Stream Synchronous Communication in UCX

Akshay Venkatesh, Sreeram Potluri, Jim Dinan, and Hessam Mirsadeghi
Acknowledgement to Yossi Itigin for UCX API Discussions

CPU Versus Stream Synchronous Communication

GPU Coordinates Data Dependencies Without CPU Involvement

GPU Integrated Communication Libraries

	Stream Synchronous	Graph Synchronous	Kernel Triggered	Kernel Initiated	Implementation
NCCL	X	X			Proxy
NVSHMEM	X	X		X	Proxy or GIC
LibMP	X	X	X		GPUDirect Async
MPI	Proposed	Proposed	Partitioned		TBD

- Goals for today's session:
  - Discuss how to support GPU integrated communication (e.g. in Open MPI) on top of UCX
  - Discuss how to enable best possible performance for UCX
    - E.g. GPU SM integrated communication, GPUDirect Async, and other technologies

Agenda

CUDA Streams and Graphs

Lessons Learned from LibMP

MPI Accelerator Extensions

UCX Stream Synchronous Communication APIs

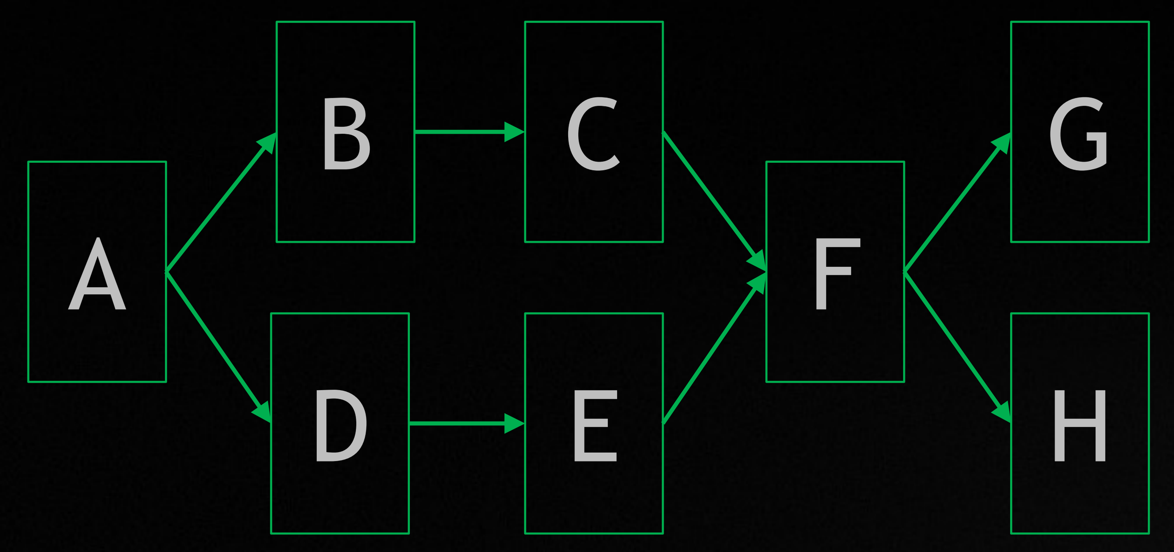
CUDA Streams and Graphs

Credit: Stephen Jones

CUDA STREAMS

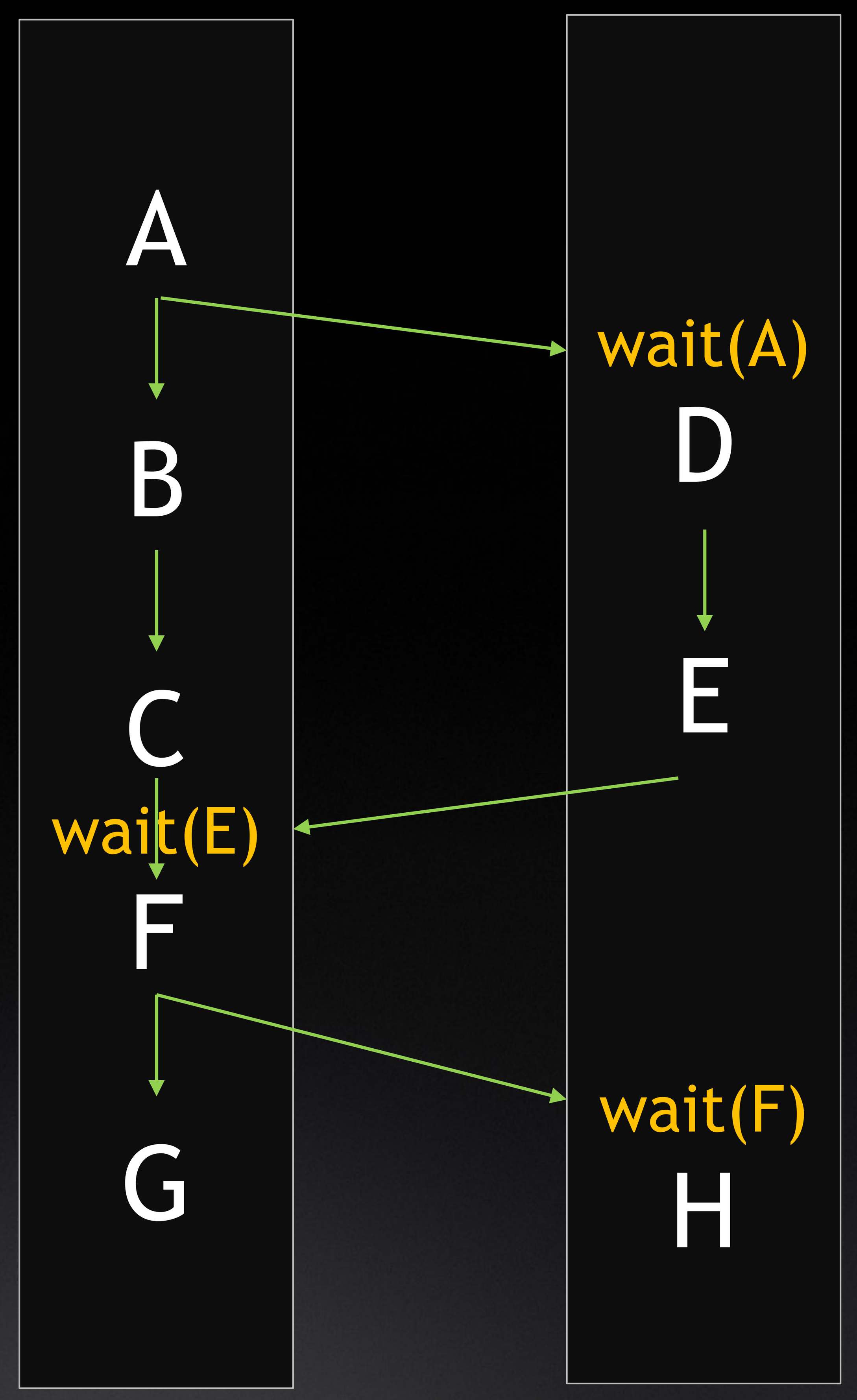
GPU Work Submission Queues

Dependency-based work



Dependencies **expressed** as CUDA streams

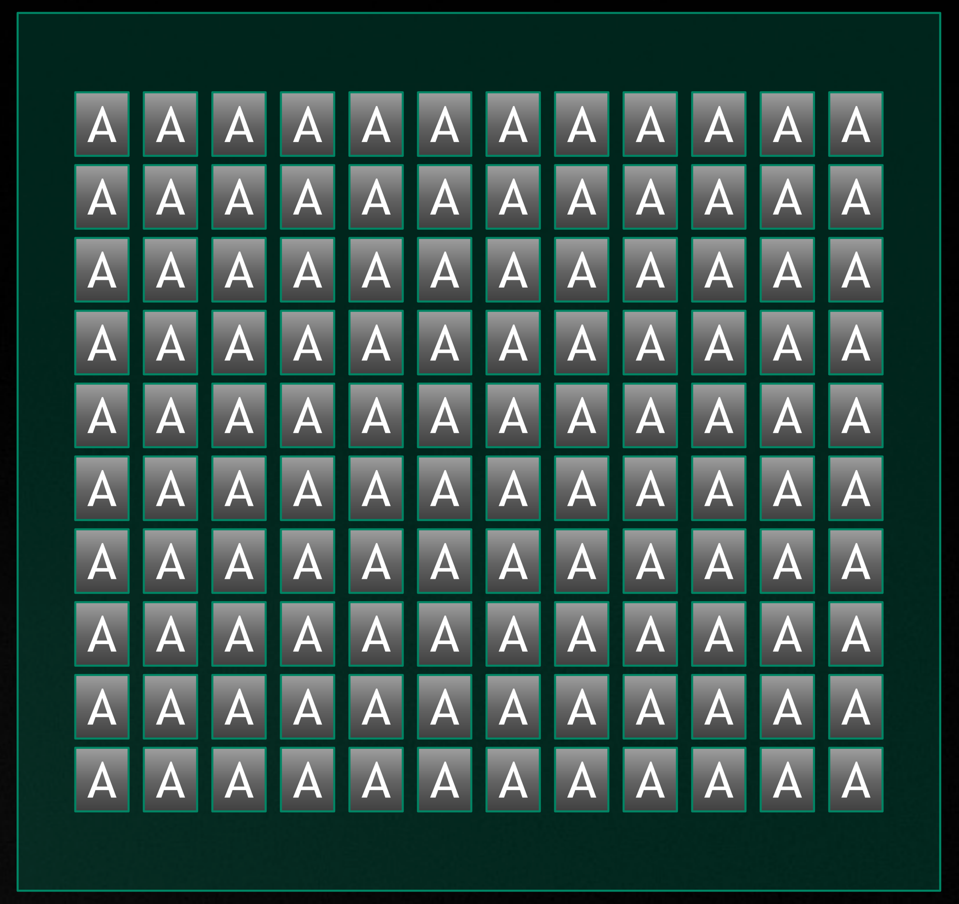
Streams have **implicit** submission-order dependencies



