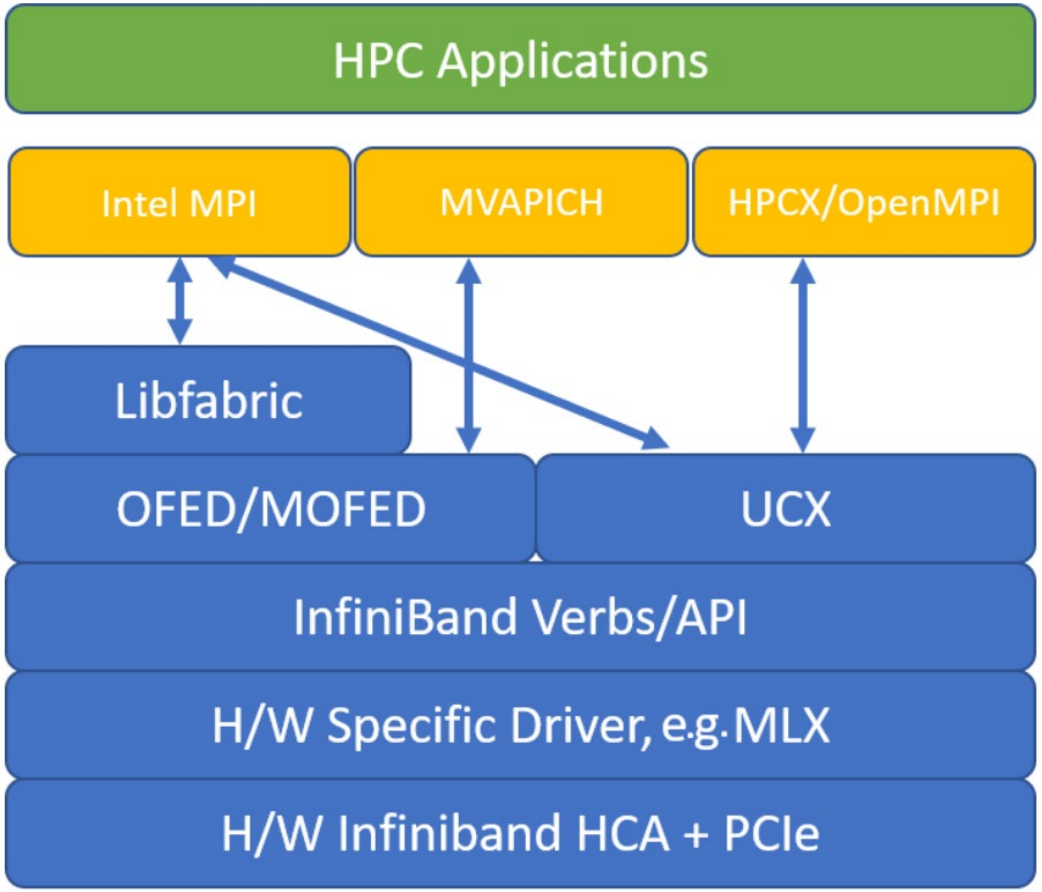
High-Performance Communication: RDMA, UCX & HPC-X

National Tsing Hua University

2025, Fall Semester

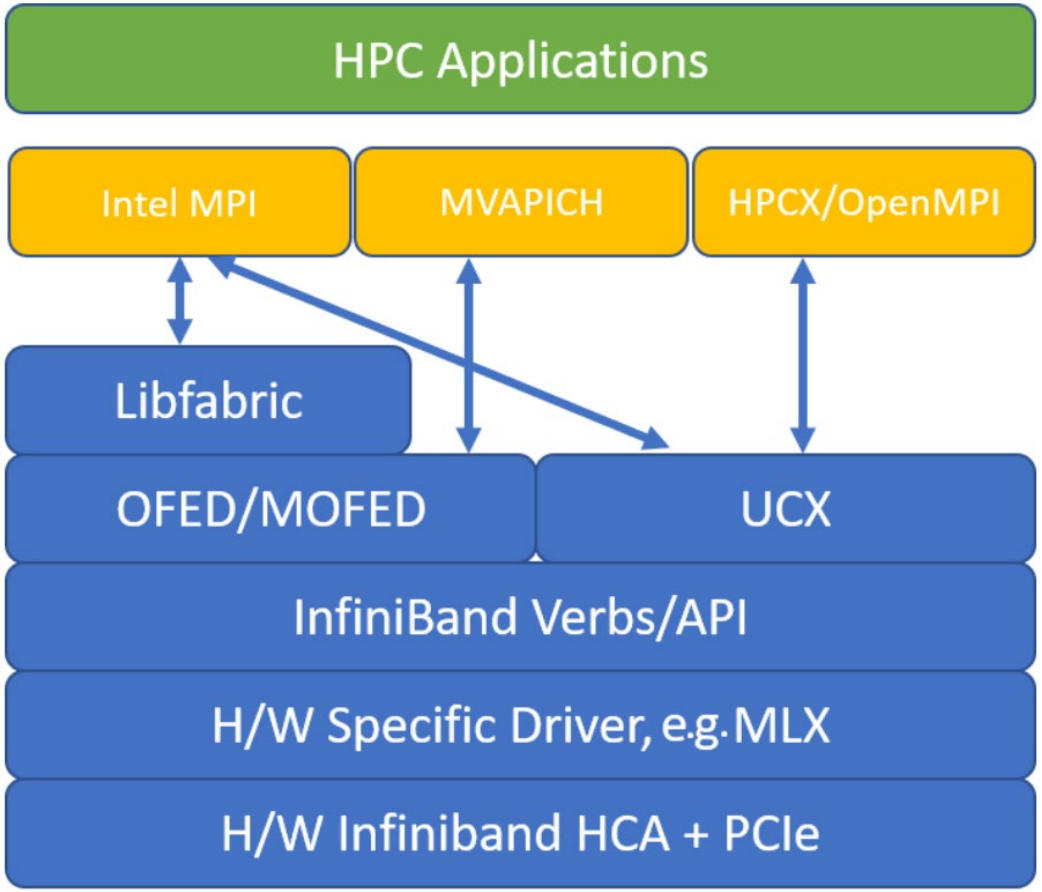
Outline

- RDMA Technology
- Verbs
- UCX
- HPC-X



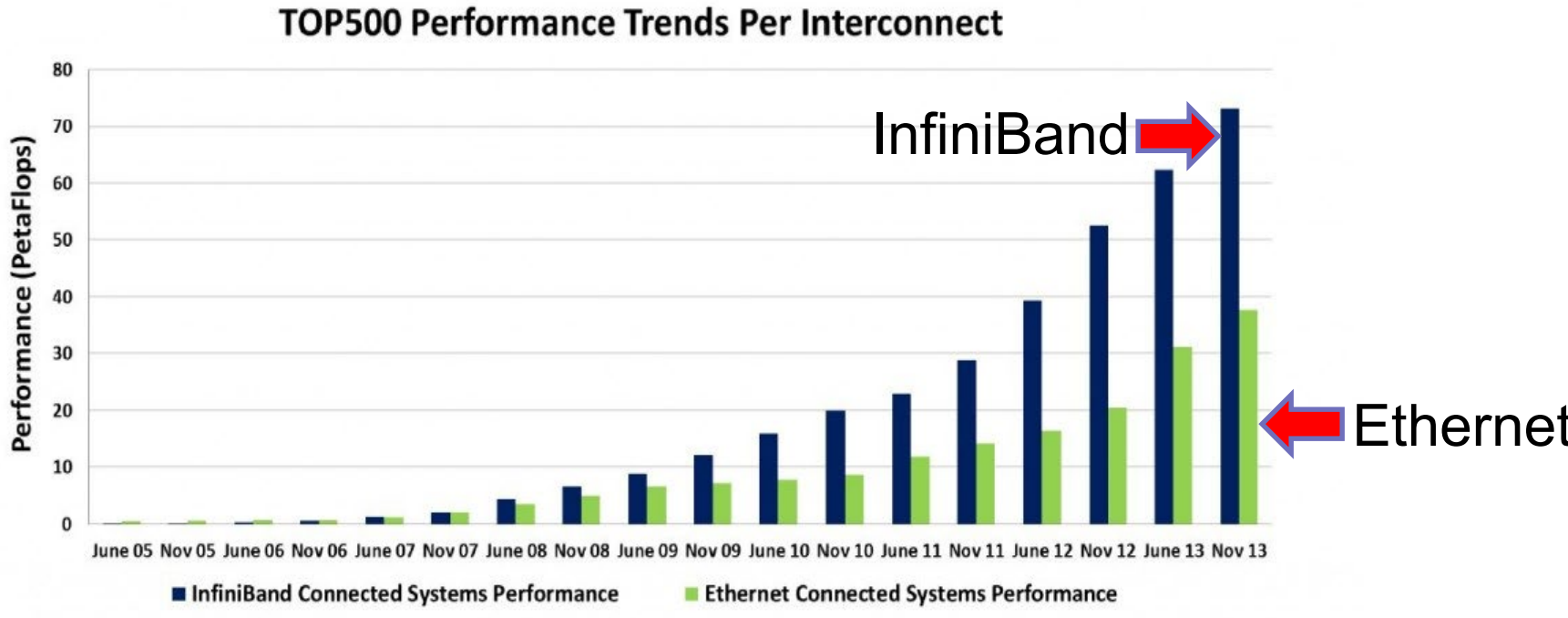
Outline

- RDMA Technology
- Verbs
- UCX
- HPC-X

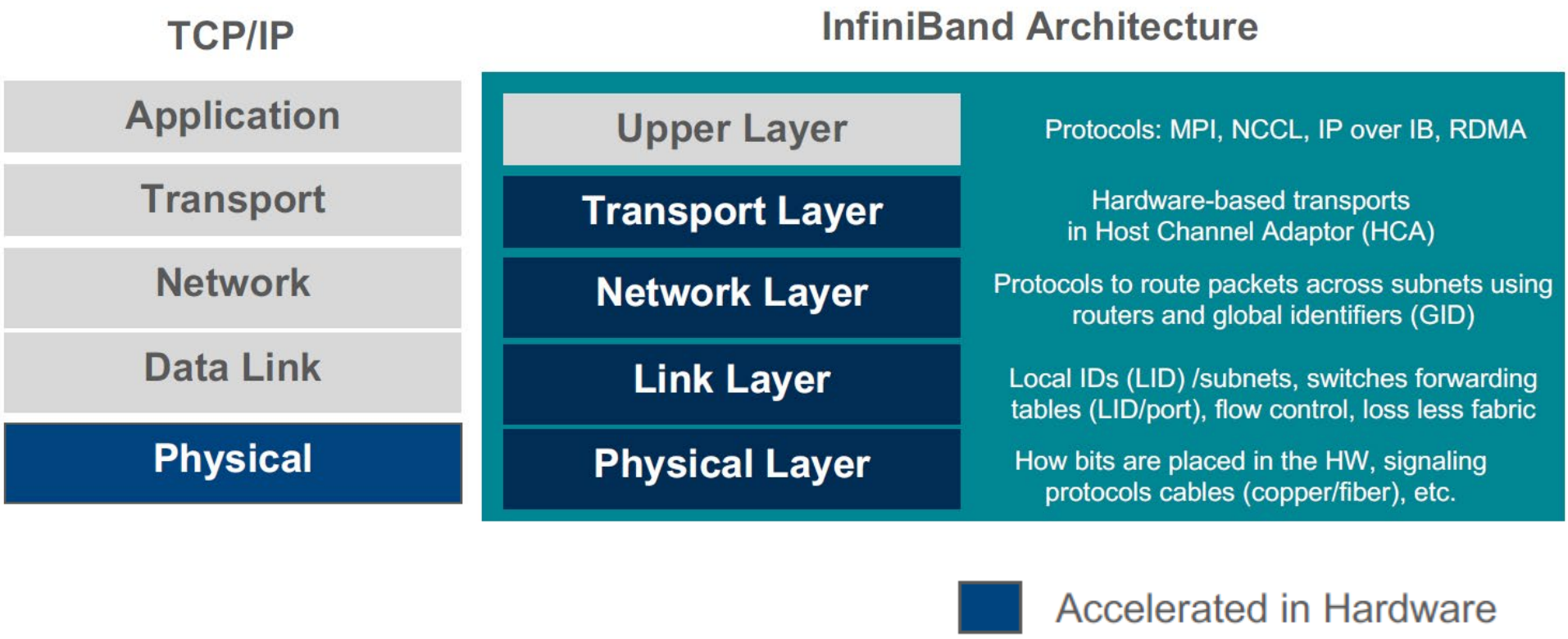


Network Device: InfiniBand

- A computer network communications link used in high-performance computing featuring very high throughput
- It is the most commonly used interconnect in supercomputers
- Manufactured by **Mellanox**



TCP/IP vs InfiniBand Architecture

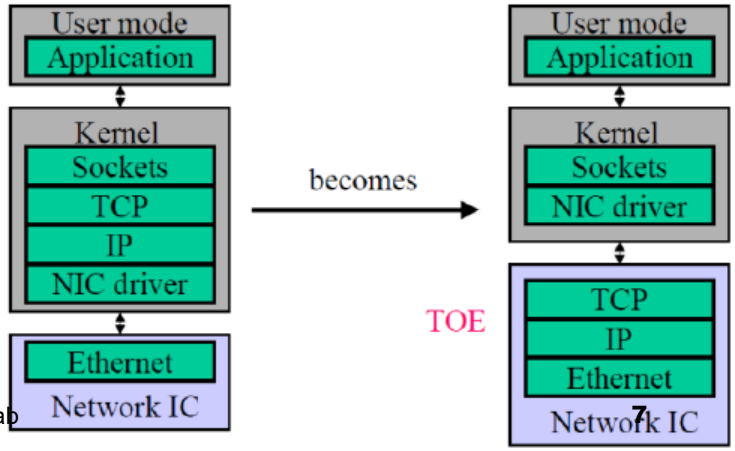


Key Features

- Transport offload
- Bypassing the OS
- Communication Model
  - Two-sided communication model
    - Send and receive model
  - One-sided communication model
    - Remote memory access and atomics
- Rendezvous
  - Two sides exchange meta-data and use one-sided operations for bulk transfer

Acceleration by Offloading

- **What is offloading? Asking somebody else to do the work!**
  - TCP offloading: Moving IP and TCP processing to the Network Interface (NIC)
  - Checksum offloading: Moving the checksum calculation to the NIC (special circuits)
- **Main justification for communication offloading**
  - Reduction of host CPU cycles for protocol header processing, checksumming
  - Fewer CPU interrupts
  - Fewer bytes copied over the memory bus
  - Potential to offload expensive features such as encryption
- **New performance metric: CPU Utilization**
  - The rate (or % of time) the CPU is used for actual work
  - Time spent on communication is time wasted…

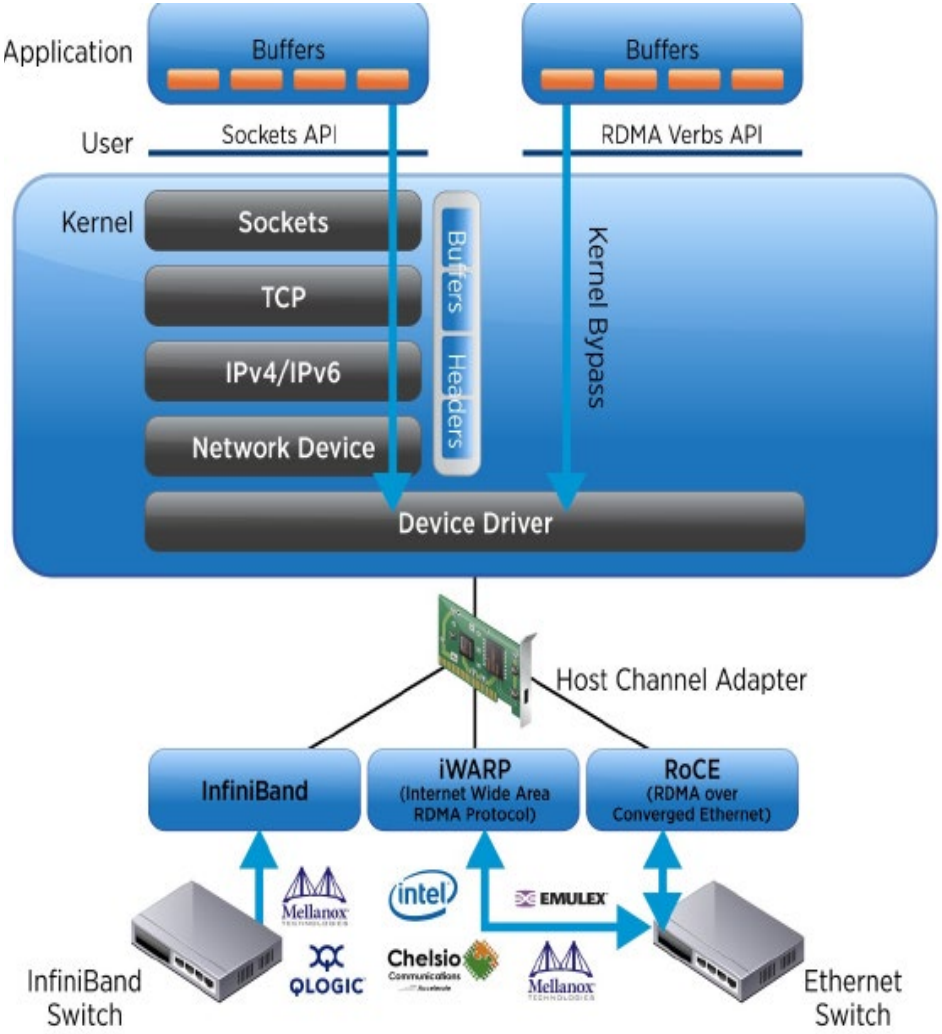


User-level Networking Concept

- **Transport offloading is not enough!**
  - Calling kernel driver still have system call overhead, context switch, memory copying
- **• U-Net:**
  - Eicken, Basu, Buch, Vogels, Cornell University, 1995
  - Virtual network interface that allows applications to send and receive messages without operating system intervention
  - Move all buffer management and packet processing to user-space ("zero-copy")
- **Efforts of U-Net eventually resulted in the *Virtual Interface Architecture* (VIA)**
  - Specification jointly proposed by Compaq, Intel and Microsoft, 1997
- **VIA architecture has led to the implementation of various high-performance networking stacks: Infiniband, iWARP, RoCE**
  - Commonly referred to as **RDMA network stacks**
  - RDMA stands for "Remote Direct Memory Access"

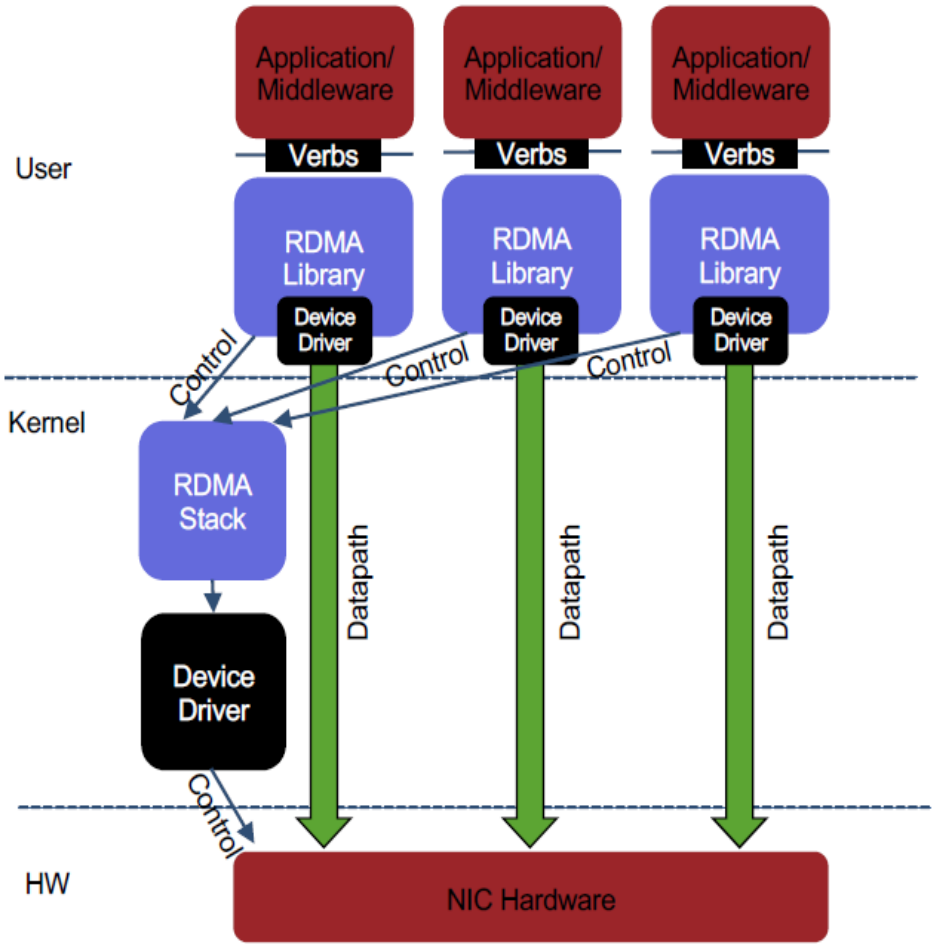
Bypassing the OS

- **Basic working principles:**
  - RDMA traffic sent directly to NIC without interrupting CPU
  - A remote memory region registers with the NIC first
  - NIC records virtual to physical page mappings.
  - When NIC receives RDMA request, it performs a **Direct Memory Access** into memory and returns the data to client.
  - Kernel bypass on both sides of traffic



