OFFLOADING COLLECTIVE OPERATIONS TO THE BLUEFIELD DATA PROCESSING UNIT

RICH GRAHAM

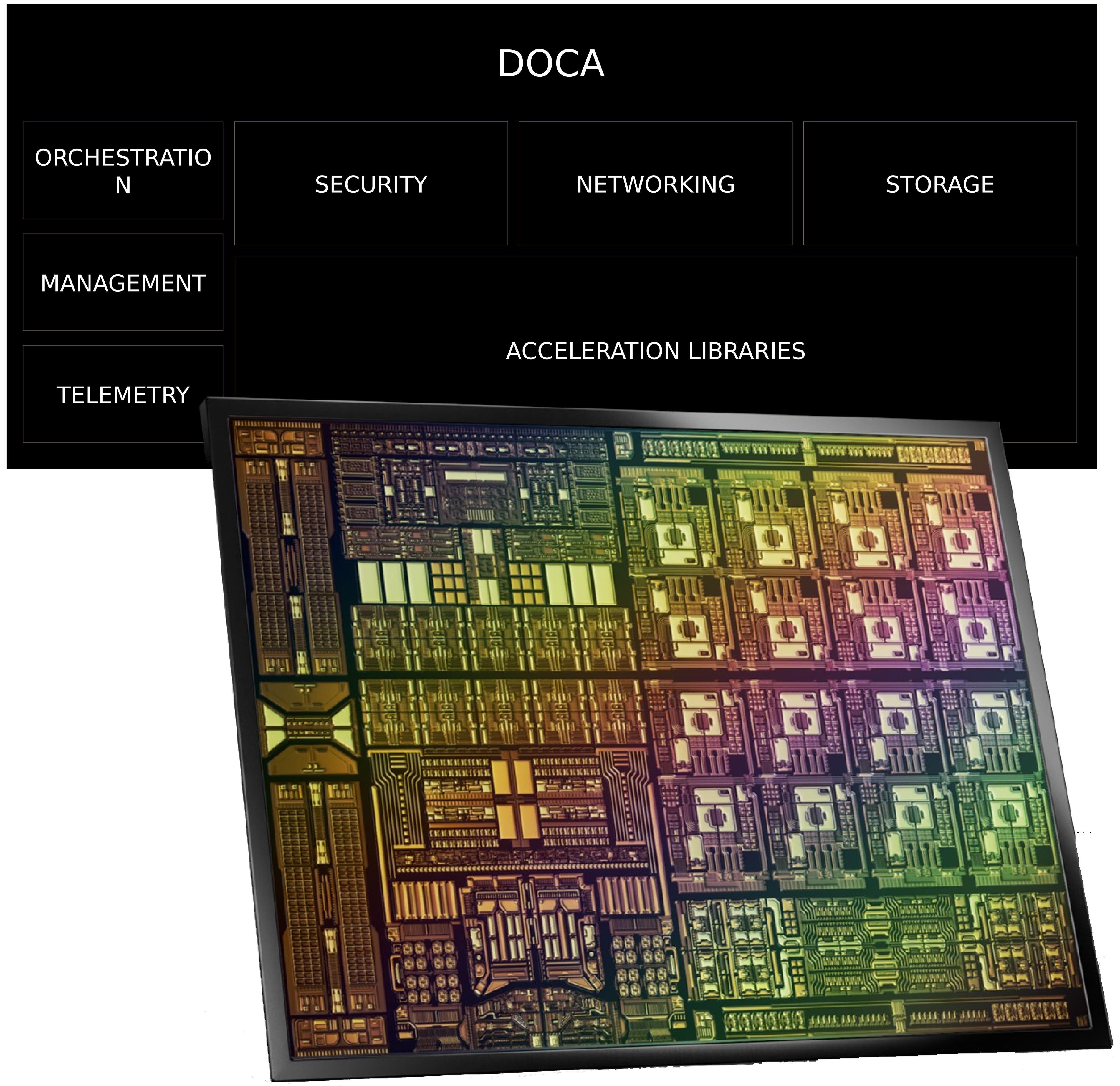
BLUEFIELD – NVIDIA’S DATA PROCESSING UNIT

NVIDIA BLUEFIELD-2

HDR Data Center On A Chip

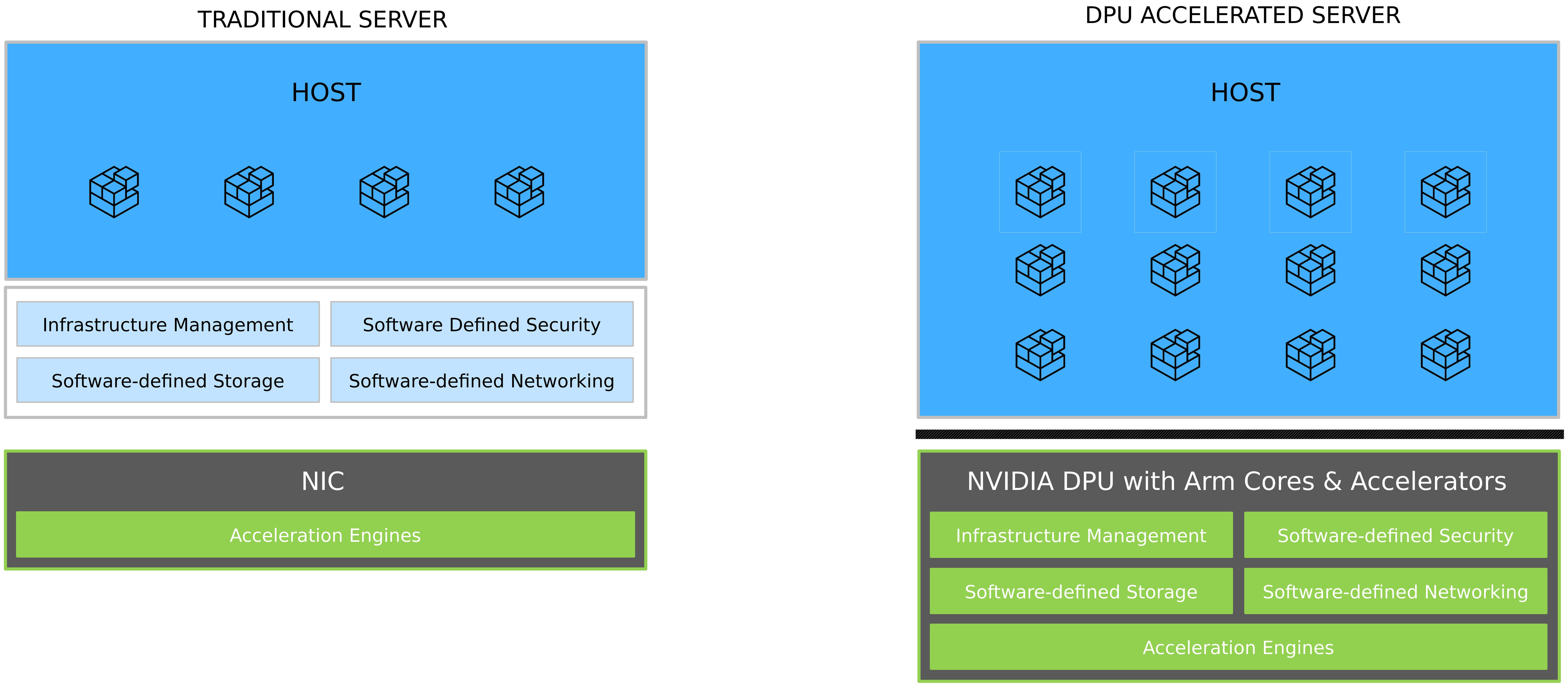
- Offloads and Accelerates Applications and Data Center Infrastructure

