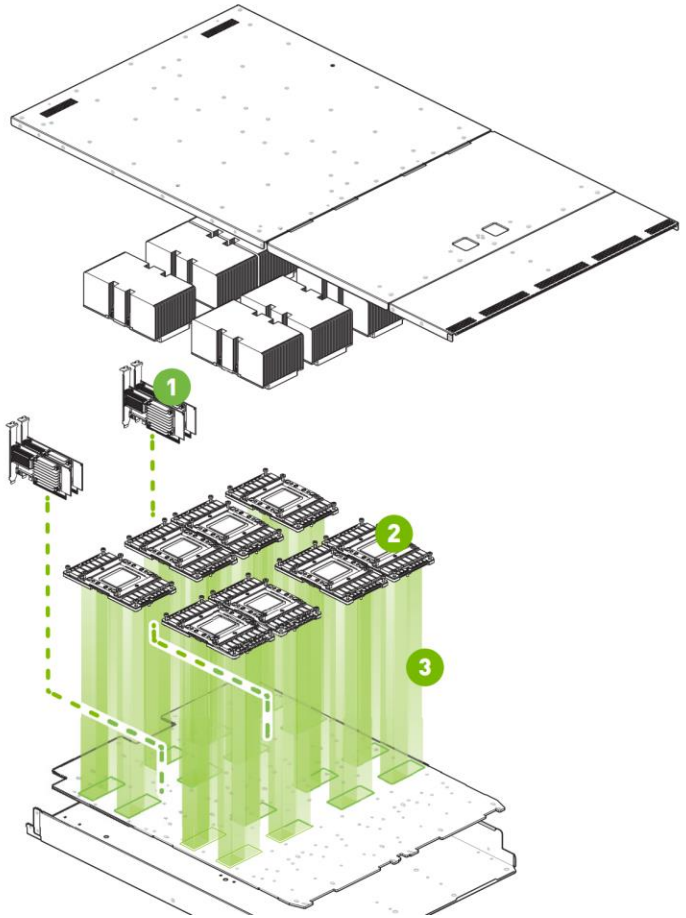
8 **Two High-Speed Ethernet**
10/25/40/100 GigE

5 **Two Intel Xeon Platinum CPUs**

6 **1.5 TB System Memory**

7 **30 TB NVME SSDs**
Internal Storage

MULTI-NODE DGX-1 ON PROM CLUSTER



1. NETWORK INTERCONNECT
4X InfiniBand 100 Gbps EDR
2X 10 GbE

2. GPUs
8X NVIDIA Tesla® V100 16 GB/GPU
40,960 Total NVIDIA CUDA® Cores
5,120 Tensor Cores

3. GPU INTERCONNECT
NVIDIA NVLink™
Hybrid Cube Mesh

SOFTWARE STACK

- UCX for communication
  - Used by Dask and many MPI implementations
  - Lower-level than MPI -> more control

- RAPIDS cuDF for local operations
  - Convenient
  - Integrate into Dask, Spark
  - Uses its own memory manager (RMM)