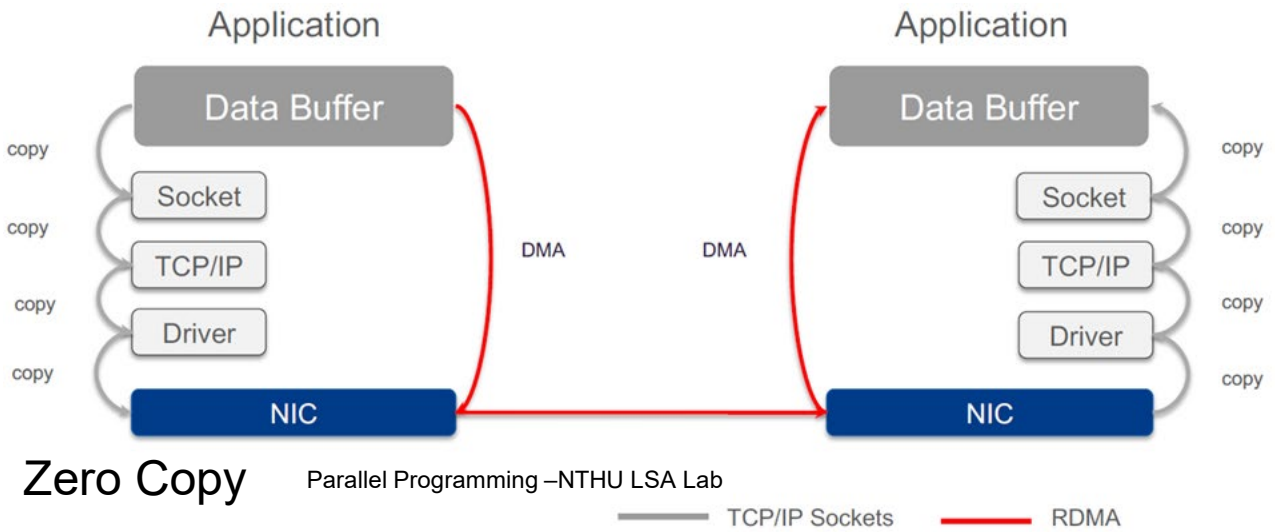
Enabling Kernel Bypass

- **Separation of Control and Data paths**
- **Control path**
  - Resource setup
  - Memory management
  - Connection establishment
- **Data path (only after control path)**
  - Post Send, Post Receive
  - **Poll** for Completion, Request event
  - Connection establishment



RDMA Communication Modes

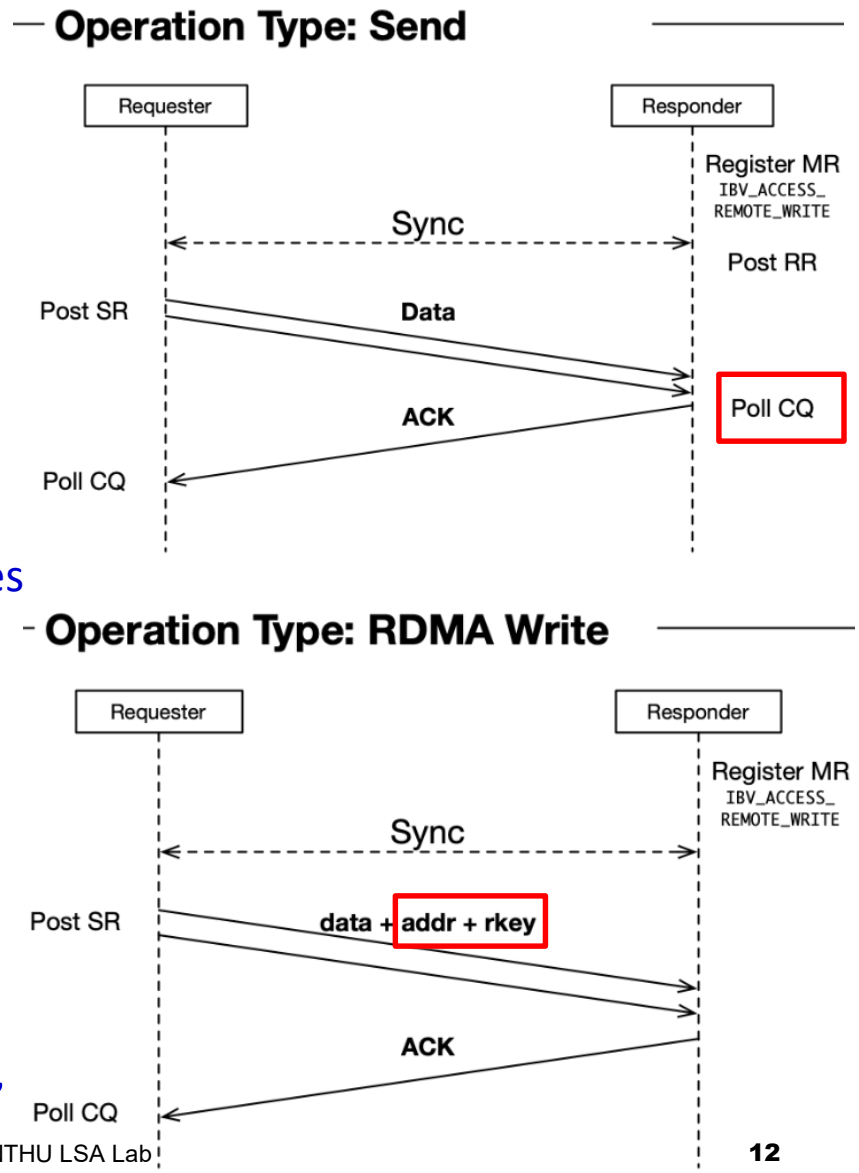
- "Zero" Copy
  - No copies
  - Sender notifies the receiver of its intent
  - Receiver notifies the sender that is ready with destination address
  - Sender sends data to remote memory address
  - Good for large messages (Take time to initiate communications)
- Buffer Copy
  - Temporary buffers on sender and receivers
  - Additional copies to/from buffers
  - Good for small messages that fit in a buffer



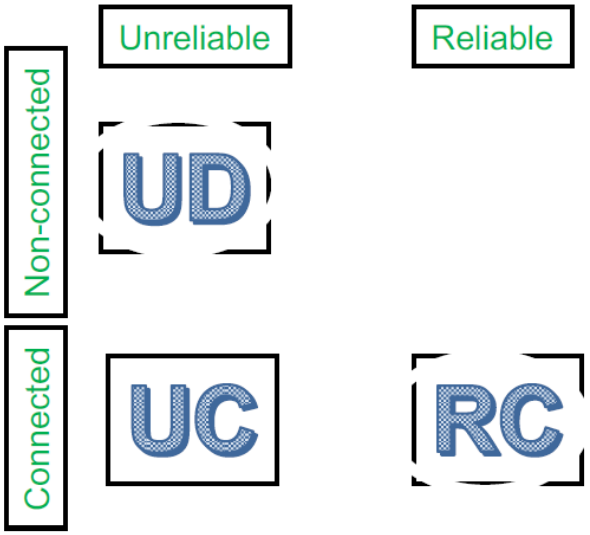
Zero Copy

Communication Semantics

- Send / Receive
  - Send / Receive with TAG matching
  - May enhanced by zero-copy
  - Two-sided communication
    - CPUs still involves on both sides
- RDMA Read and Write
  - One-sided communication
    - Only the CPU of reader/writer involves
  - Require the memory address and key on the remote side
- Atomic Operations on Remote Memory
  - SWAP, CSWAP, ADD, XOR
- Group Communication directives
  - Reduce, Allreduce, Scatter, Gather, AlltoAll



Transport Services



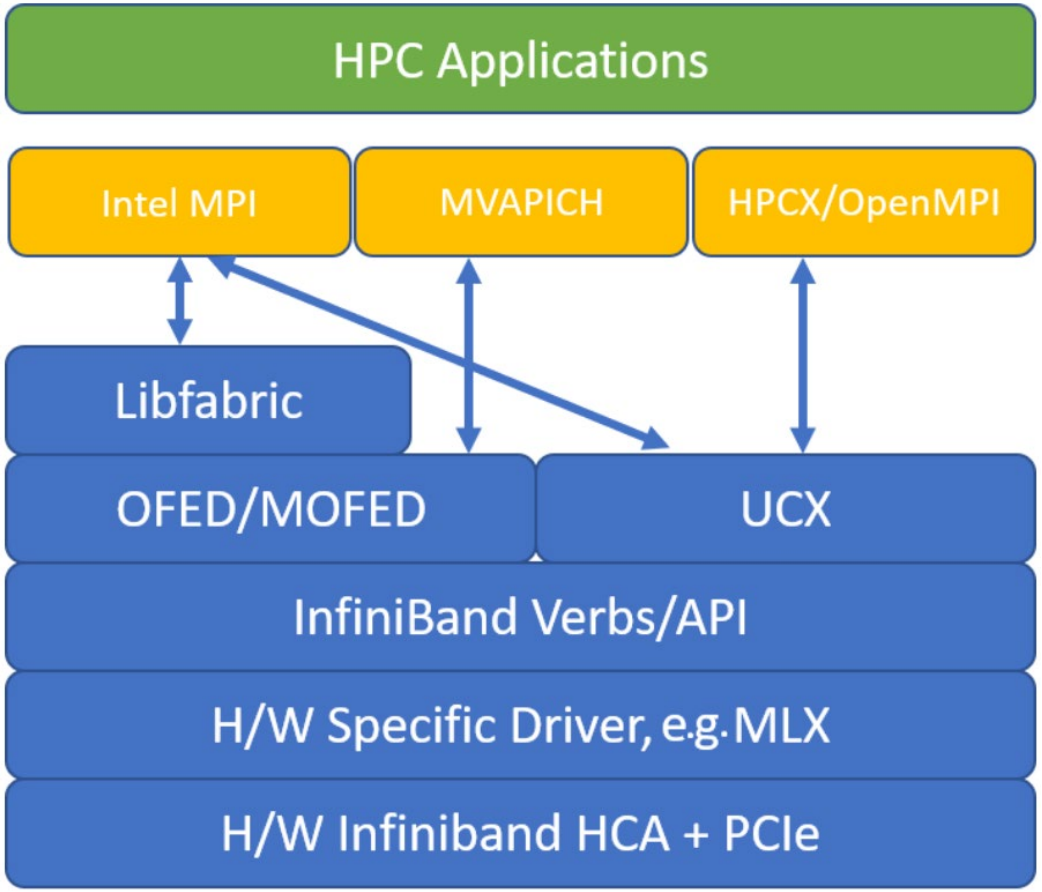
- **Reliable Connection (RC):**
  - Reliable transport, connection oriented
- **Unreliable Datagram (UD):**
  - Unreliable transport, not-connected
- **Unreliable connection (UC):**
  - Unreliable transport, connection oriented

- Reliable
  - exactly once, in-order delivery
- Connected
  - a strong paring of end-nodes

	UD	UC	RC
Send / Receive	√	√	√
RDMA Write	X	√	√
RDMA Read / Atomic	X	X	√
Max Send Size	MTU	2GB	2GB
Reliability	X	X	√
Scalability (per-process for N processes)	1	N	N

Outline

- RDMA Technology
- Verbs
- UCX
- HPC-X

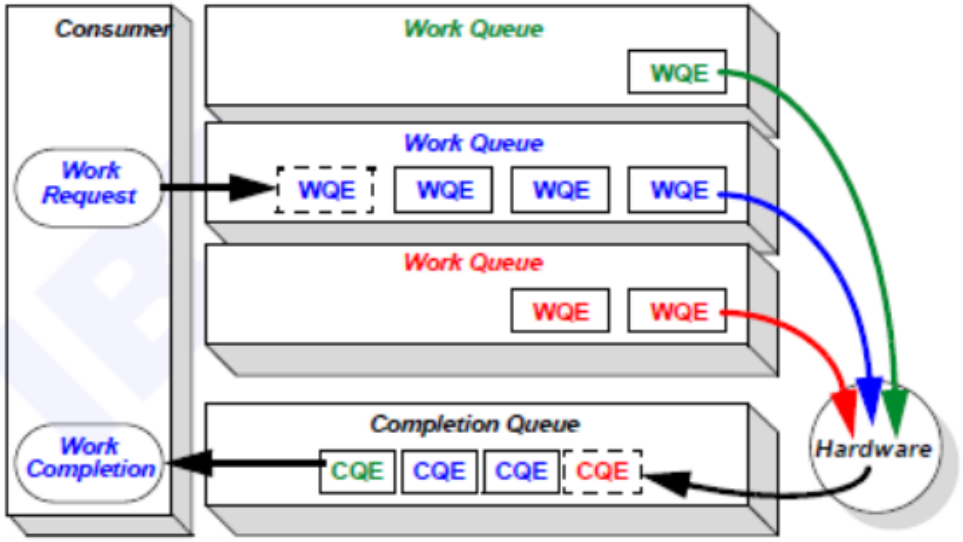


RDMA Standard - Verbs

- **Verbs is an abstract description of the functionality that is provided for applications for using RDMA.**
  - Verbs is not an API
  - There are several implementations for it
- **Verbs is a low-level description for RDMA programming**
  - Verbs are close to the "bare-metal" and provide best performance
    - Latency, BW, Message rate
  - Verbs can be used as building blocks for many applications
    - Sockets, Storage, Parallel computing
- **Any other level of abstraction over verbs *may* harm the performance**

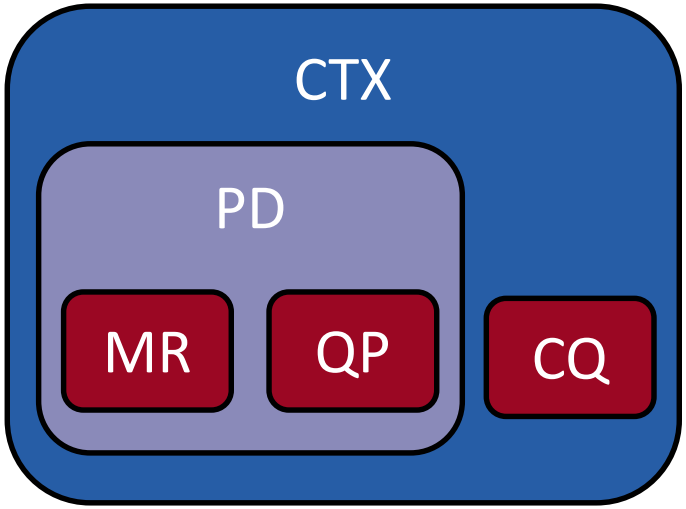
Data Transfer Model – Work Queues

- **Work Request (WR)**
  - Work items that the HW should perform
  - Encapsulated into **WQEs** (work queue elements a.k.a. wookies)
- **Work Completion (completion)**
  - Corresponding to completed WRs
  - Encapsulated into CQEs (completion queue elements a.k.a. cookies)
- **Work Queue (WQ)**
  - A queue which contains WRs
- **Completion Queue (CQ)**
  - A queue which contains completions



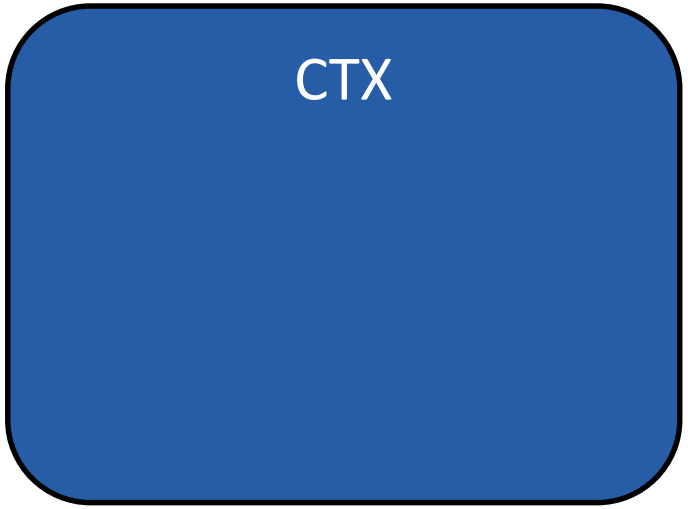
Verbs Resources for Communication

- **Context**
- **Protection Domain**
- **Memory Region**
- **Completion Queue**
- **Queue Pair**



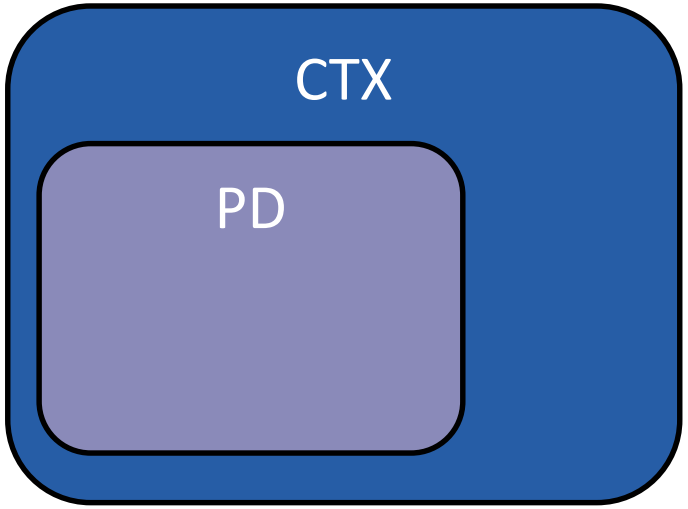
Verbs Resources for Communication

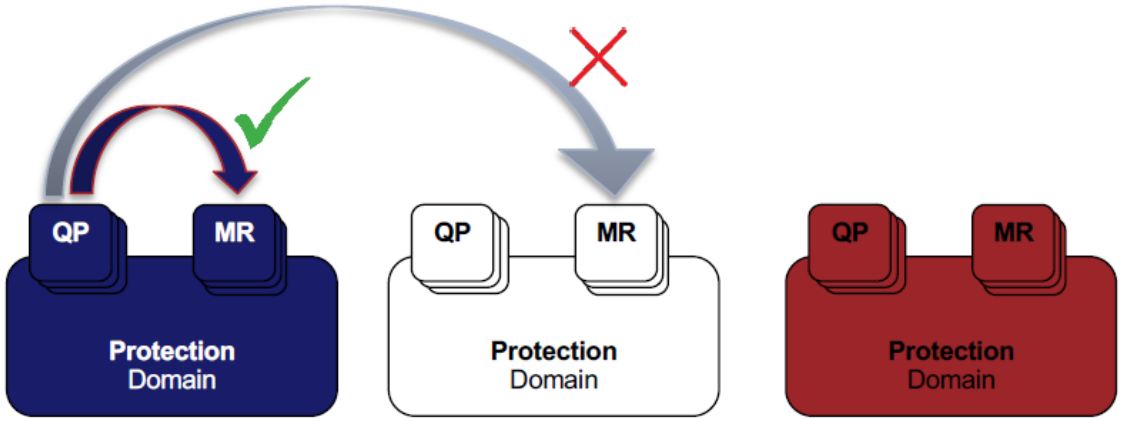
- **Context (ibv_open_device)**
  - Returns a struct ibv_context that is a representation of a device in a list returned by ibv_get_device_list()
  - API allocates resources to communicate with the NIC
  - Context object will contain all the other software resources
  - **Per process**



Verbs Resources for Communication

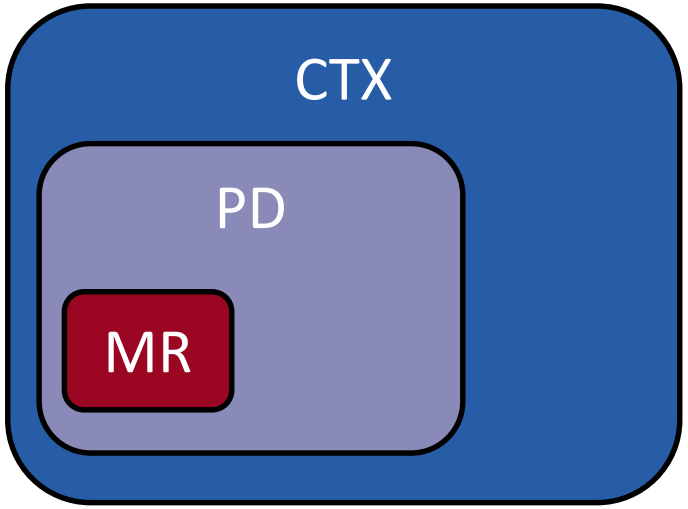
- **Context**
- **Protection Domain (ibv_alloc_pd)**
  - Provides resource isolation within a context
  - A resource can belong to only 1 PD

