



Unified  
Communication  
Framework



# Welcome to the UCX Tutorial!

Gilad Shainer

HOTI 2021

# 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](http://www.openucx.org)  
SparkUCX – [www.sparkucx.org](http://www.sparkucx.org)  
OpenSNAPI – Smart NIC Project  
UCC – Collective Library  
UCD – Advanced Datatype Engine  
HPCA Benchmark – Benchmarking Effort

## Board members

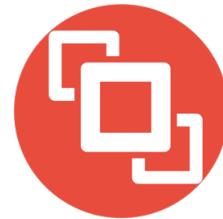
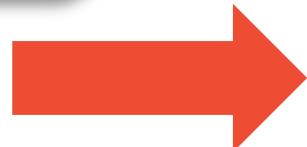
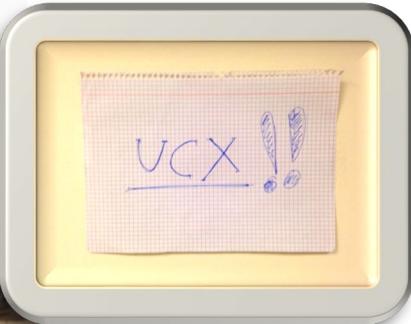
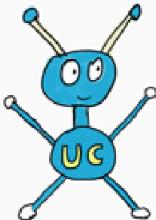
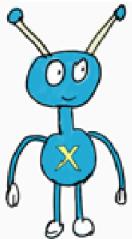
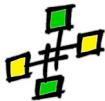
**Jeff Kuehn**, UCF Chairman (Los Alamos National Laboratory)  
**Gilad Shainer**, UCF President (NVIDIA)  
**Pavel Shamis**, UCF Treasurer (Arm)  
**Brad Benton**, Board Member (AMD)  
**Yanfei Guo**, Board Member (Argonne National Laboratory)  
**Perry Schmidt**, Board Member (IBM)  
**Dhabaleswar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



Join

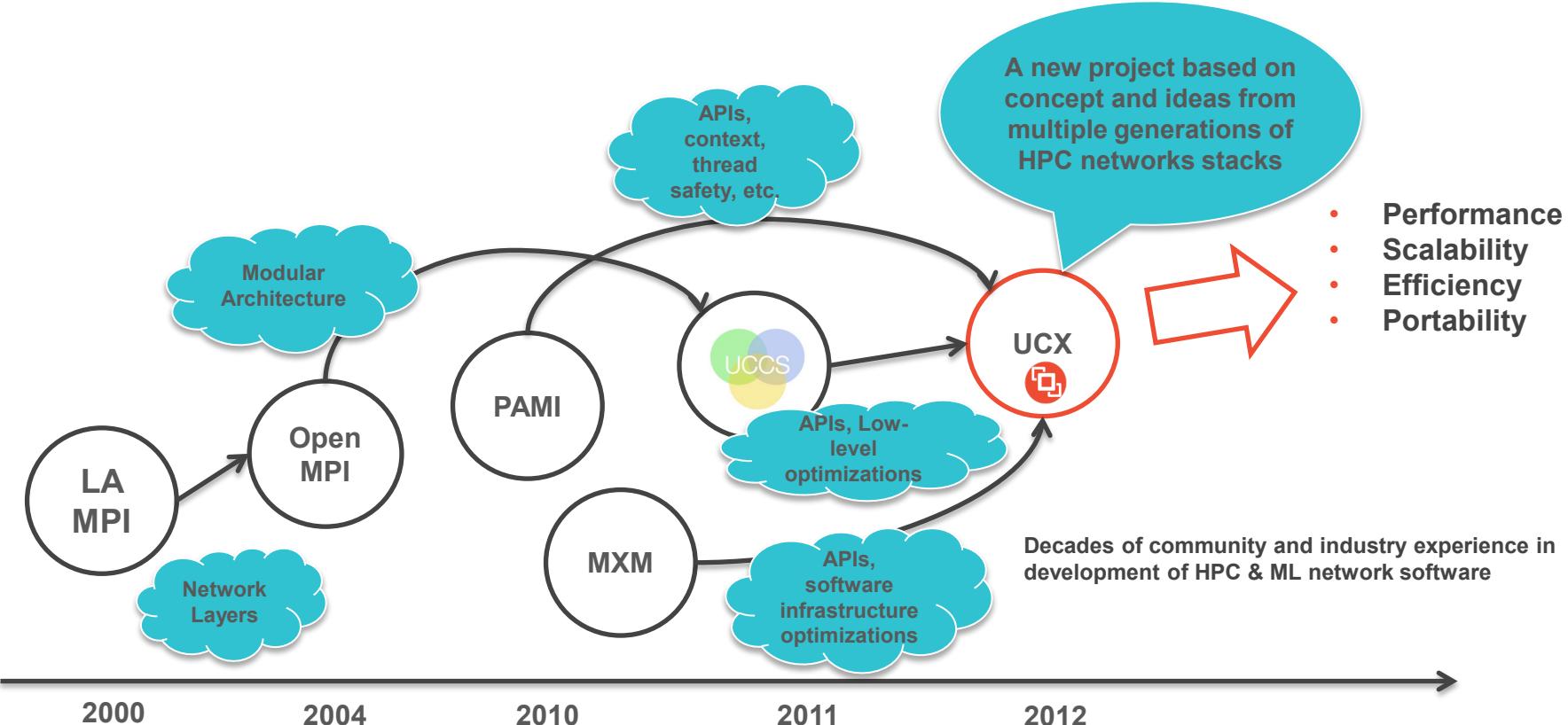
<https://www.ucfconsortium.org> or [info@ucfconsortium.org](mailto:info@ucfconsortium.org)

# Unified Communication X (UCX)

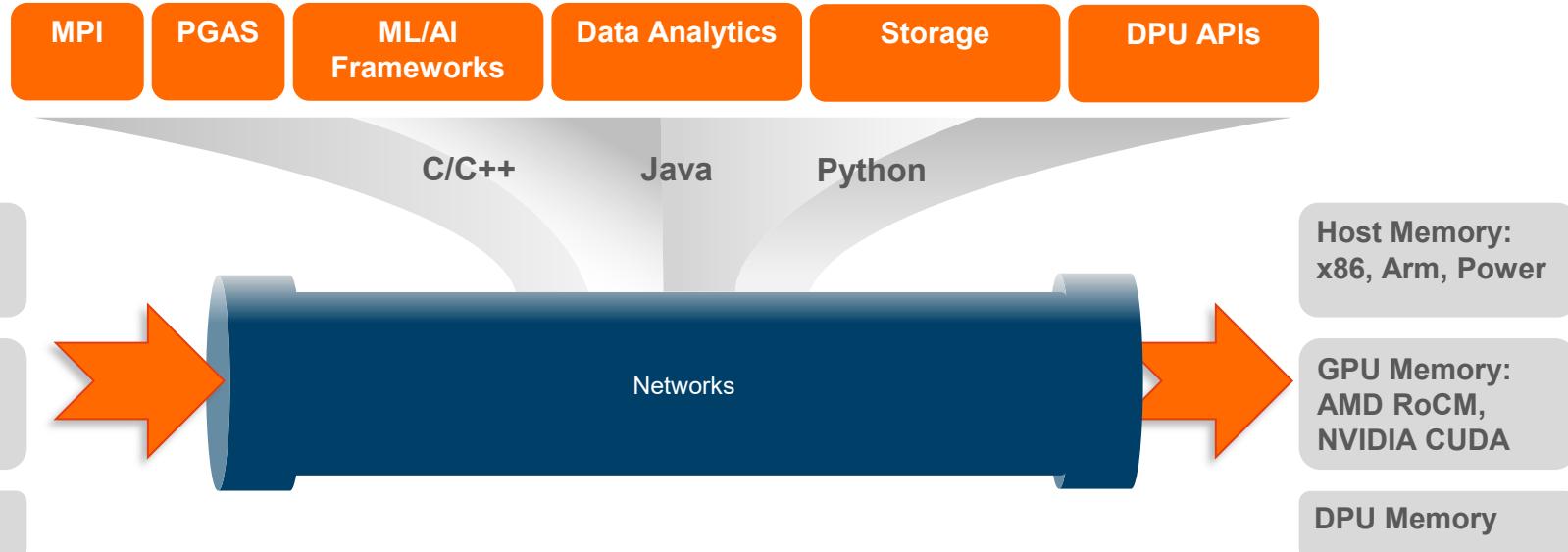


<https://www.hpcwire.com/2018/09/17/ucf-ucx-and-a-car-ride-on-the-road-to-exascale/>

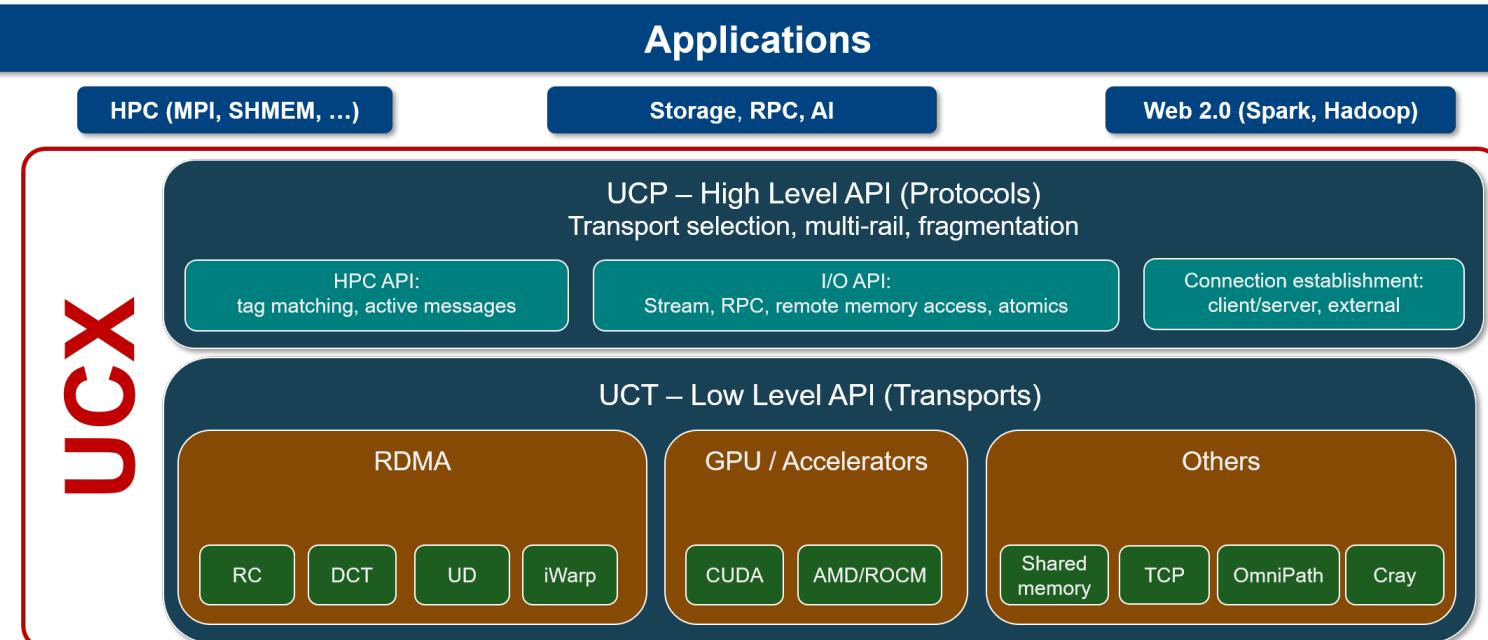
# UCX – History



# Why UCX ?



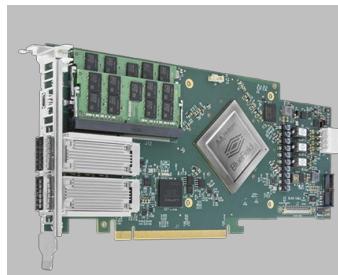
# UCX High-level Overview



## Hardware

# UCX Performance-portability

- Support for x86\_64, Power 8/9, Arm v8
- U-arch tuned code for Xeon, AMD Rome/Naples, Arm v8 (Cortex-A/N1/ThunderX2/Huawei, Fujitsu A64FX)
- First class support for AMD and NVIDIA GPUs
- Runs on Servers, Raspberry PI like platforms, SmartNIC, Nvidia Jetson platforms, etc.