ALL-TO-ALL COMMUNICATION

REPARTITIONED JOIN

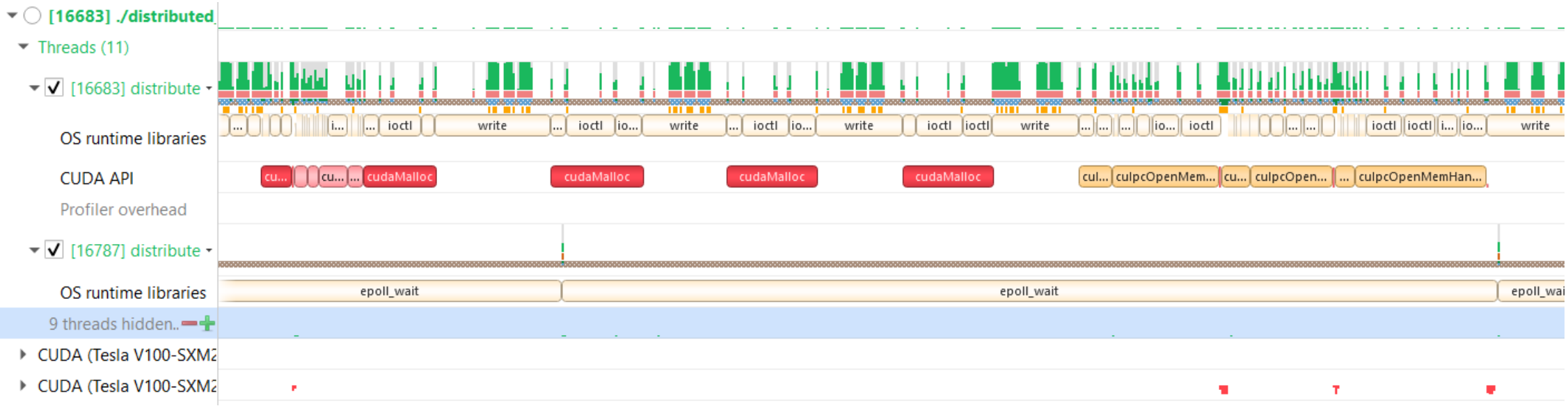
Overview

- Assume the table is already distributed.
- Step 1: Each GPU **hash** the table into N partitions. (N = number of GPUs).
- **Step 2: All-to-all communication. Send partition i to GPU i. Receive N partitions.**
- Step 3: Merge received partitions and perform **local join**.

Similar pattern also used in multi-GPU sort and multi-GPU hashing.

OPTIMIZE ALL-TO-ALL COMM

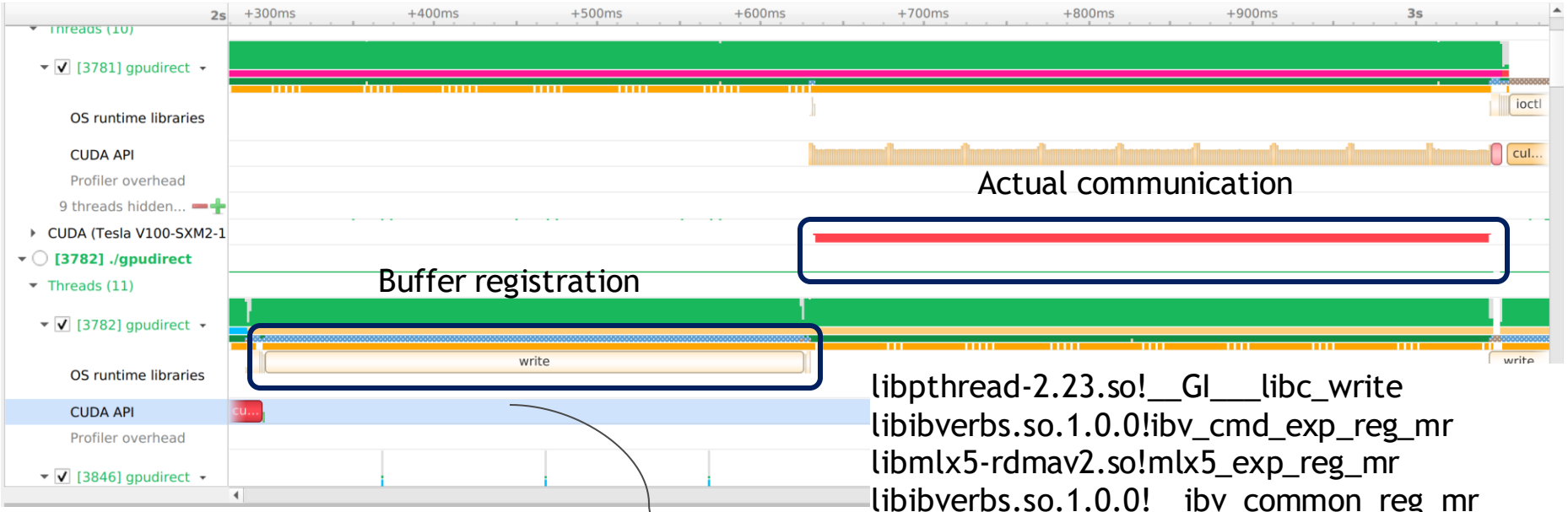
A small section of the all-to-all communication on DGX-1



- Eliminate cudaMalloc overhead: use RMM memory pool.
- Eliminate cuIpcOpenMemHandle overhead: warmup run (using smaller buffers).

BUFFER REGISTRATION

Send from 1 GPU to another on DGX-2



callstack

libpthread-2.23.so!__GI___libc_write
libibverbs.so.1.0.0!ibv_cmd_exp_reg_mr
libmlx5-rdmav2.so!mlx5_exp_reg_mr
libibverbs.so.1.0.0!__ibv_common_reg_mr
libuct_ib.so.0.0.0!uct_ib_mem_reg
libucp.so.0.0.0!ucp_mem_rereg_mds
libucp.so.0.0.0!ucp_request_memory_reg
libucp.so.0.0.0!ucp_rndv_matched
libucp.so.0.0.0!ucp_rndv_process_rts
libuct_ib.so.0.0.0!uct_rc_mlx5_iface_progress
libucp.so.0.0.0!ucp_worker_progress