Stream 1

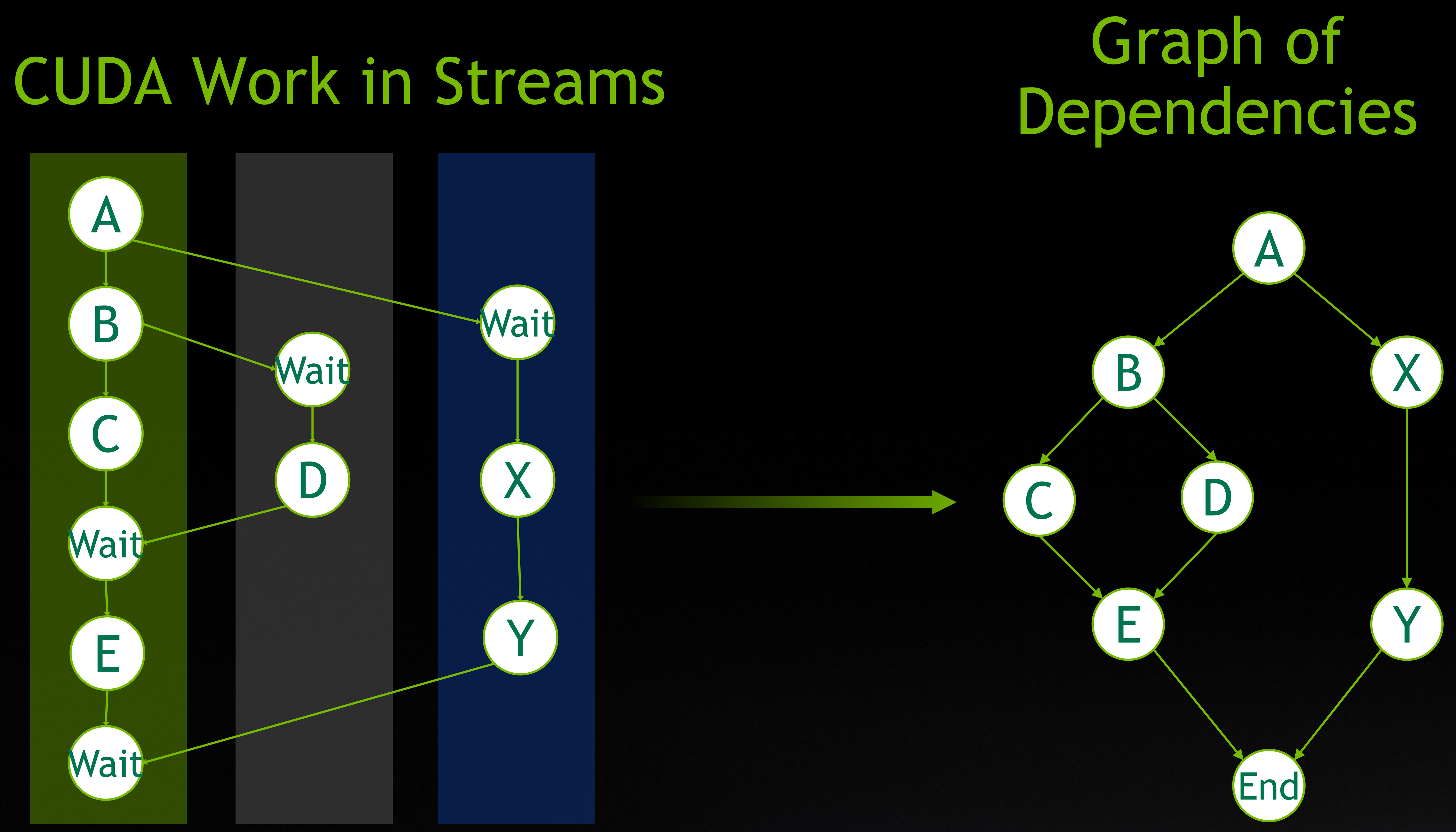
Stream 2

Hardware pops top of any available FIFO



CUDA GRAPHS

Optimize Workflows and Reduce Launch Overheads



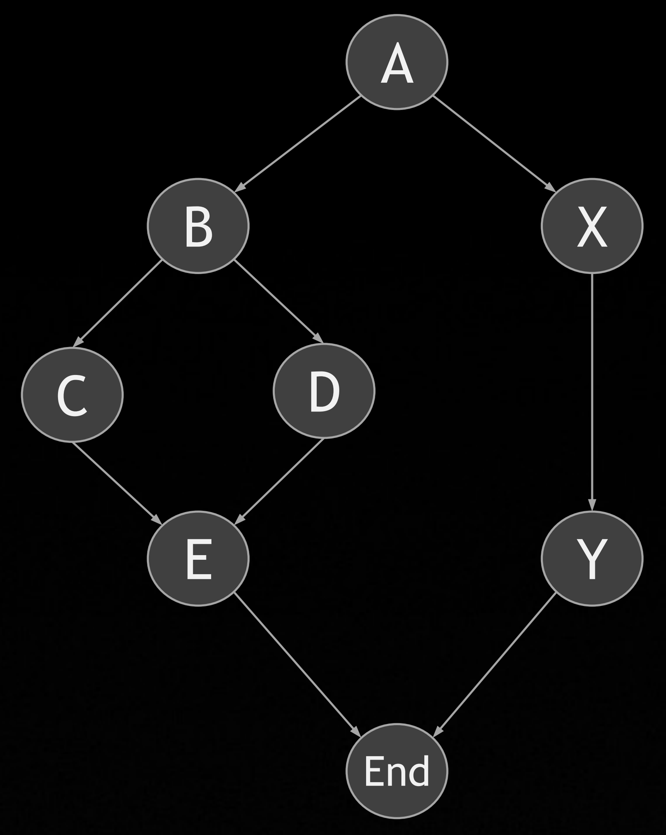
CUDA Graphs can be captured from streams (or explicitly constructed) and can be replayed multiple times

Graphs can reduce overheads:

- Launch multiple kernels with one operation (host overhead)
- Schedule work closer to GPU execution units (device overhead)

THREE-STAGE EXECUTION MODEL

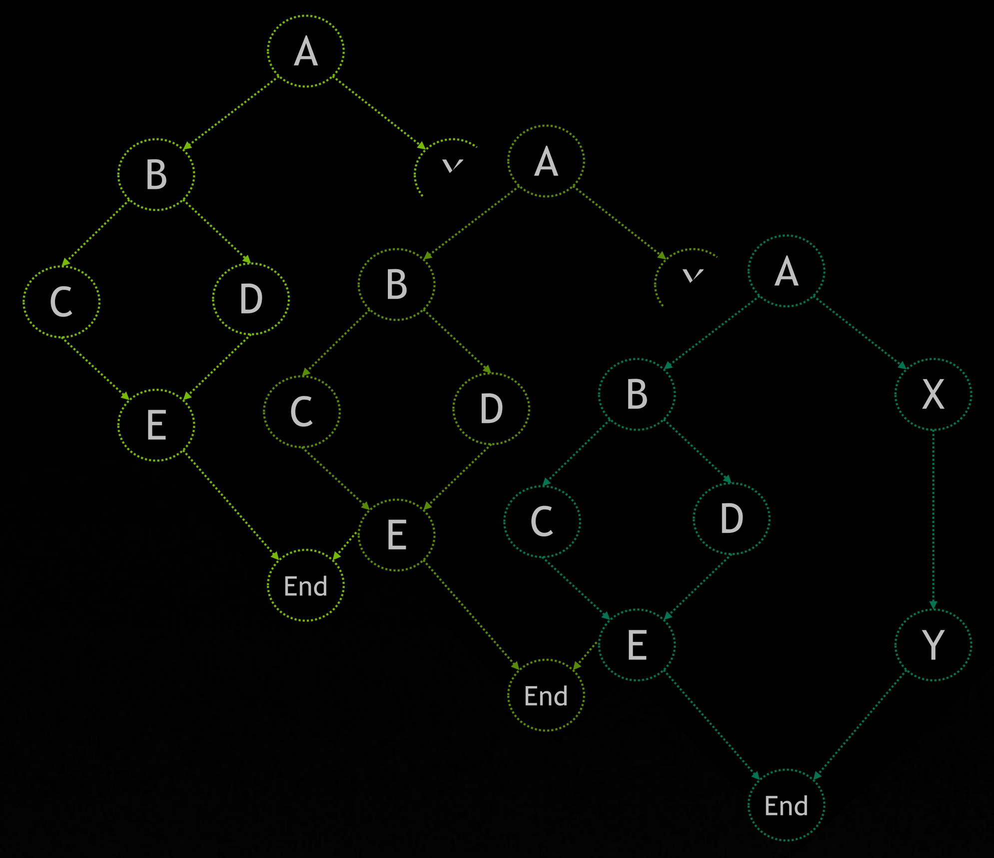
Define



Single Graph "Template"