BlueField DPU



NVIDIA Jetson



Arm ThunderX2



Odroid C2



N1 SDP

## UCX Users (Examples)

- MPI implementations: MPICH, Open MPI, NVIDIA HPC-X MPI, Huawei MPI
- PGAS: GasNET
- OpenSHMEM: OSSS SHMEM, Sandia SHMEM, Open MPI SHMEM
- Charm++
- RAPIDS / DASK
- NVIDIA's NCCL

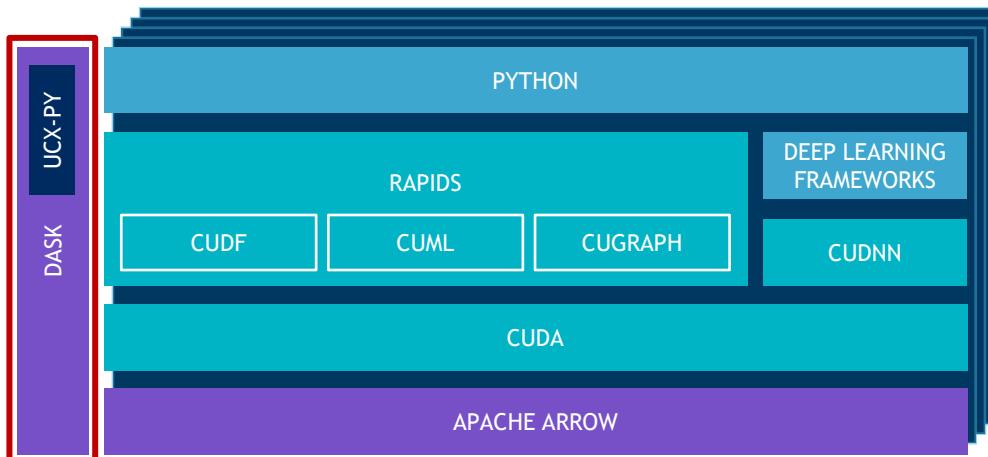


Diagram courtesy of NVIDIA

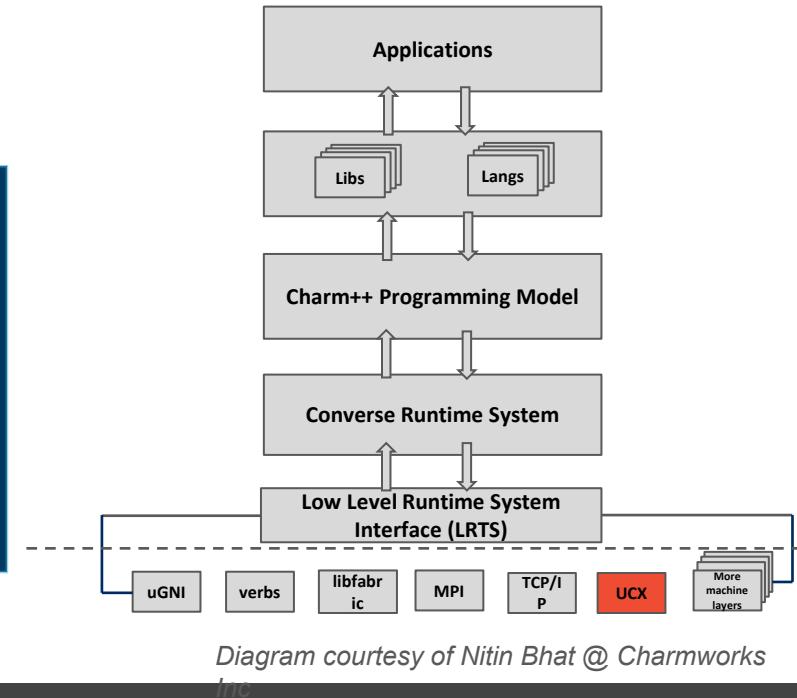


Diagram courtesy of Nitin Bhat @ Charmworks Inc

# Over 100,000 tests per commit 220,000 CPU hours per release

 **Review required**  
At least 1 approving review is required by reviewers with write access. [Learn more.](#)

 **Some checks haven't completed yet**  
22 in progress, 1 pending, and 6 successful checks

 **UCX PR (Tests althca on worker 0)** In progress — This check has started... [Details](#)

 **UCX PR (Tests althca on worker 1)** In progress — This check has started... [Details](#)

 **UCX PR (Tests althca on worker 2)** In progress — This check has started... [Details](#)

 **UCX PR (Tests althca on worker 3)** In progress — This check has started... [Details](#)

 **UCX PR (Tests gpu on worker 0)** In progress — This check has started... [Details](#)

 **Merging is blocked**  
Merging can be performed automatically with 1 approving review.

 **Codestyle**  
1 job completed 11s

 **Build**  
2/3 completed 15m 8s

 **Static checks** 15m 7s

 **Build for centos7** 2m 55s

 **Build tarball and sour...** 6m ...

 Cancel

 **Tests**  
2/22 completed 15m 10s

 **althca on worker 0** 15m 10s

 **althca on worker 1** 15m 10s

 **althca on worker 2** 15m 9s

 **althca on worker 3** 15m 8s

 **legacy on worker 0** 15m 8s

 **legacy on worker 1** 15m 8s

 **legacy on worker 2** 15m 7s

 **legacy on worker 3** 15m 7s

# 2019 R&D 100 Award



# UCX – Useful links

- **Code**
  - <https://github.com/openucx/>
- **Website**
  - [www.openucx.com](http://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/>





Unified  
Communication  
Framework



Thank You

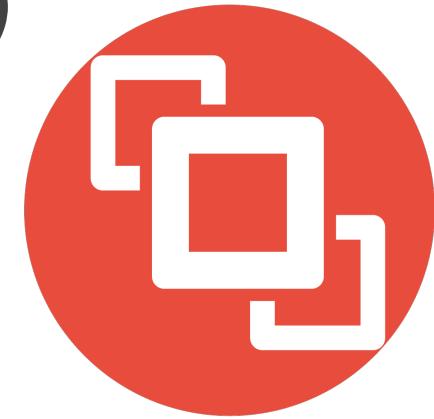
# Unified Communication-X (UCX) Tutorial

Jeff Young - Georgia Tech

Gilad Shainer - UCF / NVIDIA

Yossi Itigin, Oscar Hernandez - NVIDIA

Mathew Baker - Oak Ridge National Laboratory



Slide contributors: Pavel Shamis, Alina Sklarevich, Alex Margolis, Swen Boehm, and Oded Paz

- Slides and examples

- For this tutorial - <https://github.com/gt-crncr-rg/ucx-tutorial-hoti-21>
- UCX ReadTheDocs - <https://openucx.readthedocs.io/>

- Slack Channel

- Join the HOTI Workspace using RDMobile
- Click on the tutorial RDMobile channel link to join:
  - #tutorial-unified-communication-x-for-performance-portable-network-acceleration

- Q&A

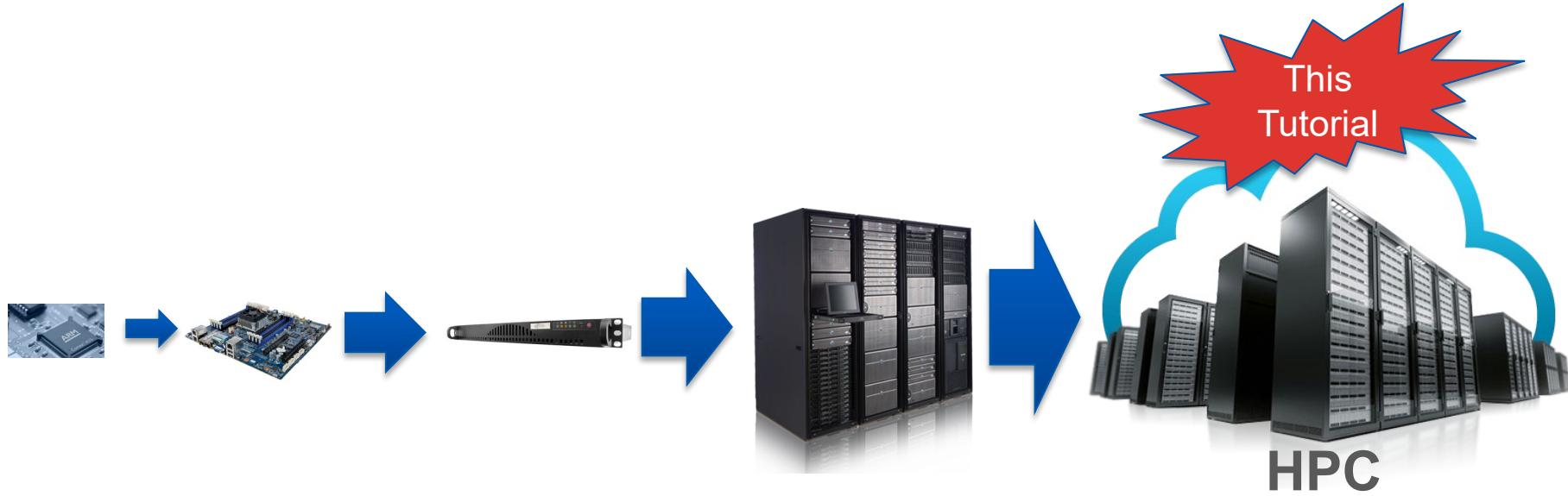
- UCX mailing list - <https://elist.ornl.gov/mailman/listinfo/ucx-group>

# Agenda

- Background: UCF Foundation
  - Overview of existing technologies
  - Unified Communication X Framework
- Introduction to UCX
  - Unified Communication Protocols (UCP)
  - UCX basic examples
- Advanced UCX examples
  - Implementing programming models with UCX

Time	Topic	Presenters	Slides	Notes
9:00 - 9:20	UCX Tutorial and Ecosystem Introduction	Gilad		
9:20 - 10:00	UCX Basics - networking overview, worker and endpoint creation	Jeff		
10:00 - 10:10	<b>BREAK</b>			
10:10 - 10:20	Hello World part 1	Matt, Jeff		
10:20 - 10:35	UCX memory management	Jeff, Oscar		
10:35 - 10:45	Hello World Demo, part 2	Matt, Jeff		
10:45 - 11:00	GPU discussion/demo	Oscar, Matt		
11:00 - 11:10	<b>BREAK</b>			
11:10 - 11:50	UCX Advanced Topics - bindings, OpenMPI and OpenSHMEM integration	Matt, Yossi, Oscar		

- Interconnects are everywhere: System-on-Chip, chip-to-chip, rack, top-of-the-rack, wide area networks



## TCP/IP

Application

Transport

Network

Data Link

Physical

## InfiniBand Architecture

Upper Layer

Transport Layer

Network Layer

Link Layer

Physical Layer

Protocols: MPI, NCCL, IP over IB, RDMA

Hardware-based transports  
in Host Channel Adaptor (HCA)

Protocols to route packets across subnets using  
routers and global identifiers (GID)

Local IDs (LID) /subnets, switches forwarding  
tables (LID/port), flow control, loss less fabric

How bits are placed in the HW, signaling  
protocols cables (copper/fiber), etc.



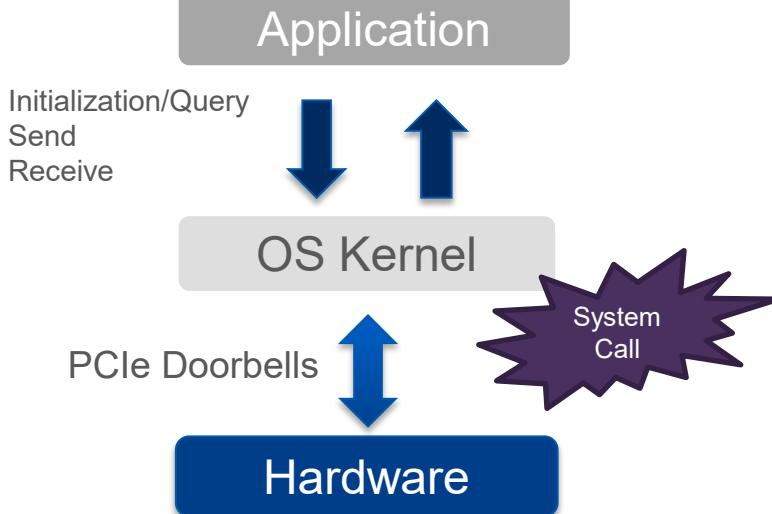
Accelerated in Hardware

# Key Concepts

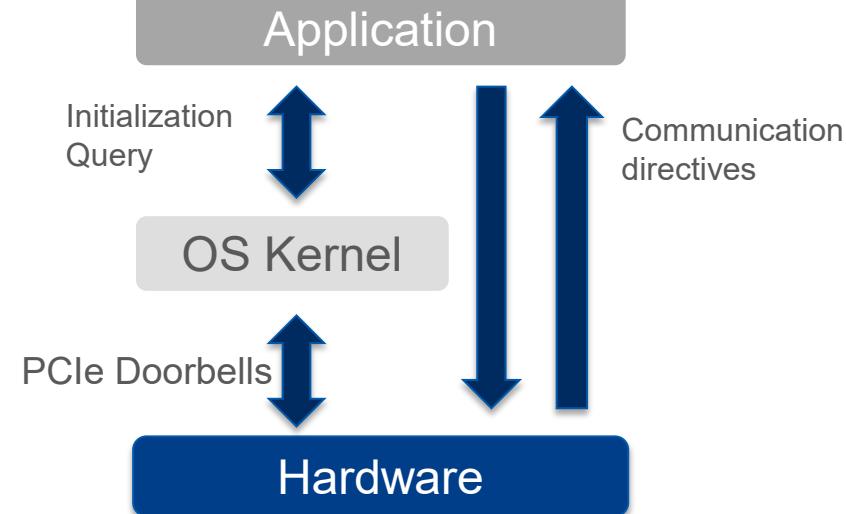
- Communication Model
  - Two-sided communication model
    - Send and receive model
  - One-sided communication model
    - Remote memory access and atomics
- Rendezvous
  - Two sides exchange meta-data and use one-sided operations for bulk transfer
- Bypassing the OS
- Transport offload