BUFFER REGISTRATION

When Infiniband is not used…

- E.g. on a single DGX-2 (16 GPUs connected with NVLink).
- Explicitly do not specify rc in UCX_TLS.
- Note: If rc is specified but not used, it will still register buffer and cause overhead.
- Could use UCX_TLS=sm,cuda_copy,cuda_ipc

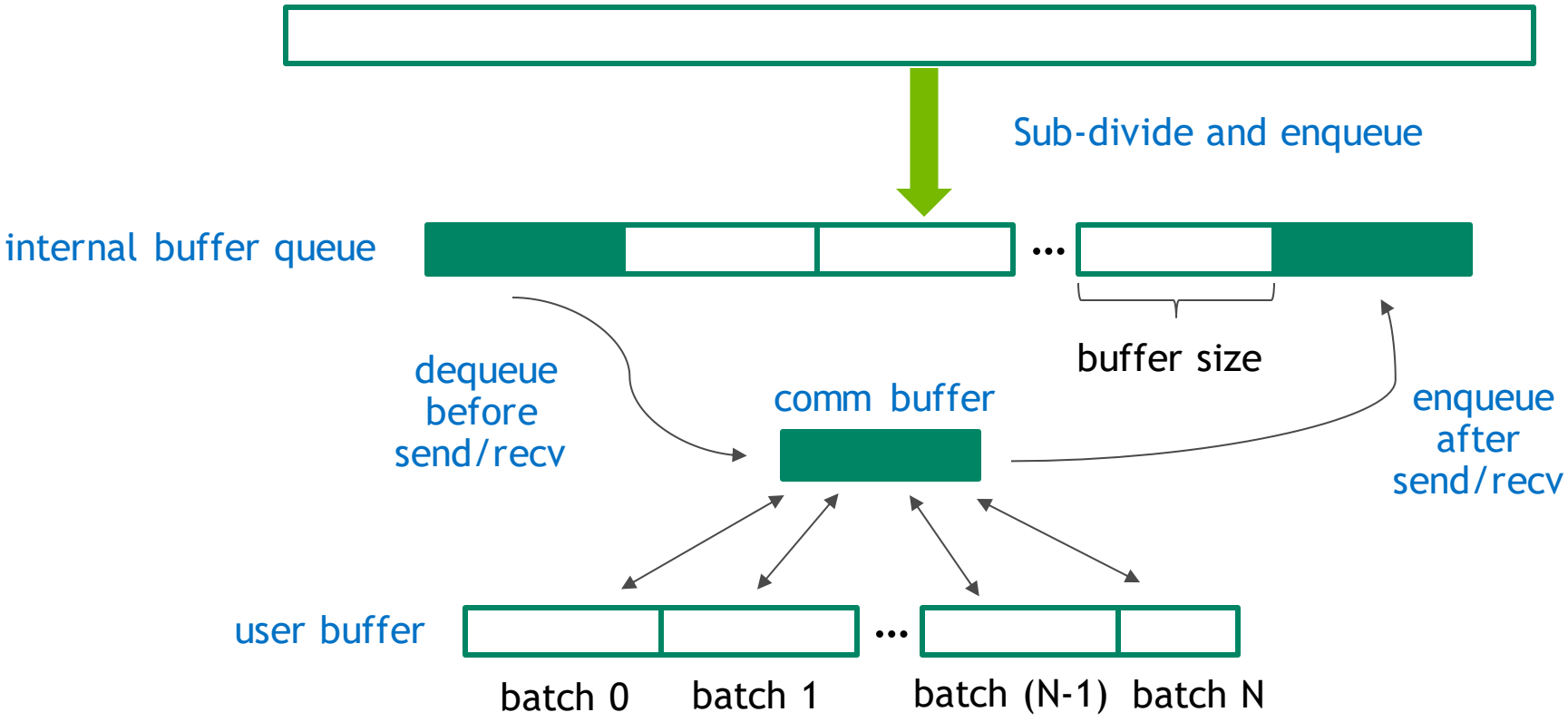
BUFFER REGISTRATION

When Infiniband is actually used…

- E.g. on Prometheus Cluster (multiple DGX-1s connected with IB)
- For GPUDirect with IB, uses UCX_TLS=rc,cuda_copy,cuda_ipc
- Repeatedly use the same communication buffer for send/recv
- Implemented using UCX callbacks
- Must be fully async, critical for scalability
- 8 DGX-1s connected with IB on Prom, buffer communicator improves all-to-all throughput from 18.90GB/s to 281.39GB/s. Theoretical peak is 387.84GB/s, so we achieve >70% network SOL.