Network Bandwidth	200Gb/s
RDMA max msg rate	215Mpps
Compute Cores	8
Compute	SPECINT2K6: 70
Memory Bandwidth	17GB/s
NVMe-OF	10M IOPs @ 4KB
NVMe SNAP	5.4M IOPs @ 4KB



BLUEFIELD DATA PROCESSING UNIT

Software-Defined, Hardware-Accelerated Data Center Infrastructure-on-a-Chip



IN-NETWORK COMPUTING ACCELERATED SUPERCOMPUTING

Software-Defined, Hardware-Accelerated, InfiniBand Network

Advanced Networking

End-to-End

High Throughput	Extremely Low Latency	High Message Rate
RDMA	GPUDirect RDMA	GPUDirect Storage
Adaptive Routing	Congestion Control	Smart Topologies

In-Network Computing

Adapter/DPU		Switch
All-to-All	MPI Tag Matching	Data Reductions (SHARP)
Programmable Datapath Accelerator	Data processing units (Arm cores)	Self Healing Network

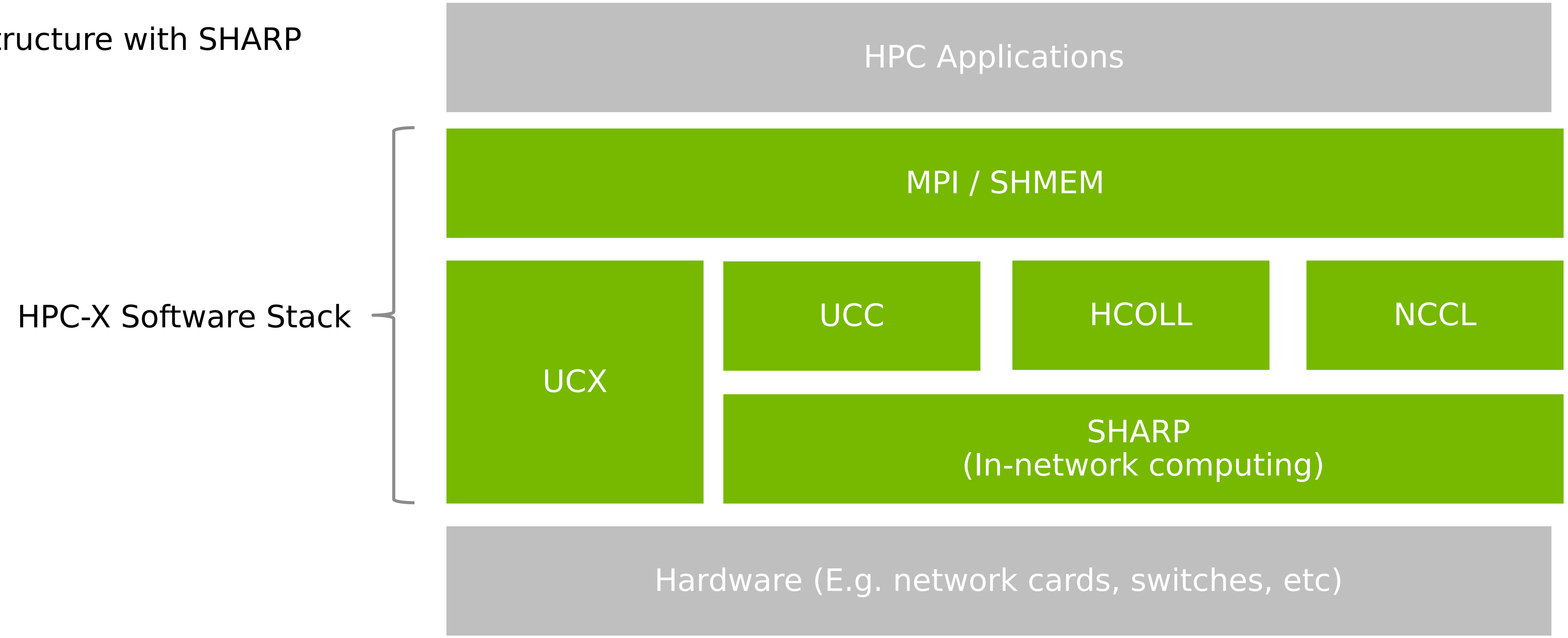
End-to-End — Security / Isolation — End-to-End

COLLECTIVE INFRASTRUCTURE - UCC

NVIDIA HPC-X

Software Stack

- 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



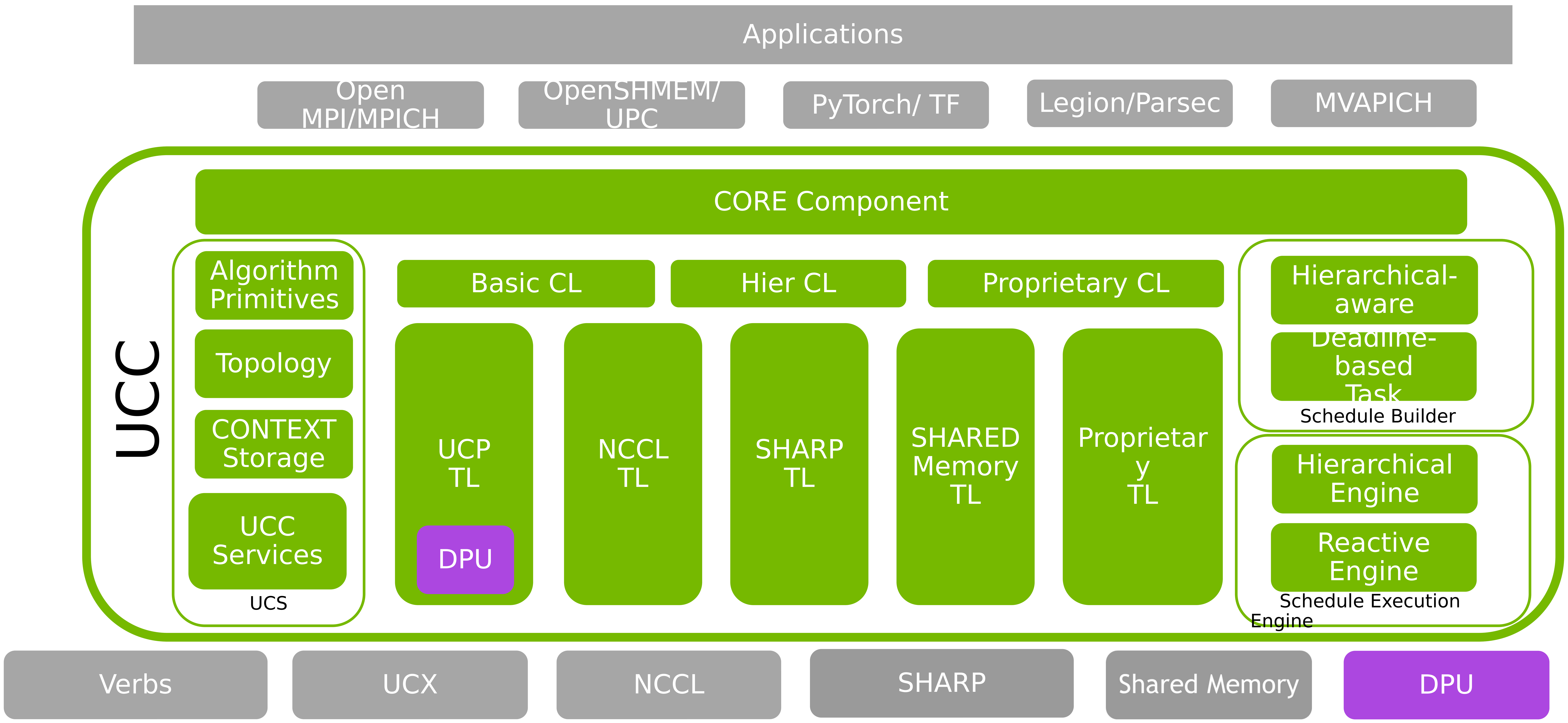
UNIFIED COLLECTIVE COMMUNICATION (UCC)