# Bypassing the OS

## No OS-bypass

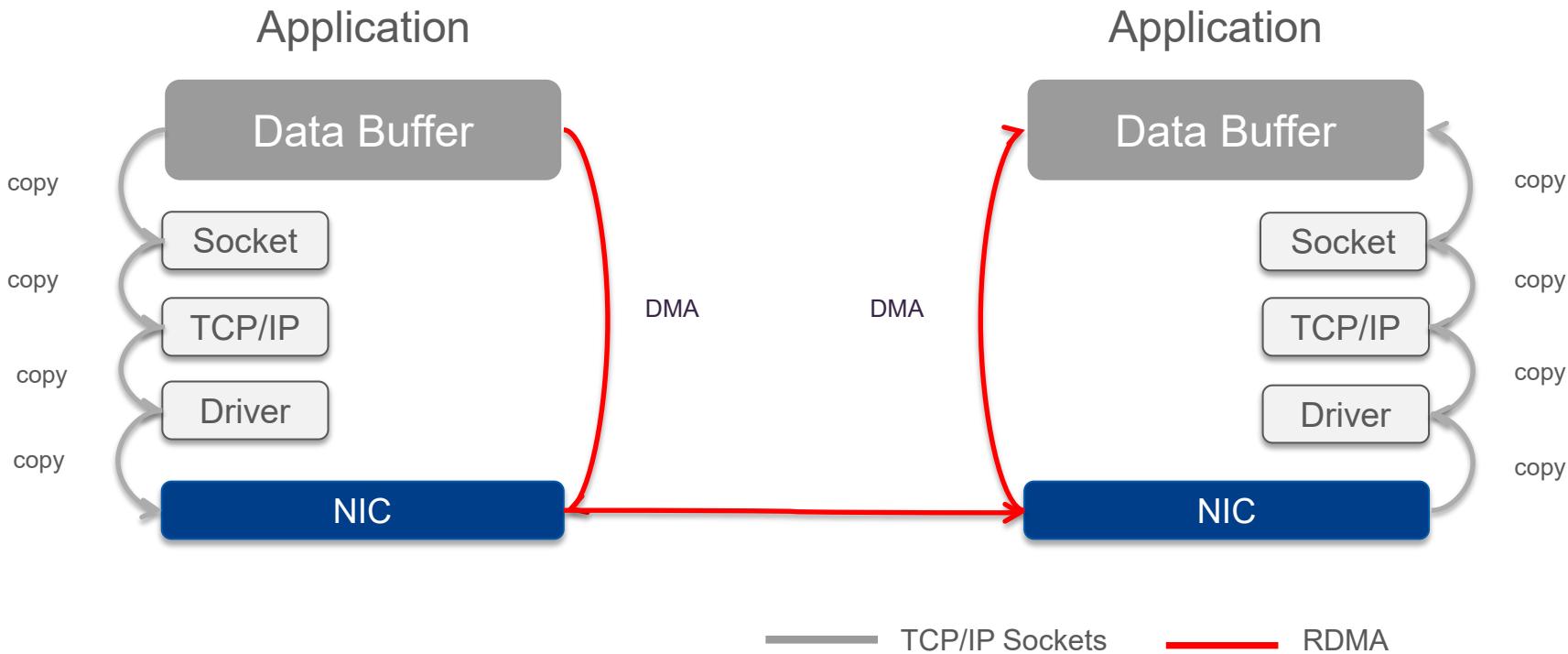


## With OS-bypass



- "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
- Buffer Copy
  - Temporary buffers on sender and receivers
  - Additional copies to/from buffers
  - Good for small messages that fit in a buffer

# RDMA – “Zero” Copy



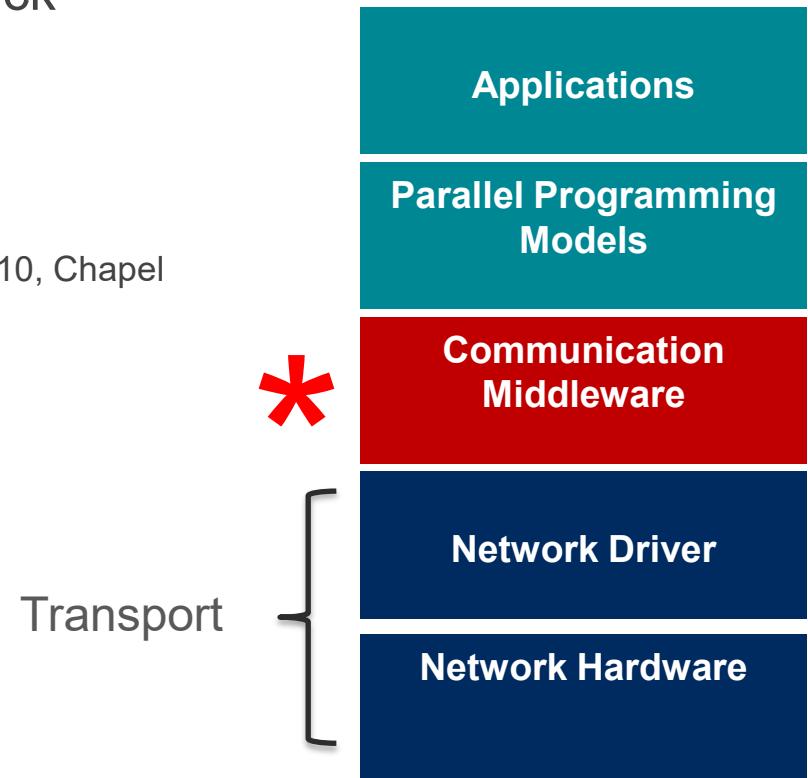
- RDMA Read and Write
- Send / Receive
  - Send / Receive with TAG matching
- Atomic Operations on Remote Memory
  - SWAP
  - CSWAP
  - ADD
  - XOR
- Group Communication directives
  - Reduce, Allreduce, Scatter, Gather, AlltoAll

**Socket API:**

*send() and recv(), or  
write() and read(), or  
sendto() and recvfrom()*

# How to improve the HPC Software Stack

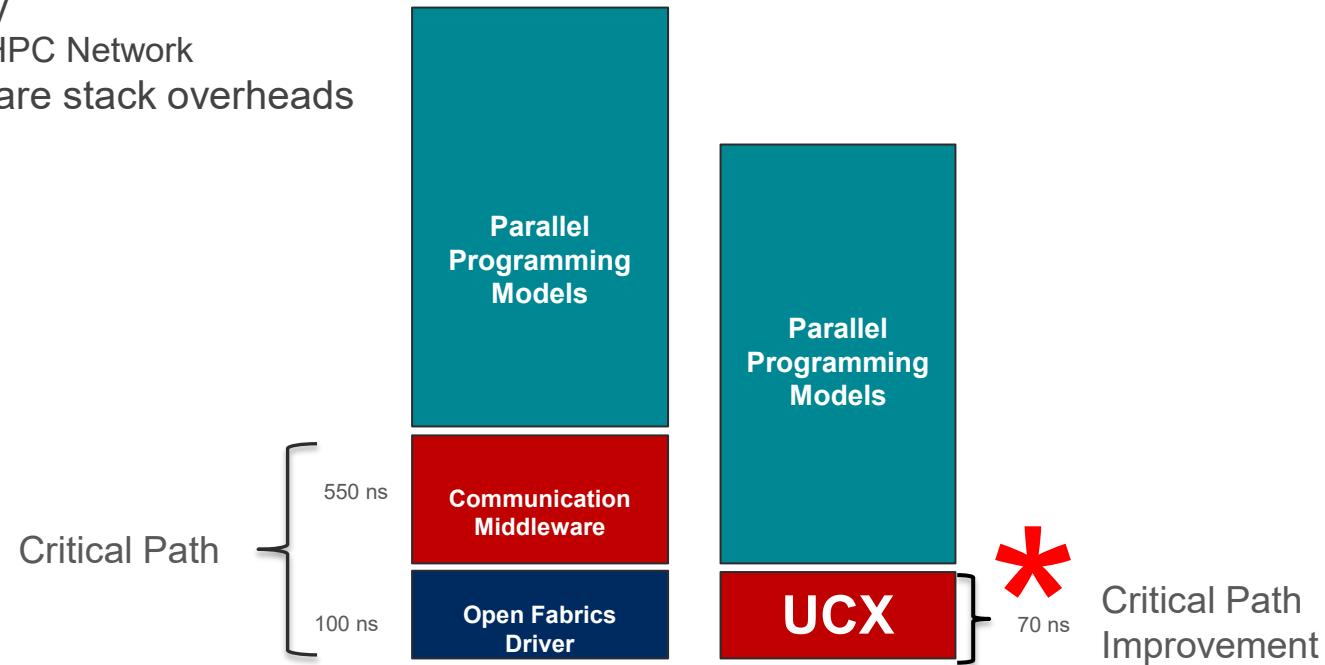
- Applications
  - GROMACS, NAMD, Fluent, Lsdyna, etc.
- Programming models
  - MPI, UPC, OpenSHMEM/SHMEM, Co-array Fortran, X10, Chapel
- Middleware
  - GasNET, MXM, ARMCI, etc.
  - Part of programming model implementation
  - Sometimes “merged” with driver
- Driver
  - OFA Verbs, Cray uGNI, etc.
- Hardware
  - InfiniBand, Cray Slingshot, etc.



# How do we improve the network stack?

- Network latency is a key
  - Sub-micro is typical for HPC Network
- Need to eliminate software stack overheads

Example: RMA PUT operation



# Network Programming Interfaces (beyond sockets)

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

## Pros

- Production Quality
- Optimized for Performance
- Support and maintenance

## Cons

- Often “vendor” locked
- Optimized for a particular technology
- Co-design lags behind

## Pros

- Community (a.k.a. user) driven
- Easy to modify and extend
- Good for research

## Cons

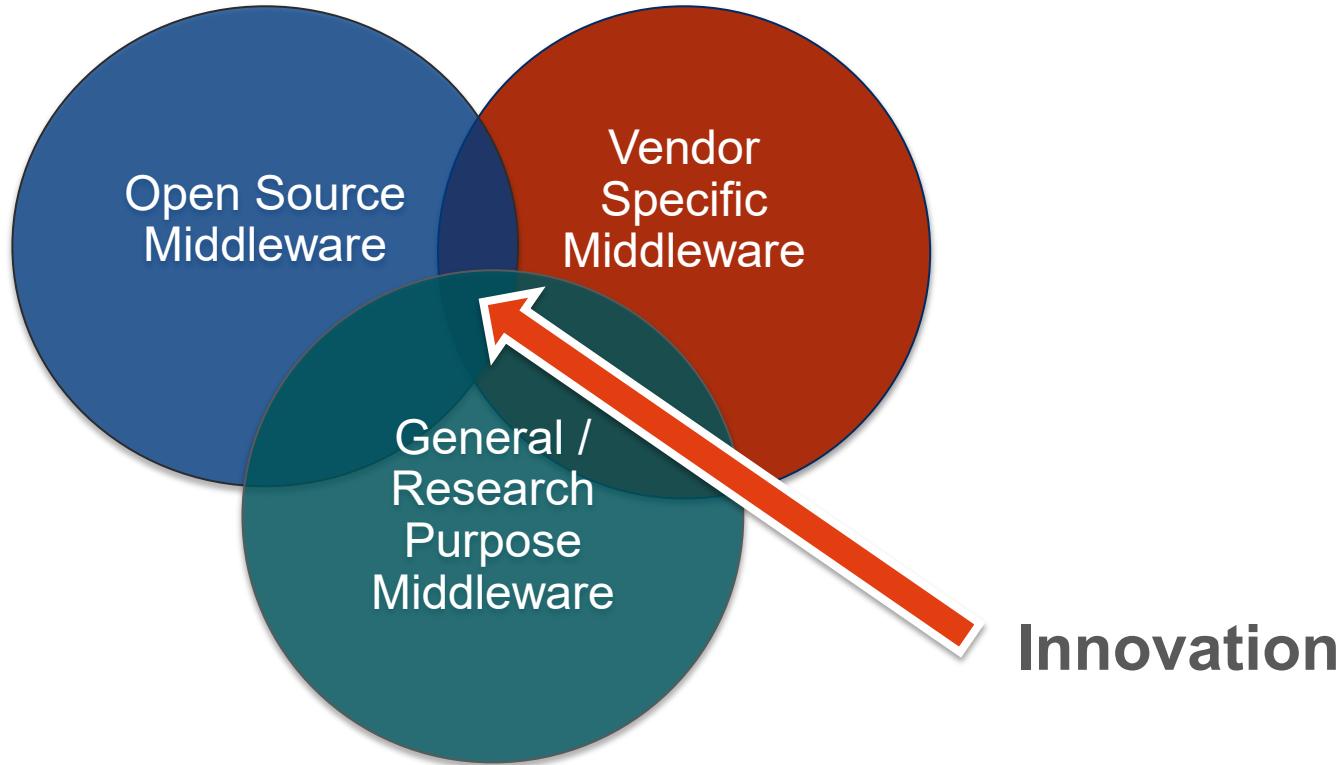
- Typically, not as optimized as commercial/vendor software
- Maintenance is challenge

## Pros

- Innovative and forward looking
  - A lot of good ideas for “free”