BUFFER COMMUNICATOR

Initialize: Allocate and register buffer of size #buffers x buffer size



BUFFER COMMUNICATOR

Send side

```
comm_handle_t UCXBufferCommunicator::send(const void *buf, int64_t count, int element_size, int dest, int tag)
{
    // Step 1: Construct communication tag from user tag (details later)
    ...

    // Step 2: Send the buffer size with callback (useful for allocating receive buffer)
    comm_handle_t request = ucp_tag_send_nb(
        ucp_endpoints[dest], &count, 1, ucp_dt_make_contig(sizeof(int64_t)), comm_tag, send_handler
    );

    // Step 3: Get the communication buffer from queue
    void *comm_buffer = buffer_cache.front();
    buffer_cache.pop();

    // Step 4: Fill in information in the request handle so that the callback can launch subsequent batches
    SendInfo *info = (SendInfo *)request;
    info->send_buffer = buf;
    info->comm_buffer = comm_buffer;
    // other infomation like count, element size, destination, tag and batch number
    ...

    return request;
}
```

BUFFER COMMUNICATOR

Send handler

```
static void send_handler(void *request, ucs_status_t status)
{
    while (nelements_remaining > 0) {
        // Copy current batch into communication buffer
        CUDA_RT_CALL(cudaMemcpyAsync(
            comm_buffer, start_addr_this_batch, nelements_current_batch * element_size,
            cudaMemcpyDeviceToDevice, copy_stream
        ));

        CUDA_RT_CALL(cudaStreamSynchronize(copy_stream));

        // Send current batch to remote node
        request = ucp_tag_send_nb(
            ucp_endpoints[dest], comm_buffer, nelements_current_batch, ucp_dt_make_contig(element_size),
            comm_tag, send_handler
        );

        if (UCS_PTR_STATUS(request) != UCS_OK) {
            // Send is not complete now. The rest of send is handled by continuation.
            break;
        }

        // more book-keeping...
    }
}
```

BUFFER COMMUNICATOR

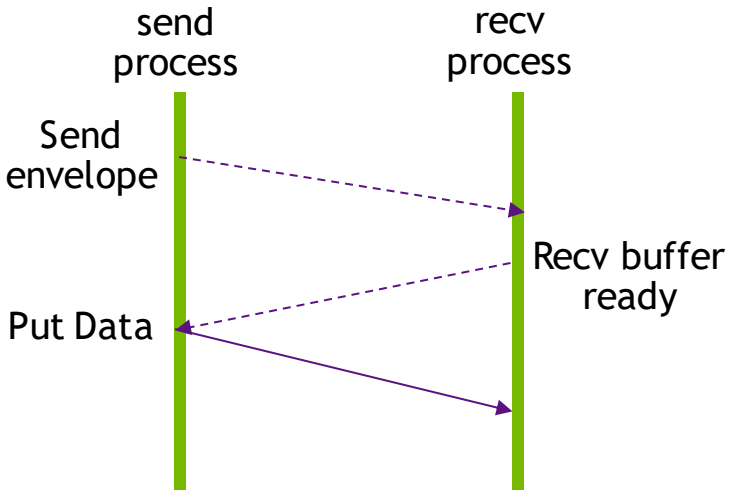
Tag design

- Unlike MPI, UCX receive does not have a source argument.
- Also unlike MPI, UCX does not guarantee order of messages to the same node with same tag.
- Could in principal uses ‘ucp_worker_fence’, but likely have performance consequences.
- Our design takes advantages of tag matching to distinguish sources/batches.

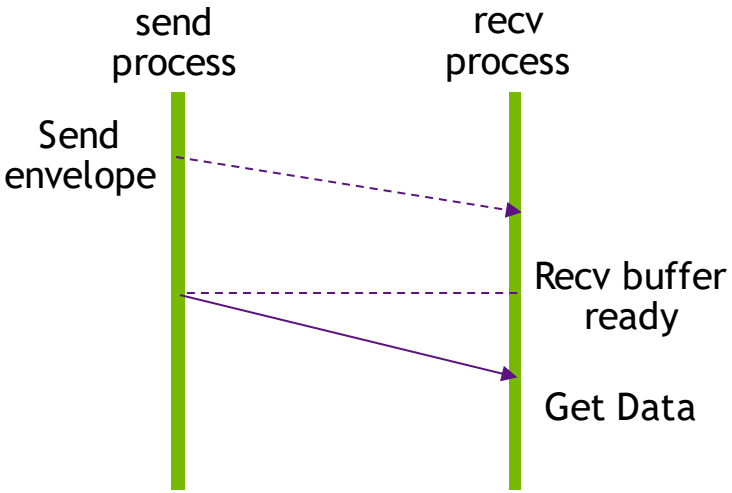
0x 00000020 001A 0002

00000020: user tag
001A: this message is sent from rank 26
0002: the second batch

OPTIMIZE ALL-TO-ALL COMM



UCX_RNDV_SCHEME=put_zcopy



UCX_RNDV_SCHEME=get_zcopy

- GPU-NIC affinity is crucial. Use nvidia-smi and “UCX_NET_DEVICES”.
- UCX_RNDV_SCHEME=put_zcopy may block communication.
- Sending a buffer to itself with ud and wakeup feature is slow.