Goals

- Unified collective stack for HPC and DL/ML workloads
  - Tunable for latency, bandwidth, throughput
- Unified collective stack for software and different networks
- Unify parallelism and concurrency
  - Concurrency – progress of a collective and the computation
  - Parallelism – progress of many independent collectives
- Unify execution models for CPU, GPU, and DPU collectives
  - Extended to supports offloading model for DPUs
- Extensible
  - Modular API and new collective algorithms can be implemented

UNIFIED COLLECTIVE COMMUNICATION (UCC)

Architecture



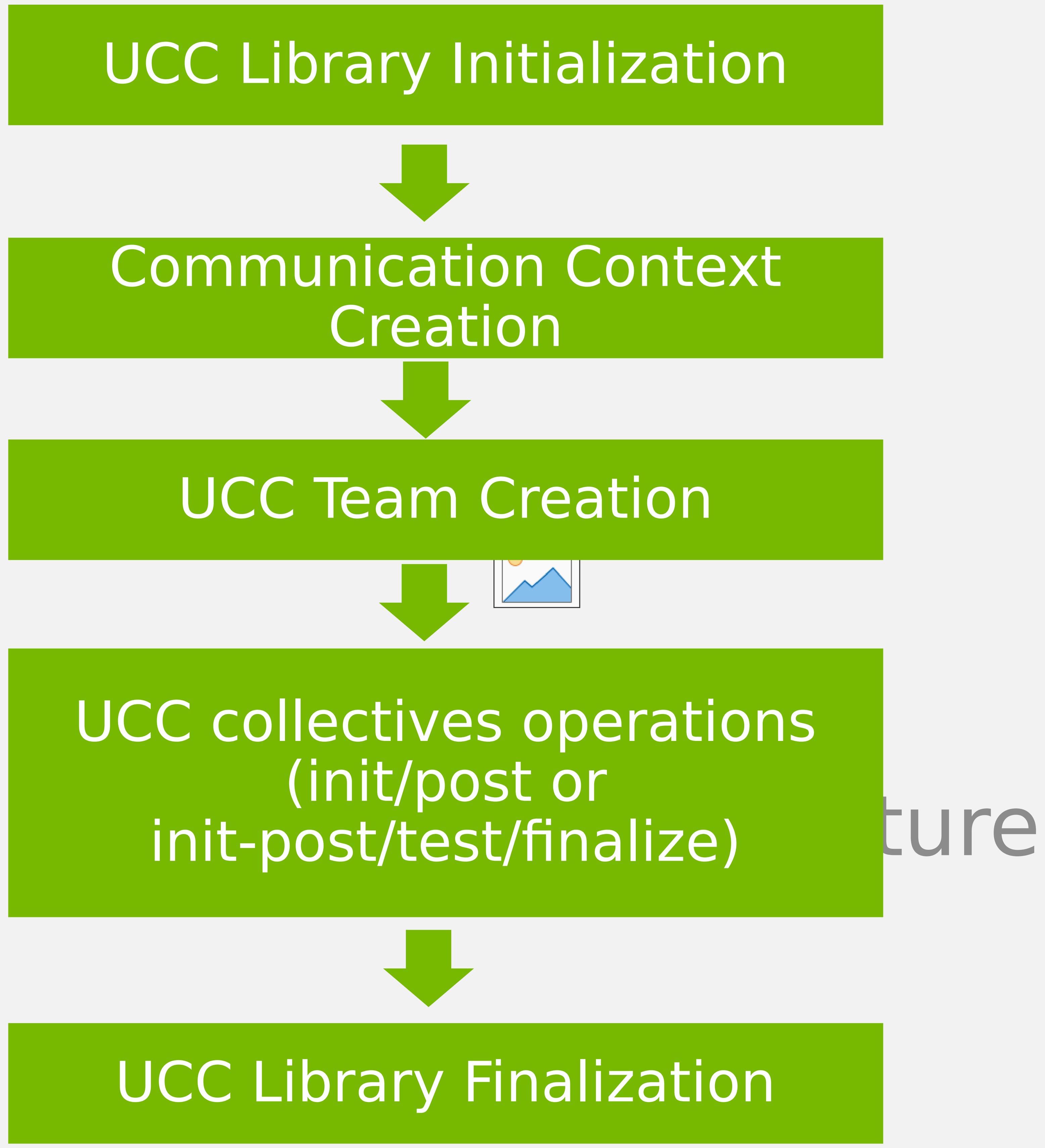
UCC CONCEPTS & CODE FLOW

UCC KEY CONCEPTS

- Abstractions for Resources
  - Collective Library
  - Communication Context
  - Teams
- Collective Operations
- Properties of Operations

UCC CODE FLOW

- Library Initialization
- Communication Context
- Team
- UCC collective operation
- Library Finalization



UCC LIBRARY

- Object that encapsulate resources
- Initialization and finalization routines
  - UCC operations should be invoked in between
- Parameters of the library
  - Thread model
  - Collective types
  - Reduction types
  - Synchronization types

- UCC API: ucc_init(), ucc_init_version(), ucc_finalize()

```c
/**
 *  @ingroup UCC_LIB
 *
 *  @brief The @ref ucc_init initializes the UCC library.
 *
 *  @param [in]  params    user provided parameters to customize the library functionality
 *  @param [in]  config    UCC configuration descriptor allocated through
 *                         @ref ucc_lib_config_read "ucc_config_read()" routine.
 *  @param [out] lib_p     UCC library handle
 *
 *  @parblock
 *
 *  @b Description
 *
 *  A local operation to initialize and allocate the resources for the UCC
 *  operations. The parameters passed using the ucc_lib_params_t and
 *  @ref ucc_lib_config_h structures will customize and select the functionality of the
 *  UCC library. The library can be customized for its interaction with the user
 *  threads, types of collective operations, and reductions supported.
 *  On success, the library object will be created and ucc_status_t will return
 *  UCC_OK. On error, the library object will not be created and corresponding
 *  error code as defined by @ref ucc_status_t is returned.
 *
 *  @endparblock
 *
 *  @return Error code as defined by @ref ucc_status_t
 */

static inline ucc_status_t ucc_init(const ucc_lib_params_t *params,
                                    const ucc_lib_config_h config,
                                    ucc_lib_h *lib_p)
{
    return ucc_init_version(UCC_API_MAJOR, UCC_API_MINOR, params, config,
                            lib_p);
}
```

```c
/**
 *
 *  @ingroup UCC_LIB_INIT_DT
 *
 *  @brief Structure representing the parameters to customize the library
 *
 *  @parblock
 *
 *  Description
 *
 *  @ref ucc_lib_params_t defines the parameters that can be used to customize
 *  the library. The bits in "mask" bit array is defined by @ref
 *  ucc_lib_params_field, which correspond to fields in structure @ref
 *  ucc_lib_params_t. The valid fields of the structure is specified by the
 *  setting the bit to "1" in the bit-array "mask". When bits corresponding to
 *  the fields is not set, the fields are not defined.
 *
 *  @endparblock
 *
 */
typedef struct ucc_lib_params {
    uint64_t                mask;
    ucc_thread_mode_t       thread_mode;
    uint64_t                coll_types;
    uint64_t                reduction_types;
    ucc_coll_sync_type_t    sync_type;
    ucc_reduction_wrapper_t reduction_wrapper;
} ucc_lib_params_t;
```