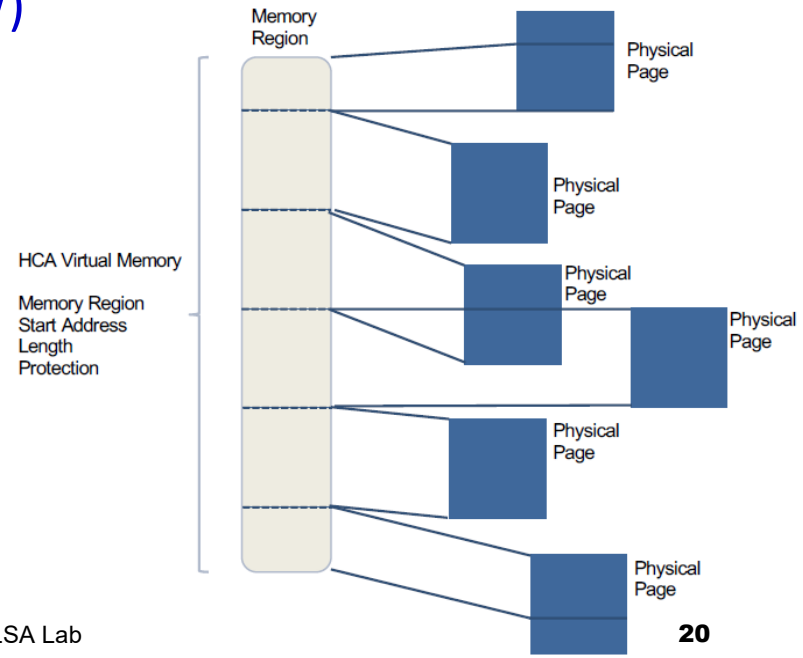




Verbs Resources for Communication

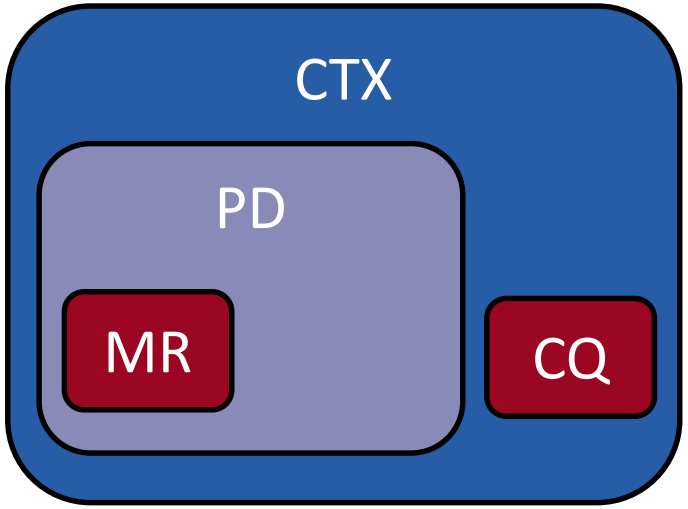
- **Context**
- **Protection Domain**
- **Memory Region (ibv_reg_mr)**
  - Representation of memory that is registered with the NIC
  - Byte level permission protection (R/W)
  - Page level translation
  - Memory Pinning & DMA Mapping



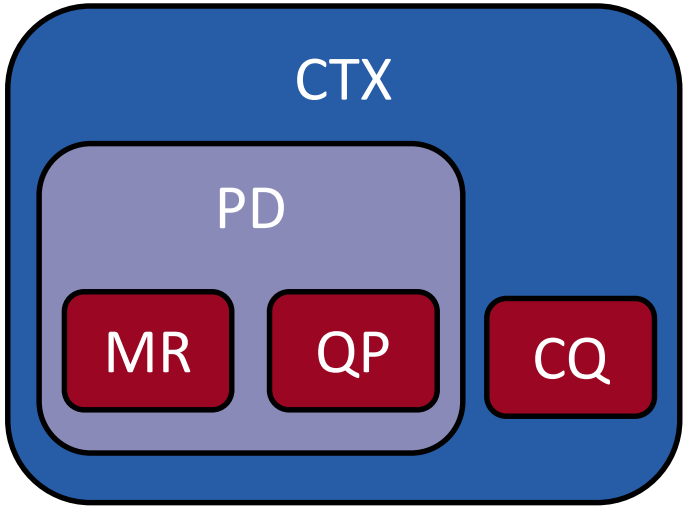


Verbs Resources for Communication

- **Context**
- **Protection Domain**
- **Memory Region**
- **Completion Queue (ibv_create_cq)**
  - Object that will hold completions (CQEs) corresponding to work requests (WQEs)
  - Applications **Poll** for Completion and Request event



Verbs Resources for Communication



- **Context**
- **Protection Domain**
- **Memory Region**
- **Completion Queue**
- **Queue Pair (ibv_create_qp)**
  - It encapsulates both Send and Receive Queue
    - **Send Queue (SQ):** A **Work Queue** that handles sending messages
    - **Receive Queue (RQ):** A **Work Queue** that handles incoming messages
    - Each of them is completely independent & Full duplex
  - A QP represent a real HW resource
    - Every QP is associated with a **Partition Key (P_Key)**

libibverbs

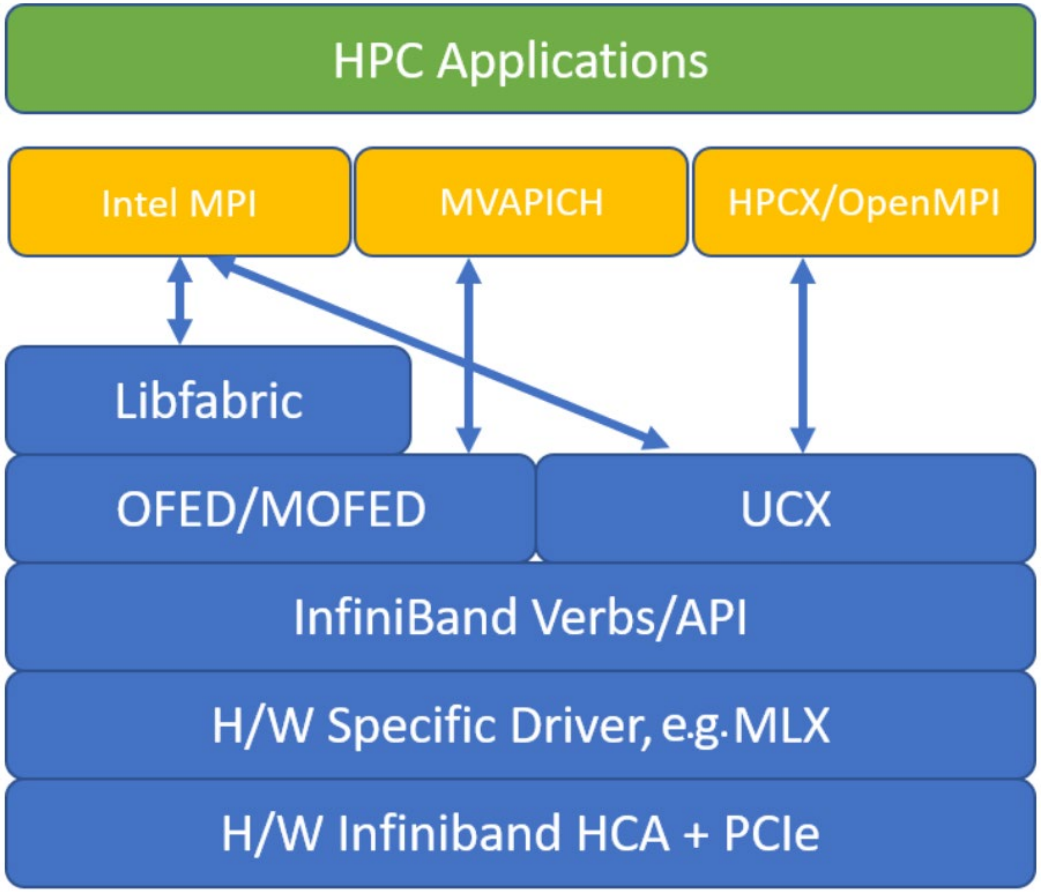
- **libibverbs, developed and maintained by Roland Dreier since 2006, are de-facto the verbs API standard in *nix**
  - Developed as an Open source, community project
  - The kernel part of the verbs is integrated in the Linux kernel since 2005
  - There are low-level libraries from several HW vendors
- **Same API for all RDMA-enabled transport protocols**
  - **Infiniband Networks**
    - Used extensively in HPC machines (Supercomputers)
    - Expensive, requires specialized hardware (physical network and NIC)
  - **RoCE: RDMA done over Ethernet instead of Infiniband** (RDMA over Converged Ethernet)
    - Still requires specialized hardware
    - Cheaper because need only specialized NICs
    - RoCE seems to perform worse at scale (Ethernet is lossy)
  - **iWARP**
    - RDMA over TCP
    - Once again, cheaper; only needs specialized NICs

Network Programming Interfaces (beyond sockets)

- Open Fabric Alliance: Verbs, Udapl, SDP, libfabrics, …
- Research: Portals, CCI, UCCS
- Vendors: Mellanox MXM, Cray uGNI/DMAPP, Intel PSM, Atos Portals, IBM PAMI, OpenMX
- Programming model driven: MVAPICH-X, GasNET, ARMCI
- Enterprise App oriented: OpenDataPlane, DPDK, Accelio

Outline

- RDMA Technology
- Verbs
- UCX
- HPC-X

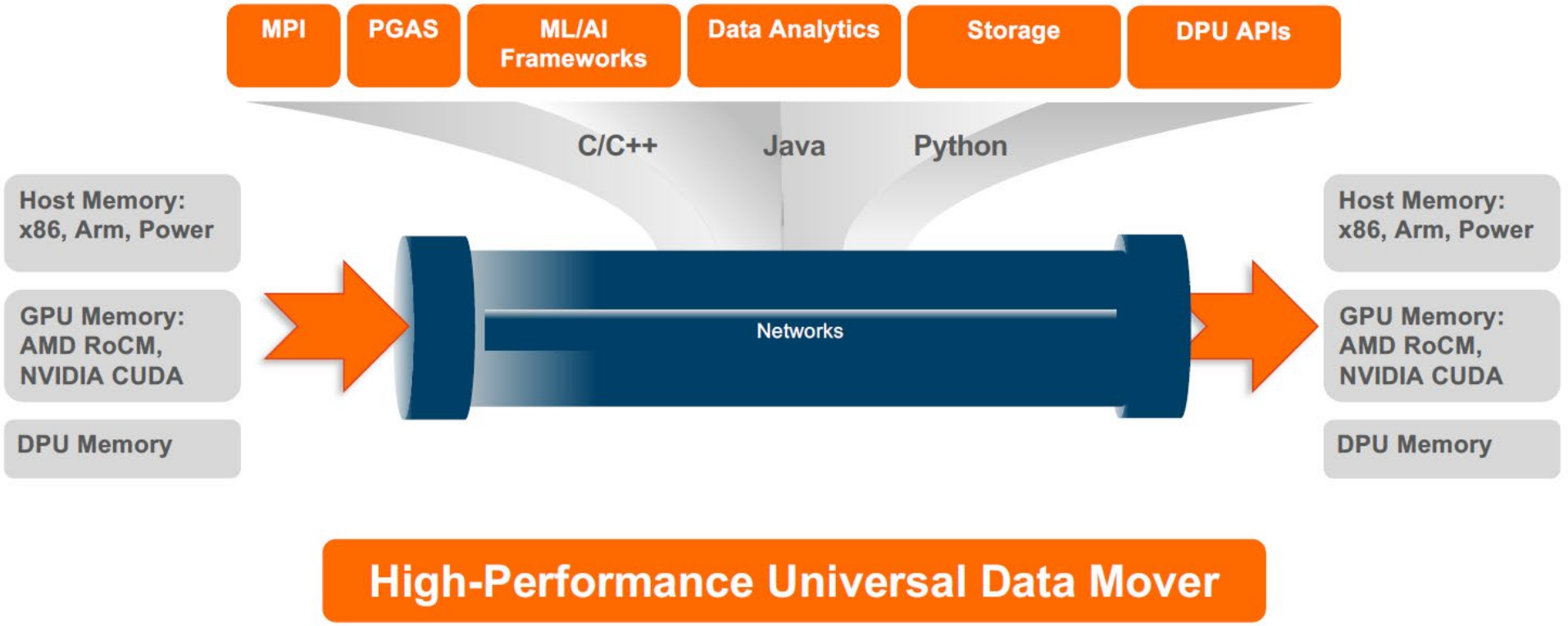


Oscar Hernandez
on behalf of Gilad Shainer/UCF

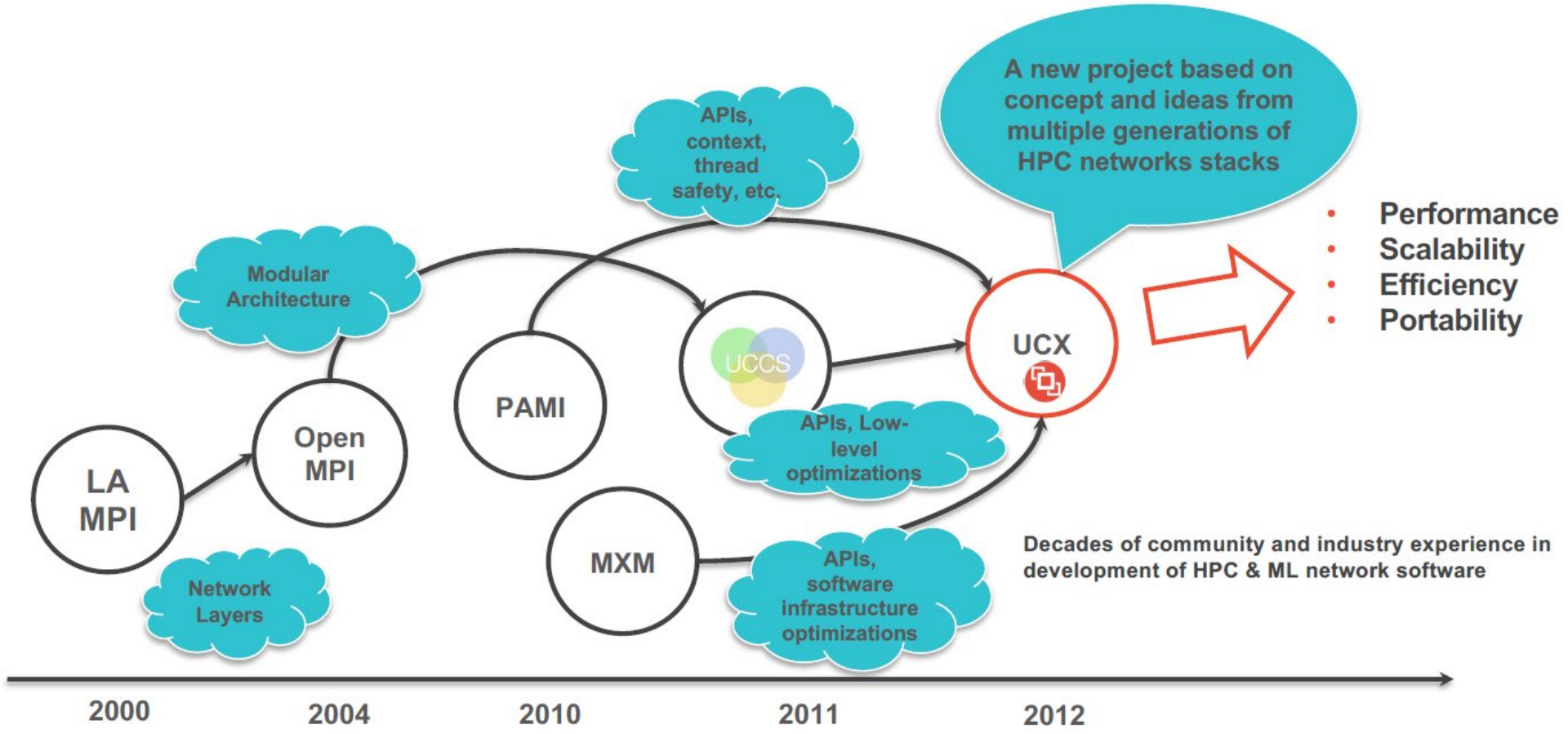
https://github.com/gt-crnch-rg/ucx-tutorial-hot-interconnects

HOTI 2022: UCX TUTORIAL

What is UCX?


MPI
PGAS
ML/AI Frameworks
Data Analytics
Storage
DPU APIs
C/C++
Java
Python
Host Memory: x86, Arm, Power
GPU Memory: AMD RoCM, NVIDIA CUDA
DPU Memory
Networks
Host Memory: x86, Arm, Power
GPU Memory: AMD RoCM, NVIDIA CUDA
DPU Memory
High-Performance Universal Data Mover

UCX History


A new project based on concept and ideas from multiple generations of HPC networks stacks
APIs, context, thread safety, etc.
Modular Architecture
PAMI
UCCS
UCX
Performance
Scalability
Efficiency
Portability
LA MPI
Open MPI
APIs, Low-level optimizations
MXM
APIs, software infrastructure optimizations
Network Layers
Decades of community and industry experience in development of HPC & ML network software
2000
2004
2010
2011
2012

Unified Communication Framework (UCF) Consortium

MISSION: Collaboration between industry, laboratories, and academia to create production grade communication frameworks and open standards for data centric, ML/AI, and high-performance applications

Projects & Working Groups

- **UCX – Unified Communication X – www.openucx.org**
- SparkUCX – www.sparkucx.org
- OpenSNAPI – Smart NIC Project
- UCC – Collective Library
- UCD – Advanced Datatype Engine
- HPCA Benchmark – Benchmarking Effort

Board members

- **Jeff Kuehn**, UCF Chairman (AMD)
- **Gilad Shainer**, UCF President (NVIDIA)
- **Pavel Shamis**, UCF Treasurer (Arm)
- **Yanfei Guo**, Board Member (Argonne National Laboratory)
- **Perry Schmidt**, Board Member (IBM)
- **Dhabaleswar K. (DK) Panda**, Board Member (Ohio State University)
- **Steve Poole**, Board Member (Open Source Software Solutions)

Join

https://www.ucfconsortium.org or info@ucfconsortium.org

Unified Communication X (UCX)

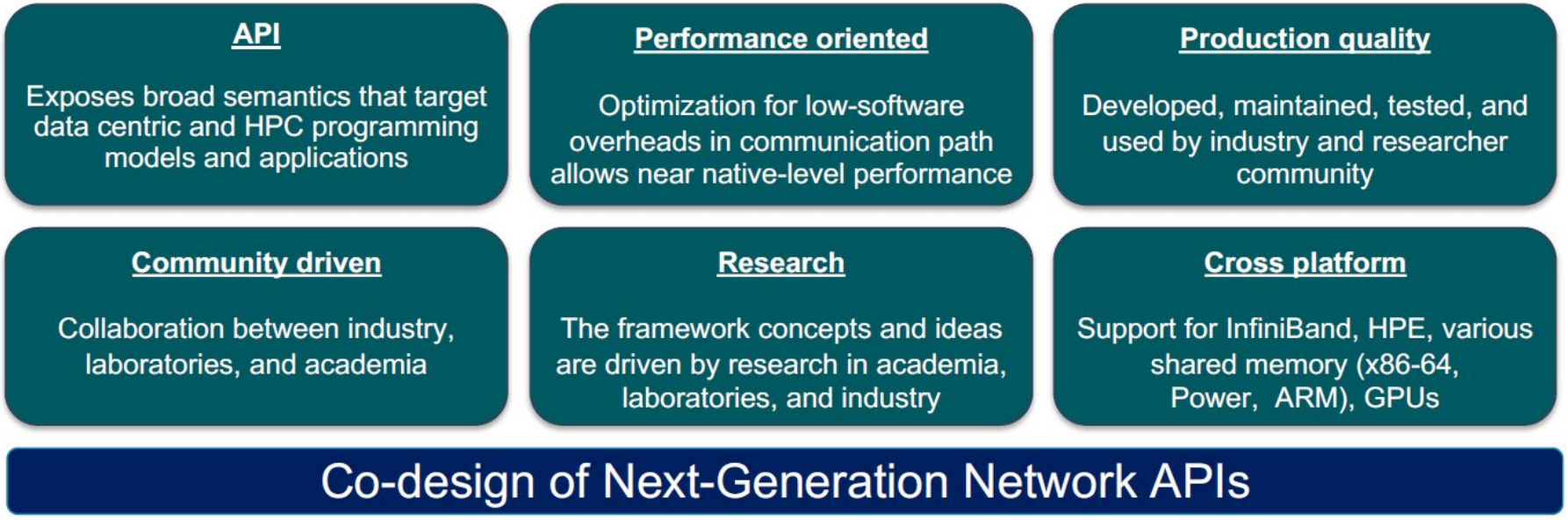
https://www.hpcwire.com/2018/09/17/ucf-ucx-and-a-car-ride-on-the-road-to-exascale/