Created in host code
or built up from libraries

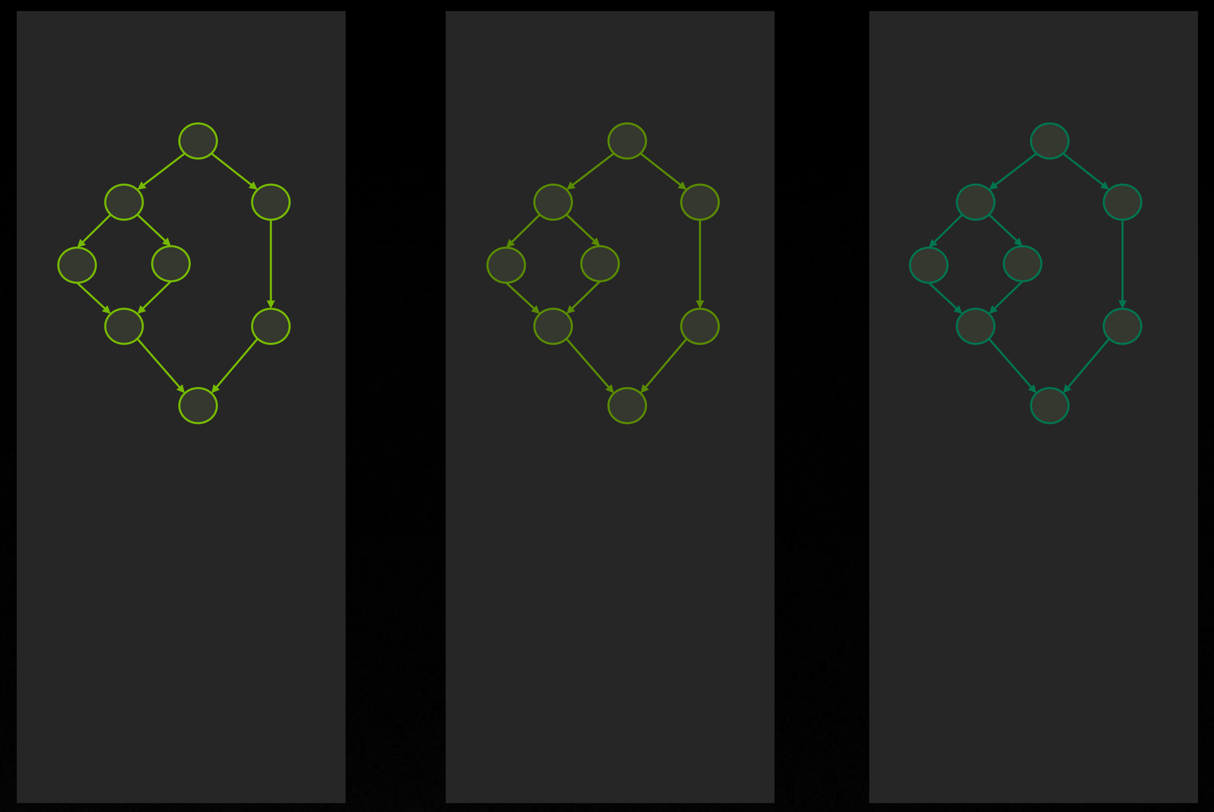
Instantiate



Multiple "Executable Graphs"

Snapshot of template
Sets up & initializes GPU
execution structures
(create once, run many times)

Execute



Executable Graphs
Running in CUDA Streams

Concurrency in graph
is not limited by stream

WORKFLOW EXECUTION OPTIMIZATIONS

Reducing System Overheads Around Short-Running Kernels

Breakdown of time spent during execution

WORKFLOW EXECUTION OPTIMIZATIONS

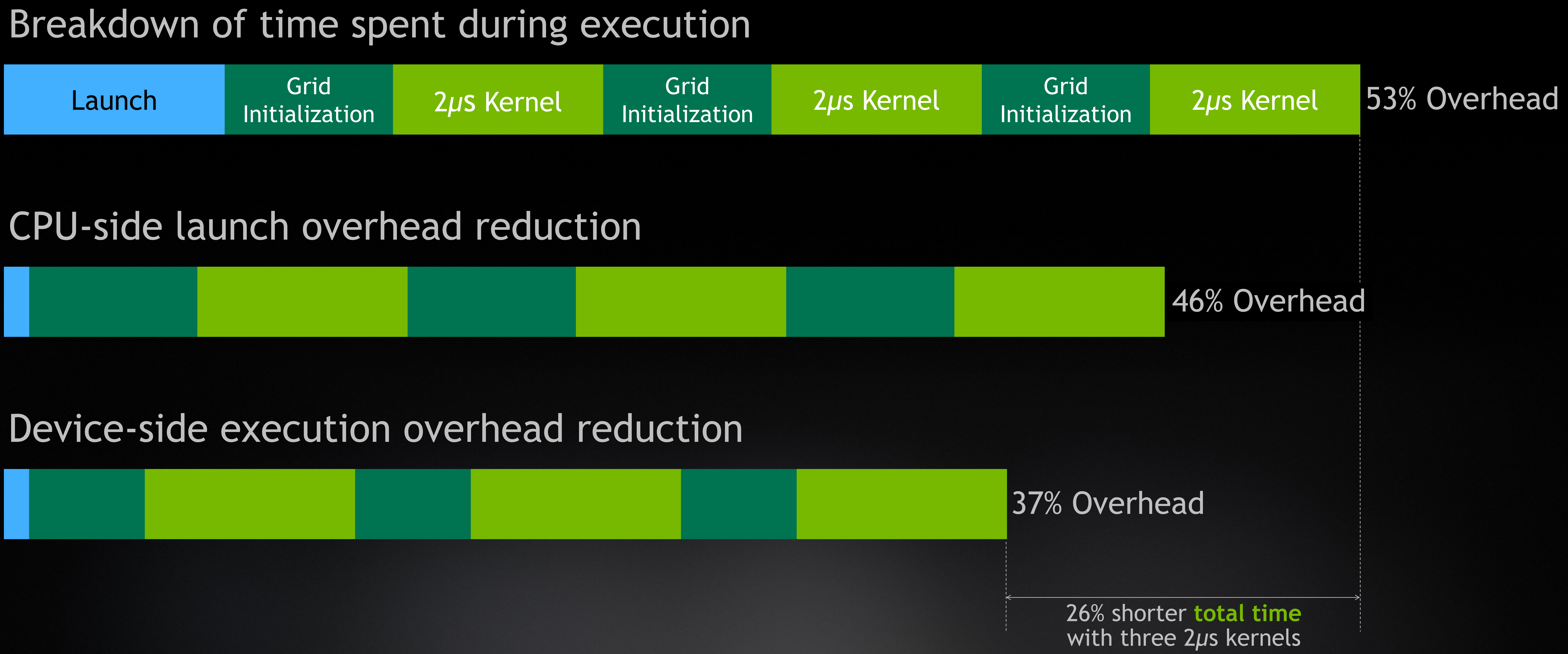
Reducing System Overheads Around Short-Running Kernels

Breakdown of time spent during execution

CPU-side launch overhead reduction

WORKFLOW EXECUTION OPTIMIZATIONS

Reducing System Overheads Around Short-Running Kernels



LibMP Lessons Learned

Credit: Pak Markthub and Davide Rosetti

LibMP Overview

https://github.com/gpudirect/libmp

- LibMP is a lightweight messaging library
  - Point-to-point and one-sided communications
- LibMP is a thin layer on top of GPUDirect Async
  - No tags, no wildcards, no data types
  - No synchronization protocol, e.g. back pressuring, credit exchange, ready to receive, etc.
- Intended to easily combine GPUDirect Async with GPUDirect RDMA
- Uses MPI as an out-of-band mechanism to bootstrap execution
  - MPI is not used during actual communication