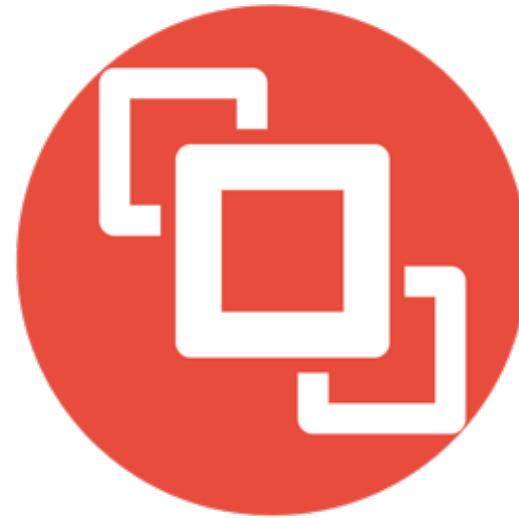
## Cons

- Support, support, support
- Typically, narrow focus



## Unified Communication - X Framework

# UCX



- 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

## API

Exposes broad semantics that target data centric and HPC programming models and applications

## Performance oriented

Optimization for low-software overheads in communication path allows near native-level performance

## Production quality

Developed, maintained, tested, and used by industry and researcher community

## Community driven

Collaboration between industry, laboratories, and academia

## Research

The framework concepts and ideas are driven by research in academia, laboratories, and industry

## Cross platform

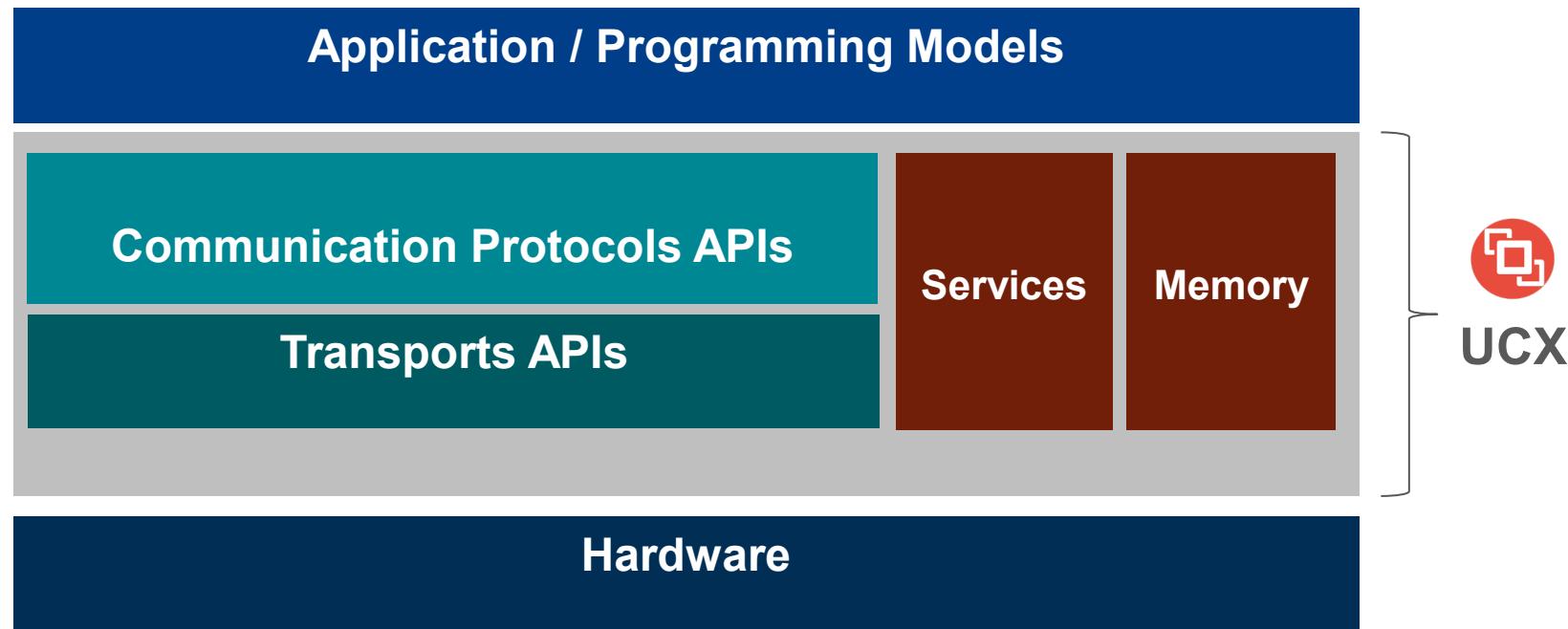
Support for InfiniBand, HPE, various shared memory (x86-64, Power, ARM), GPUs

## Co-design of Next-Generation Network APIs



- Mellanox co-designs network API and contributes MXM technology
  - Infrastructure, transport, shared memory, protocols, integration with OpenMPI/SHMEM, MPICH
- ORNL co-designs network API and contributes to the UCCS project
  - InfiniBand optimizations, Cray devices, shared memory
- LANL co-designs network API
- ARM co-designs the network API and contributes optimizations for ARM eco-system
- NVIDIA co-designs high-quality support for GPU devices
  - GPUDirect, GDR copy, etc.
- IBM co-designs network API and contributes ideas and concepts from PAMI
- UH/UTK focus on integration with their research platforms

- **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.



## 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

## Applications / Programming Models

High-Level API

### UCP - Protocols

Transport selection, multi-rail support, fragmentation  
HPC and I/O protocols (tag matching, active messages, RMA, atomics, etc)  
Connection establishment (client/server, external)

Low-Level API

### UCT - Transports

RDMA (RC, DC, UD, iWarp), Aries (GNI), Accelerators (CUDAm ROCm),  
Shared Memory (XPMEM, etc), Others (TCP/IP, Omnipath, etc)

### UCS - Services

### UCM - Memory

## Hardware

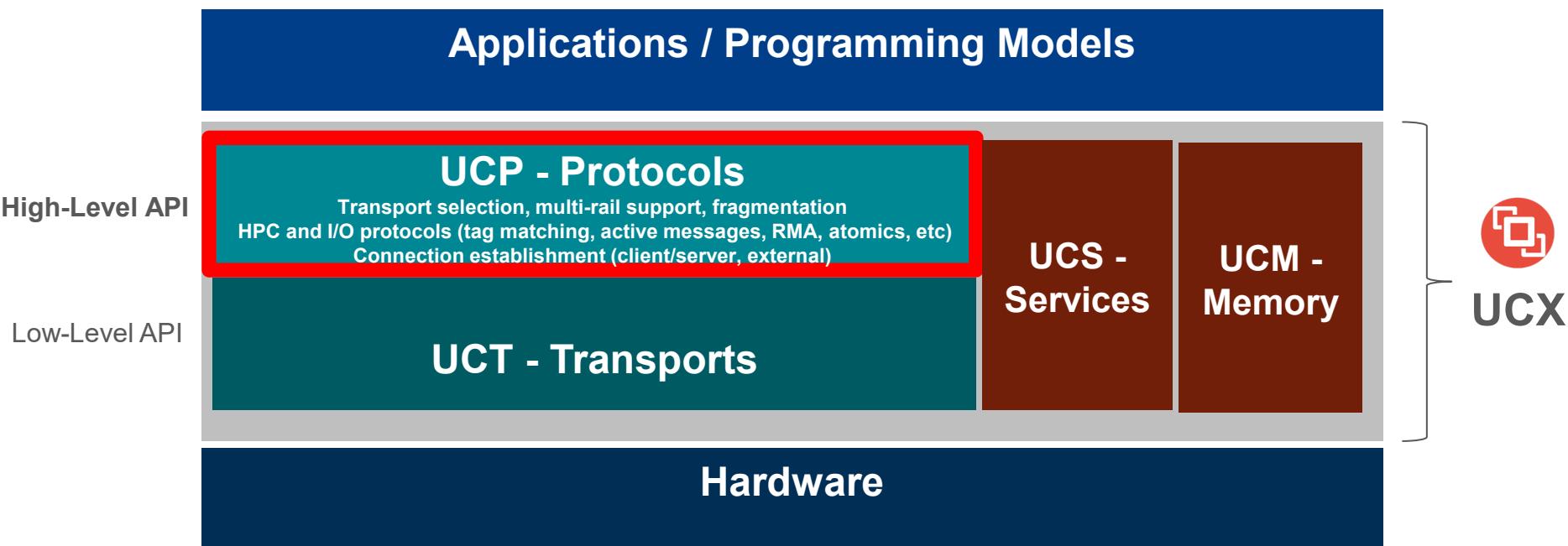


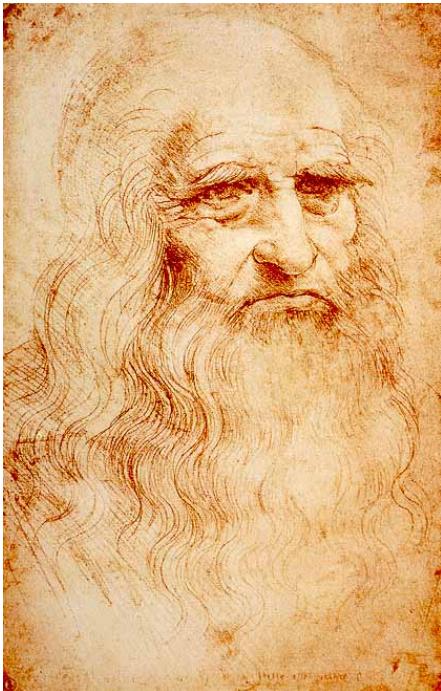
**UCX**

# Clarifications

- UCX is not a device driver
- UCX is a communication framework
  - Close-to-hardware API layer
  - Providing an access to hardware's capabilities
- UCX relies on drivers supplied by vendors

# API Overview





“Simplicity is the ultimate sophistication.”

*Leonardo da Vinci*

# Protocol Layer

- Selects the best network for the application
  - Does not have to be the same vendor
- Optimized by default
  - Protocols are optimized for the message size and underlying network semantics
  - Intelligent fragmentation
- Multi-rail, multi-interconnect communication
- Emulates unsupported semantics in software
  - No “ifdefs” in user code
  - Software atomics, tag-matching, etc.
- Abstracts connection setup
- Handles 99% of “corner” cases
  - Network out of resources
  - Reliability
  - No message size limit
  - ....and many more

# UCP Objects

## ucp\_context\_h

- A global context for the application; holds the memory registrations and global device context. For example, a hybrid MPI/SHMEM library may create one context for MPI, and another for SHMEM.

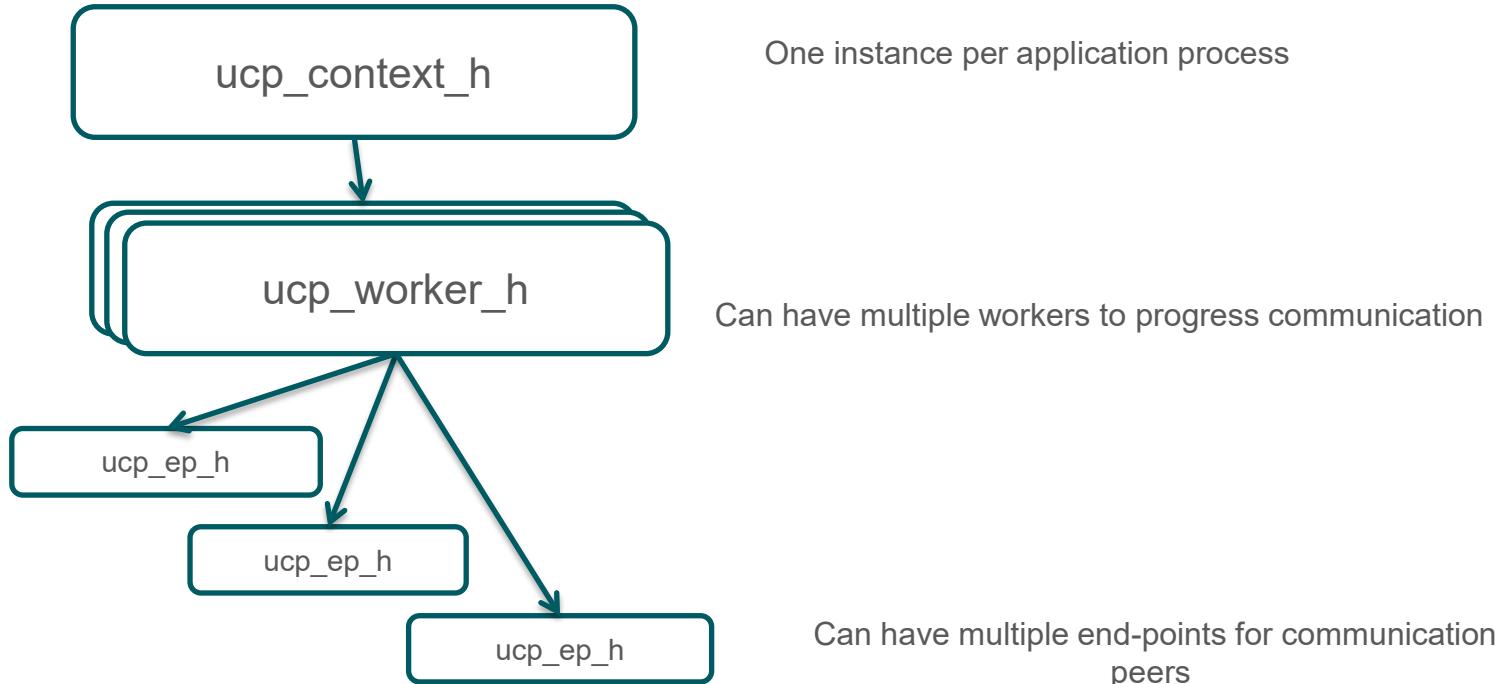
## ucp\_worker\_h

- 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.