UCX Useful Links

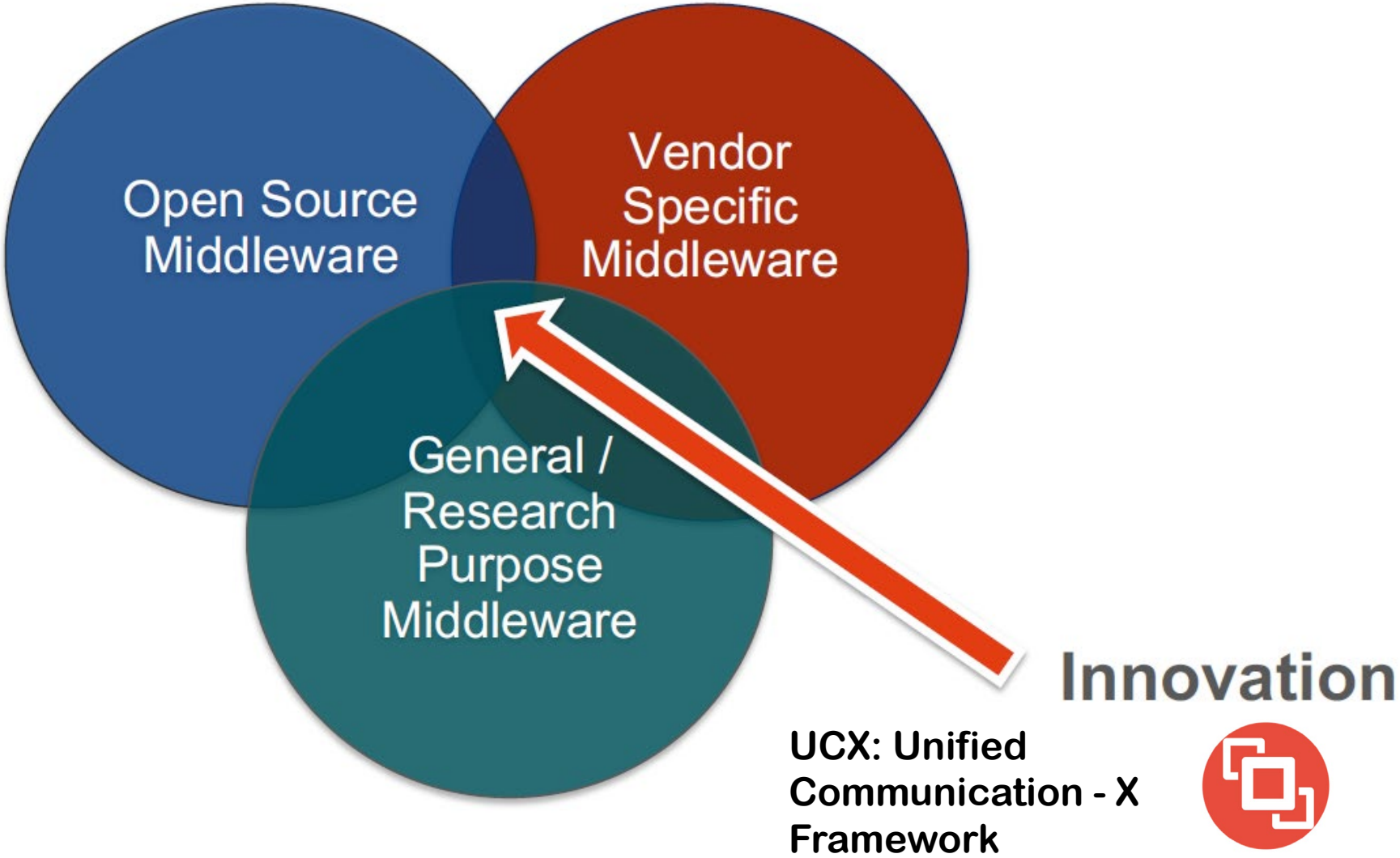
- Code
  - https://github.com/openucx/
- Website
  - www.openucx.com
- Mailing list
  - https://elist.ornl.gov/mailman/listinfo/ucx-group
- Contributor agreement
  - https://www.openucx.org/license/
- User documentation
  - https://openucx.readthedocs.io/

UCX Framework Mission

- Collaboration between industry, laboratories, and academia
- Create open-source production grade communication framework for HPC applications
- Enable the highest performance through co-design of software-hardware interfaces
- Unify industry - national laboratories - academia efforts



Network Programming Interfaces



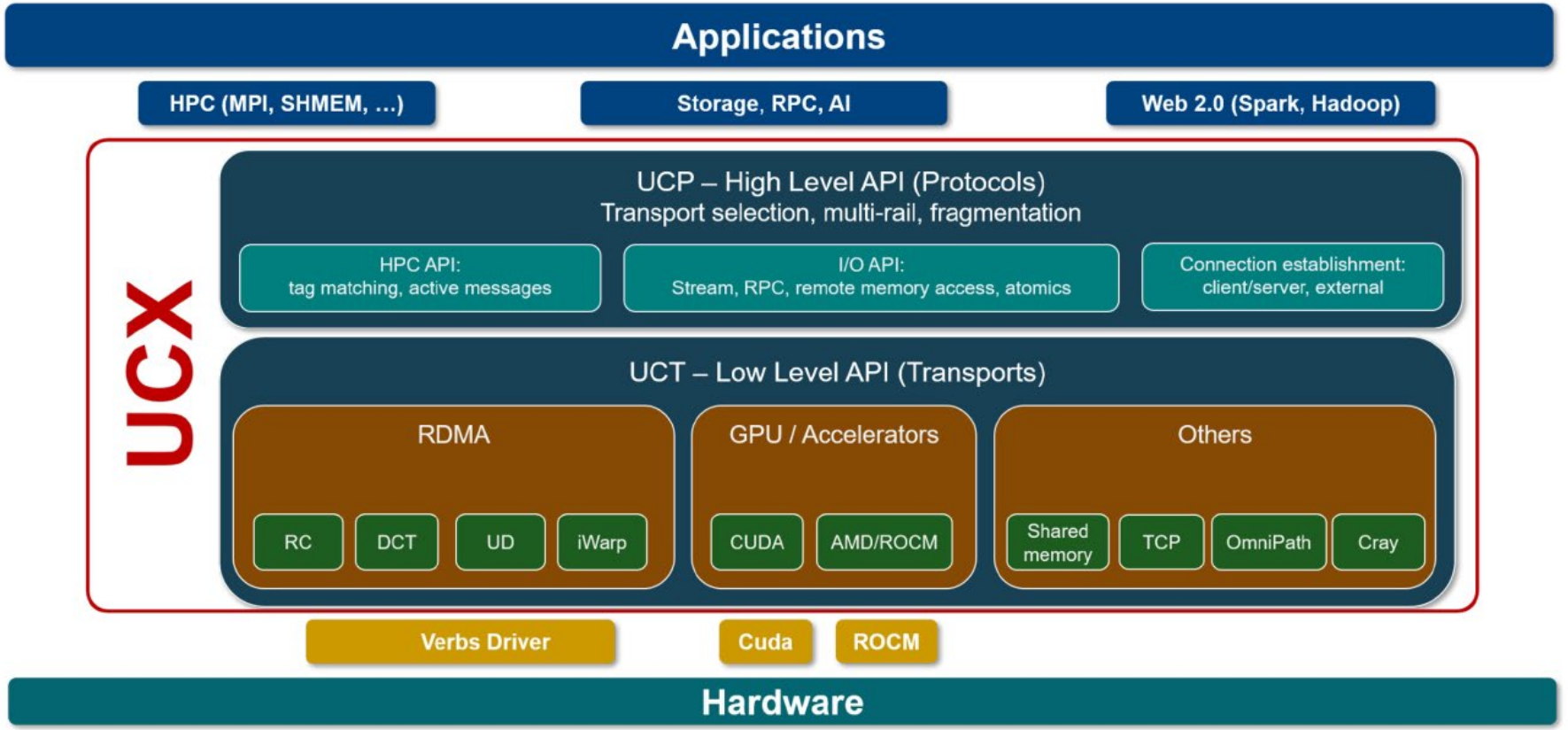
Network Programming Interfaces

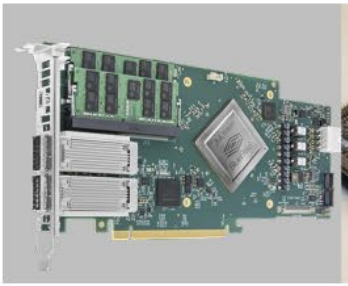
	Pros	Cons
Vendor-Specific APIs	▪ Production Quality ▪ Optimized for Performance ▪ Support and maintenance	▪ Often “vendor” locked ▪ Optimized for a particular technology ▪ Co-design lags behind
Open-Source APIs	▪ Community (a.k.a. user) driven ▪ Easy to modify and extend ▪ Good for research	▪ Typically, not as optimized as commercial/vendor software ▪Maintenance is challenge
Research API	▪ Innovative and forward looking • A lot of good ideas for “free”	▪ Support, support, support ▪ Typically, narrow focus

What's innovative about UCX?

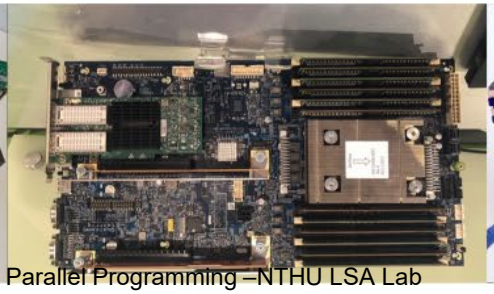
- **Simple, consistent, performance portable unified API**
- Choosing between low-level and high-level API allows easy integration with a wide range of applications and middleware.
- Protocols and transports are selected by capabilities and performance estimations, rather than hard-coded definitions.
- Support thread contexts and dedicated resources, as well as fine-grained and coarse-grained locking.
- Accelerators are represented as a transport, driven by a generic "glue" layer, which will work with all communication networks.

UCX High-level Overview




BlueField DPU

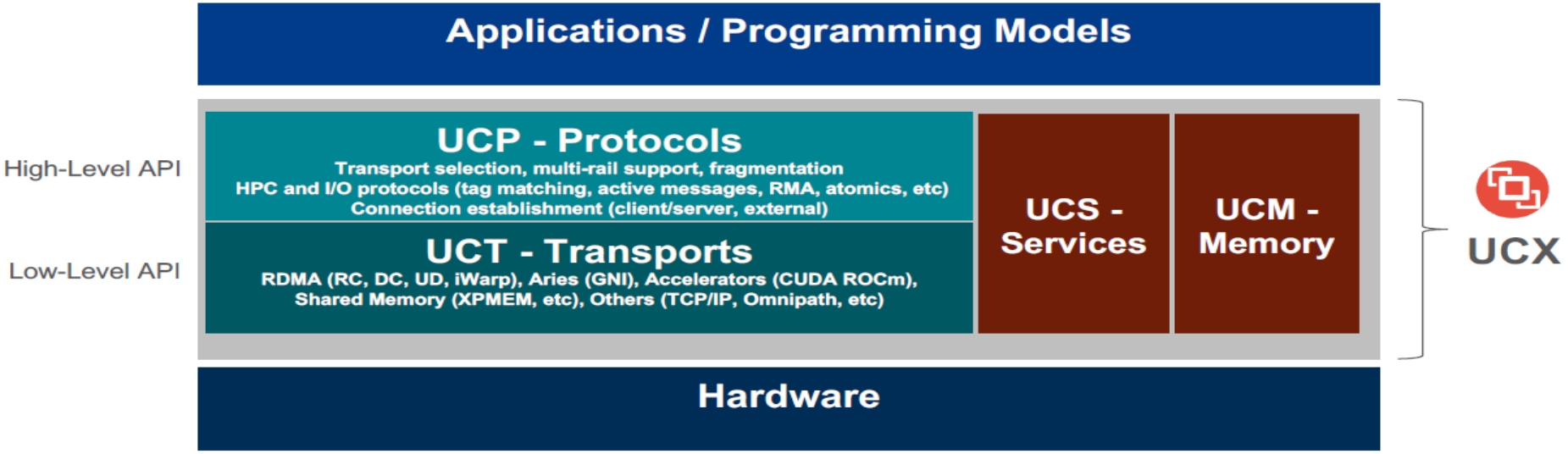

NVIDIA Jetson


Arm ThunderX2


Odroid C2


N1 SDP

UCX Framework



UC-P for Protocols

High-level API uses UCT framework to construct protocols commonly found in applications

Functionality:
Multi-rail, device selection, pending queue, rendezvous, tag-matching, software-atomics, etc.

UC-T for Transport

Low-level API that expose basic network operations supported by underlying hardware. Reliable, out-of-order delivery.

Functionality:
Setup and instantiation of communication operations.

UC-S for Services

This framework provides basic infrastructure for component-based programming, data structure support, and useful system utilities

Functionality:
Platform abstractions, data structures support, debug facilities.

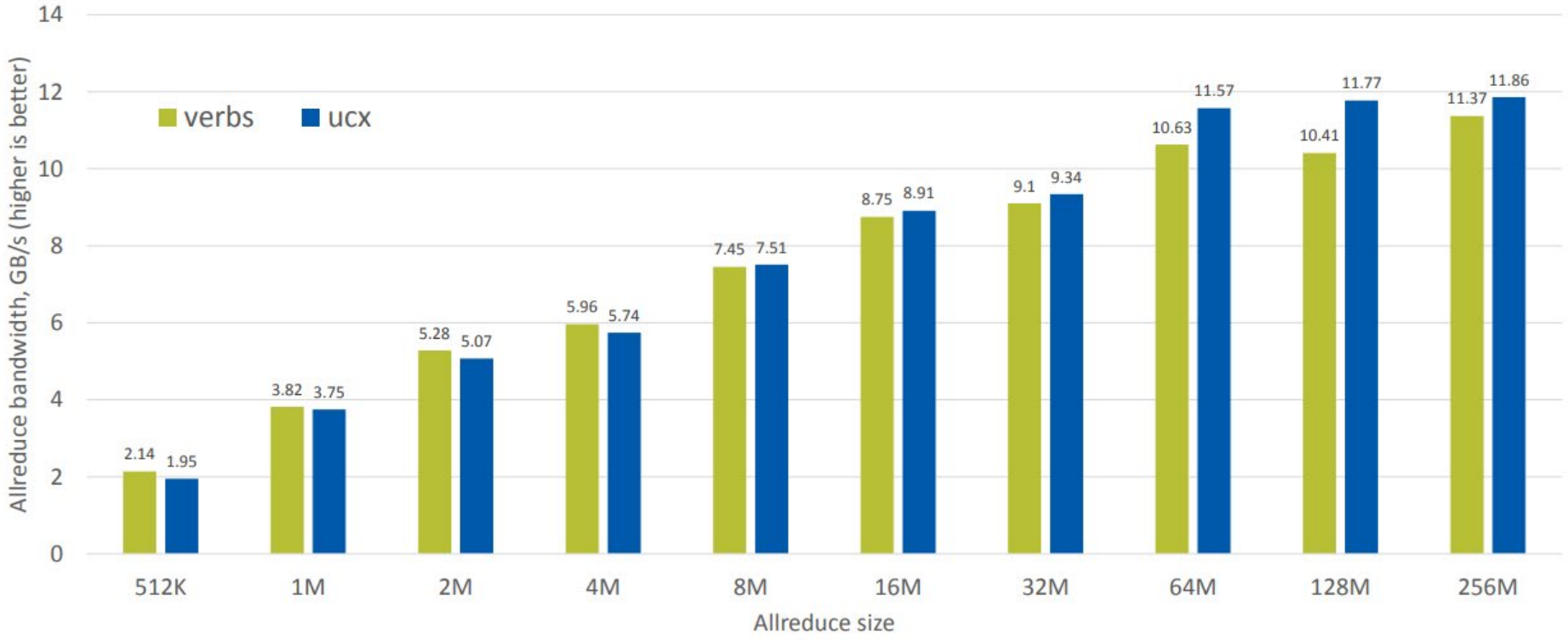
UC-M for Memory

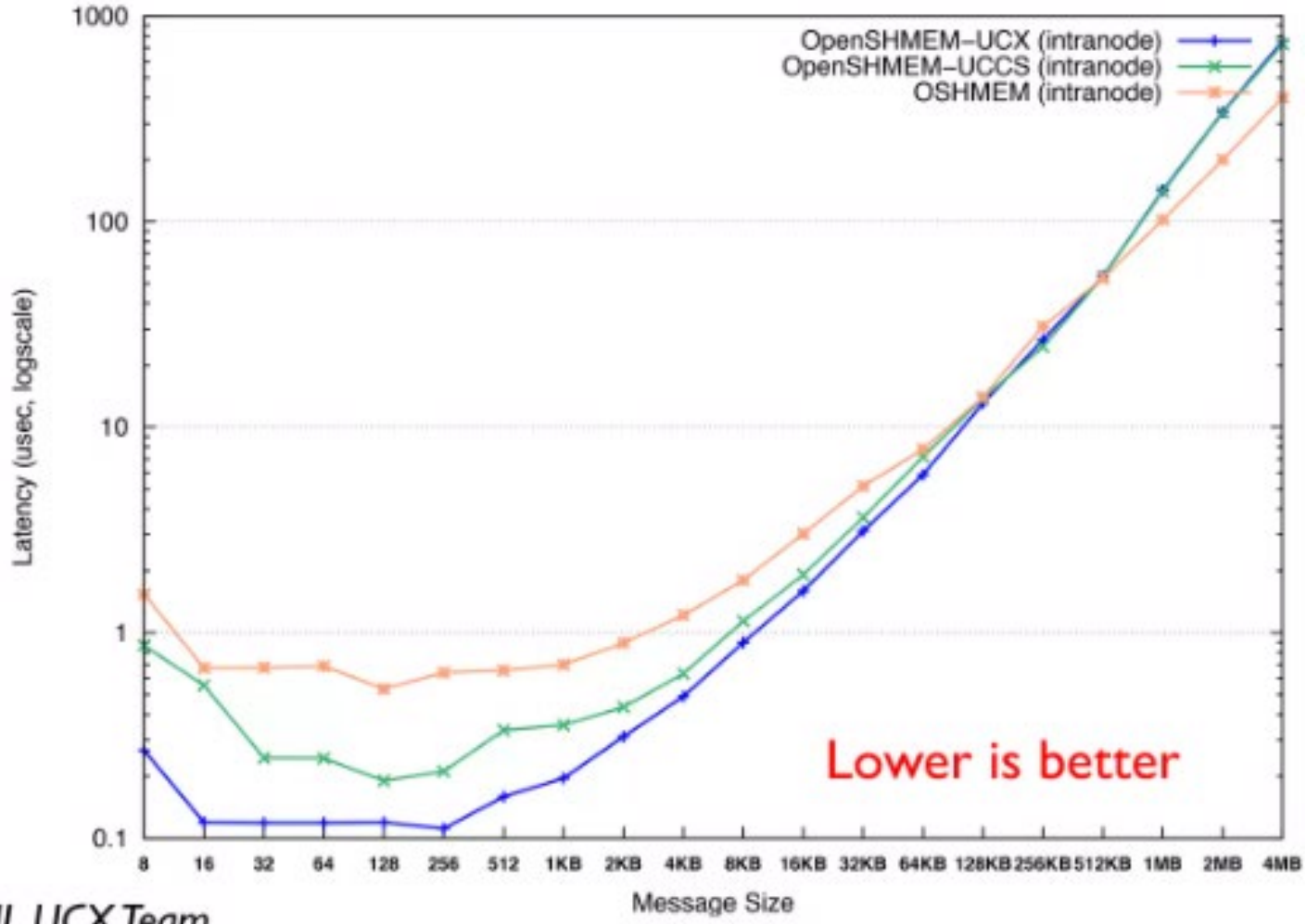
This framework provides infrastructure for getting notifications about memory allocate and release events

Functionality:
Platform for memory allocations/dealloc. notifications across devices