CUDA Interaction With External Depenencies

- Interaction with external dependencies through flags in CUDA accessible memory

1. Kernels
    - Kernels can update and spin on flags
    - Blocks any dependent work in the CUDA stream/graph

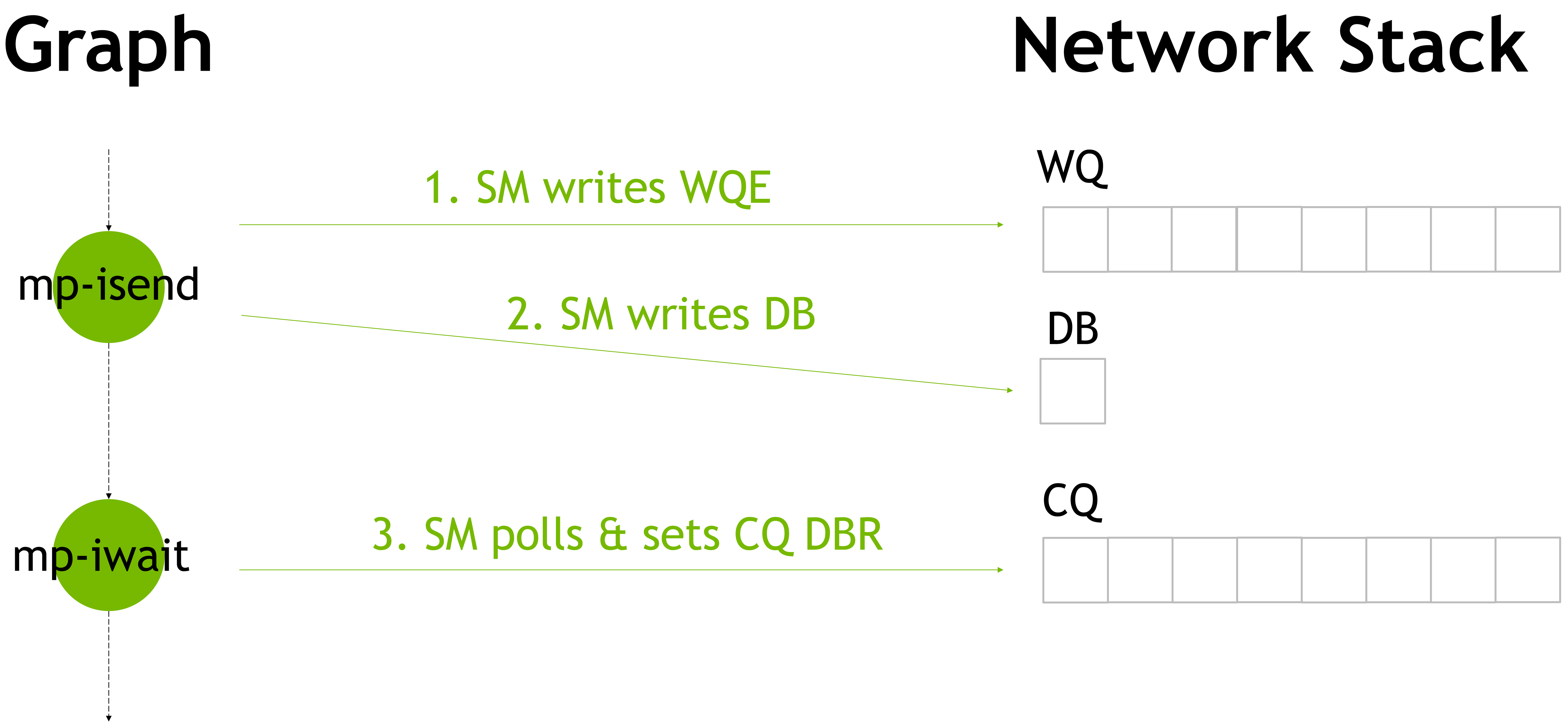
2. CUDA Memory Operations
    - cuStreamWriteValue32/64 – Update a flag in CUDA accessible memory when execution reaches this task
    - cuStreamWaitValue32/64 – Wait for a flag in CUDA accessible memory memory to satisfy condition
        - Conditions: Equal, greater-or-equal, AND, NOR