# A diagram on how different UCP Objects are related to each other



# 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.

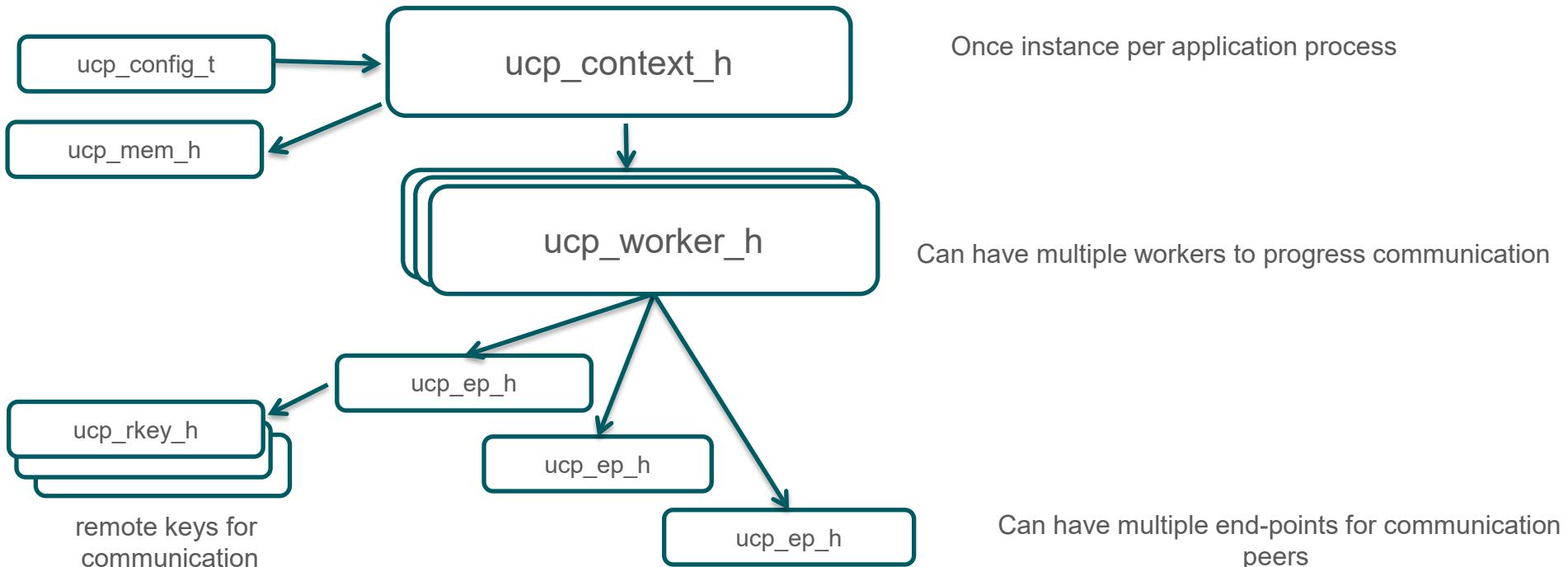
## **`ucp_rkey_h`**

- 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.

# A diagram on how different UCP Objects are related to each other



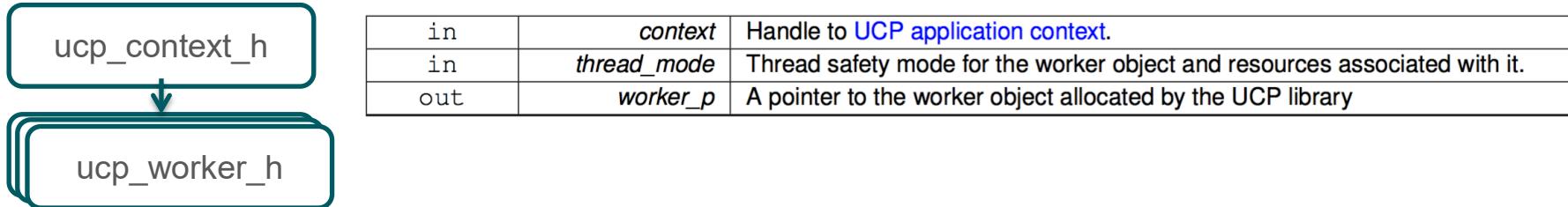
- `ucp_init ( const ucp_params_t * params, const ucp_config_t * config, ucp_context_h * context_p )`

`ucp_context_h`

in	<i>config</i>	UCP configuration descriptor allocated through <code>ucp_config_read()</code> routine.
in	<i>params</i>	User defined <code>tunings</code> for the UCP application context.
out	<i>context_p</i>	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`

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



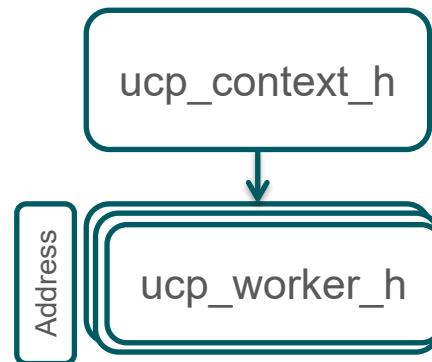
- This routine allocates and initializes a worker object. Each worker is associated with one and only one application context. At the same time, an application context can create multiple workers in order to enable concurrent access to communication resources. For example, application can allocate a 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 )`

<i>in</i>	<i>worker</i>	Worker object whose address to return.
<i>out</i>	<i>address_p</i>	A pointer to the worker address.
<i>out</i>	<i>address_length_p</i>	The size in bytes of the address.

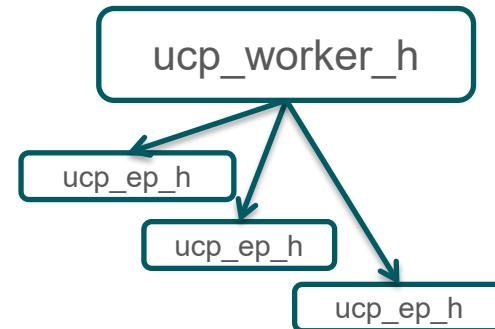
- This routine returns the address of the worker object. This address can be passed to remote instances of the UCP library in order to connect to this worker. The memory for the address handle is allocated by this function and must be released by using `ucp_worker_release_address()` routine.

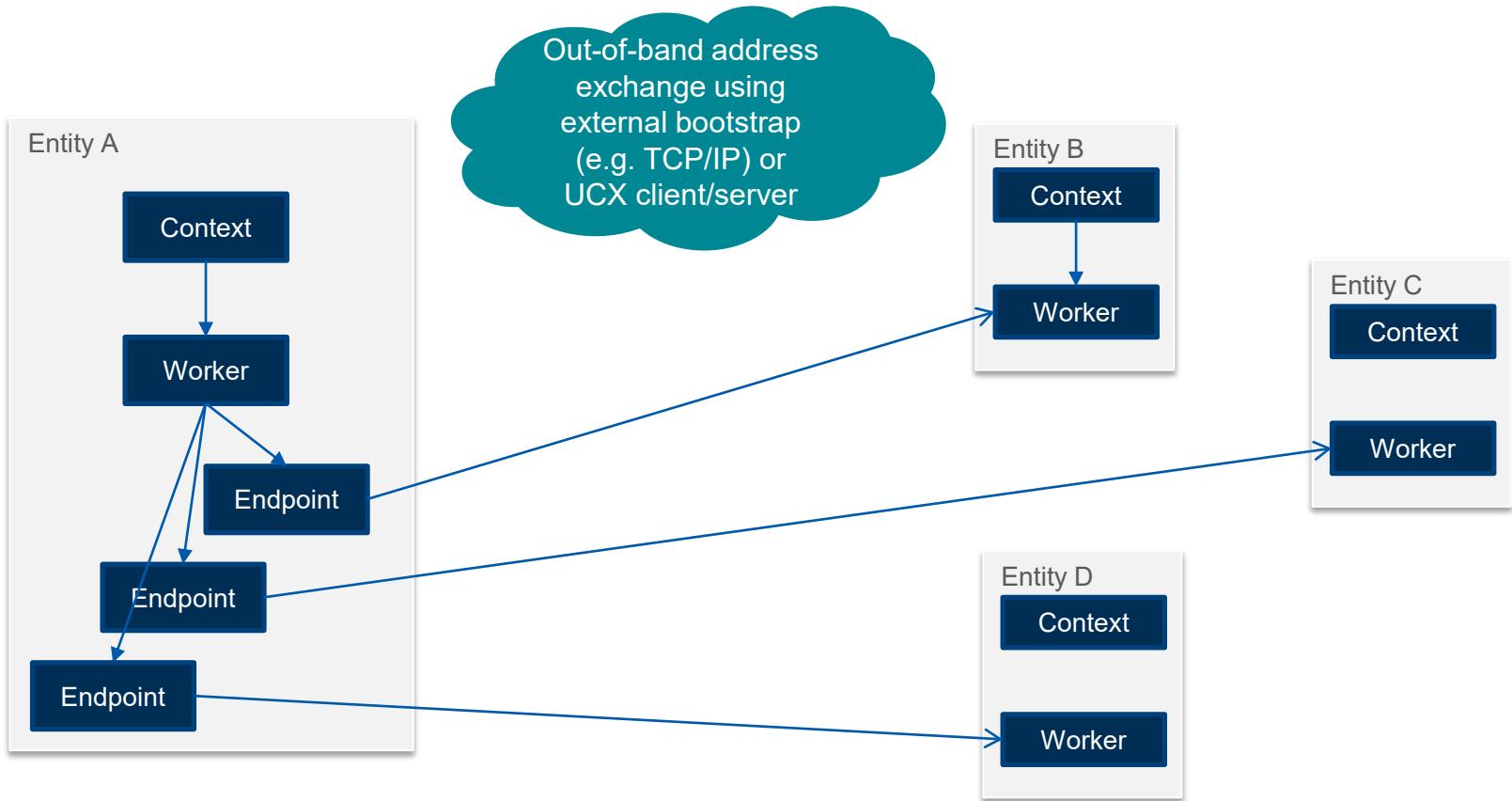


- `ucs_status_t ucp_ep_create ( ucp_worker_h worker, const ucp_address_t * address, ucp_ep_h * ep_p )`

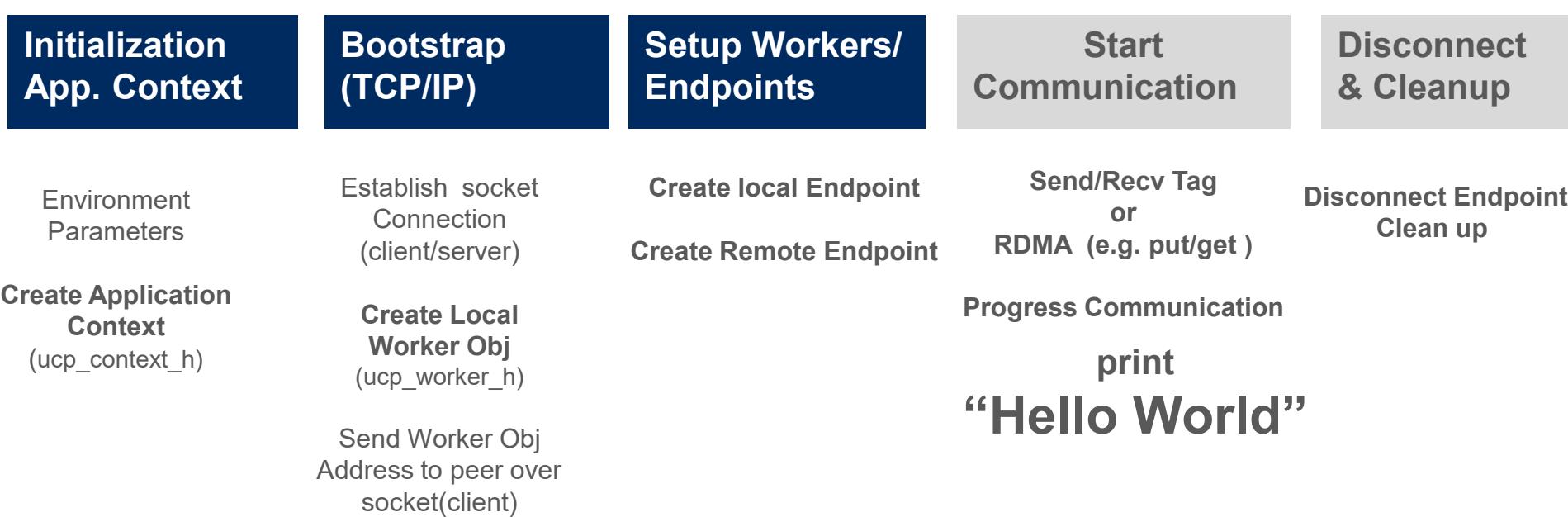
in	<i>worker</i>	Handle to the worker; the endpoint is associated with the worker.
in	<i>address</i>	Destination address; the address must be obtained using <code>ucp_worker_get_address()</code> routine.
out	<i>ep_p</i>	A handle to the created endpoint.