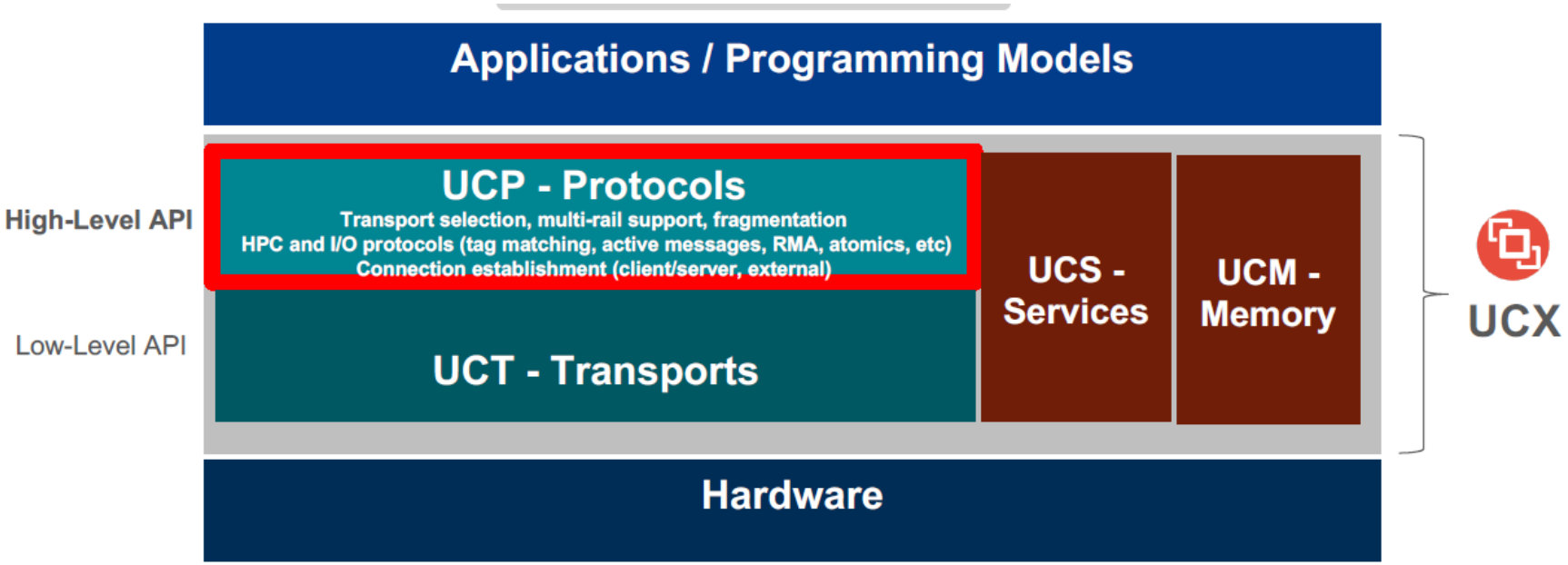
NCCL Internal Verbs vs NCCL UCX Plugin

- UCX plugin outperforms NCCL Verbs implementation up to 13% on large messages





Slide courtesy of ORNL UCX Team



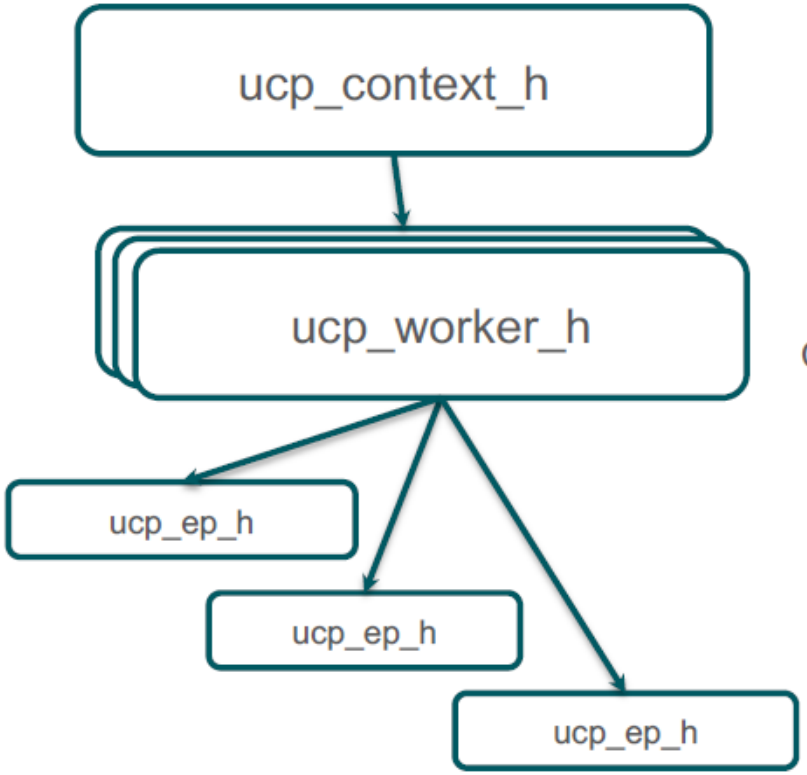
API OVERVIEW

UCP – PROTOCOL LAYER

UCP Objects

- ucp_context_h
  - A global context for the application; holds the memory registrations and global device context.
- ucp_worker_h
  - Communication and progress engine context; handles transport related resources: handling interrupts, connection establishment, completion events, etc. One possible usage is to create one worker per thread.
- ucp_ep_h
  - Represents a connection from a local worker to a remote worker. All send operations are performed on an end point.

UCP Objects



One instance per application process

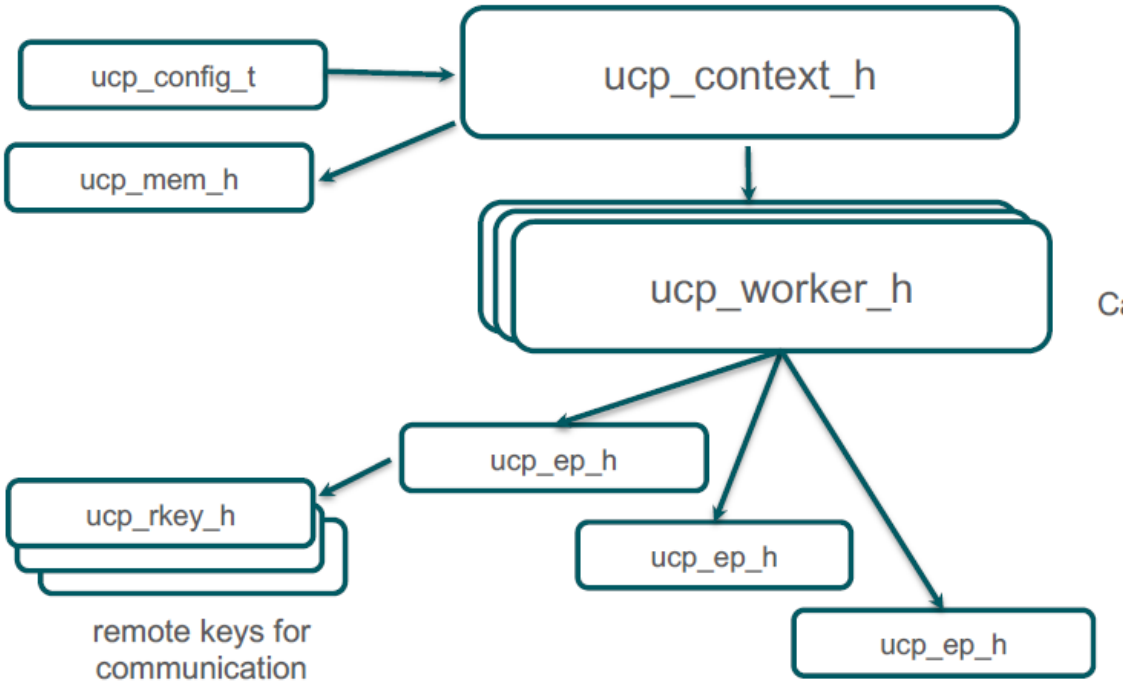
Can have multiple workers to progress communication

Can have multiple end-points for communication peers

UCP Objects (Cont…)

- **ucp_mem_h**
  - A handle to an allocated or registered memory in the local process. Contains details describing the memory, such as address, length etc.
- **ucp_rkey_h**
  - Remote key handle, communicated to remote peers to enable access to the memory region. Contains an array of uct_rkey_t's.
- **ucp_config_t**
  - Configuration for ucp_context_h. Loaded from the run-time to set environment parameters for UCX.

UCP Objects (Cont…)

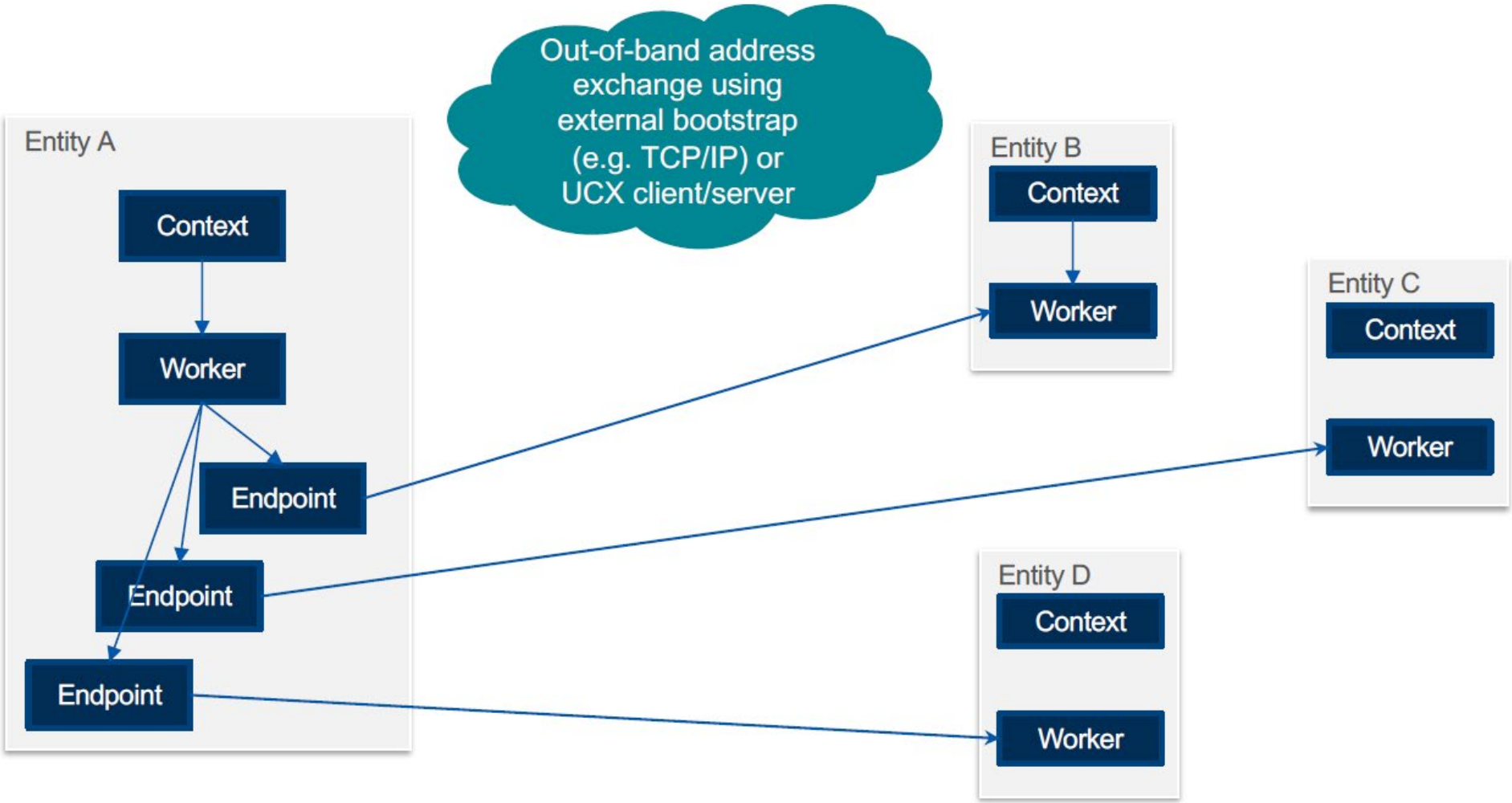


Once instance per application process

Can have multiple workers to progress communication

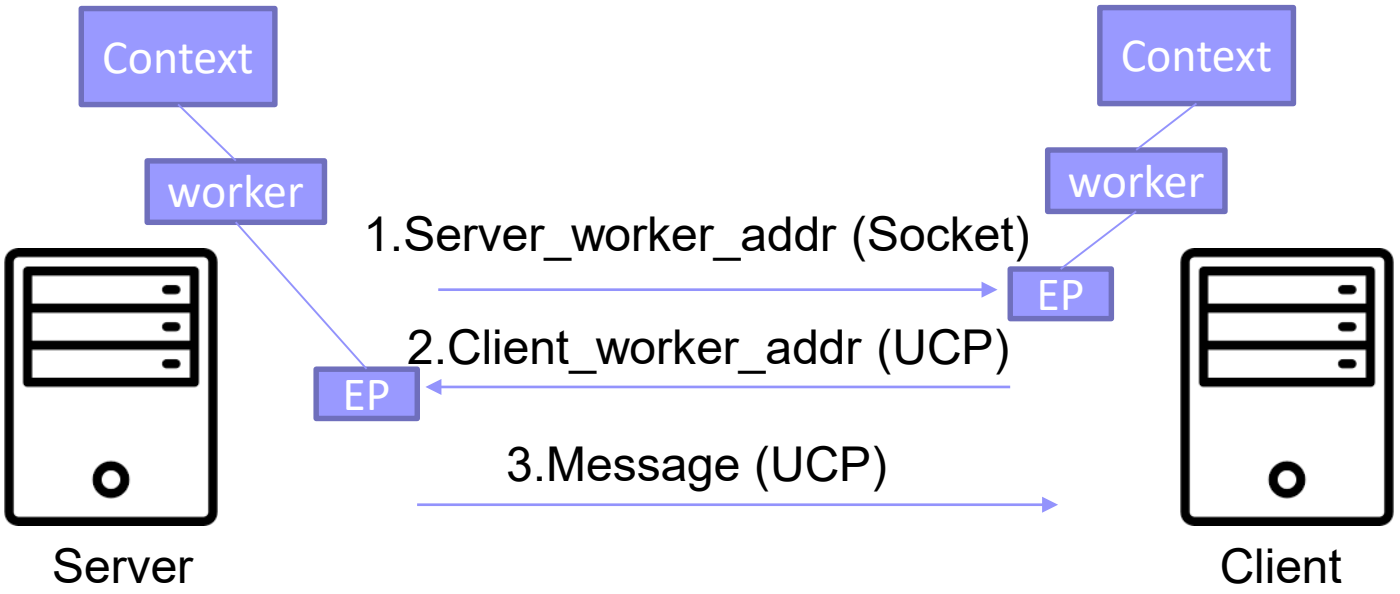
Can have multiple end-points for communication peers

UCP API

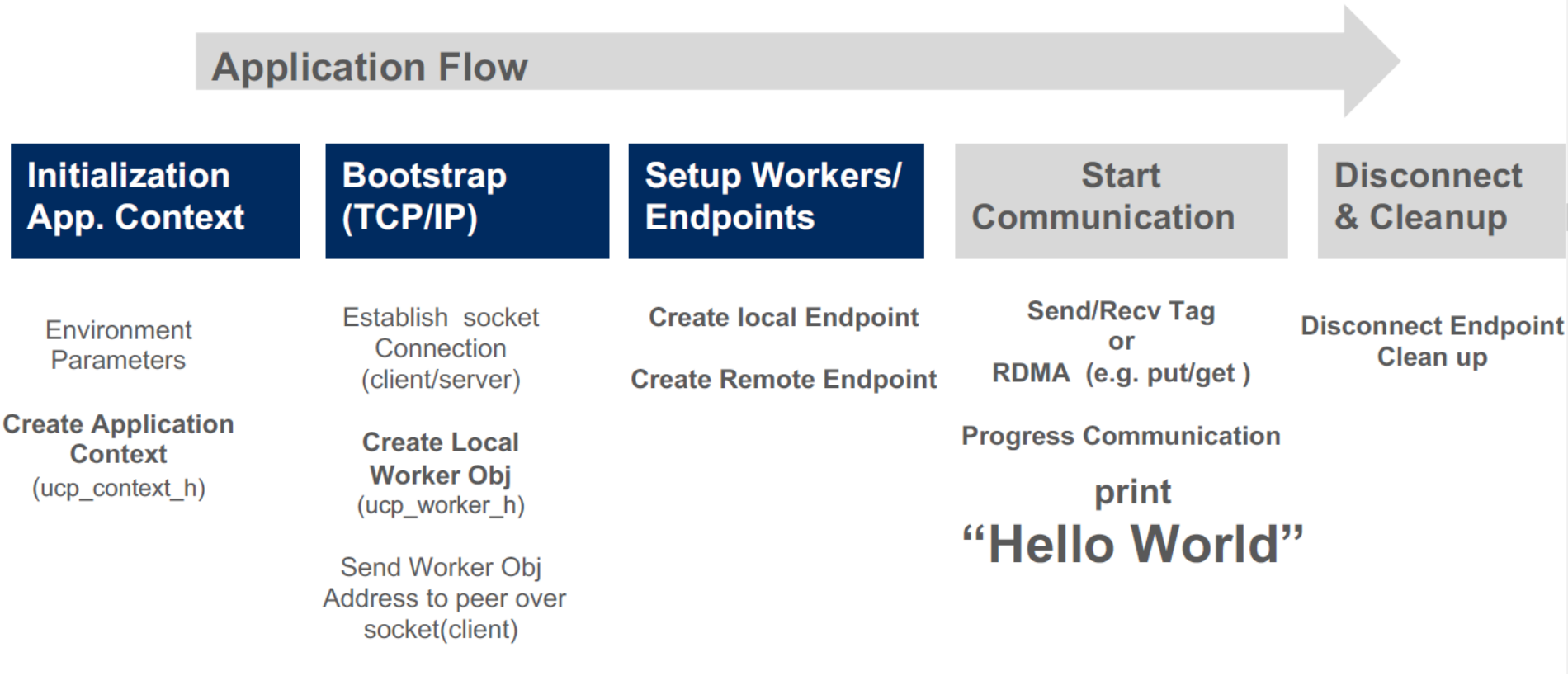

Out-of-band address exchange using external bootstrap (e.g. TCP/IP) or UCX client/server
Entity A
Context
Worker
Endpoint
Endpoint
Endpoint
Entity B
Context
Worker
Entity C
Context
Worker
Entity D
Context
Worker

Hello word Example

- Clip: [1:05:30 ~ 1:21:10]
- Code: **ucp_hello_world.c**



Putting it together: Implementing “Hello World” using UCP



UCP Initialization

- ucp_init (const ucp_params_t * **params**,
  const ucp_config_t * **config**, ucp_context_h * **context_p**)

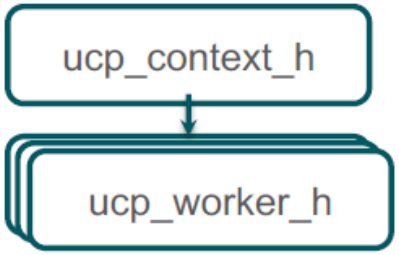
ucp_context_h

in	*config*	UCP configuration descriptor allocated through ucp_config_read() routine.
in	*params*	User defined tunings for the UCP application context.
out	*context_p*	Initialized UCP application context.

- This routine creates and initializes a UCP application context.
- This routine checks API version compatibility, then discovers the available network interfaces, and initializes the network resources required for discovering of the network and memory related devices.
- This routine is responsible for initialization all information required for a particular application scope, for example, MPI application, OpenSHMEM application, etc.
- Related routines: ucp_cleanup, ucp_get_version

UCP Initialization

- ucs_status_t ucp_worker_create (ucp_context_h **context**, ucs_thread_mode_t **thread_mode**, ucp_worker_h ***worker_p**)



in	*context*	Handle to UCP application context.
in	*thread_mode*	Thread safety mode for the worker object and resources associated with it.
out	*worker_p*	A pointer to the worker object allocated by the UCP library

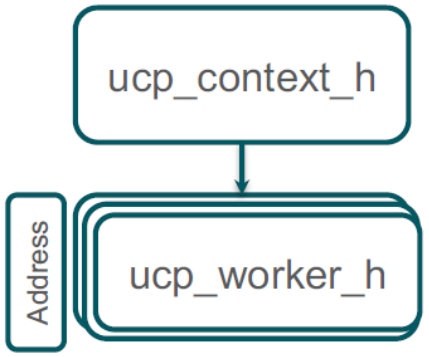
- This routine allocates and initializes a worker object. Each worker is associated with one and only one application context. At the same time, an application context can create multiple workers in order to enable concurrent access to communication resources.
  - For example, application can allocate a dedicated worker for each application thread, where every worker can be progressed independently of others.
- Related routines: ucp_worker_destroy, ucp_worker_get_address, ucp_worker_release_address, ucp_worker_progress, ucp_worker_fence, ucp_worker_flush

UCP Initialization

- ucs_status_t ucp_worker_get_address (ucp_worker_h **worker**, ucp_address_t ** **address_p**, size_t * **address_length_p**)

in	*worker*	Worker object whose address to return.
out	*address_p*	A pointer to the worker address.
out	*address_length_p*	The size in bytes of the address.