PERF SUMMARY - ALL TO ALL COMM

	DGX-2	Prom 8 nodes
Theoretical Peak	2400.00GB/s	387.84GB/s
No optimization	33.04GB/s	12.88GB/s
Memory pool	38.96GB/s	18.90GB/s
Memory pool + warmup	2130.48GB/s	18.42GB/s
Memory pool + warmup + buffer communicator	742.01GB/s	281.39GB/s

- On Prom, buffer communicator is crucial for achieving good bandwidth.
- On single DGX-2, non-buffer communication should be used as buffer communicator has significant overhead.

[1]: DGX-2 test uses UCX_TLS=sm,cuda_copy,cuda_ipc
[2]: Prom 8 node test uses UCX_TLS=rc,cuda_copy,cuda_ipc

COMPUTATION & COMMUNICATION OVERLAP

REPARTITIONED JOIN

Overview

- Assume the table is already distributed.
- Step 1: Each GPU hash the table into N buckets. (N = number of GPUs).
- **Step 2: All-to-all communication. Send bucket i to GPU i. Receive N buckets.**
- **Step 3: Merge received buckets and perform local join.**

COMPUTATION AND COMM OVERLAP

Overview

- Over-decomposition: hash into (MxN) partitions
- M – the number of batches, N – the number of GPUs

COMPUTATION AND COMM OVERLAP

- Try 1: Make all-to-all communication async
  - UCX needs CPU to explicitly progress communication through ‘ucx_worker_progress’ (where callbacks are called)
  - Cannot progress communication as CPU is waiting on join
- Try 2: Make join async as well
  - Not easy since the CPU needs to wait for estimated size for allocating result buffer
- Our strategy: use one CPU thread for communication and another for local join

COMPUTATION AND COMM OVERLAP

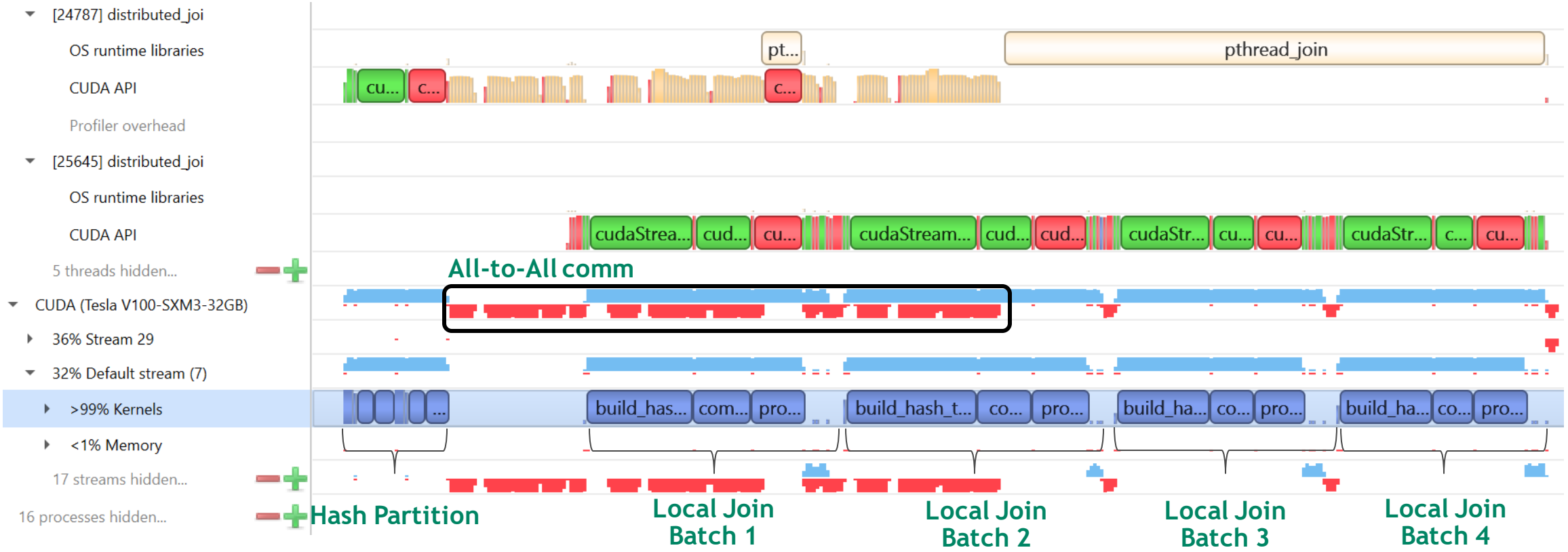
Using multiple threads

COMPUTATION AND COMM OVERLAP

- Mutex locks prevent communication and computation threads to run in parallel
  - Eliminate all cudaMallocHost, cudaFreeHost, cudaMallocManaged, cudaFree
- Both buffer loading in communication and kernel in computation uses SM (For buffered communicator)
  - Use priority stream for device-to-device memory copy