COMMUNICATION CONTEXT

- Object to encapsulate local resource and express network parallelism
- Local resources
  - E.g. Injection queues or network endpoints
- Can be used to specify affinity
  - Can be bound to a specific core, socket, accelerator
- Contexts can be created for:
  - Processes - E.g. single MPI process can have multiple contexts
  - Threads – E.g. a thread can be coupled with multiple contexts
  - Tasks
- Controls resource sharing
  - EXCLUSIVE
    - E.g. single team
  - SHARED
    - E.g. shared across teams
- UCC API: ucc_context_create()

UCC TEAMS

- Encapsulates the resources required for group of operations
- Created by processes, threads or tasks
  - Each process/thread passes a context (local resource object)
- Properties
  - Synchronization Model
    - On_Entry, On_Exit or On_Both
  - Ordering
    - Must invoke collective in the same order (e.g MPI)
    - TensorFlow and persistent collectives can be invoked in different orders
  - Datatype
    - Can be customized for contiguous, strided or non-contiguous data types
- UCC API: ucc_team_create_post()
  - Non-blocking call
  - Only one active call at any given instance
  - It is a collective operation

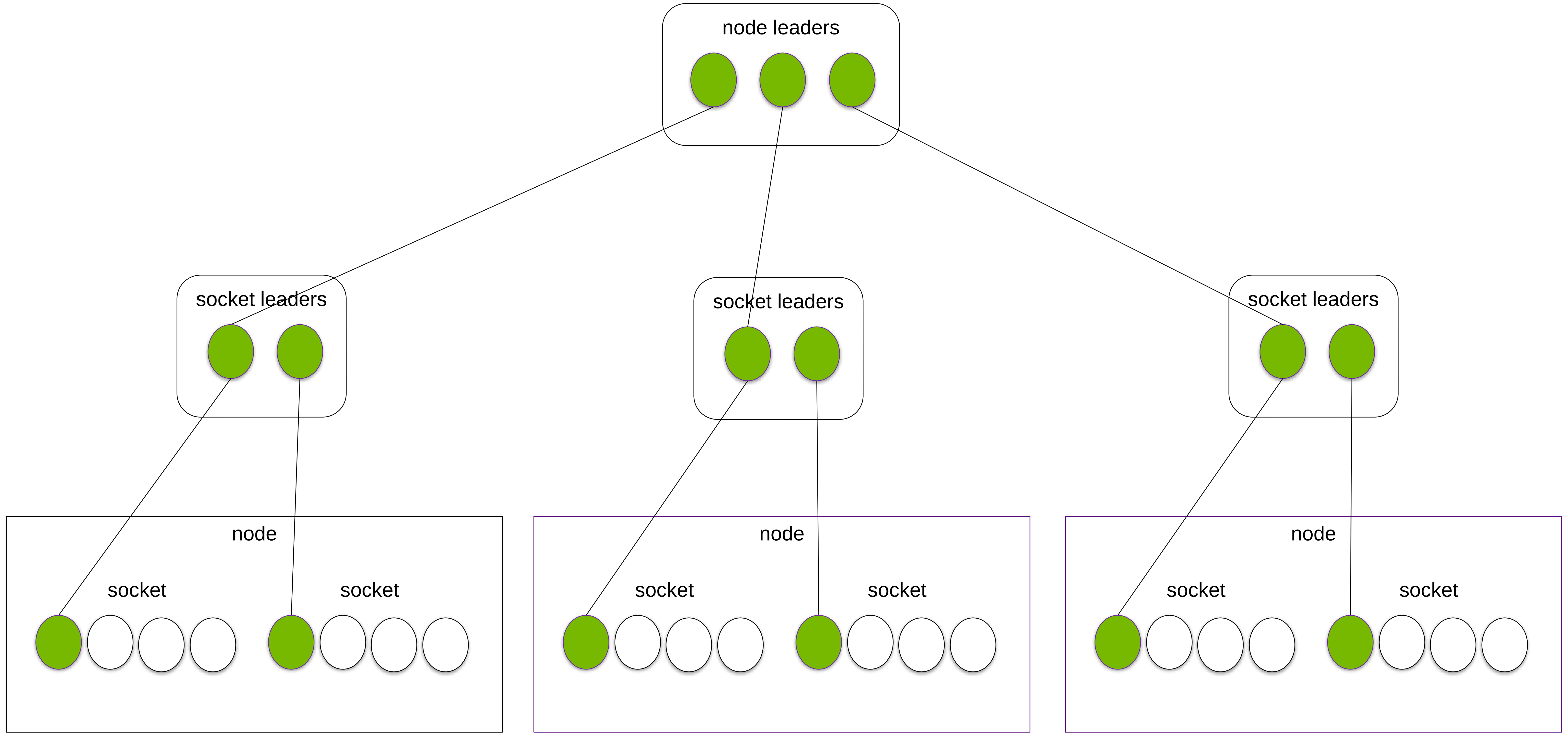
```
/**
 *  @ingroup UCC_TEAM
 *
 *  @brief The routine is a method to create the team.
 *
 *  @param
  [in]  contexts           Communication contexts abstracting the resources
 *  @param
  [in]  num_contexts       Number of contexts passed for the create operation
 *  @param  [in]  team_params        User defined configurations for the team
 *  @param  [out] new_team           Team handle
 *
 *  @parblock
 *
 *  @b Description
 *
 *  @ref ucc_team_create_post is a nonblocking collective operation to create
 *  the team handle. It takes in parameters ucc_context_h and ucc_team_params_t.
 *  The ucc_team_params_t provides user configuration to customize the team and,
 *  ucc_context_h provides the resources for the team and collectives.
 *  The routine returns immediately after posting the operation with the
 *  new team handle. However, the team handle is not ready for posting
 *  the collective operation. ucc_team_create_test operation is used to learn
 *  the status of the new team handle. On error, the team handle will not
 *  be created and corresponding error code as defined by @ref ucc_status_t is
 *  returned.
 *
 *  @endparblock
 *
 *  @return Error code as defined by @ref ucc_status_t
 */
ucc_status_t ucc_team_create_post(ucc_context_h *contexts,
                                  uint32_t num_contexts,
                                  const ucc_team_params_t *team_params,
                                  ucc_team_h *new_team);
```

Click icon to add picture

```
typedef struct ucc_team_params {
    uint64_t                mask;
    ucc_post_ordering_t     ordering;
    uint64_t                outstanding_colls;
    uint64_t                ep;
    uint64_t                *ep_list;
    ucc_ep_range_type_t     ep_range;
    uint64_t                team_size;
    ucc_coll_sync_type_t    sync_type;
    ucc_team_oob_coll_t     oob;
    ucc_team_p2p_conn_t     p2p_conn;
    ucc_mem_map_params_t    mem_params;
    ucc_ep_map_t            ep_map;
    uint64_t                id;
} ucc_team_params_t;
```