  - This routine returns the address of the worker object. This address can be passed to remote instances of the UCP library in order to connect to this worker.
  - The worker **address structure can work with various transportation layer**, including IB, TCP/IP, NVLink.
  - The memory for the address handle is allocated by this function and must be released by using ucp_worker_release_address() routine.



UCP Initialization

- ucs_status_t ucp_ep_create (ucp_worker_h **worker**, const const ucp_ep_params_t ***address**, ucp_ep_h * **ep_p**)

in	*worker*	Handle to the worker; the endpoint is associated with the worker.
in	*address*	Destination address; the address must be obtained using ucp_worker_get_↩ address() routine.
out	*ep_p*	A handle to the created endpoint.

  - This routine creates and connects an endpoint on a local worker for a destination address that identifies the remote worker.
  - The destination address (peer address) should be sent by out of bound communication (e.g., socket)
  - This function is non-blocking, and communications may begin immediately after it returns. If the connection process is not completed, communications may be delayed.
  - The created endpoint is associated with one and only one worker.
  - Related routines: ucp_ep_flush, ucp_ep_fence, ucp_ep_destroy

ucp_hello_world.c

```
struct ucx_context {
    int             completed;
};
static void request_init(void *request)
{
    struct ucx_context *ctx = (struct ucx_context *) request;
    ctx->completed = 0;
}
 /* UCP initialization */
 status = ucp_config_read(NULL, NULL, &config);
 CHKERR_JUMP(status != UCS_OK, "ucp_config_read\n", err);

 ucp_params.field_mask   = UCP_PARAM_FIELD_FEATURES |
                           UCP_PARAM_FIELD_REQUEST_SIZE |
                           UCP_PARAM_FIELD_REQUEST_INIT;
 ucp_params.features     = UCP_FEATURE_TAG;
 ucp_params.request_size    = sizeof(struct ucx_context);
 ucp_params.request_init    = request_init;
 status = ucp_init(&ucp_params, config, &ucp_context);

 worker_params.field_mask  = UCP_WORKER_PARAM_FIELD_THREAD_MODE;
 worker_params.thread_mode = UCS_THREAD_MODE_SINGLE;
 status = ucp_worker_create(ucp_context, &worker_params, &ucp_worker);
 status = ucp_worker_get_address(ucp_worker, &local_addr, &local_addr_len);
```

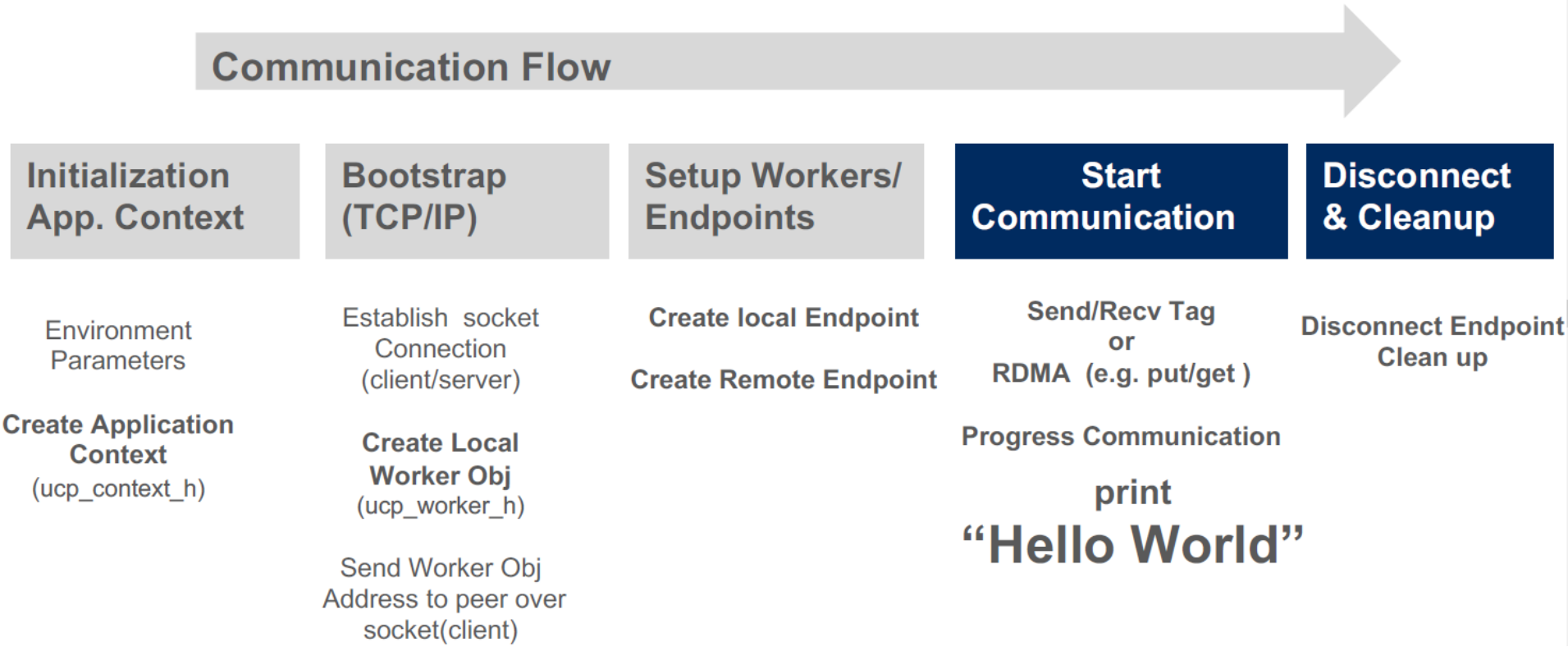
Out of bound communication to send *peer_addr* & *peer_addr_len*

```
if (client_target_name) {
    peer_addr_len = local_addr_len;

    oob_sock = client_connect(client_target_name, server_port);
    CHKERR_JUMP(oob_sock < 0, "client_connect\n", err_addr);

    ret = recv(oob_sock, &addr_len, sizeof(addr_len), MSG_WAITALL);
    CHKERR_JUMP_RETVAL(ret != (int)sizeof(addr_len),
                       "receive address length\n", err_addr, ret);

    peer_addr_len = addr_len;
    peer_addr = malloc(peer_addr_len);
    CHKERR_JUMP(!peer_addr, "allocate memory\n", err_addr);

    ret = recv(oob_sock, peer_addr, peer_addr_len, MSG_WAITALL);
    CHKERR_JUMP_RETVAL(ret != (int)peer_addr_len,
                       "receive address\n", err_peer_addr, ret);
} else {
    oob_sock = server_connect(server_port);
    CHKERR_JUMP(oob_sock < 0, "server_connect\n", err_peer_addr);

    addr_len = local_addr_len;
    ret = send(oob_sock, &addr_len, sizeof(addr_len), 0);
    CHKERR_JUMP_RETVAL(ret != (int)sizeof(addr_len),
                       "send address length\n", err_peer_addr, ret);

    ret = send(oob_sock, local_addr, local_addr_len, 0);
    CHKERR_JUMP_RETVAL(ret != (int)local_addr_len, "send address\n",
                       err_peer_addr, ret);
}
```

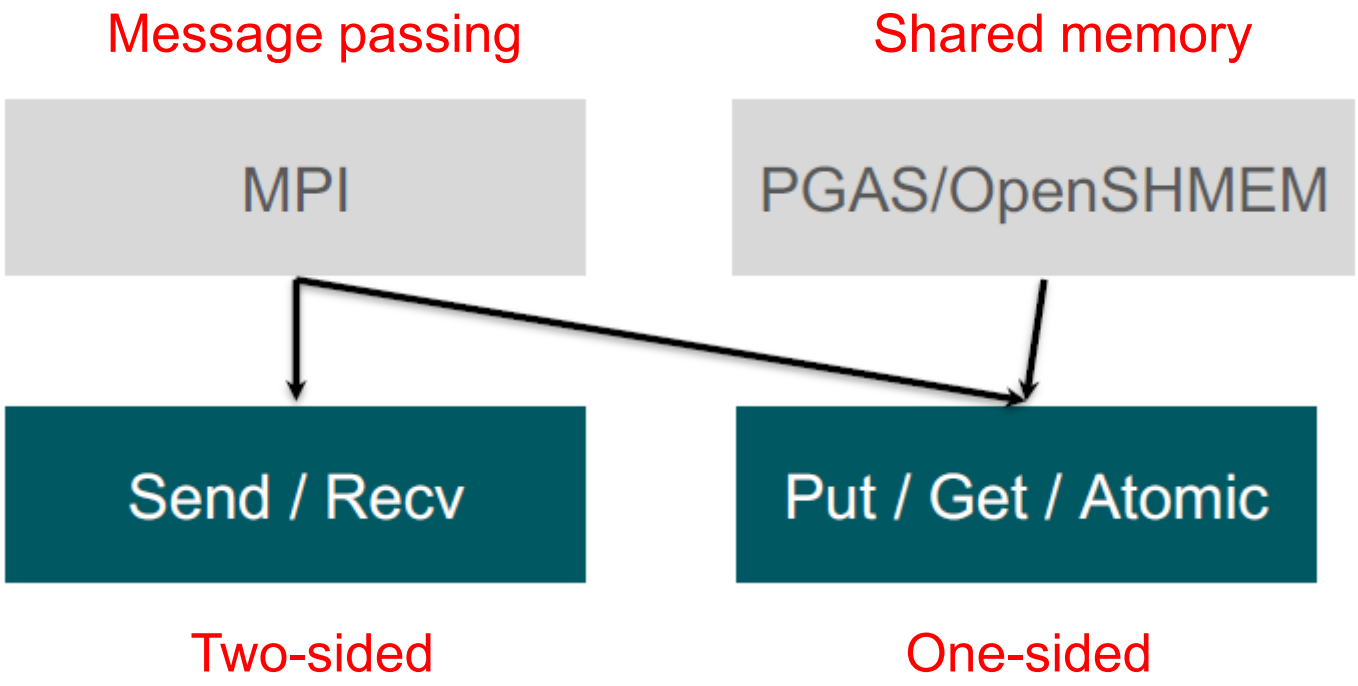
Create ep

```
ep_params.field_mask      = UCP_EP_PARAM_FIELD_REMOTE_ADDRESS;
ep_params.address         = peer_addr;
status = ucp_ep_create(ucp_worker, &ep_params, &server_ep);
```

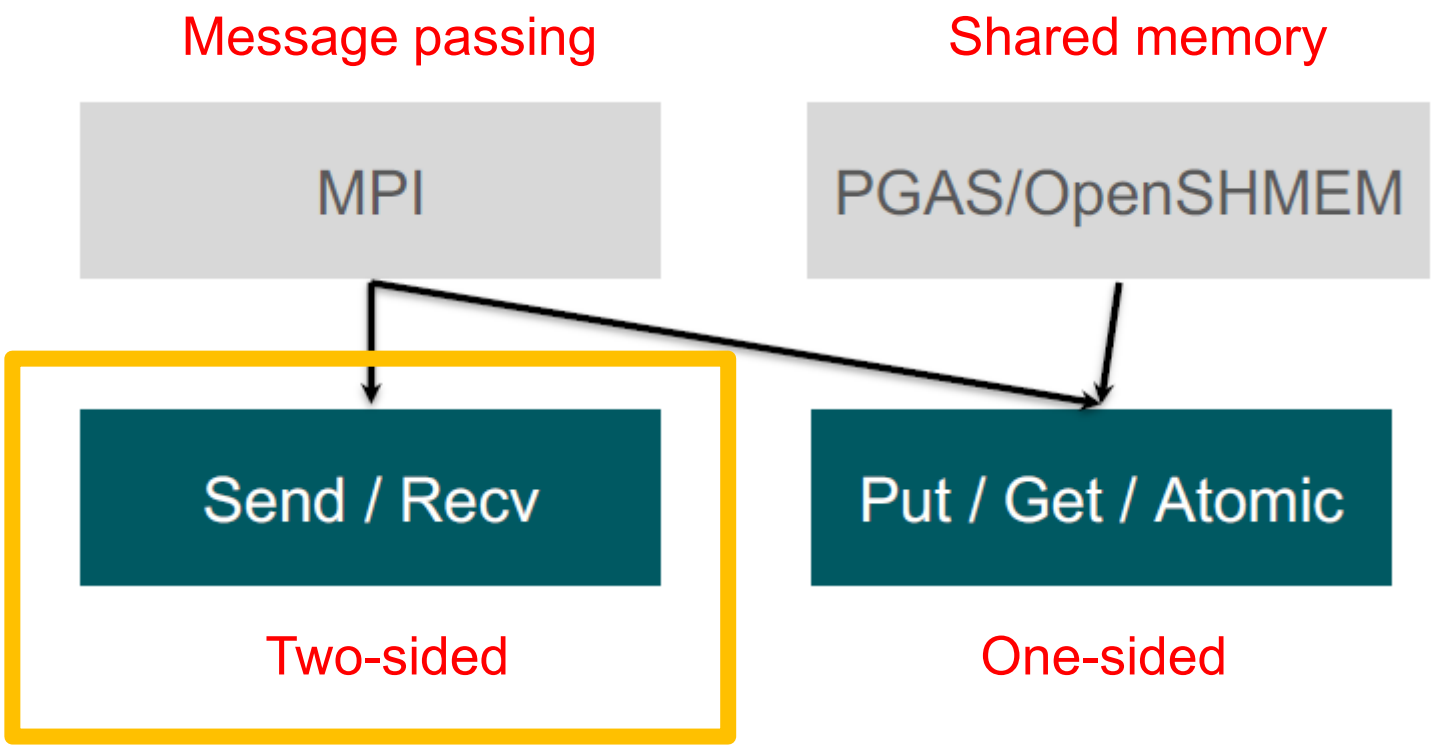
Putting it together: Implementing “Hello World” using UCP



Communication Methods



Communication Methods



UCP Completion / Progress Semantics

- **Operation completion is "local":**
  - Send -when the send buffer can be released/reused
  - Receive -when the data is ready in the receive buffer
- **UCP Worker Progress:ucp_worker_progress()**
  - Advances state of the communication stack (poll CQs, check shared memory queues, etc.)
  - Invokes application callbacks when sends/receives are completed, connection requests arrive, etc.
  - Usually should be called in the application main loop.

```
static void ucx_wait(ucp_worker_h ucp_worker, struct ucx_context *context)
{
    while (context->completed == 0) {
        ucp_worker_progress(ucp_worker);
    }
}
```

Send (Non-blocking)

- ucs_status_ptr_t ucp_tag_send_nbx (ucp_ep_h **ep**, const void * **buffer**, size_t **count**, ucp_tag_t **tag**, const ucp_request_param_t * **param**)

in	*ep*	Destination endpoint handle.
in	*buffer*	Pointer to the message buffer (payload).
in	*count*	Number of elements to send
in	*tag*	Message tag.
in	*param*	Operation parameters, see ucp_request_param_t

Receive (Non-blocking)

- ucs_status_ptr_t ucp_tag_recv_nbx (ucp_worker_h **worker**, void * **buffer**, size_t **count**, ucp_tag_t **tag**, ucp_tag_t **tag_mask**, const ucp_request_param_t * **param**)

in	*worker*	UCP worker that is used for the receive operation.
in	*buffer*	Pointer to the buffer to receive the data to.
in	*count*	Number of elements to receive
in	*tag*	Message tag to expect.
in	*tag_mask*	Bit mask that indicates the bits that are used for the matching of the incoming tag against the expected tag.
in	*param*	Operation parameters, see ucp_request_param_t

ucs_status_ptr_t

- UCS_OK - The send operation was completed immediately.
  - UCS_PTR_IS_ERR(_ptr) - The send operation failed.
  - otherwise - Operation was scheduled for send and can be completed in any point in time. The request handle is returned to the application in order to track progress of the message. The application is responsible to release the handle using ucp_request_release() routine.
- Request handling
  - int ucp_request_is_completed (void * request)
  - void ucp_request_release (void * request)
  - void ucp_request_cancel (ucp_worker_h worker, void * request)

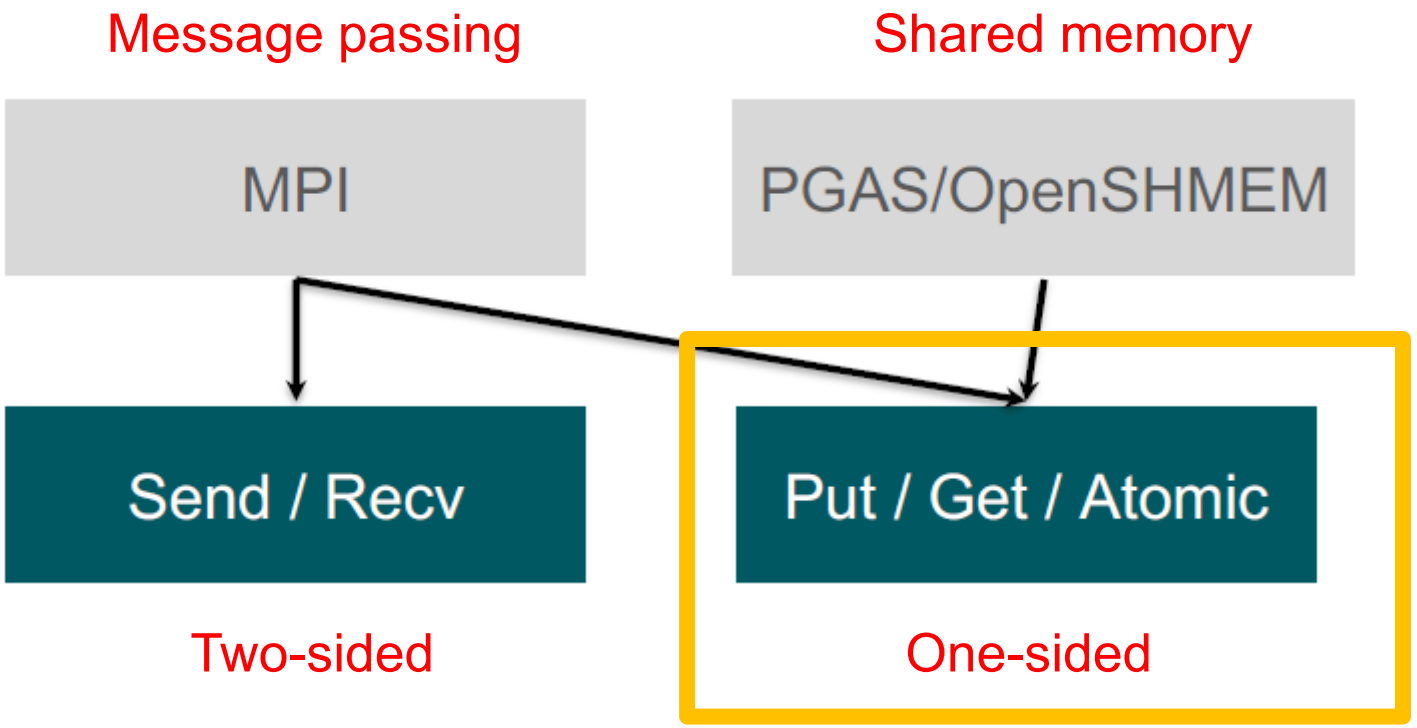
```
request                 = ucp_tag_send_nbx(server_ep, msg, msg_len, tag,
                                           &send_param);
if (UCS_PTR_IS_ERR(request)) {
    fprintf(stderr, "unable to send UCX address message\n");
    free(msg);
    goto err_ep;
} else if (UCS_PTR_IS_PTR(request)) {
    ucx_wait(ucp_worker, &ctx);
    ucp_request_release(request);
}
```

Client side

Server side

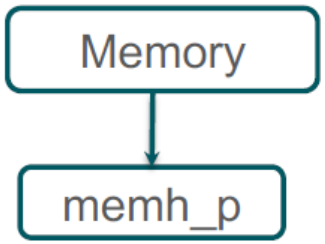
```
request = ucp_tag_msg_recv_nb(ucp_worker, msg, info_tag.length,
                              ucp_dt_make_contig(1), msg_tag, recv_handler);

if (UCS_PTR_IS_ERR(request)) {
    fprintf(stderr, "unable to receive UCX address message (%s)\n",
            ucs_status_string(UCS_PTR_STATUS(request)));
    free(msg);
    ret = -1;
    goto err;
} else {
    /* ucp_tag_msg_recv_nb() cannot return NULL */
    assert(UCS_PTR_IS_PTR(request));
    ucx_wait(ucp_worker, request);
    request->completed = 0;
    ucp_request_release(request);
    printf("UCX address message was received\n");
}
```

Communication Methods



UCP Memory Management

- ucs_status_t ucp_mem_map (ucp_context_h **context**, void **address_p, size_t **length**, unsigned **flags**, ucp_mem_h *memh_p)

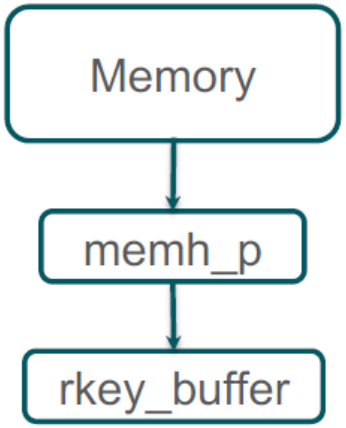


in	*context*	Application context to map (register) and allocate the memory on.
in,out	*address_p*	If the pointer to the address is not NULL, the routine maps (registers) the memory segment. if the pointer is NULL, the library allocates mapped (registered) memory segment and returns its address in this argument.
in	*length*	Length (in bytes) to allocate or map (register).
in	*flags*	Allocation flags (currently reserved - set to 0).
out	*memh_p*	UCP handle for the allocated segment.