- This routine creates and connects an endpoint on a local worker for a destination address that identifies the remote worker. 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`





## Application Flow



# TUTORIAL BREAK

Time	Topic	Presenters	Slides	Notes
9:00 - 9:20	UCX Tutorial and Ecosystem Introduction	Gilad		
9:20 - 10:00	UCX Basics - networking overview, worker and endpoint creation	Jeff		
10:00 - 10:10	<b>BREAK</b>			
10:10 - 10:20	Hello World part 1	Matt, Jeff		
10:20 - 10:35	UCX memory management	Jeff, Oscar		
10:35 - 10:45	Hello World Demo, part 2	Matt, Jeff		
10:45 - 11:00	GPU discussion/demo	Oscar, Matt		
11:00 - 11:10	<b>BREAK</b>			
11:10 - 11:50	UCX Advanced Topics - bindings, OpenMPI and OpenSHMEM integration	Matt, Yossi, Oscar		

We will return at **10:10 US PDT**

- Code walk-through

## Application Flow

### Initialization App. Context

### Bootstrap (TCP/IP)

### Setup Workers/ Endpoints

### Start Communication

### Disconnect & Cleanup

Environment  
Parameters

Establish socket  
Connection  
(client/server)

Create local Endpoint  
  
Create Remote Endpoint

Send/Recv Tag  
or  
RDMA (e.g. put/get )

Disconnect Endpoint  
Clean up

Create Application  
Context  
(ucp\_context\_h)

Create Local  
Worker Obj  
(ucp\_worker\_h)

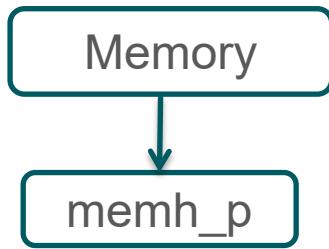
Progress Communication

print

**“Hello World”**

Send Worker Obj  
Address to peer over  
socket(client)

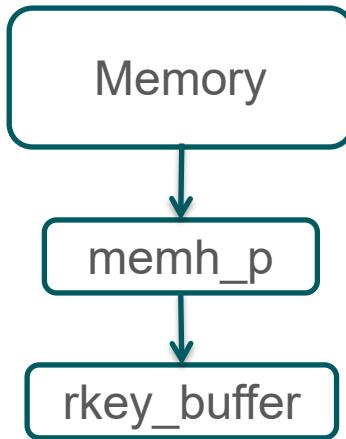
- `ucs_status_t ucp_mem_map (ucp_context_h context, void **address_p, size_t length, unsigned flags, ucp_mem_h *memh_p)`



in	<i>context</i>	Application <b>context</b> to map (register) and allocate the memory on.
in,out	<i>address_p</i>	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	<i>length</i>	Length (in bytes) to allocate or map (register).
in	<i>flags</i>	Allocation flags (currently reserved - set to 0).
out	<i>memh_p</i>	UCP <b>handle</b> for the allocated segment.

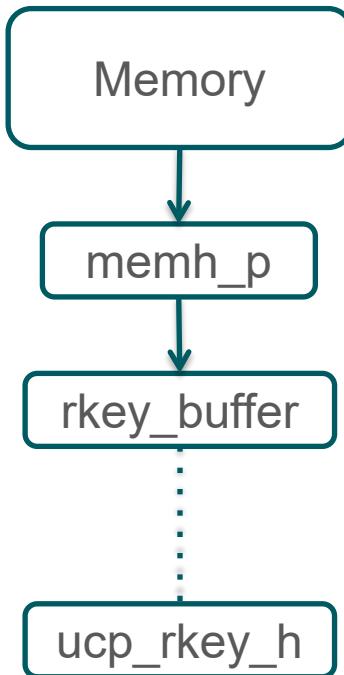
- This routine maps or/and allocates a user-specified memory segment with UCP application context and the network resources associated with it.
- Related routines: `ucp_mem_unmap`

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



in	<i>context</i>	Application <a href="#">context</a> which was used to allocate/map the memory.
in	<i>memh</i>	<a href="#">Handle</a> to memory region.
out	<i>rkey_buffer_p</i>	Memory buffer allocated by the library. The buffer contains packed RKEY.
out	<i>size_p</i>	Size (in bytes) of the packed RKEY.

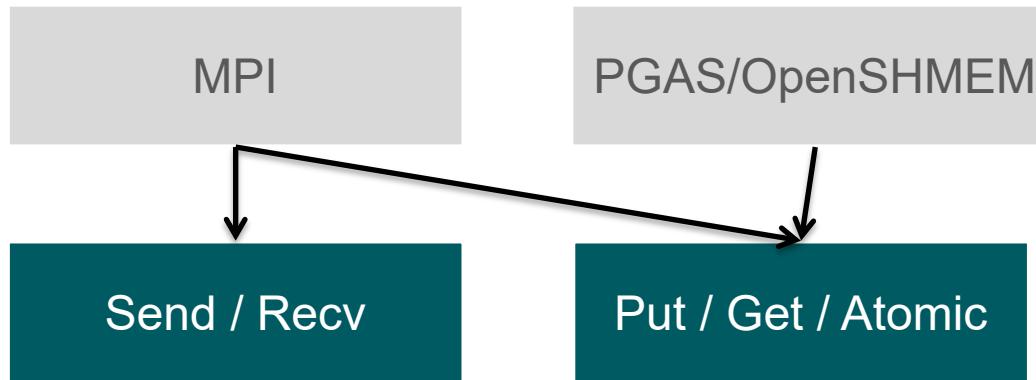
- This routine allocates memory buffer and packs into the buffer a remote access key (RKEY) object. RKEY is an opaque object that provides the information that is necessary for remote memory access. This routine packs the RKEY object in a portable format such that the object can be unpacked on any platform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



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

in	<i>ep</i>	Endpoint to access using the remote key.
in	<i>rkey_buffer</i>	Packed rkey.
out	<i>rkey_p</i>	Remote key handle.

- This routine unpacks the remote key (RKEY) object into the local memory such that it can be accessed 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`



# Put operation

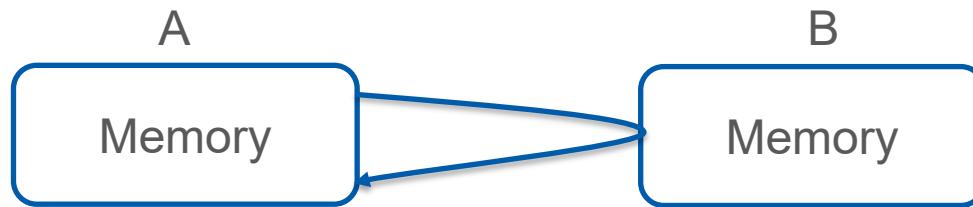
- `ucs_status_ptr_t ucp_put_nb( ucp_ep_h ep, const void * buffer, size_t count, uint64_t remote_addr, ucp_rkey_h rkey, const ucp_request_param_t * param )`

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



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

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



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

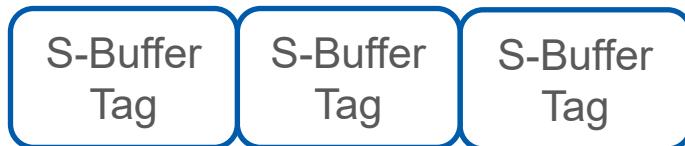
in	<i>ep</i>	Destination endpoint handle.
in	<i>buffer</i>	Pointer to the message buffer (payload).
in	<i>count</i>	Number of elements to send
in	<i>tag</i>	Message tag.
in	<i>param</i>	Operation parameters, see <a href="#">ucp_request_param_t</a>



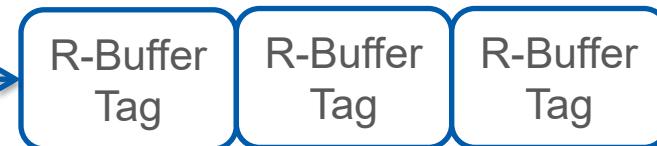
- `ucs_status_ptr_t ucp_tag_recv_nb(ucp_worker_h worker, void * buffer, size_t count, ucp_tag_t tag, ucp_tag_t tag_mask, const ucp_request_param_t * param)`

in	<i>worker</i>	UCP worker that is used for the receive operation.
in	<i>buffer</i>	Pointer to the buffer to receive the data to.
in	<i>count</i>	Number of elements to receive
in	<i>tag</i>	Message tag to expect.
in	<i>tag_mask</i>	Bit mask that indicates the bits that are used for the matching of the incoming tag against the expected tag.
in	<i>param</i>	Operation parameters, see <a href="#">ucp_request_param_t</a>

Sender



Receiver



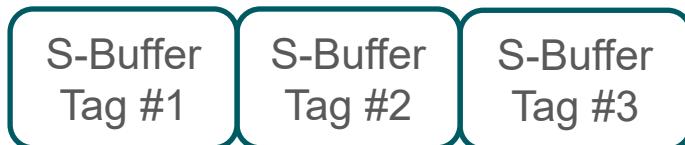
## ucs\_status\_ptr\_t

- 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 )

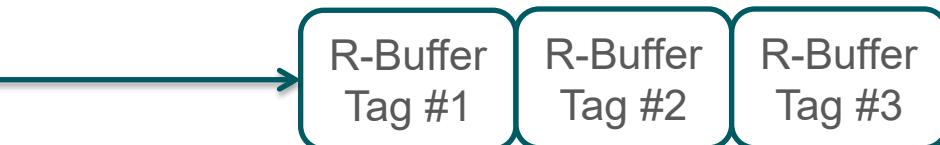
- `ucs_status_ptr_t ucp_tag_send_sync_nb( ucp_ep_h ep, const void * buffer, size_t count, ucp_tag_t tag, const ucp_request_param_t * param )`

in	<i>ep</i>	Destination endpoint handle.
in	<i>buffer</i>	Pointer to the message buffer (payload).
in	<i>count</i>	Number of elements to send
in	<i>tag</i>	Message tag.
in	<i>param</i>	Operation parameters, see <a href="#">ucp_request_param_t</a>

Sender

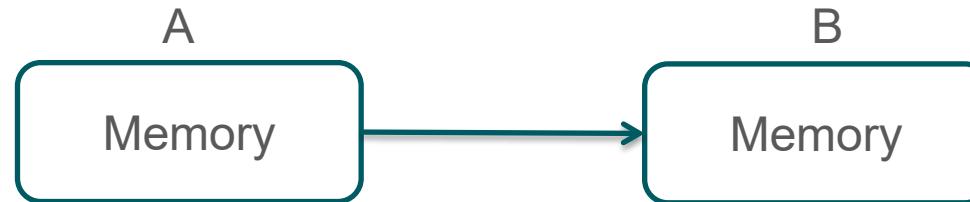


Receiver



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



## Communication Flow

### Initialization App. Context

Environment Parameters

Create Application Context  
(ucp\_context\_h)

### Bootstrap (TCP/IP)

Establish socket Connection (client/server)

Create Local Worker Obj  
(ucp\_worker\_h)

Send Worker Obj Address to peer over socket(client)

### Setup Workers/ Endpoints

Create local Endpoint  
Create Remote Endpoint

### Start Communication

Send/Recv Tag or RDMA (e.g. put/get )

Progress Communication  
print

**“Hello World”**

### Disconnect & Cleanup

Disconnect Endpoint Clean up

- “Hello World with UCP”
  - Using two nodes with IB
  - UCX for communication - send/recv
  - Finalization

## Communication Flow

### Initialization App. Context

Environment Parameters

Create Application Context  
(ucp\_context\_h)

### Bootstrap (TCP/IP)

Establish socket Connection (client/server)

Create Local Worker Obj  
(ucp\_worker\_h)

Send Worker Obj Address to peer over socket(client)

### Setup Workers/ Endpoints

Create local Endpoint  
Create Remote Endpoint

### Start Communication

Send/Recv Tag or RDMA (e.g. put/get )

Progress Communication  
print

**“Hello World”**

### Disconnect & Cleanup

Disconnect Endpoint Clean up

# 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 UCP using GPUs”
  - Show how to use UCP to communicate between CPU and GPUs

# TUTORIAL BREAK

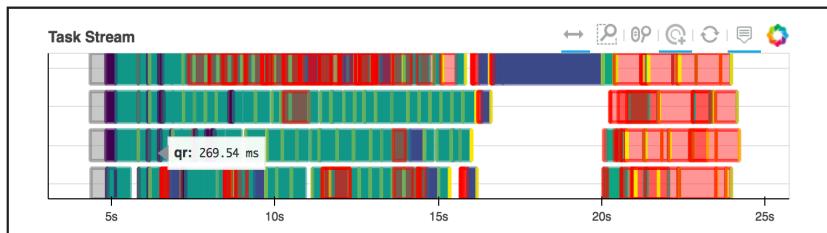
Time	Topic	Presenters	Slides	Notes
9:00 - 9:20	UCX Tutorial and Ecosystem Introduction	Gilad		
9:20 - 10:00	UCX Basics - networking overview, worker and endpoint creation	Jeff		
10:00 - 10:10	<b>BREAK</b>			
10:10 - 10:20	Hello World part 1	Matt, Jeff		
10:20 - 10:35	UCX memory management	Jeff, Oscar		
10:35 - 10:45	Hello World Demo, part 2	Matt, Jeff		
10:45 - 11:00	GPU discussion/demo	Oscar, Matt		
11:00 - 11:10	<b>BREAK</b>			
11:10 - 11:50	UCX Advanced Topics - bindings, OpenMPI and OpenSHMEM integration	Matt, Yossi, Oscar		