HIERARCHICAL TEAMS

Example of subgrouping



UCC COLLECTIVE OPERATIONS

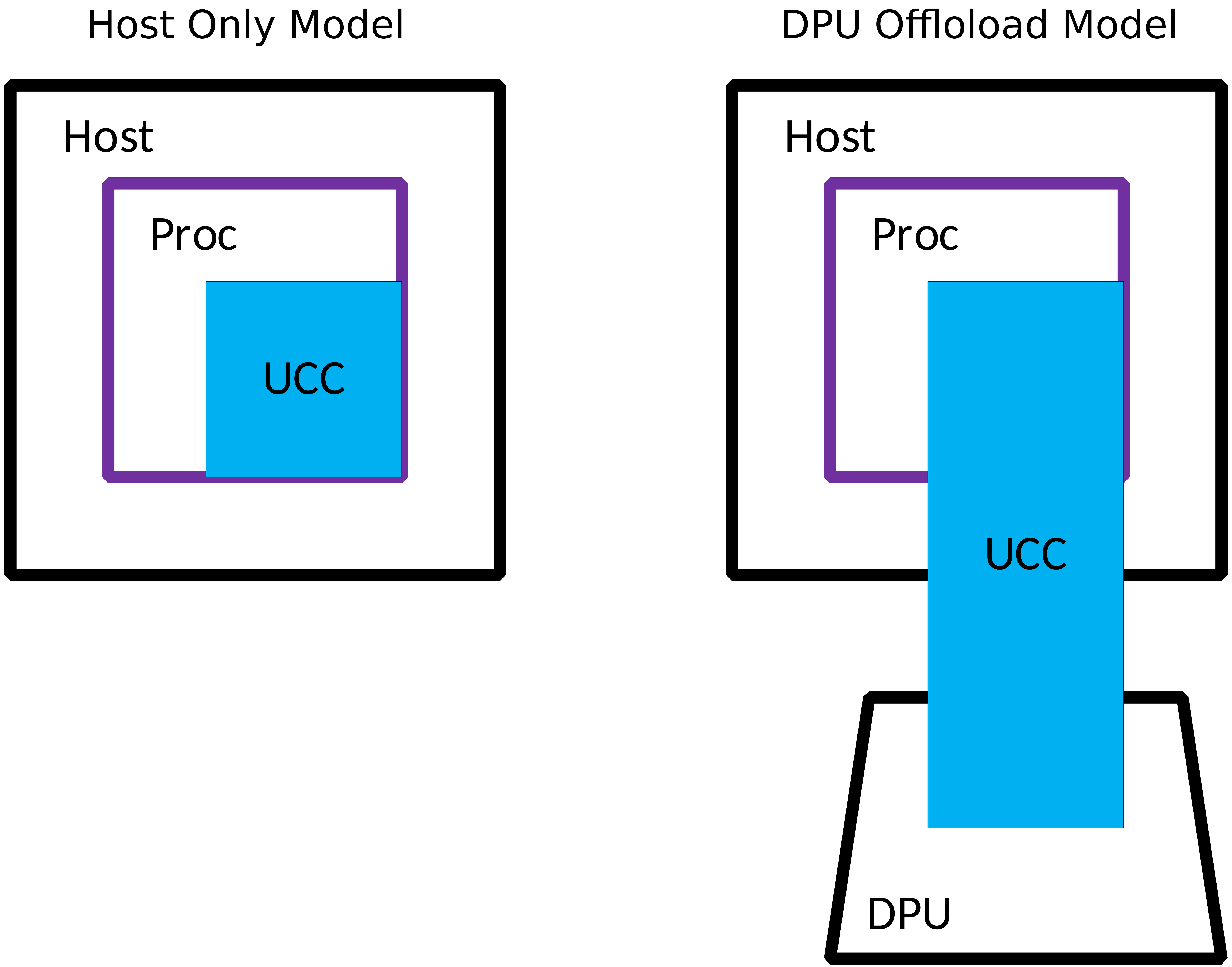
Building blocks

- Collective operations : ucc_collective_init(…) and ucc_collective_init_and_post(…)
  - Local operations: ucc_collective_post, test, and finalize
- Initialize with ucc_collective_init(…)
  - Initializes the resources required for a particular collective operation, but does not post the operation
- Completion
  - The test routine provides the status
- Finalize
  - Releases the resources for the collective operation represented by the request
  - The post and wait operations are invalid after finalize
- Implementing collectives:
  - Blocking collectives:
    - Can be implemented with Init_and_post and test+finalize
  - Persistent Collectives:
    - Can be implemented using the building blocks - init, post, test, and finalize
  - Split-Phase
    - Can be implemented with Init_and_post and test+finalize

```
ucc_status_t ucc_collective_init(ucc_coll_op_args_t* coll_args,
                                 ucc_coll_req_h* request, ucc_team_h team);

ucc_status_t ucc_collective_post(ucc_coll_req_h request);

ucc_status_t ucc_collective_init_and_post(ucc_coll_op_args_t* coll_args,
                                          ucc_coll_req_h* request,
                                          ucc_team_h team);

ucc_status_t ucc_collective_finalize(ucc_coll_req_h request);
```