LibMP on Stream

NVIDIA

LibMP on Graph

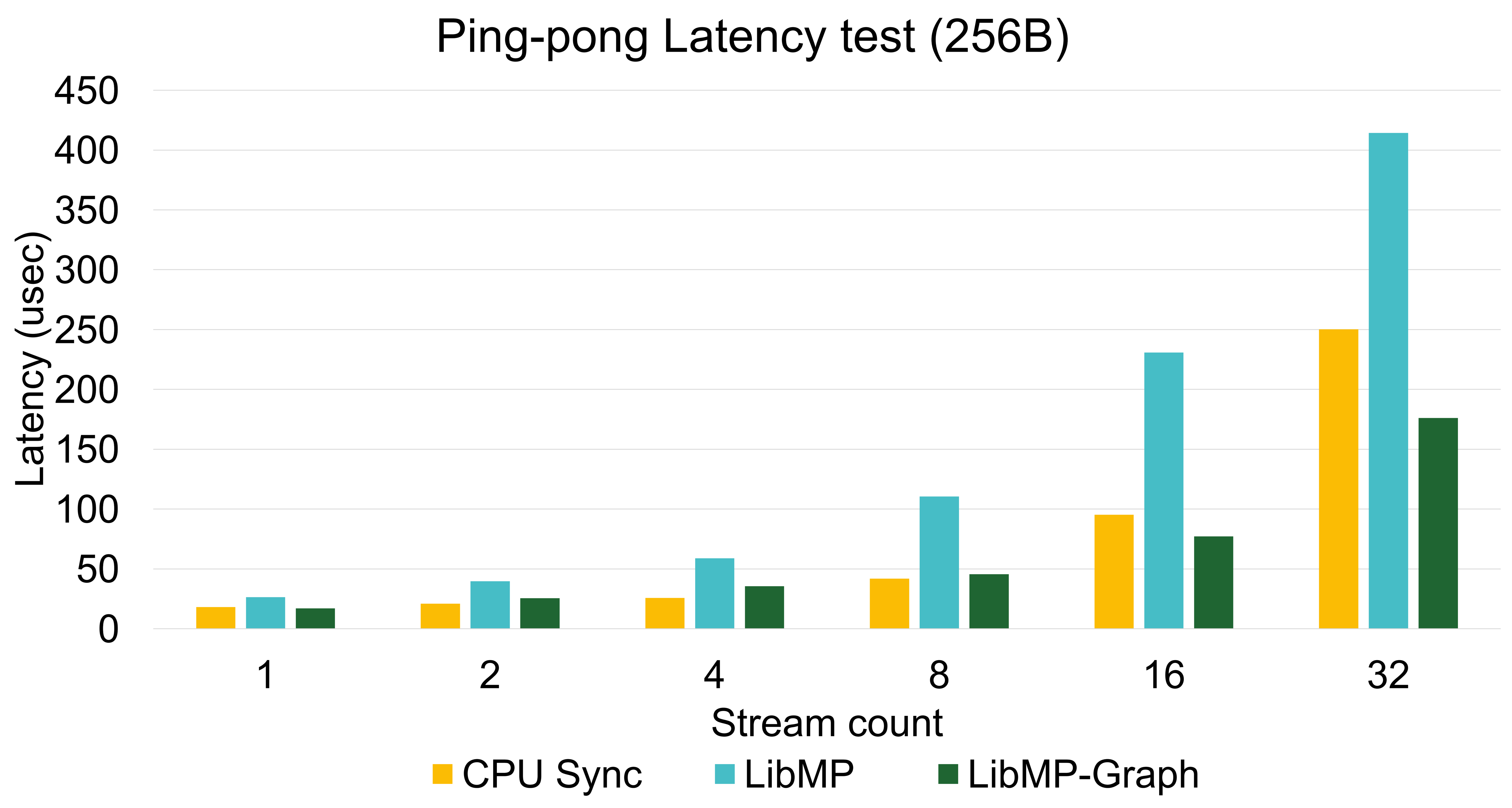
Prologue & Epilogue by GPU



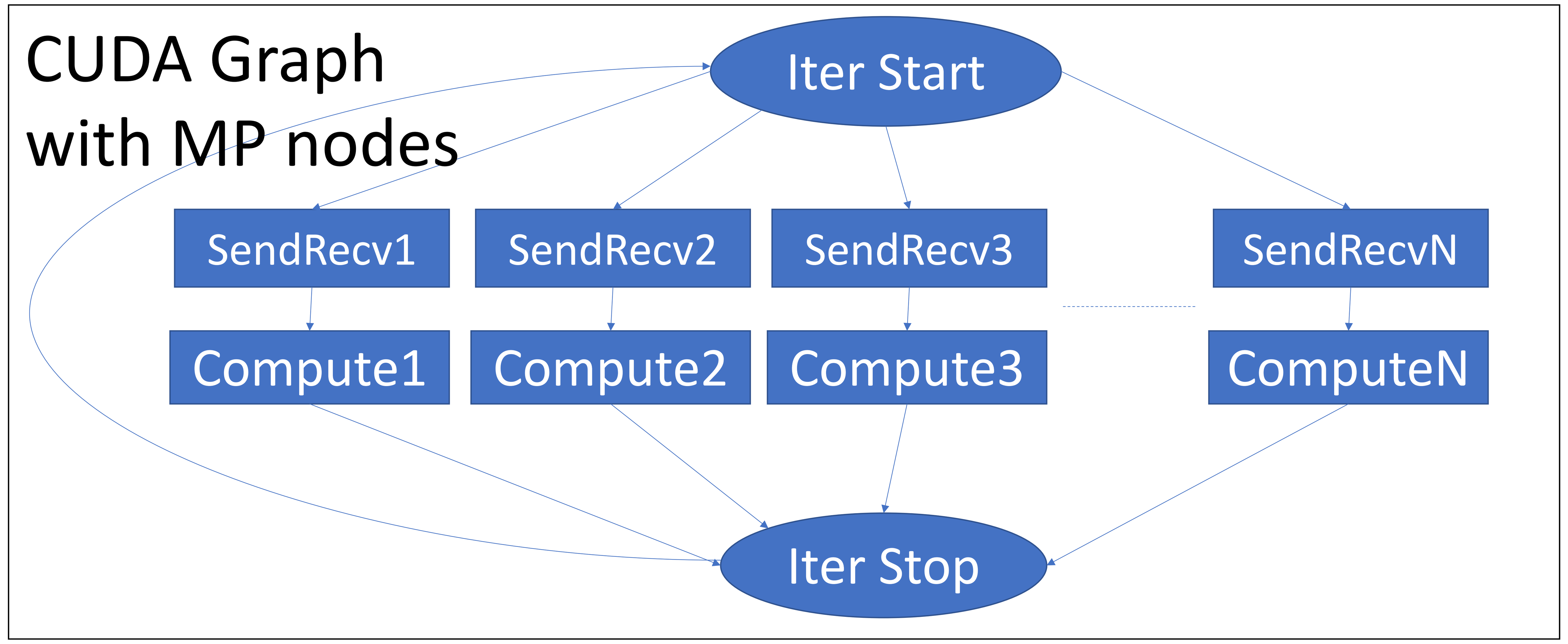
- Put WQ, CQ, DBR on GPU memory for better performance.
- Software stack is ready on Coral (P9).
  - Need patches on other systems.
- QP is incompatible with ibverbs.

LIBMP PERFORMANCE ANALYSIS

Multi-stream ping-pong benchmark



CUDA 11.0, DGX 1V



- CPU sync:
  - Communication from CPU, compute offloaded to GPU
- LibMP Stream:
  - KernelOps version used, not StreamMemOps
- LibMP+Graphs vs. LibMP: clear gains [35,67]%
- LibMP+Graphs vs. CPU sync: [-37,30]%
  - Gains from direct triggering via memory overwhelms the communication kernel invocation overheads

Lessons Learned

- Simple protocols enable efficient integration of communication with CUDA
  - Memory registration, matching, protocol progression, etc.
  - Some simplifications (e.g. no crediting, rendezvous, etc.) hard for applications to adopt