COMPUTATION AND COMM OVERLAP

Profile on DGX-2



COMPUTATION AND COMM OVERLAP

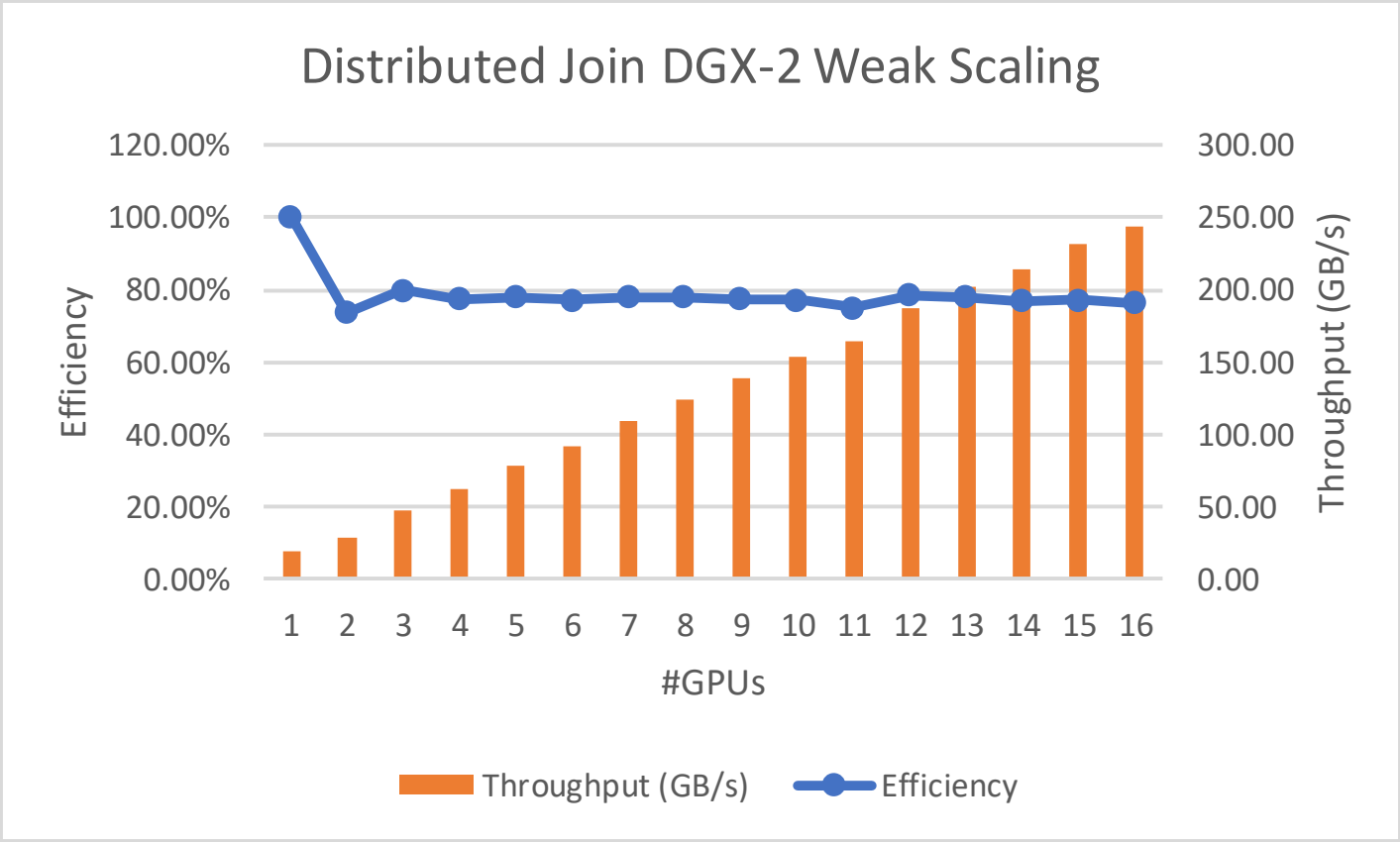
Performance Impact

- Single-node DGX-2:
  - Running time decreases from 491ms with no overlap, to 419ms with 4 batches. (~15% improvement).
- Multi-node DGX-1Vs:
  - Small improvement when the number of GPUs is small.
    - Running time decreases from 715ms with 1 batch to 686ms with 4 batches for 4 nodes on Prom. (~4% improvement).
  - No difference or even worse performance when the number of GPUs is large.
    - Running time increases from 772ms with 1batch to 792ms with 4 batches for 8 nodes on Prom.
    - Speculation: more partitions => less data per partition => more overhead for communication