We will return at **11:10 US PDT**

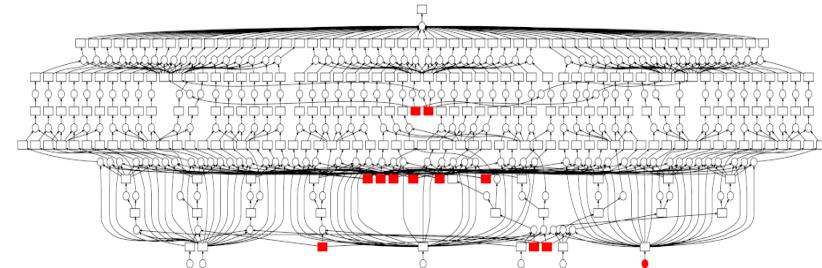
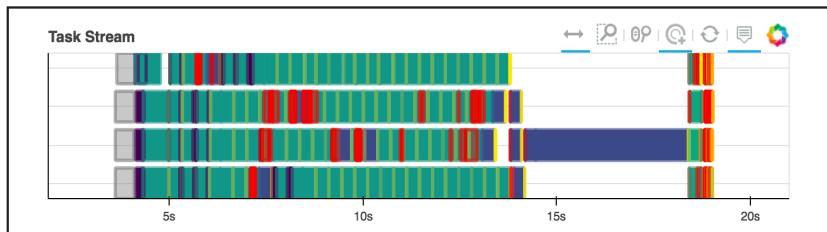
# Other bindings: UCX-py, UCX-java, UCX-go

Helping converge HPC to Data Sciences,  
AI, and BigData

Before UCX:



After UCX:



Science loves Python

- Easy to learn, free; numpy is similar to MATLAB

RAPIDS

- Open-source data science python libraries
- Single thread multi accelerator

DASK

- Distributed tasking framework for python

UCX

- Client-server model, python bindings, efficient RDMA, etc.

UCX-py: <https://github.com/rapidsai/ucx-py>

UCX-java, UCX-go: <https://github.com/openucx/ucx>

Source: Matthew Rocklin, Rick Zamora, **Experiments in High Performance Networking with UCX and DGX**  
<https://blog.dask.org/2019/06/09/ucx-dgx>

# Example of UCX-py (via RAPIDS)

```
async def test_send_recv_numpy(size, dtype, blocking_progress_mode):
    ucp.init(blocking_progress_mode=blocking_progress_mode)

    msg = np.arange(size, dtype=dtype)
    msg_size = np.array([msg.nbytes], dtype=np.uint64)

    listener = ucp.create_listener(
        make_echo_server(lambda n: np.empty(n, dtype=np.uint8))
    )
    client = await ucp.create_endpoint(ucp.get_address(), listener.port)
    await client.send(msg_size)
    await client.send(msg)
    resp = np.empty_like(msg)
    await client.recv(resp)
    np.testing.assert_array_equal(resp, msg)
```

Example extracted from [https://github.com/rapidsai/ucx-py/blob/branch-0.22/tests/test\\_send\\_recv.py](https://github.com/rapidsai/ucx-py/blob/branch-0.22/tests/test_send_recv.py)

# AGENDA UPDATE

Time	Topic	Presenters	Slides	Notes
9:00 - 9:20	UCX Tutorial and Ecosystem Introduction	Gilad		
9:20 - 10:00	UCX Basics - networking overview, worker and endpoint creation	Jeff		
10:00 - 10:10	<b>BREAK</b>			
10:10 - 10:20	Hello World part 1	Matt, Jeff		
10:20 - 10:35	UCX memory management	Jeff, Oscar		
10:35 - 10:45	Hello World Demo, part 2	Matt, Jeff		
10:45 - 11:00	GPU discussion/demo	Oscar, Matt		
11:00 - 11:10	<b>BREAK</b>			
11:10 - 11:50	UCX Advanced Topics - bindings, OpenMPI and OpenSHMEM integration	Matt, Yossi, Oscar		

# OPEN MPI INTEGRATION WITH UCX

# Overview

- Open MPI (OMPI) supports UCX starting OMPI v1.10.
- UCX is a PML component in OMPI.  
To enable UCX:
  - mpirun -mca pml ucx ... <APP>
- OMPI is integrated with the UCP layer.

# Overview

- UCX mca parameters:
  - pml\_uxc\_verbose, pml\_uxc\_priority
- UCX environment parameters:
  - ucx\_info -f
- For example:  
`mpirun -mca pml ucx -x UCX_NET_DEVICES=mlx5_0:1 ... <APP>`

# Overview

- UCX will select the best transport to use.
- Supported transports include
  - IB - ud, rc, dc, accelerated verbs
  - shared memory
  - uGNI
- OMPI uses Full/Direct modes with UCX.
- UCX will connect the ranks.

# UCP Main Objects - Recap

- **`ucp_context_h`**

A global context for the application - a single UCP communication instance.  
Includes communication resources, memory and other communication information directly associated with a specific UCP instance.

- **`ucp_worker_h`**

Represents an instance of a local communication resource and an independent progress of communication. It contains the `uct_iface_h`'s of all selected transports.  
One possible usage is to create one worker per thread.

- **`ucp_ep_h`**

Represents a connection to a remote worker.  
It contains the `uct_ep_h`'s of the active transports.

# UCX Main Objects - Recap

- **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 an 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.

# OMPI - UCX Stack



# Init Stage

`MPI_Init` → `ompi_mpi_init`  
`mca_pml_ucx_open`

} OpenMPI

`ucp_config_read`  
`ucp_init`  
`ucp_config_release`

} UCX

# Init Stage - Cont

mca\_pml\_ucx\_init

ucp\_worker\_create

ucp\_worker\_get\_address

ucp\_worker\_release\_address

# Init Stage - Cont

```
opal_progress_register(mca_pml_uxc_progress)
```



```
ucp_worker_progress
```

# Send Flow

MPI\_Isend → mca\_pml\_ucx\_isend

mca\_pml\_ucx\_add\_proc

ucp\_ep\_create

ucp\_tag\_send\_nb



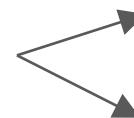
# Send Flow - Cont

- If the send request isn't completed  
→ progress it.
- Once completed  
→ callback function is invoked
- Contiguous and non-contiguous datatypes are supported.

# Receive Flow

`MPI_Irecv` → `mca_pml_ucx_irecv`

`ucp_tag_recv_nb`



- Eager Receive
- Rendezvous

# Receive Flow - Cont

- Expected & Unexpected queues are used
- Can probe with `ucp_tag_probe_nb`  
→ `ucp_tag_msg_recv_nb`
- If the receive request isn't completed  
→ progress it.
- Once completed  
→ callback function is invoked

# Progress Flow

opal\_progress

mca\_pml\_uxc\_progress

ucp\_worker\_progress → uct\_worker\_progress

\* Send/Receive Finished:

mca\_pml\_uxc\_send/recv\_completion

# Finalization Stage

`MPI_Finalize` → `ompi_mpi_finalize`

`mca_pml_ucx_del_procs`

\* Per remote peer:

`ucp_ep_destroy`

# Finalization Stage - Cont

mca\_pml\_uxc\_cleanup

opal\_progress\_unregister (mca\_pml\_uxc\_progress)

ucp\_worker\_destroy

mca\_pml\_uxc\_close

ucp\_cleanup

# OPENSHMEM INTEGRATION WITH UCX

- PGAS Library
- One-sided Communication
- Atomic operations
- Collectives

# Symmetric Memory

- All Processing Elements (PE's) share an address space (symmetric heap)
- Symmetric heap is allocated on startup
- Heapsize can be customized via environment variable SMA\_SYMMETRIC\_SIZE
- Symmetric data objects must be allocated with shmem\_malloc
- Symmetric data objects are accessible by remote PEs

# Shared global address space

- Global and static variables are symmetric objects
- Accessible by remote PEs

## *OpenSHMEM API*

*Atomics   RMA   Collectives   Symetric Memory*

*Core Components*

*Utils*

*Comms*

*UCX*

*GASNet*



**OpenSHMEM**

<http://www.openshmem.org/>

# Initialization

## ■ shmem\_init

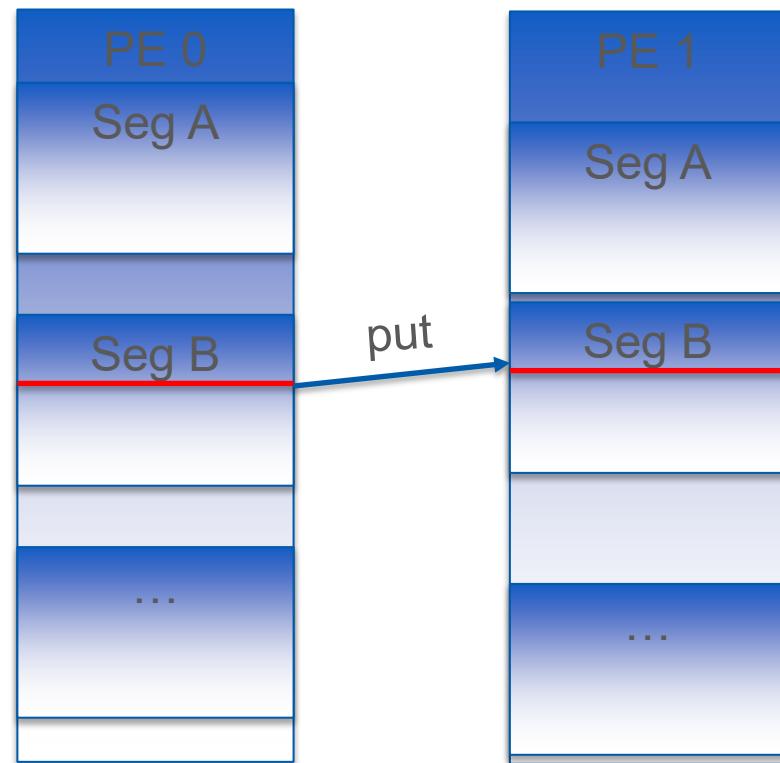
- shmemi\_comms\_init
  - ...
  - ucp\_config\_read
  - ucp\_init
  - ucp\_config\_release
  - ucp\_worker\_create
  - init\_memory\_regions (more on this later)
  - ...
  - ucp\_ep\_create (for each PE)
  - ucp\_config\_release
  - ...

- OpenSHMEM registers global data (data and bss segment) and the symmetric heap
- `ucp_mem_map` maps the memory with the ucp context ( returning `ucp_mem_h` )



# Translating symmetric addresses

- For RMA operations UCP needs Remote Memory Handle (remote key or rkey)
- To access a remote address the rkey is needed
- Look up rkey with *find\_seg*
- Translate local buffer address into remote buffer address



# RMA put / get / atomics

- `shmem_<TYPENAME>_<put, get, atomic_op>`

- `find_seg`
- `translate_symmetric_address`
- depending on the API pick:
  - `ucp_put`
  - `ucp_get`
  - `ucp_atomic_<op,size>`

UCP picks the right internal UCT calls depending on the message size, and operation (e.g. atomics)

# Compiling UCX for OpenMPI and OpenSHMEM

```
$ ./autogen.sh  
$ ./contrib/configure-release --prefix=$PWD/install  
$ make -j && make install
```

**UCX Build**

```
<openmpi_root>$ ./autogen.pl  
$ --prefix=$PWD/install \  
--with-ucx=$PWD/ucx  
$ make -j && make install
```

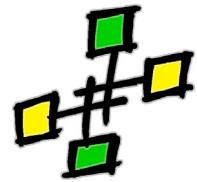
**OpenMPI UCX Build**

```
<openshmem_root>$ ./autogen.pl  
$ ./contrib/configure-release --with-comms-layer=ucx \  
--with-ucx-root=$PWD/install \  
--with-rte-root=$PWD/install \  
--prefix=$PWD/install  
$ make -j && make install
```

**OpenSHMEM UCX Build**

# Summary

- We covered an overview of UCX with a focus on UCP
  - Demonstrated simple examples
- UCX has been integrated with multiple programming models and frameworks:
  - MPI: Open MPI, MPICH
  - OpenSHMEM: Reference Implementation, OSHMEM
  - Dask, Spark, BlazingSQL, etc
- Supports multiple transports and accelerators
  - IB/RoCE: RC, UD, DCT, CM
  - Aries, Slingshot: FMA, SMSG, BTE
  - Shared Memory: SysV, Posix, CMA, KNEM, XPMEM
  - Accelerators: CUDA, ROCm
- Bindings and GPU integration allow for focus on user-level programming rather than on low-level communication APIs
  - UCX-Py and other bindings provide paths to use UCX for HPC apps
  - GPU integration helps to accelerate DGX cluster communication