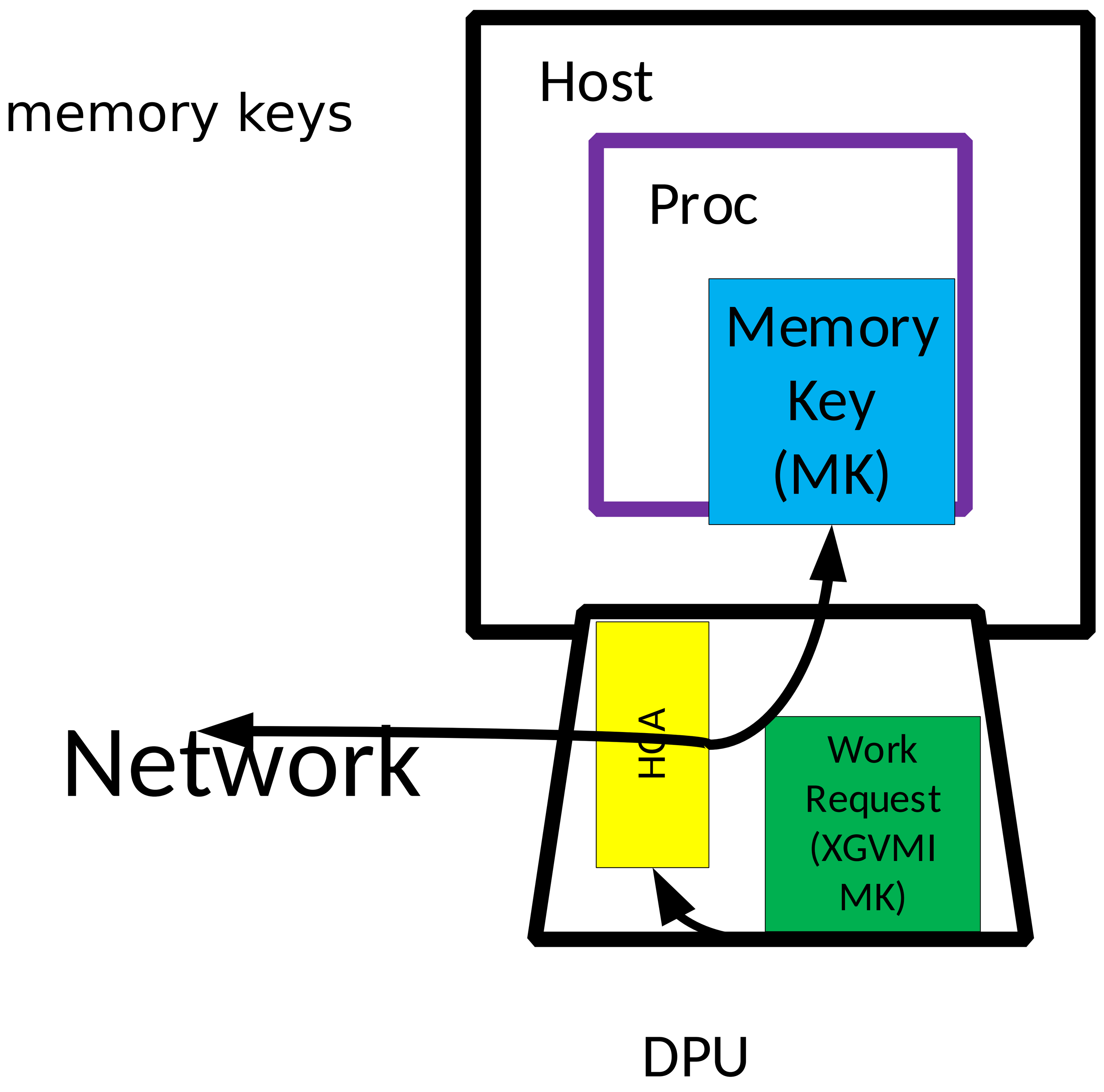
OFFLOADING CONCEPTS

UCC DPU OFFLOAD MODEL



DESIGN CONSIDERATIONS

- DPU is an asynchronous agent
- Number of host cores is on the order of 10X those of the DPU – need work sharing
- DPU cores less powerful computationally with respect to the host compute engines
- DPU have targeted acceleration engines
- Host and DPU need to be “in sync”
- BlueField enhancements
  - Work requests can be posted on behalf of memory that is host-resident – Cross-GVMI memory keys
  - Some optimized data paths between the host and the BlueField - GGA



OFFLOAD AND LIBRARY INFRASTRUCTURE

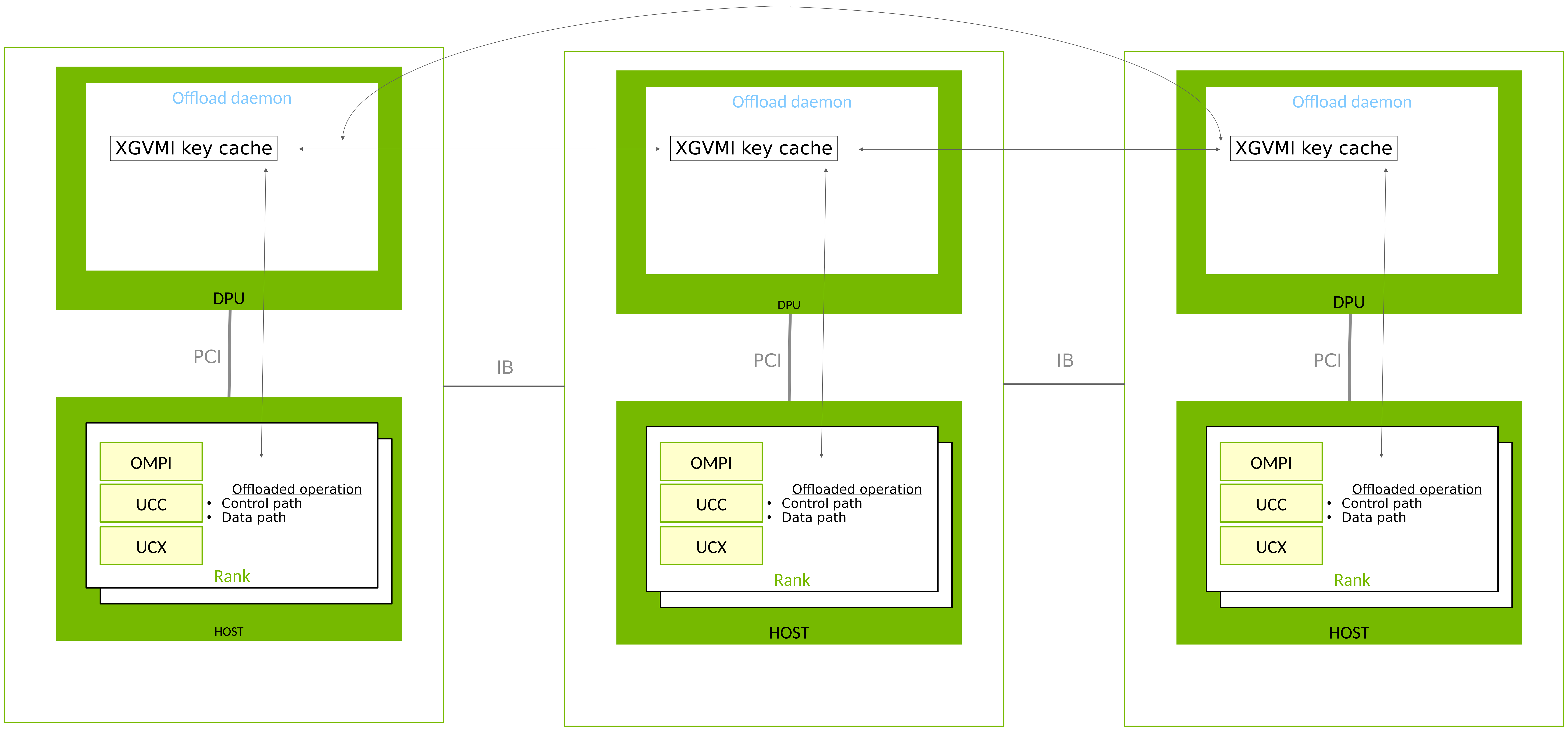
TERMINOLOGY

- Library/building block
  - A set of APIs and the library code that goes with it, not an instantiation
  - Does NOT refer to how I use it in an implementation
- Daemon/service process
  - An executable binary, based on building blocks, that can be executed on the DPU
  - Multiple service processes can run on a single DPU
- Service/service API: everything necessary to extend an existing software component (e.g., UCC) to benefit from DPU offloading
- Local/remote DPU
  - Local DPU: DPU with a PCI physical connection to the core where the rank is running
  - Remote DPU: DPU with a IB-only physical connection to the core where the rank is running
- Endpoint (EP): handle from the communication layer to initiate a communication (send, receive, one-sided)

OFFLOAD AND LIBRARY INFRASTRUCTURE

- Goals
  - Provide an infrastructure for the offloading of operations to DPUs
  - Provide generic APIs, not limited to a programming language
  - Currently used in conjunction with Open MPI + UCC for the offloading of MPI collectives
- Model relevant to this presentation
  - An offloading service is running on the DPUs
  - For offloaded collectives, MPI ranks connect to the service on the DPU
  - The offloaded algorithm is split between the MPI/UCC component running on the host; and the service on the DPU
- Key concepts
  - Offloading engine
  - Execution contexts
  - Events and notifications
  - Endpoint cache (for X-GVMI)
- What is needed to offload an operation?
  - Identify what piece of the algorithm is supposed to run on the DPUs and on the hosts
  - Extend the host code to initiate the offloading to the DPU
  - Coordinate the flow of the algorithm between the hosts and DPUs using control notifications
  - Rely on XGVMI for efficient data path

ARCHITECTURE OVERVIEW



OFFLOADING ENGINE

- Required on both DPUs and hosts for the implementation of a service
- Meant to separate offloading service; in our context, only one required
- Option to use a configuration file to specify details about the platform where to run the job
- Highest level handle
  - Enable the creation of one or more execution contexts
  - Provides a special execution context for self
  - A default notification system, for example for local events
  - A buddy buffer system for efficient memory management
- Two functions
  - dpu_offload_status_t offload_engine_init(offloading_engine_t **engine);
  - void offload_engine_fini(offloading_engine_t **engine);