- Overheads from enqueueing communication must be,
  - Less than gains from directly triggering communication
  - Minimized by enqueueing in batches (e.g. batch memOps)
  - Hidden by overlapping with computation

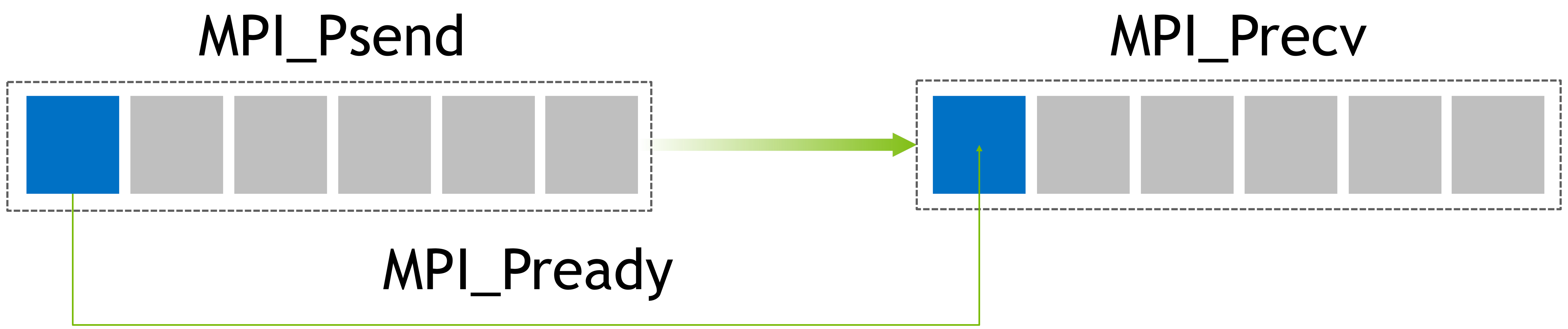
- MemOp parallelism is limited by the number of FIFOs assigned to the CUDA context
  - CUDA_DEVICE_MAX_CONNECTIONS - 1 to 32 (default is 8)

- Graphs can naturally resolve these issues:
  1. Protocols – Declaring "persistent" communication ahead of time
  2. Offloading overheads – Submitting graph to GPU as a single request
  3. Parallelism – Scheduling dependencies close to GPU where greater parallelism is possible

MPI Accelerator Extensions

Accelerator Triggered Communication

Using the MPI 4.0 Persistent Partitioned Communication API



Send/Recv data buffers are broken into equal-sized partitions

- MPI_Pready: mark partition as ready to send
- MPI_Parrived: query if partition has arrived