- This routine maps or/and allocates a user-specified **memory segment** with UCP application context and the network resources associated with it.
- Related routines: ucp_mem_unmap

UCP Memory Management

- ucs_status_t ucp_rkey_pack (ucp_context_h **context**, ucp_mem_h **memh**, void **)**rkey_buffer_p**, size_t *__size_p__)



in	*context*	Application context which was used to allocate/map the memory.
in	*memh*	Handle to memory region.
out	*rkey_buffer_p*	Memory buffer allocated by the library. The buffer contains packed RKEY.
out	*size_p*	Size (in bytes) of the packed RKEY.

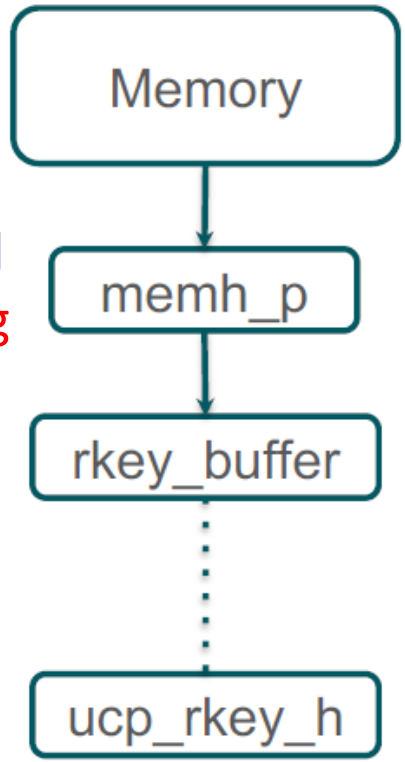
  - This routine allocates memory buffer and packs into the buffer a remote access key (RKEY) object. RKEY is an opaque object that provides the information that is necessary for remote memory access. This routine packs the RKEY object in a portable format such that the object can be unpacked on any platform supported by the UCP library.
  - Related routines: ucp_rkey_buffer_release

UCP Memory Management

- ucs_status_t ucp_ep_rkey_unpack (ucp_ep_h **ep**, void ∗**rkey_buffer**, ucp_rkey_h ∗**rkey_p**)

in	*ep*	Endpoint to access using the remote key.
in	*rkey_buffer*	Packed rkey.
out	*rkey_p*	Remote key handle.

  - This routine unpacks the remote key (RKEY) object into the local memory such that it can be accesses and used by UCP routines. The RKEY object must be packed using the ucp_rkey_pack() routine. Application code should not make any alterations to the content of the RKEY buffer.
  - Related routines: ucp_rkey_destroy



Atomic operations

- ucs_status_ptr_t ucp_atomic_op_nbx(ucp_ep_h **ep**, ucp_atomic_op_t **opcode**, const void ***buffer**, size_t **count**, uint64_t **remote_addr**, ucp_rkey_h **rkey**, const ucp_request_param_t ***param**)

in	*ep*	UCP endpoint.
in	*opcode*	One of ucp_atomic_op_t.
in	*buffer*	Address of operand for the atomic operation. See 6.142 for exact usage by different atomic operations.
in	*count*	Number of elements in *buffer* and *result*. The size of each element is specified by ucp_request_param_t::datatype
in	*remote_addr*	Remote address to operate on.
in	*rkey*	Remote key handle for the remote memory address.
in	*param*	Operation parameters, see ucp_request_param_t.

Put operation

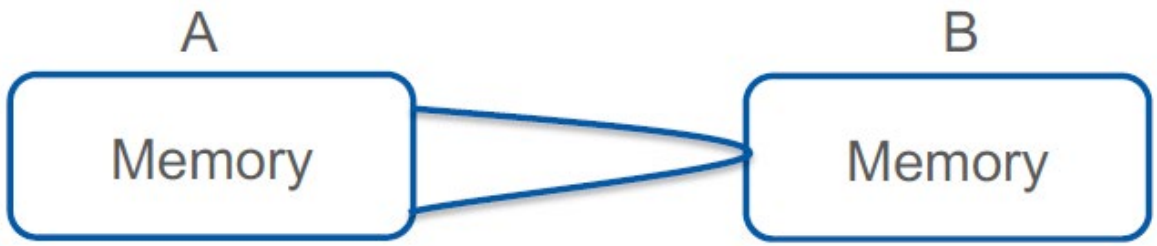
- ucs_status_ptr_t ucp_put_nbx (ucp_ep_h **ep**, const void * **buffer**, size_t **count**, uint64_t **remote_addr**, ucp_rkey_h **rkey**, const ucp_request_param_t * **param**)

in	*ep*	Remote endpoint handle.
in	*buffer*	Pointer to the local source address.
in	*count*	Number of elements of type ucp_request_param_t::datatype to put. If ucp_request_param_t::datatype is not specified, the type defaults to ucp_dt_make_contig(1), which corresponds to byte elements.
in	*remote_addr*	Pointer to the destination remote memory address to write to.
in	*rkey*	Remote memory key associated with the remote memory address.
in	*param*	Operation parameters, see ucp_request_param_t

Get operation

- ucs_status_ptr_t ucp_get_nbx (ucp_ep_h **ep**,
  void * **buffer**, size_t **count**, uint64_t **remote_addr**,
  ucp_rkey_h **rkey**, const ucp_request_param_t * **param**)

in	*ep*	Remote endpoint handle.
in	*buffer*	Pointer to the local destination address.
in	*count*	Number of elements of type ucp_request_param_t::datatype to put. If ucp_request_param_t::datatype is not specified, the type defaults to ucp_dt_make_contig(1), which corresponds to byte elements.
in	*remote_addr*	Pointer to the source remote memory address to read from.
in	*rkey*	Remote memory key associated with the remote memory address.
in	*param*	Operation parameters, see ucp_request_param_t.



Support for Accelerators (e.g. GPUs)

- UCX was designed from the beginning with accelerators in mind
- Accelerators are structured as transports
  - The accelerator transport abstractions is responsible to transferring data between the host and accelerator memory
  - Protocols are agnostic to the types of accelerators
  - **No need to change higher level logic** in order to add new accelerators
- Communicating accelerator memory **works the same as passing host memory**
  - Passing a pointer to the accelerator memory in UCP API works
  - The user can pass runtime flags to override internal memory detection
- The programming model doesn't need to deal with accelerator logic/code

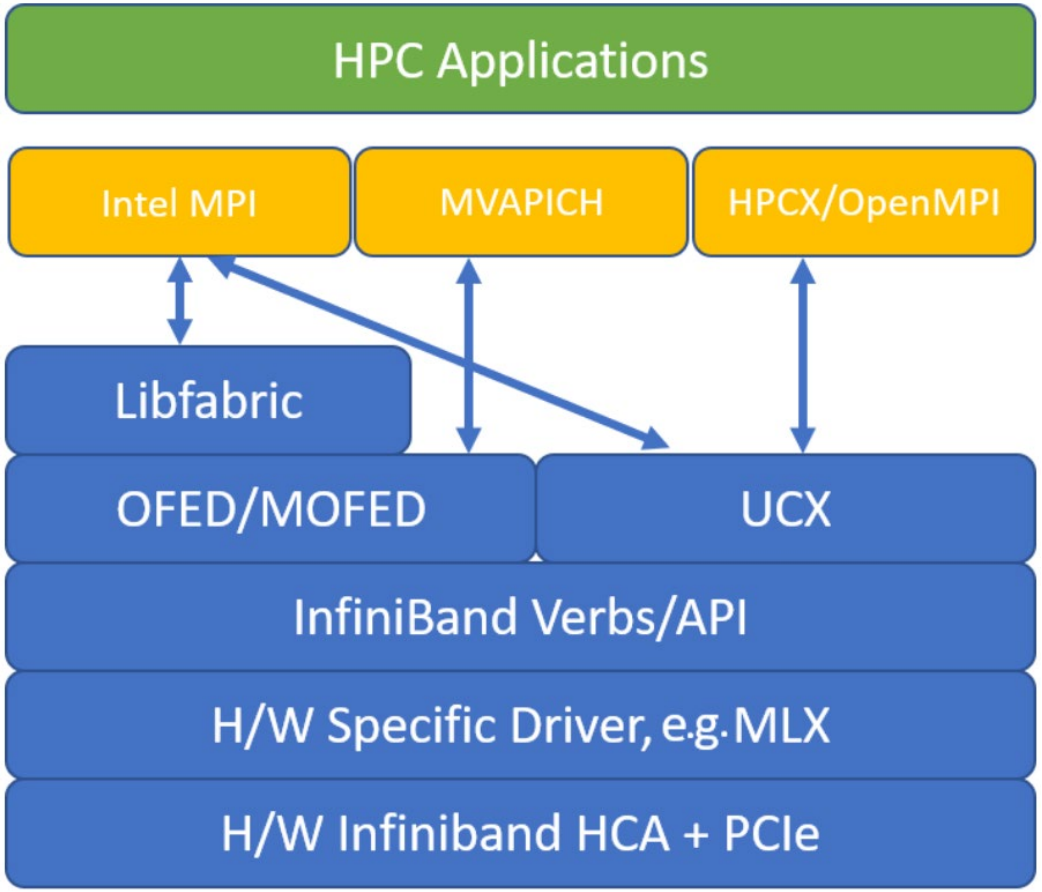
Hello world with GPU memory

- **ucx-tutorial-hot-interconnects**
  /examples/02_ucx_gpu_example/**ucp_hello_world.c**

```
//msg = malloc(info_tag.length);
CUDA_FUNC(cudaMallocManaged((void **)&msg, info_tag.length, cudaMemAttachGlobal));
CHKERR_JUMP(msg == NULL, "allocate memory\n", err_ep);

request = ucp_tag_msg_recv_nb(ucp_worker, msg, info_tag.length,
                              ucp_dt_make_contig(1), msg_tag,
                              recv_handler);
```

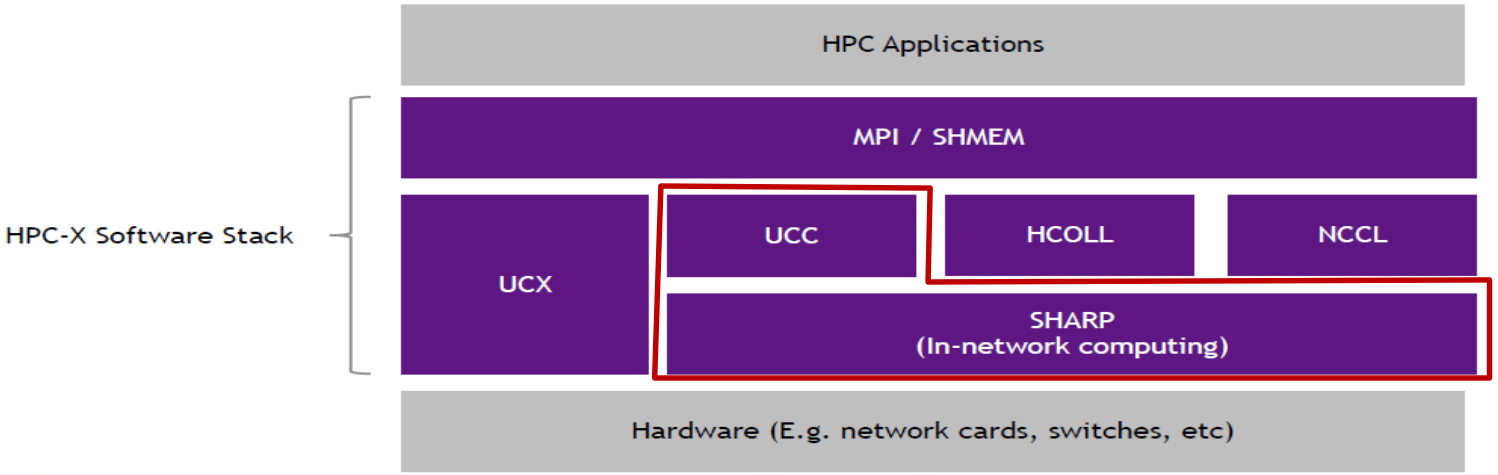
Outline

- RDMA Technology
- Verbs
- UCX
- HPC-X

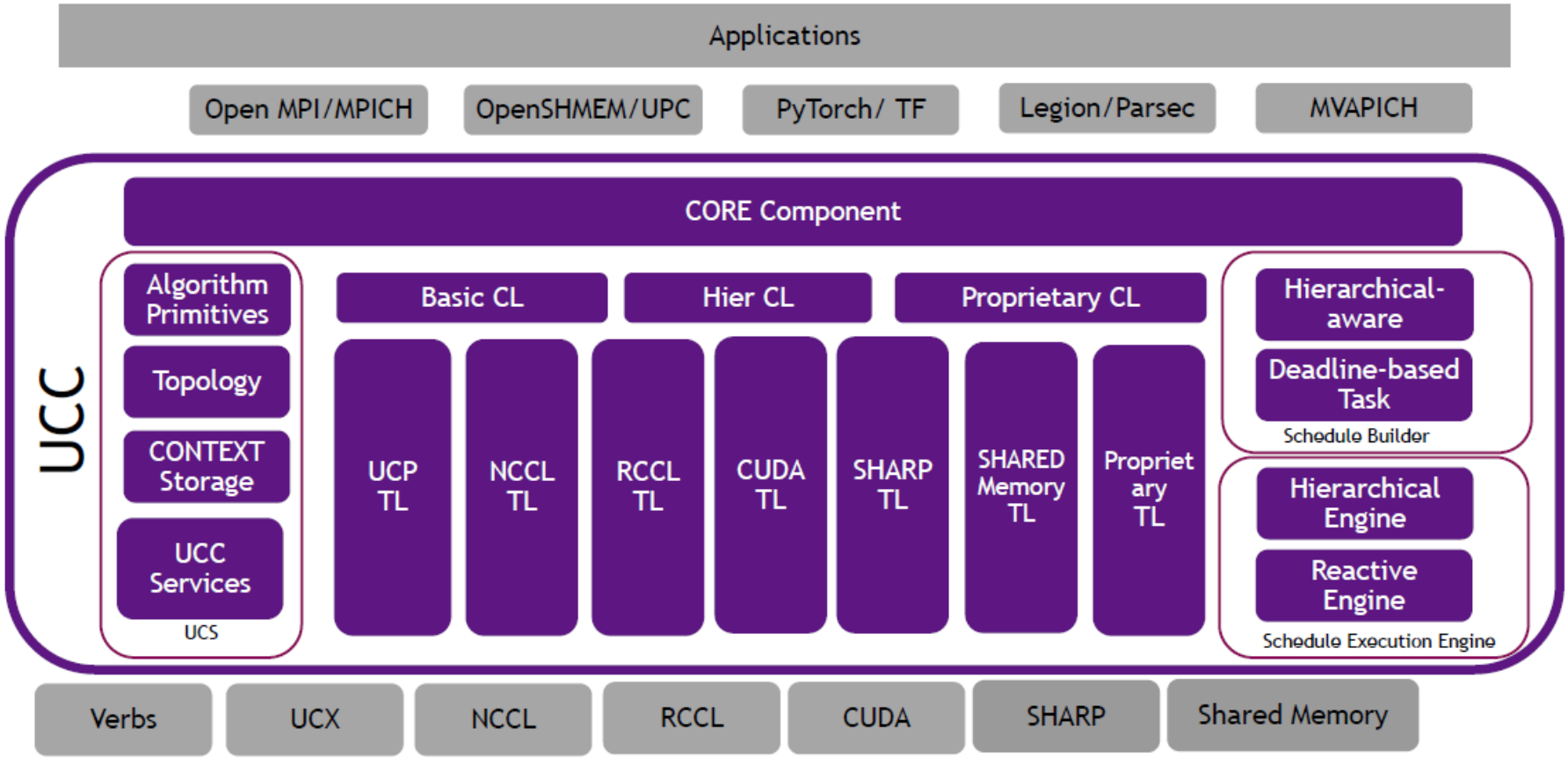


HPC-X Software Stack

- HPC-X is the Mellanox solution for HPC communication libraries to improve the performance and scalability of your HPC cluster
  - MPI /SHMEM implementation
  - UCX –Unified Communication X
  - UCC –Unified Collective Communication
  - HCOLL –Hierarchical Collectives (Note: UCC will replace this in the future)
  - NCCL/SHARP hardware collectives
  - In-network computing infrastructure with SHARP



UNIFIED COLLECTIVE COMMUNICATION (UCC) Architecture



UCC: Unified Collective Communication Library

- Proposal : Collective communication operations API that is flexible, complete, and feature-rich for current and emerging programming models and runtimes.

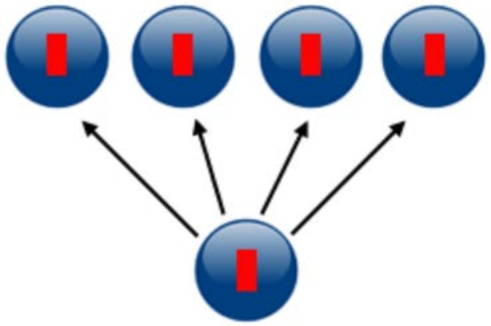
High-level Features

- Blocking and Nonblocking collective operations
- Hierarchical collectives are a first-class citizen
  - Well-established design for achieving performance and scalability
- Hardware collectives are a first-class citizen
  - Well-established model and have demonstrated to achieve performance and scalability
- Flexible resource allocation model
  - Support for lazy, local and global resource allocation decisions
- Support for relaxed ordering model
  - For AI/ML application domains
- Flexible synchronous model
  - Highly synchronized collective operations (MPI model)
  - Less synchronized collective operations (OpenSHMEM and PGAS model)
- Repetitive collective operations (init once and invoke multiple times)
  - AI/ML collective applications, persistent collectives
- Point-to-point operations in the context of group
- Global memory management
  - OpenSHMEM PGAS, MPI, and CORAL2 (RFP)

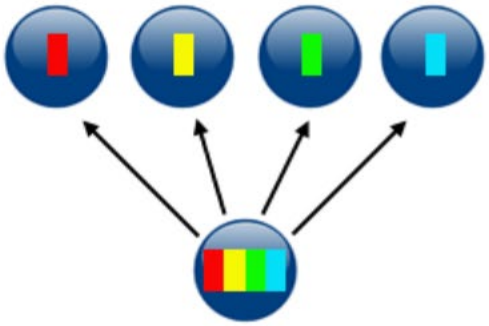
UCC : A convergence of multiple implementations

- UCC API has emerged as this convergence
- Now working towards converged implementation
- Particularly XUCG and XCCL
- XCCL
  - Driven by NVIDIA/Mellanox and hierarchical based design
  - https://github.com/openucx/xccl
- XUCG
  - Driven by Huawei and reactive based design
  - https://github.com/openucx/xucg
- HCOLL, PAMI and other collectives design and implementation

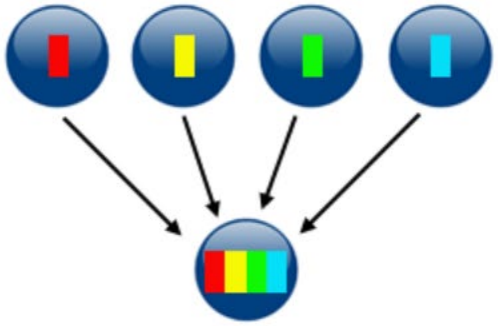
Collective Communication Routines



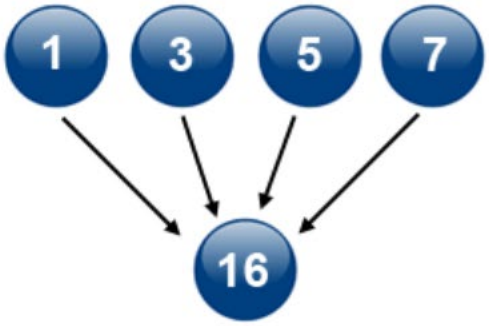
broadcast



scatter



gather

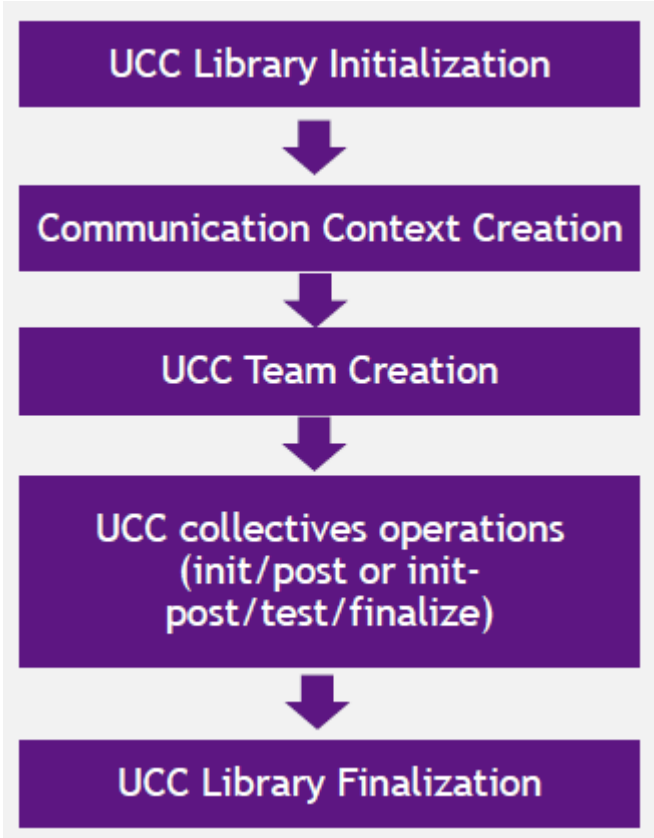


reduction

Types of Collective Operations:

UCC Code Flow

- Library Initialization
- Communication Context
- Team
  - An object to encapsulate the resources required for group operations
- UCC collective operation
- Library Finalization



Reference:
[1] https://mug.mvapich.cse.ohio-state.edu/static/media/mug/presentations/21/gorentla_bureddy_ucc_sharp_mug21.pdf\
[2] https://openucx.github.io/ucc/api/v1.0/html/md_docs_doxygen_library.html
[3] https://openucx.github.io/ucc/
*[4] https://ucfconsortium.org/wp-content/uploads/2020/02/Manjunath_GV_gorentla_ucx_collectives.pdf
*recommended

Open MPI with UCX & UCC

- https://github.com/openucx/ucc
  - **Compile UCX**
  - **Compile UCC**
  - **Compile Open MPI**

```
$ git clone https://github.com/open-mpi/ompi
$ cd ompi
$ ./autogen.pl; ./configure --prefix=<ompi-install-path> --with-ucx=<ucx-install-path>
--with-ucc=<ucc-install-path>; make -j install
```

  - **Run MPI programs**

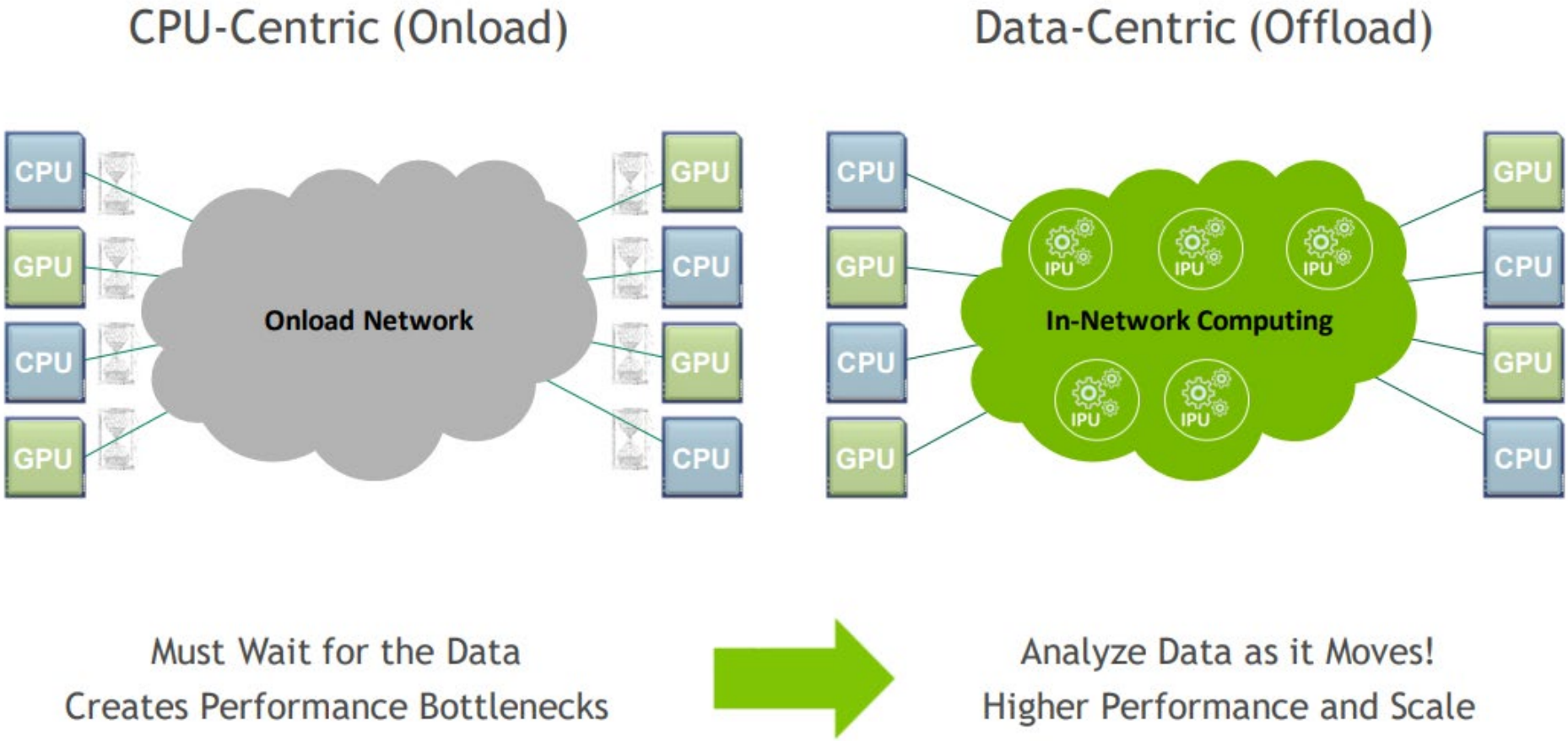
```
$ mpirun -np 2 --mca coll_ucc_enable 1 --mca coll_ucc_priority 100 ./my_mpi_app
```

  - **SUPPORTED Transports**
    - UCX/UCP: InfiniBand, ROCE, Cray Gemini and Aries, Shared Memory
    - SHARP、CUDA、NCCL、RCCL

https://mug.mvapich.cse.ohio-state.edu/static/media/mug/presentations/20/bureddy-mug-20.pdf

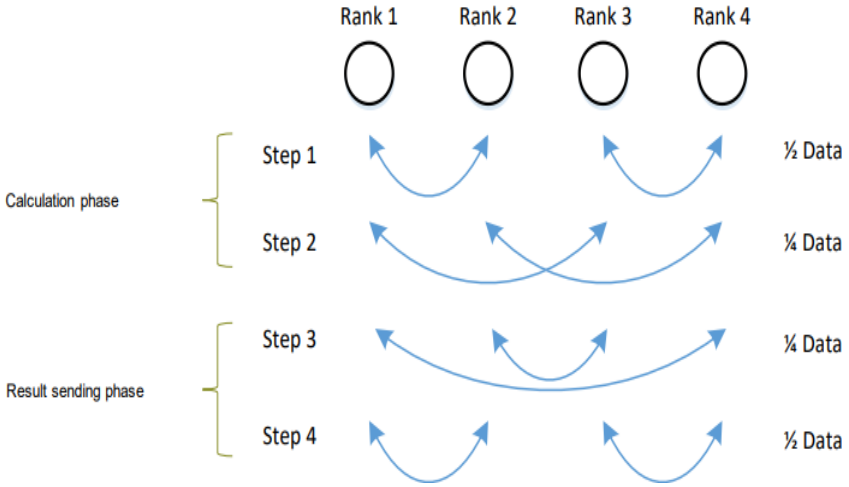
SHARP: A IN-NETWORK SCALABLE STREAMING HIERARCHICAL AGGREGATION AND REDUCTION PROTOCOL

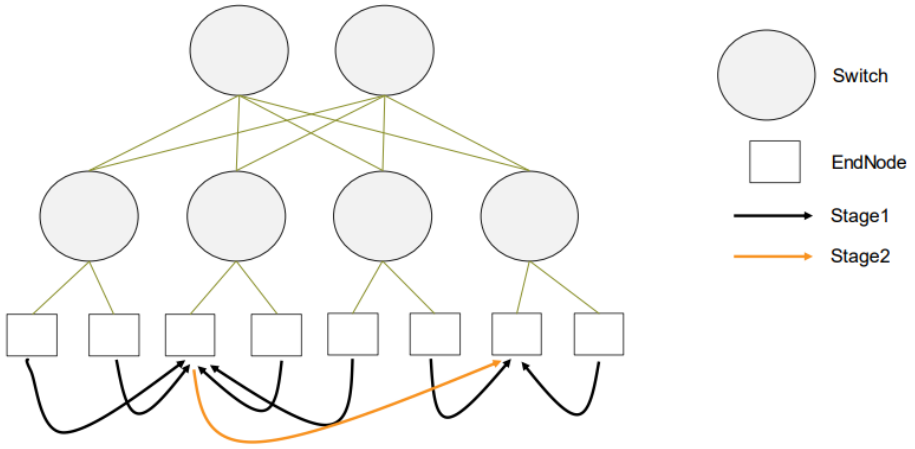
The Need For Intelligent And Faster Interconnect



Collective Operations

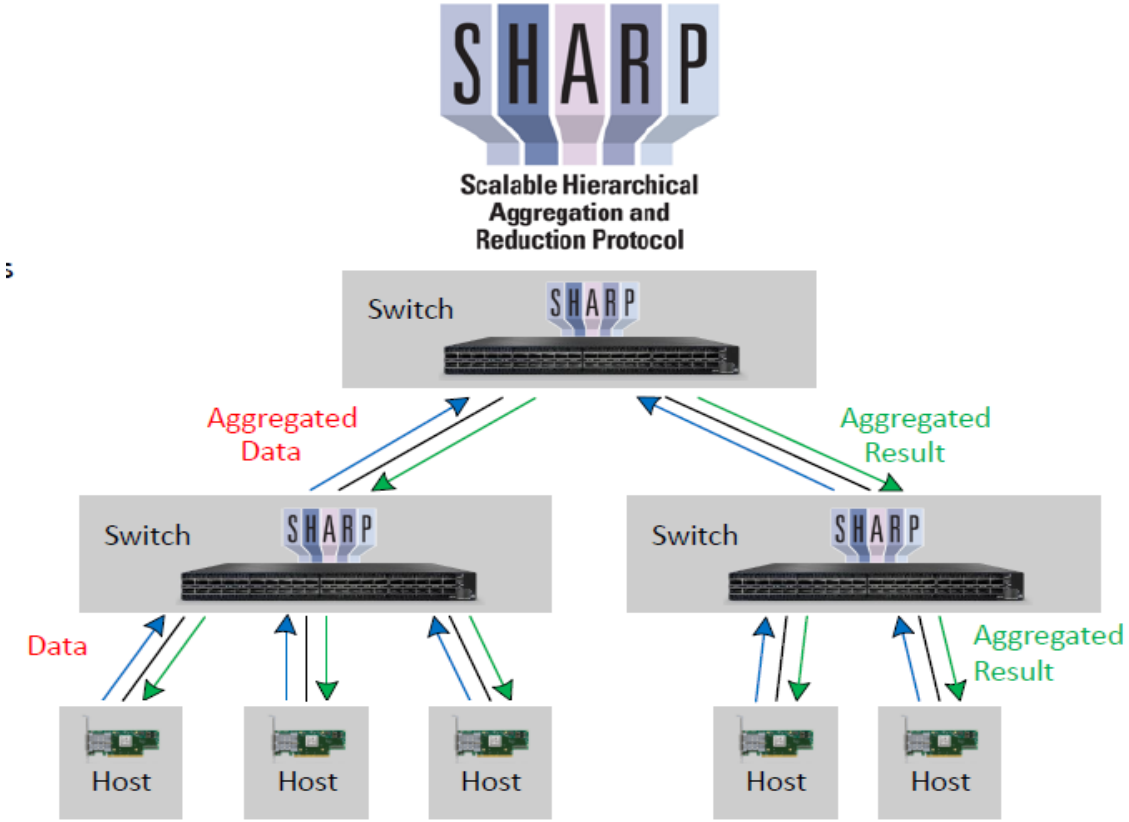
- Many2One and One2Many traffic patterns – possible network congestion
- Probably not a good solution for large data
- Large scale requires higher tree / larger radix
- Result distribution – over the tree / MC





SHARP

- SHARP is a in-network scalable streaming hierarchical aggregation and reduction protocol



Features

- In-network Tree based aggregation mechanism
- Multiple simultaneous outstanding operations
- For HPC (MPI / SHMEM) and Distributed Machine Learning applications
- Scalable High Performance Collective Offload
- Barrier, Reduce, All-Reduce, Broadcast and more
- Sum, Min, Max, Min-loc, max-loc, OR, XOR, AND
- Integer and Floating-Point, 8/16/32/64 bits

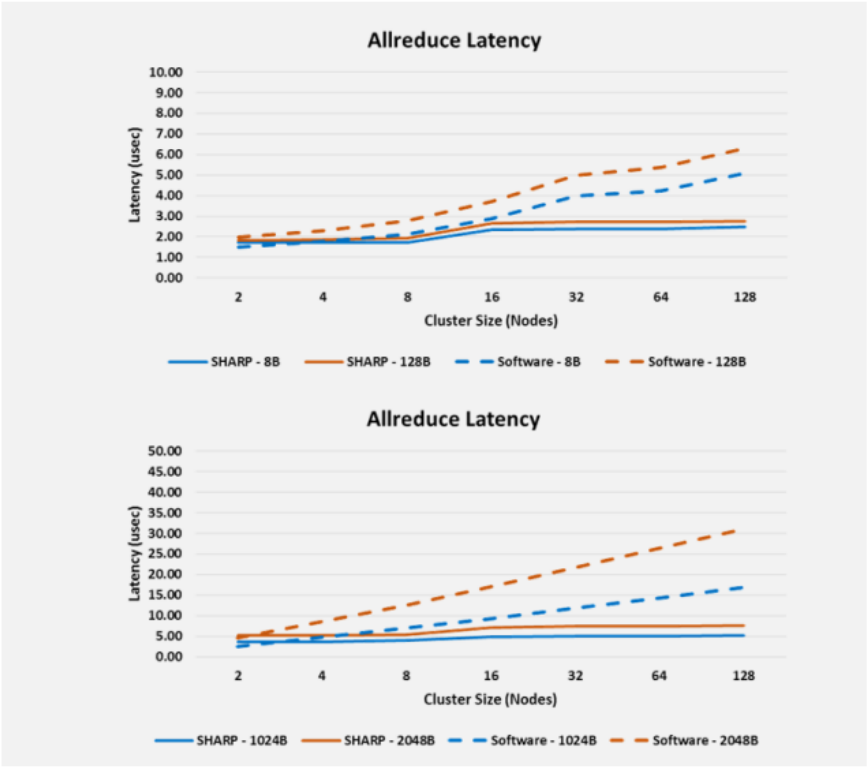
HW Support

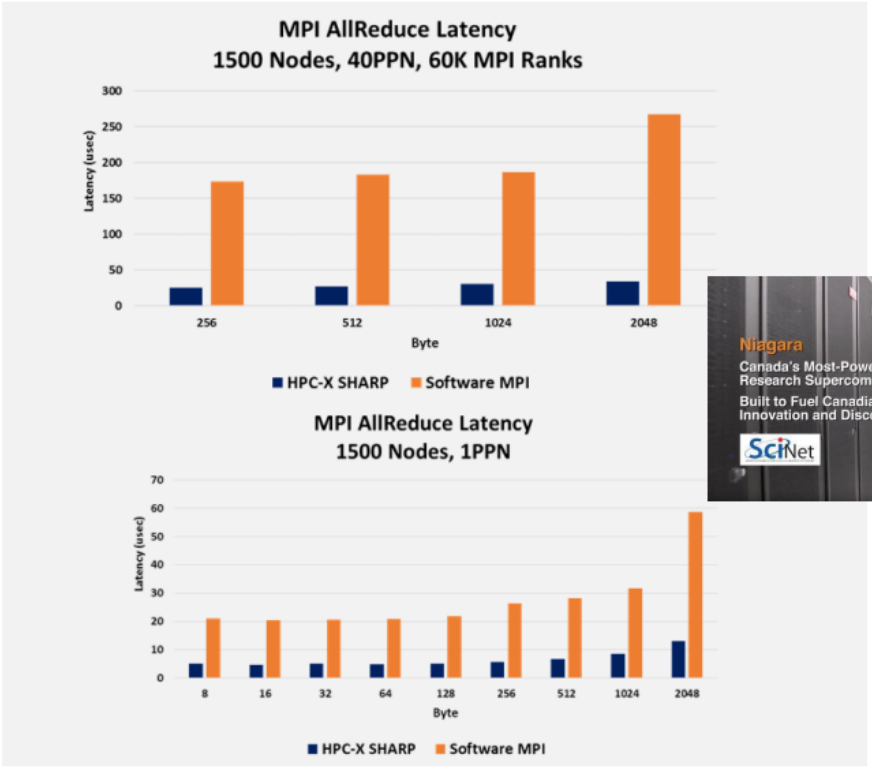
- Switch hardware-based network-level reduction supporting for the full range of message sizes
  - EDR InfiniBand Switch-IB-2 switch introduced support for short message reductions
    - Referred to as Low Latency Transmission (LLT) SHARP
    - Latency optimized, fully offloaded to the switches –asynchronous
  - HDR InfiniBand Quantum switch added support for long vector reduction
    - Referred to as Streaming Aggregation (SAT) SHARP
    - Bandwidth optimized, fully offloaded to the network hardware – asynchronous
  - NDR InfiniBand Quantum switch added support for multiple flows per switch (flow per port)

Performance

SHARP ALLREDUCE PERFORMANCE ADVANTAGES

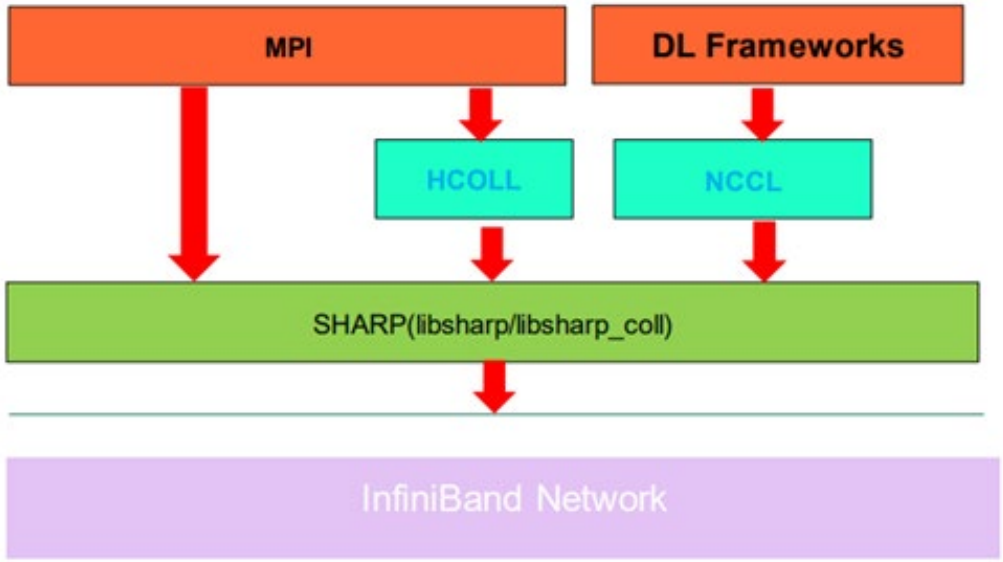
Providing Flat Latency, 7X Higher Performance





SW Architecture

- MPI
  - Open MPI/Specturm MPI, MVAPICH
  - Application programming interface
- HCOLL
  - Optimized collective library
  - Integrated with UCC
- NCCL
  - Optimized GPU collective library
- SHARP
  - Easy to use high level API



Using SHARP WITH MPI

- Integrated with multiple MPI libraries
  - MVAPICH2
    - MV2_ENABLE_SHARP
  - OMPI (HPC-X, Spectrum MPI)
    - HCOLL_ENABLE_SHARP
    - SHARP_COLL_ENABLE_SAT

Reference

- https://github.com/gt-crnch-rg/ucx-tutorial-hot-interconnects
- https://mug.mvapich.cse.ohio-state.edu/static/media/mug/presentations/21/gorentla_bureddy_ucc_sharp_mug21.pdf
- https://openucx.github.io/ucc/
- https://ucfconsortium.org/wp-content/uploads/2020/02/Manjunath_GV_gorentla_ucx_collectives.pdf
- https://mug.mvapich.cse.ohio-state.edu/static/media/mug/presentations/20/bureddy-mug-20.pdf