EXECUTION CONTEXT

- Execution contexts provide all the capabilities for interactions with another execution context
- In charge of bootstrapping, by ensuring
  - Two execution contexts connect to each other
  - All capabilities related to interaction between execution contexts are initialized and available to users
- Based on client/server concepts to simplify the design of new solutions
- Example
  - A server execution context is running on the DPU and client execution contexts running in the context of MPI ranks connects to it
  - A series of server/client execution contexts are running on the DPUs to enable the cross-connection of service processes
- APIs
  - execution_context_t *client_init(offloading_engine_t *engine, init_params_t *init_params);
  - void client_fini(execution_context_t **ctx);
  - execution_context_t *server_init(offloading_engine_t *engine, init_params_t *init_params);
  - void server_fini(execution_context_t **ctx);
  - Get the current phase of the bootstrapping process

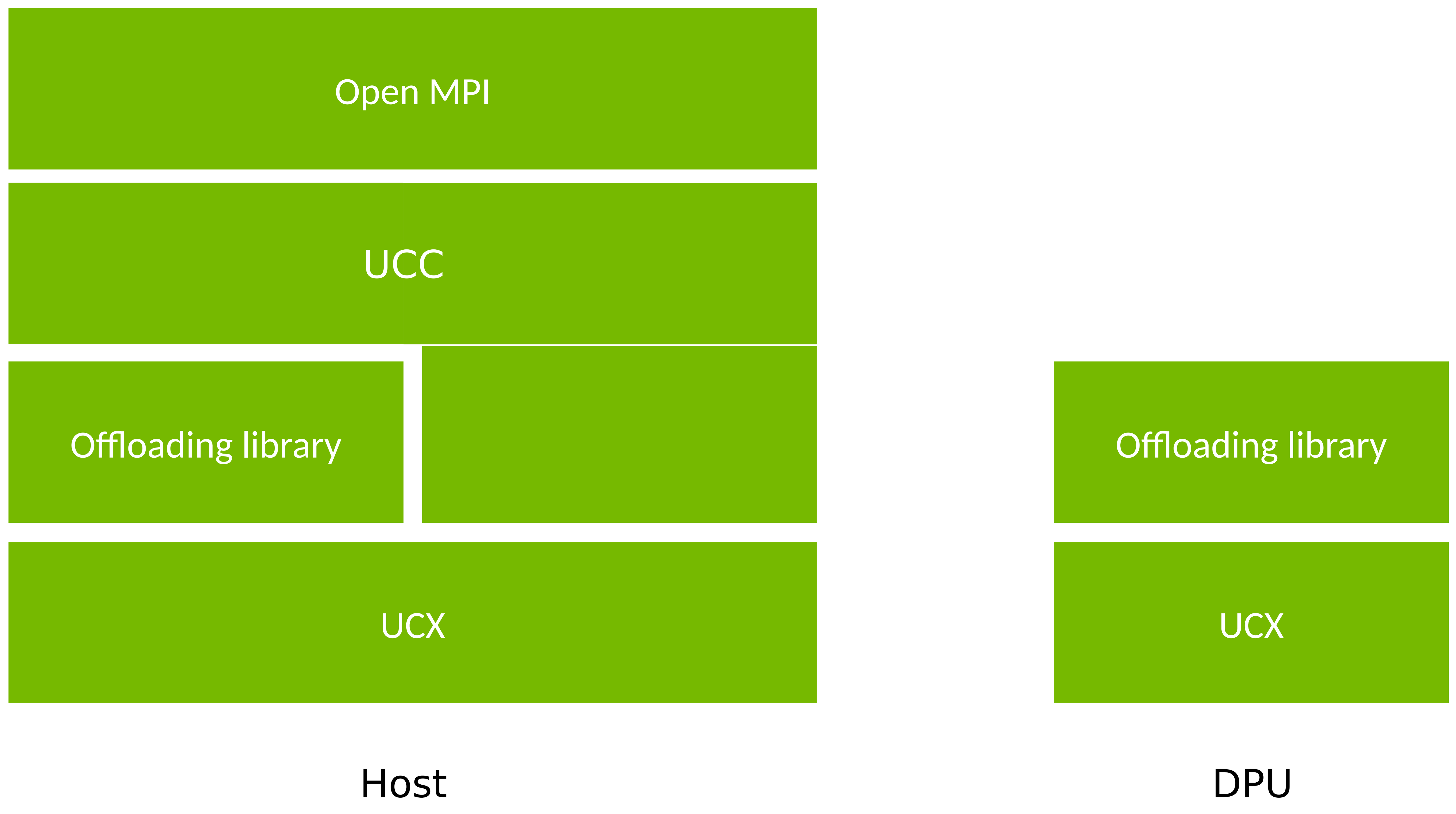
  GET_ECONTEXT_BOOTSTRAPING_PHASE(execution_context)
- Bootstrapping is asynchronous and does not require any action from users other than progress
- Once bootstrapping completed, the type of the execution context (client or server) is less relevant
- More details in the documentation

EVENTS & NOTIFICATIONS

- Mainly used to implement the control path between hosts and service processes, as well as between service processes
- Available from an execution context
  - All execution contexts provide an event/notification system
  - On the receive side
    1. Choose a unique identifier for your custom notification, called a *notification type*
    2. Register a unique handler for the notification type
  - On the sender side
    1. Get an event
    2. Optionally set the payload
    3. Get the destination information
    4. Emit the event
- By default
  - All events are added to a list for progress
  - When an event completes, it is implicitly returned
  - If the event is associated to a payload, the payload is released
  - When a handler is invoked upon reception of a notification, the buffer is only valid throughout the execution of the said handler and then released
- Other features
  - Manual management of events' lifecycles (not put on the ongoing list, not implicitly returned)
  - Possible to specify pool of memories to efficiently use payload buffers with events and notification handlers
- See documentation for more details

OVERVIEW OF THE SOFTWARE STACK

- Offloading libraries
  - A set of shared libraries (.so files) with their headers
  - A binary to instantiate the offloading service on the DPU
- A modified version of UCC that support offloading for (some) MPI collectives
- A modified version of UCX that support XGVMI