Partitioned operations match once in own matching space based on order of init calls

Kernel Triggered Communication Usage

Partitioned Neighbor Exchange

Host Code

```
MPI_Request req[2];
MPI_Prequest preq;
MPI_Psend_init(..., &req[0]);
MPI_Precv_init(..., &req[1]);
MPI_Prequest_create(req[0], MPI_INFO_NULL, &preq);
while (...) {
  MPI_Startall(2, req);
  kernel<<<..., s>>>(..., preq);
  MPI_Waitall(2, req);
}
MPI_Prequest_free(&preq);
MPI_Request_free(&req[0]);
MPI_Request_free(&req[1]);
```

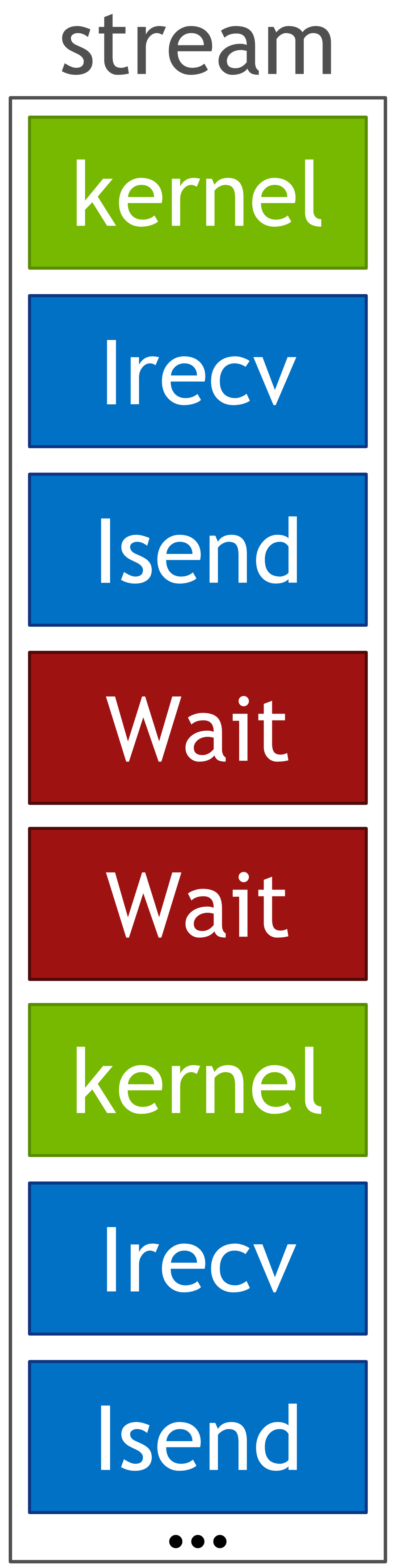
Device Code

```
__global__ kernel(..., MPI_Prequest preq) {
  int i = my_partition(...);
  // Compute and fill partition i
  // then mark i as ready
  MPI_Pready(i, preq);
}
```

MPI STREAM TRIGGERED API PROPOSAL

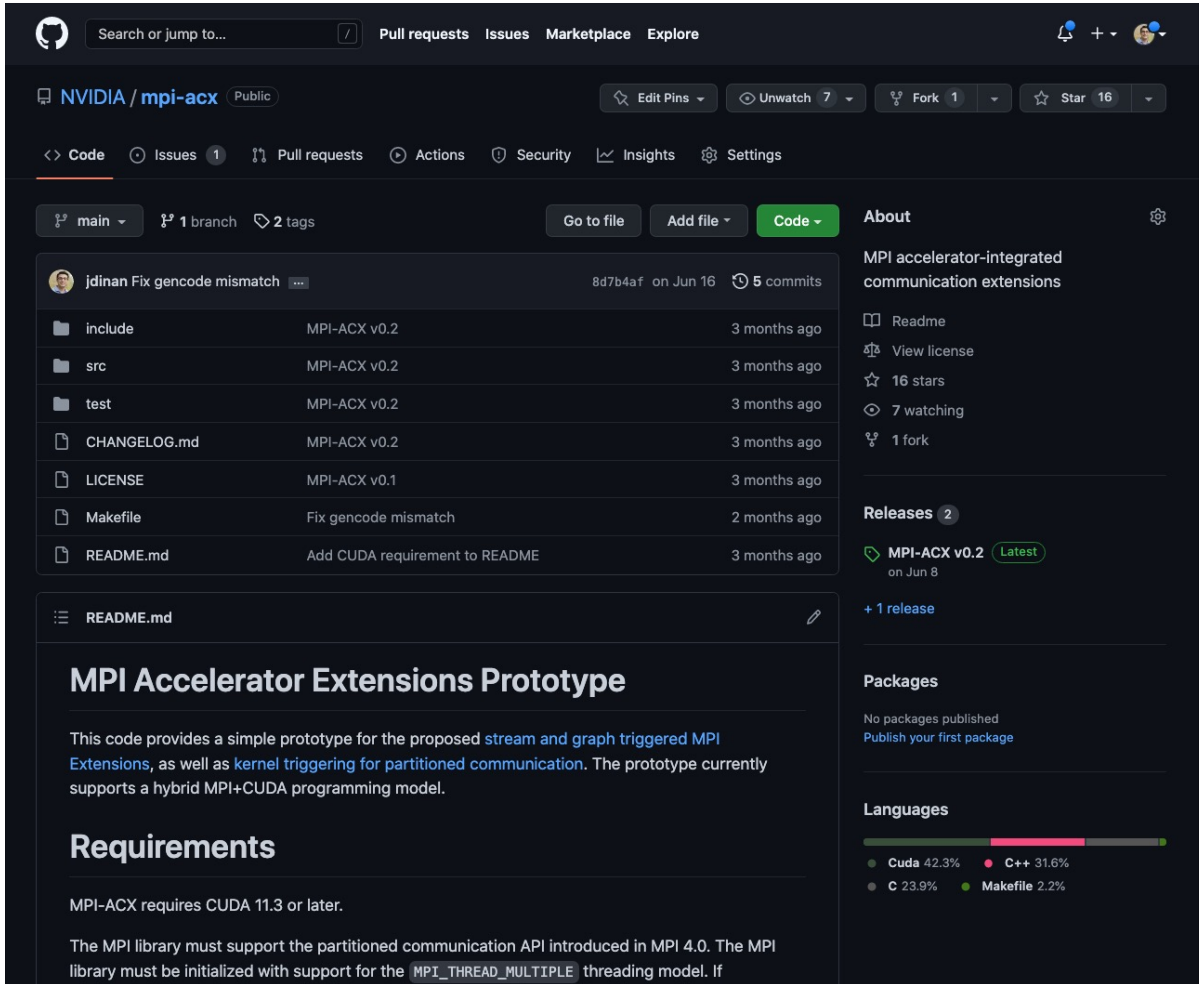
Simple Ring Exchange

```
MPI_Request send_req;
MPI_Request recv_req;

for (i = 0; i < NITER; i++) {
  if (i > 0) {
    MPI_Wait_enqueue(recv_req, &rstatus, MPI_CUDA_STREAM, stream);
    MPI_Wait_enqueue(send_req, &sstatus, MPI_CUDA_STREAM, stream);
  }

  kernel<<<..., stream>>>(send_buf, recv_buf, …);

  if (i < NITER - 1) {
    MPI_Irecv_enqueue(&recv_buf, …, recv_req, MPI_CUDA_STREAM, stream);
    MPI_Isend_enqueue(&send_buf, …, send_req, MPI_CUDA_STREAM, stream);
  }
}
```