## Unified Communication - X Framework

Web:

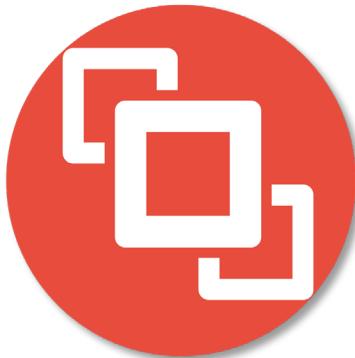
[www.openucx.org](http://www.openucx.org)

<https://github.com/openucx/ucx>

Mailing List:

<https://elist.ornl.gov/mailman/listinfo/ucx-group>

[ucx-group@elist.ornl.gov](mailto:ucx-group@elist.ornl.gov)





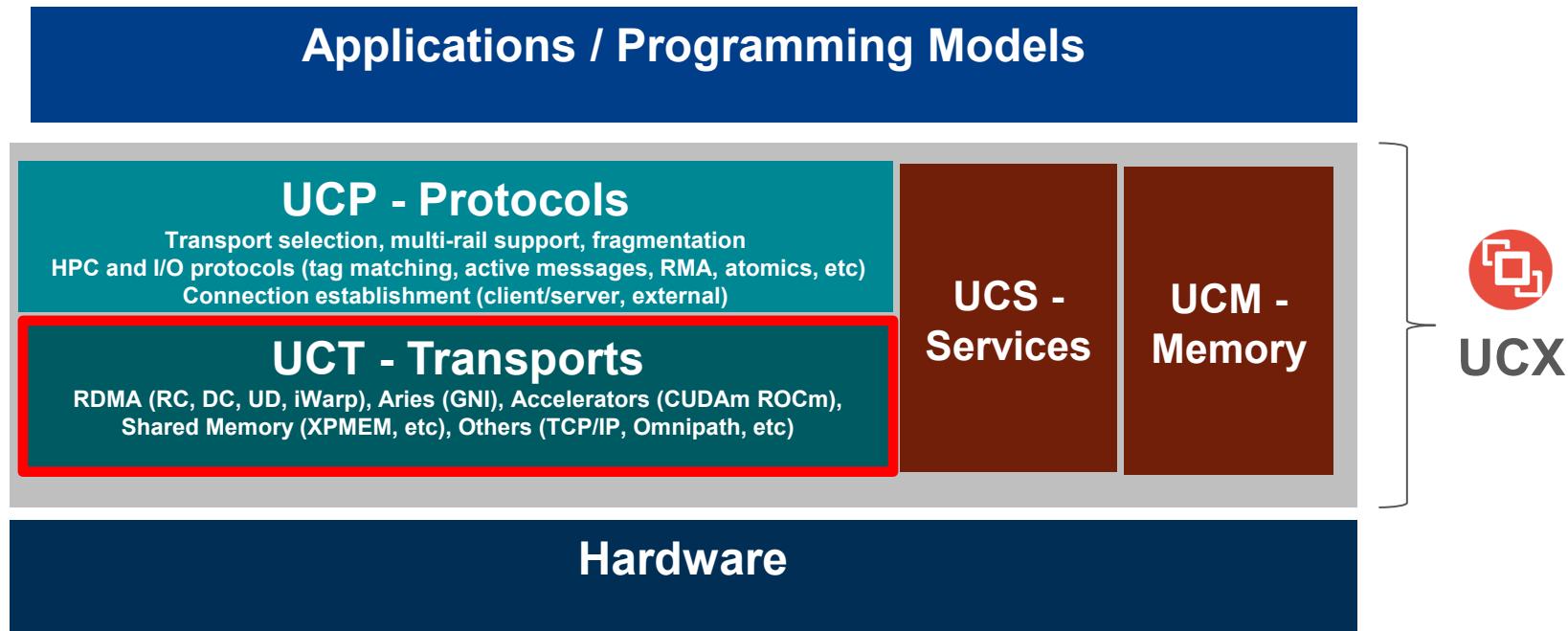
# Acknowledgements

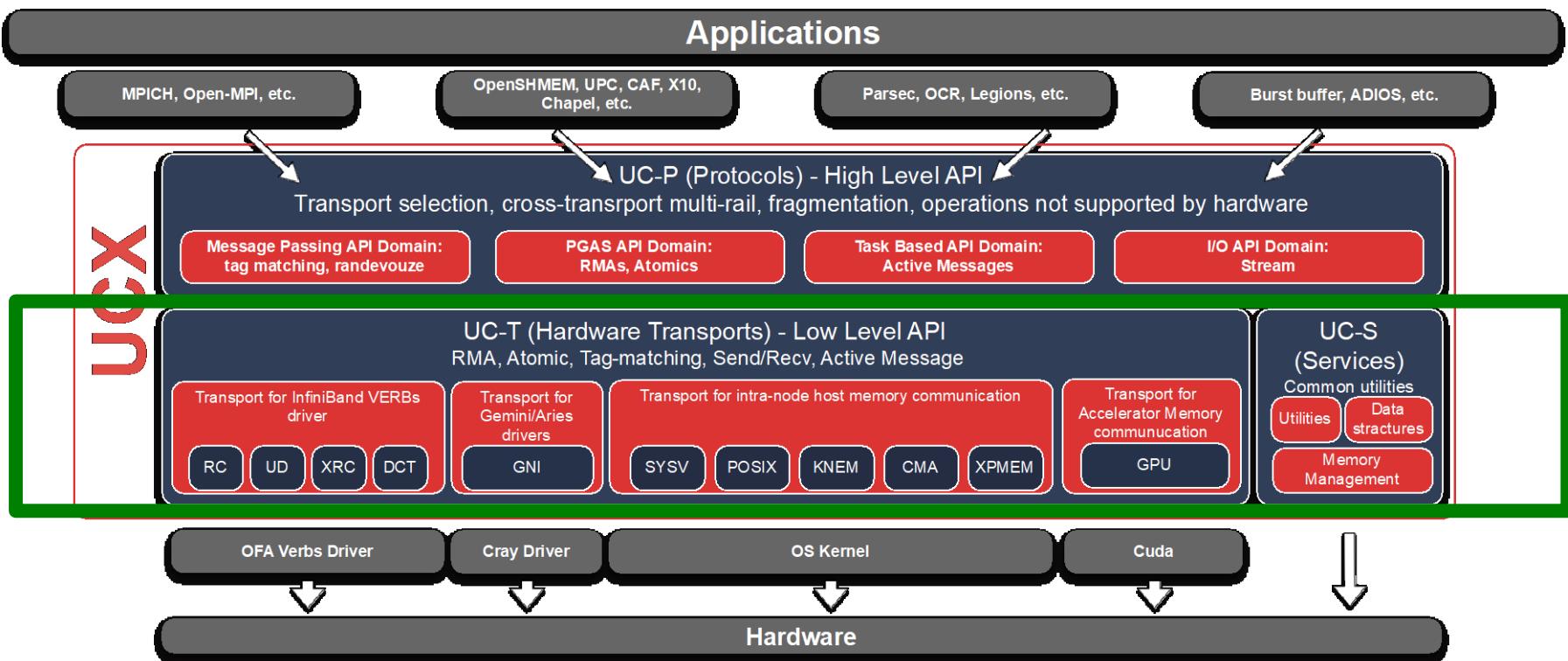
- This work was supported by the United States Department of Defense & used resources at Oak Ridge National Laboratory.
- This work was also partially supported by NSF CNS #2016701, "Rogues Gallery: A Community Research Infrastructure for Post-Moore Computing".



# Backup Slides



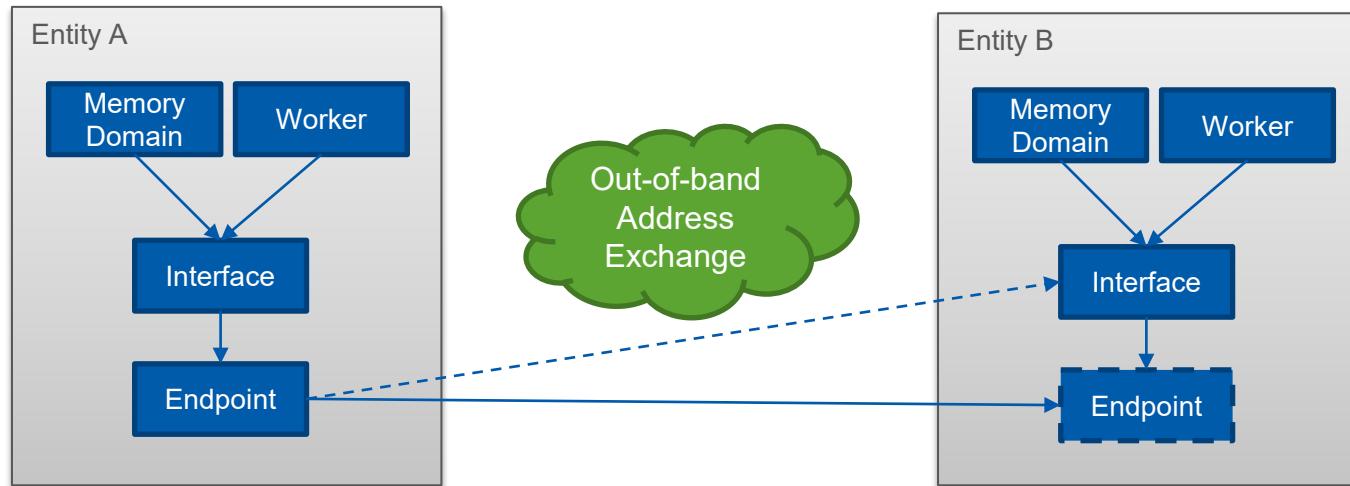




## UCT (Transport layer) objects

- `uct_worker_h` - A context for separate progress engine and communication resources. Can be either thread-dedicated or shared
- `uct_md_h` - Memory registration domain. Can register user buffers and/or allocate registered memory
- `uct_iface_h` - Communication interface, created on a specific memory domain and worker. Handles incoming active messages and spawns connections to remote interfaces
- `uct_ep_h` - Connection to a remote interface. Used to initiate communications

# UCT initialization



- Register/de-register memory within the domain
  - Can potentially use a cache to speedup memory registration
- Allocate/de-allocate registered memory
- Pack memory region handle to a remote-key-buffer
  - Can be sent to another entity
- Unpack a remote-key-buffer into a remote-key
  - Can be used for remote memory access

- Not everything has to be supported
  - Interface reports the set of supported primitives
  - UCF uses this info to construct protocols
  - UCF implement emulation of unsupported directives
- Send active message (active message id)
- Put data to a remote memory (virtual address, remote key)
- Get data from a remote memory (virtual address, remote key)
- Perform an atomic operation on a remote memory:
  - Add
  - Fetch-and-add
  - Swap
  - Compare-and-swap
- Communication Fence and Flush (Quiet)

- UCT communications have a size limit
  - Interface reports max. allowed size for every operation
  - Fragmentation, if required, should be handled by user / UCP
- Several data “classes” are supported
  - “short” – small buffer
  - “bcopy” – a user callback which generates data (in many cases, “memcpy” can be used as the callback)
  - “zcopy” – a buffer and it’s memory region handle. Usually large buffers are supported.
- Atomic operations use a 32 or 64 bit values

- All operations are non-blocking
- Return value indicates the status:
  - OK – operation is completed
  - INPROGRESS – operation has started, but not completed yet
  - NO\_RESOURCE – cannot initiate the operation right now. The user might want to put this on a pending queue, or retry in a tight loop
  - ERR\_xx – other errors
- Operations which may return INPROGRESS (get/atomics/zcopy) can get a completion handle
  - User initializes the completion handle with a counter and a callback
  - Each completion decrements the counter by 1, when it reaches 0 – the callback is called

# Memory

- It is a limited resource
  - The goal is to maximize the availability of memory for the application
- Avoid  $O(n)$  memory allocations, where  $n$  is the number communication peers (endpoints)
- Keep the endpoint object as small as possible
- Keep the memory pools size limited
- Allocation has to be proportional to the number of in-flight-operations

- Three main data paths:
  - Short messages – critical path
  - Medium messages
  - All the rest

- Take care of the small-message case first
- Avoid function calls
- Avoid extra pointer dereference, especially store operations
- Avoid adding conditionals, if absolutely required use `ucs_likely/ucs_unlikely` macros
- Avoid bus-locked instructions (atomics)
- Avoid branches
- No `malloc()/free()` nor system calls
- Limit the scope of local variables (the time from first to last time it is used) - larger scopes causes spilling more variables to the stack
- Use benchmarks and performance analysis tools to analyze the impact on the latency and message rate

- Avoid locks if possible. If needed, use spinlock, no mutex.
- Reduce function calls
- Move error and slow-path handling code to non-inline functions, so their local variables will not add overhead to the prologue and epilogue of the fast-path function.

- Performance is still important
- No system calls / malloc() / free()
- It's ok to reasonable add pointer dereferences, conditionals, function calls, etc.
  - Having a readable code here is more important than saving one conditional or function call.
- Protocol-level performance considerations are more important here, such as fairness between connections, fast convergence, etc.
- Avoid O(n) complexity. As a thumb rule, all scheduling mechanisms have to be O(1).