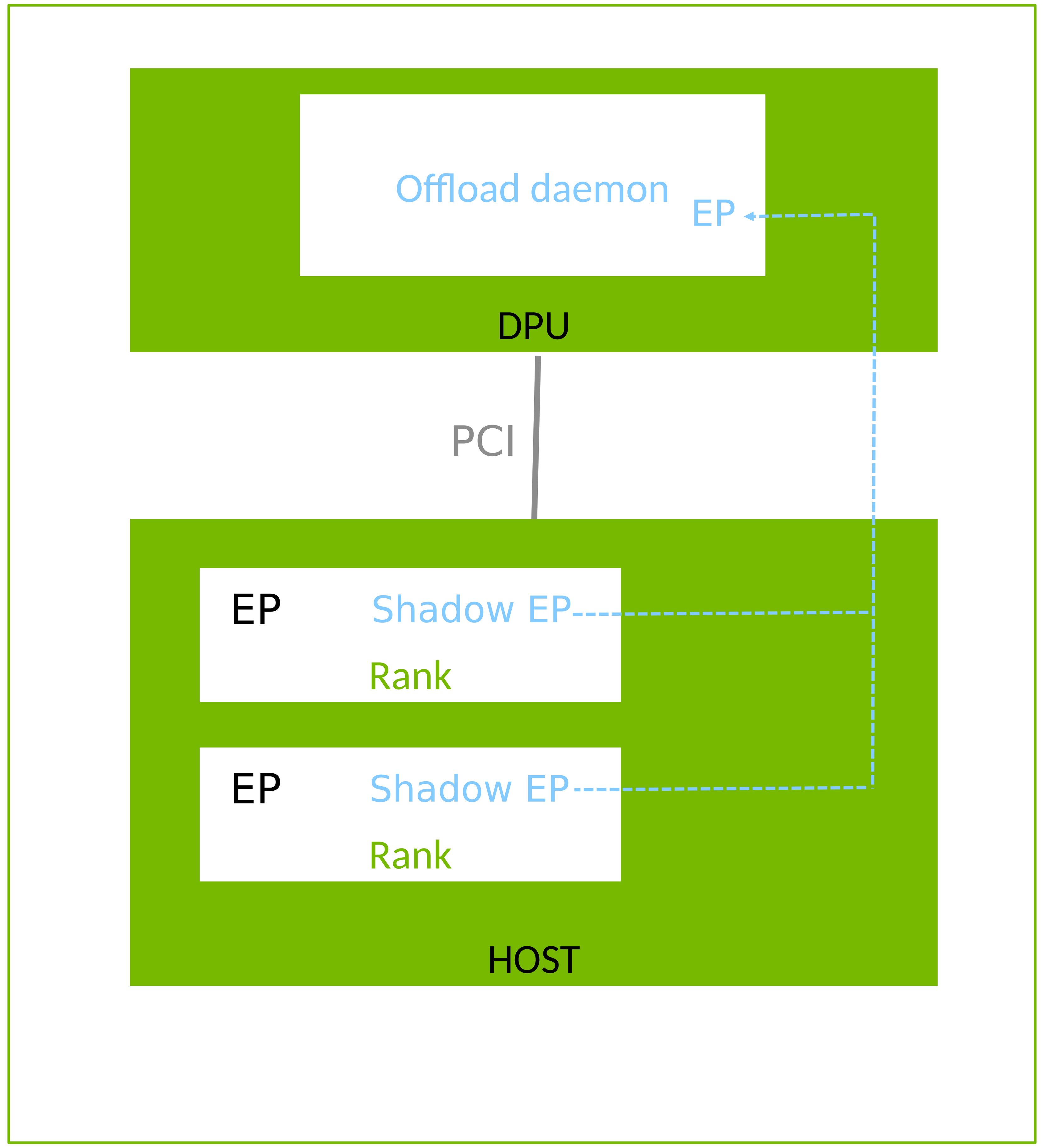
DPUS & SHADOW ENDPOINTS

What are the local DPUs associated to a MPI rank

- Reminder: all operations are in the context of a group; in the remaining of the slides, rank means “rank in a group”
- Need to know what are the local DPUs for all ranks in the operation. No limitation on communication patterns that collective developers can use
  - rank-to-rank
  - rank-to-DPU
  - DPU-to-DPU
  - DPU-to-rank
- Concept of ghost endpoints: All the data required to communicate with a local DPUs for a given rank
- Related functions for the implementation of offloaded operations:
  - Find the service process associated to a remote rank

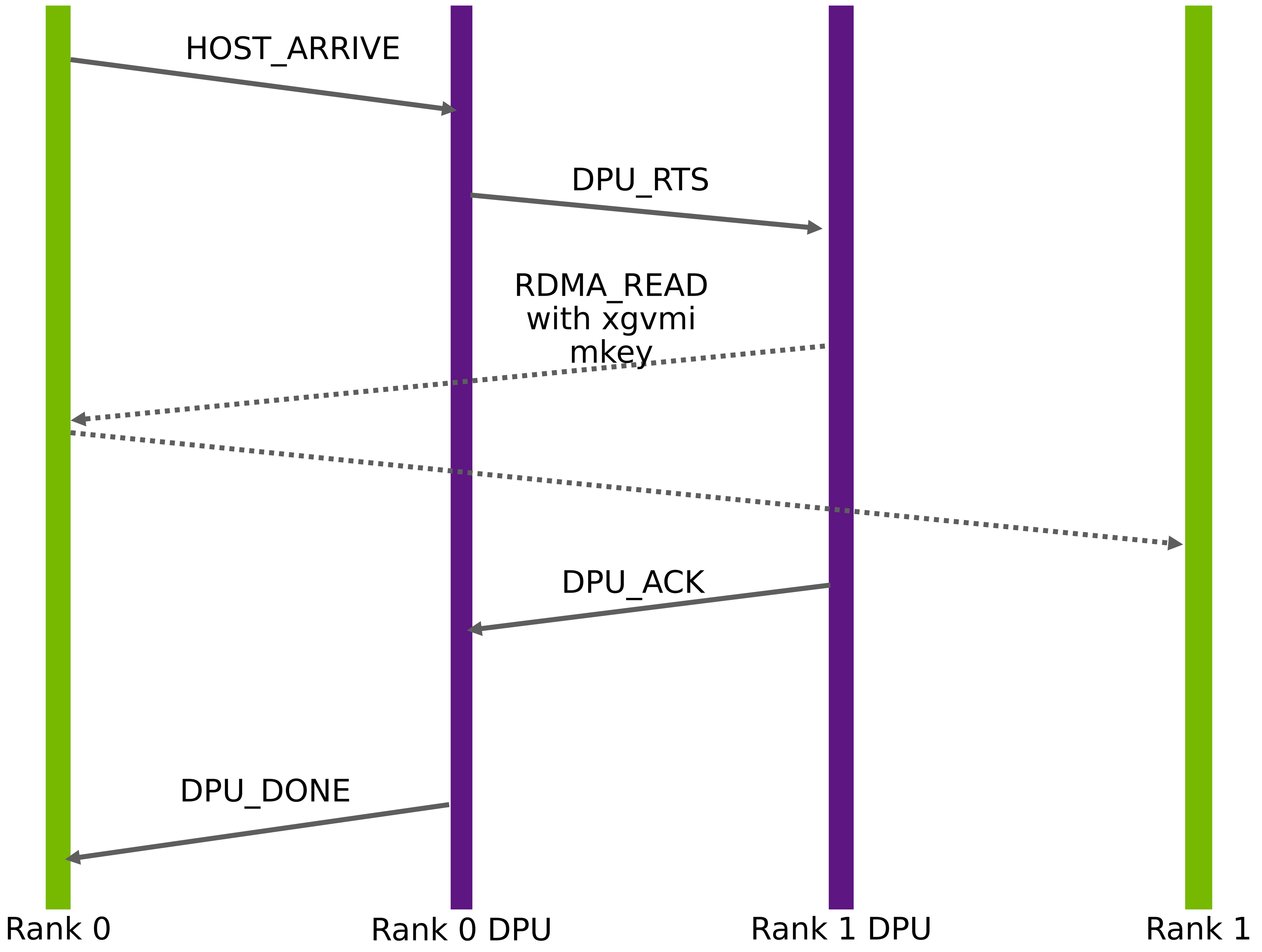
    get_sp_id_by_group_rank(engine, group_id, rank, service_proc_idx, &service_proc_id, &ev);
  - Find the endpoint for a service process

    get_sp_ep_by_id(engine, sp_id, sp_ep, &econtext_comm, &dest_id);
  - event_get(*ev_sys, *info, **ev)
  - event_channel_emit(**event, type, dest_ep, dest_id, *ctx)



OFFLOAD DATA EXCHANGE – PART OF A COLLECTIVE ALGORITHM

Sender: Rank 0 Receiver: Rank 1



NVIDIA