DISTRIBUTED JOINS: EVALUATION

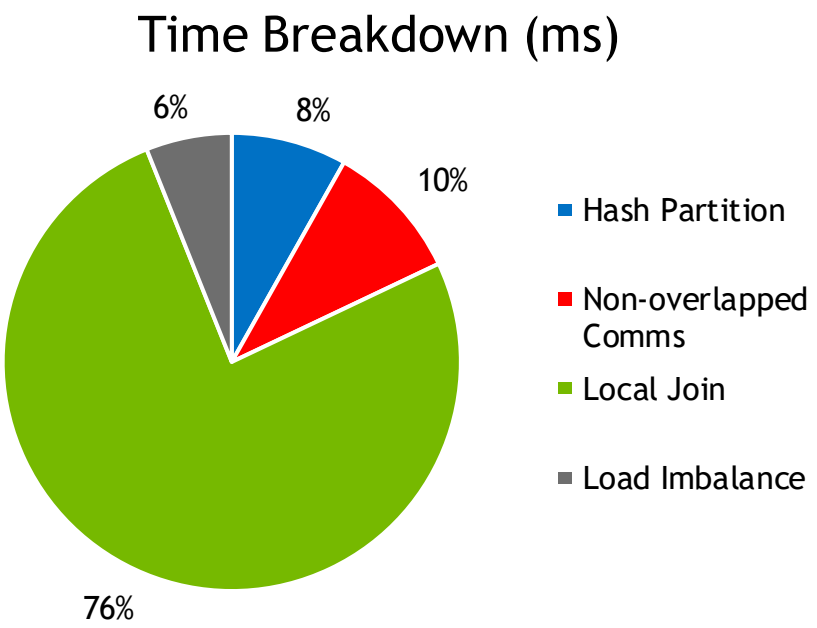
WEAK SCALING ON DGX-2

- 200M/table/GPU
- 64 bits keys/payloads
- Selectivity = 0.3
- Build keys unique



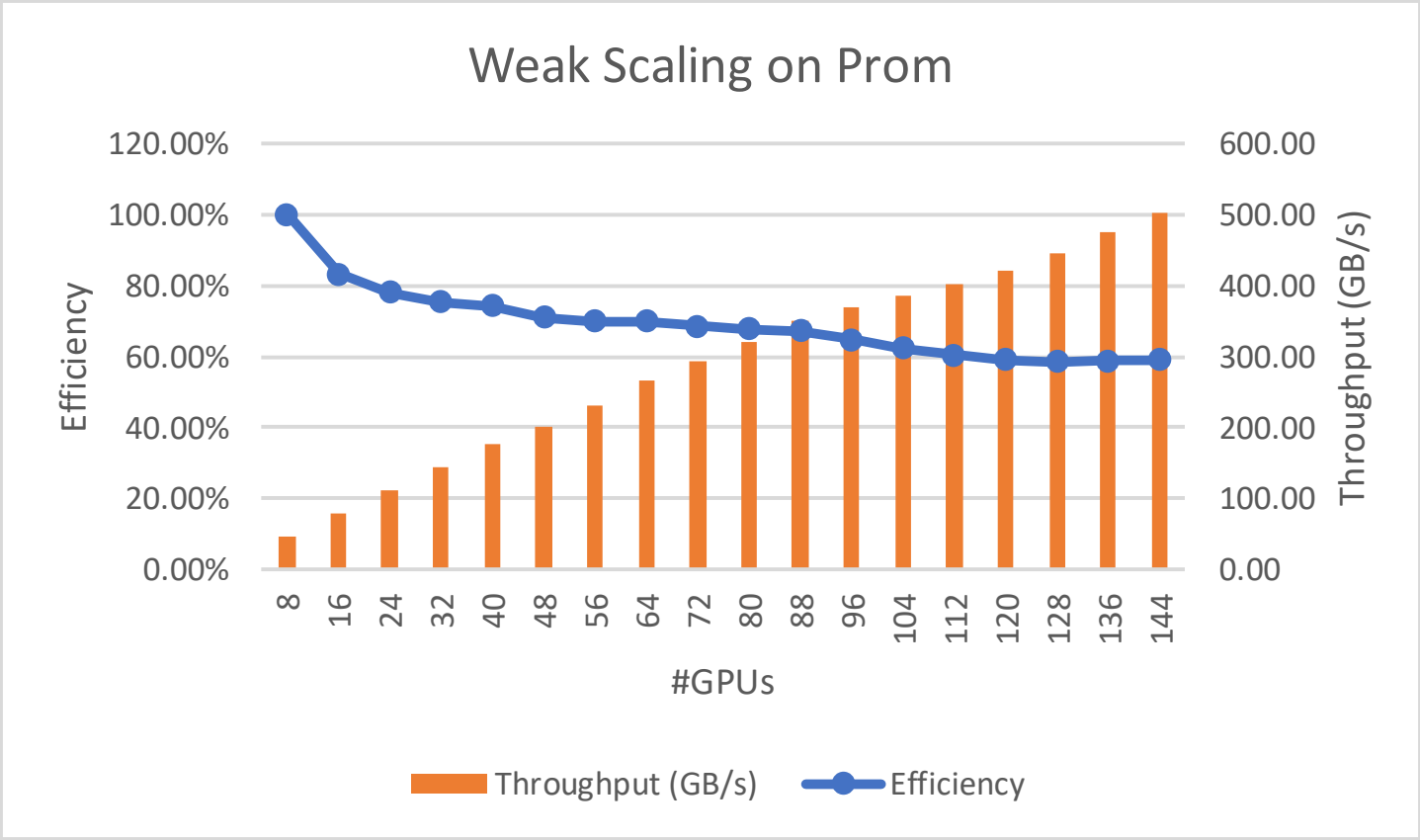
DGX-2 PERFORMANCE ANALYSIS

- With 16 GPUs on DGX-2, the throughput is 243.81GB/s, which is comparable with 100 CPU nodes (800 CPU cores), in the best-known CPU implementation.
- Our result is also comparable (within 10%) with the state-of-the-art GPU partitioned join in GTC 2019, but our implementation has the following advantages:
  - Use cuDF => more modular and easier to maintain
  - Materialize the final output
  - Extend to distributed-memory systems



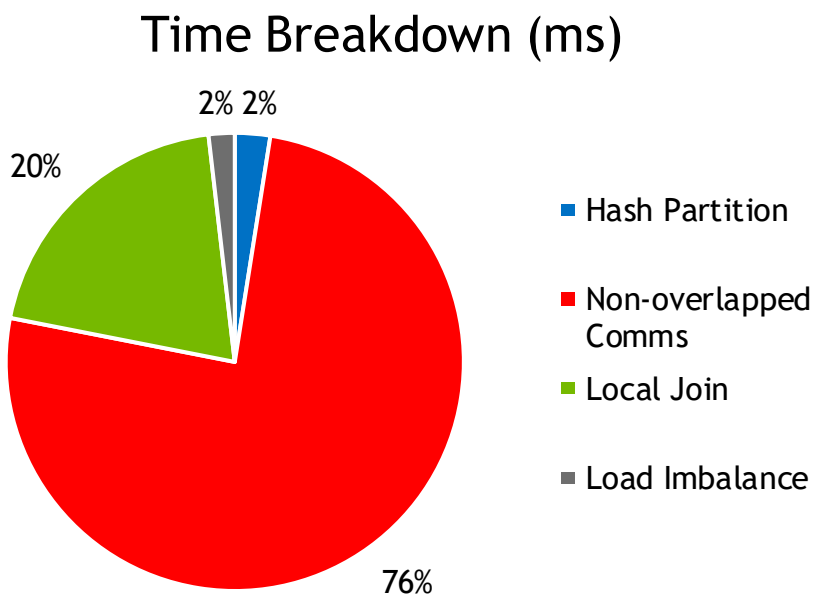
WEAK SCALING ON PROM

- 100M/table/GPU
- 64 bits keys/payloads
- Selectivity = 0.3
- Build keys unique



MULTI-NODE DGX-1 PERFORMANCE ANALYSIS

- With 18 nodes on Prom (144 GPUs), the achieved throughput is 503.61GB/s, which is comparable with 244 CPU nodes (1952 cores) in the best-known CPU implementation.
- Best-known CPU result achieves 780GB/s using 512 CPU nodes (4096 cores).



TAKEAWAYS

Buffer registration is costly, we implemented our own pipelining scheme

- Can we use UCX pipelining for this? Is it ready?
- Can we pass custom CUDA steams to UCX (to use priorities)?

Multi-node DGX-1 performance is **limited by network BW**

- We'll try compression next (compressing on GPU/CPU)

Perf issues: Dask/UCX << C++/UCX, also C++/UCX overlap is not ideal

- Continue to analyze & improve UCX-PY
- **Need better profiling tools** for in-depth performance analysis

NVIDIA®