MPI-ACX

Prototype of Stream, Graph, and Kernel Triggered Operations



- Proxy thread issues communication
  - Calls the triggered MPI function
  - Uses one flag per operation in host registered memory

- Supports:
  - Kernel triggered bindings for partitioned communication
  - Stream/graph synchronous Isend, Irecv, and Wait

https://github.com/NVIDIA/mpi-acx

Proposed UCX Stream Synchronous Communication APIs

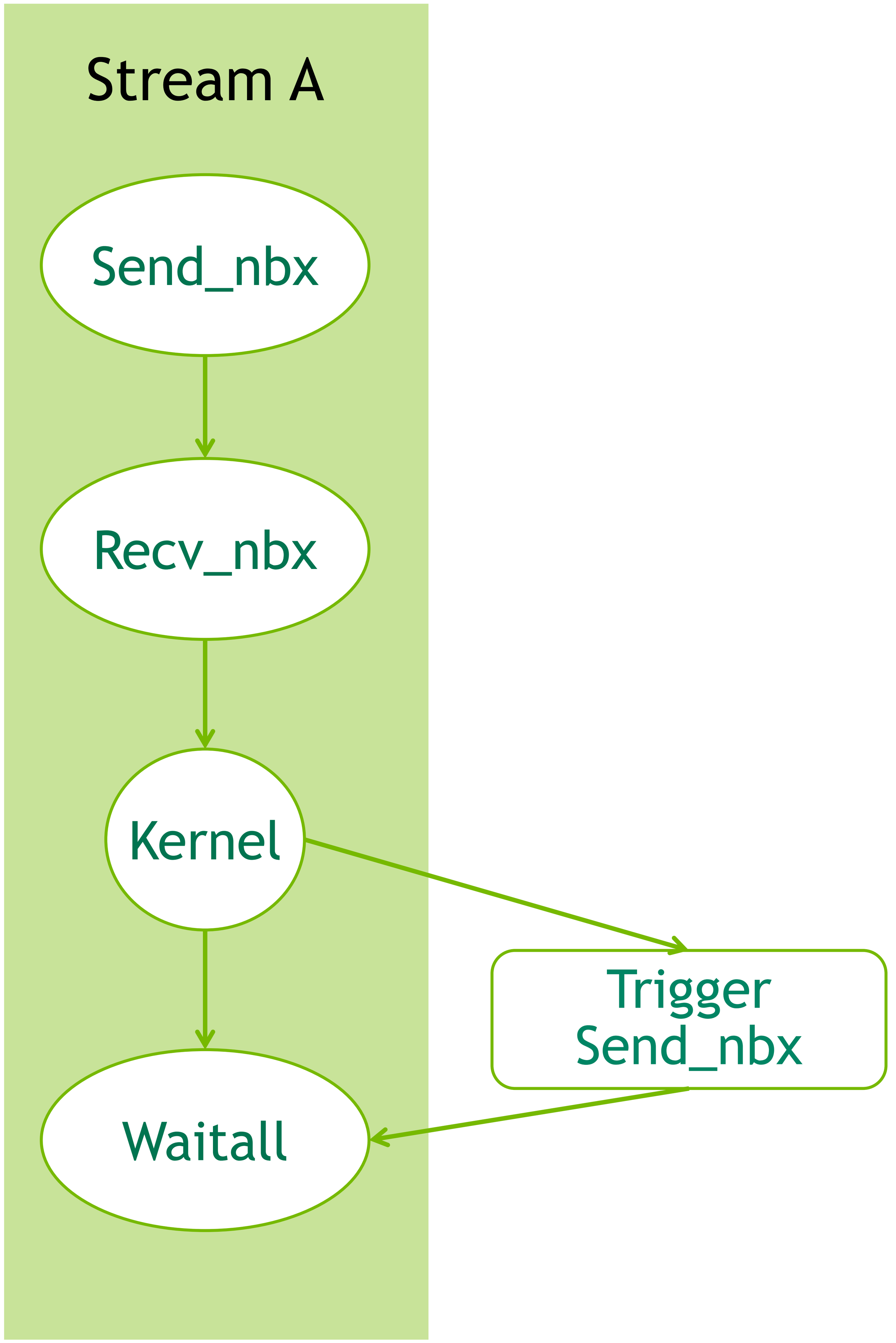
Goals

1. UCX support for stream *and graph* synchronous communication
   - Enqueue UCX operations on an external synchronization resource
   - Communication is not initiated until dependencies are met
   - Each stream is treated similarly to a separate thread

2. UCX support for kernel triggered communication
   - Calls to MPI_Pready / MPI_Parrived on GPU

3. Simplify protocols to enable efficient implementaitons
   - Do as much control/setup on CPU (e.g. memory mapping/registration) ahead of time
   - Data transfers / RDMA ops triggered or performed by CUDA

4. Progress should be external to the GPU
   - Accessing MPI internal state and advancing operations from GPU can be inefficient



UCP API Extension

Extend ucp_request_param_t with Condition

```
typedef enum {
    UCP_CONDITION_CATEGORY_STREAM,
    UCP_CONDITION_CATEGORY_GRAPH,
    UCP_CONDITION_CATEGORY_KERNEL_TRIGGERED
} ucp_condition_category_t;

typedef struct {
    ucp_condition_category_t category,
    void *context,
    ...
} ucp_condition_param_t;

typedef struct {
    uint32_t       op_attr_mask;
    uint32_t       flags;
    void          *request;
    …
    /* UCP condition to be met before
       initiating the operation */
    ucp_condition_h condition;
} ucp_request_param_t;
```

- ucp_condition_h links UCX op with an external task scheduler
  - Operation is performed after the given condition is satisfied
    - CUDA/HIP stream execution reaches a certain point
    - CUDA graph dependencies are satisfied
    - Kernel triggers the operation

- APIs to create UCP condition variables:

```
ucs_status_t ucp_create_condition(
    ucp_condition_h *condition,
    ucp_condition_param_t *param);

ucs_status_t ucp_destroy_condition(
    ucp_condition_h *condition);
```

- Condition context used as follows:
  - STREAM – Input the stream handle
  - GRAPH – Output graph node handle
  - KERNEL_TRIGGERED – Output handle passed to triggering fn

UCP Example

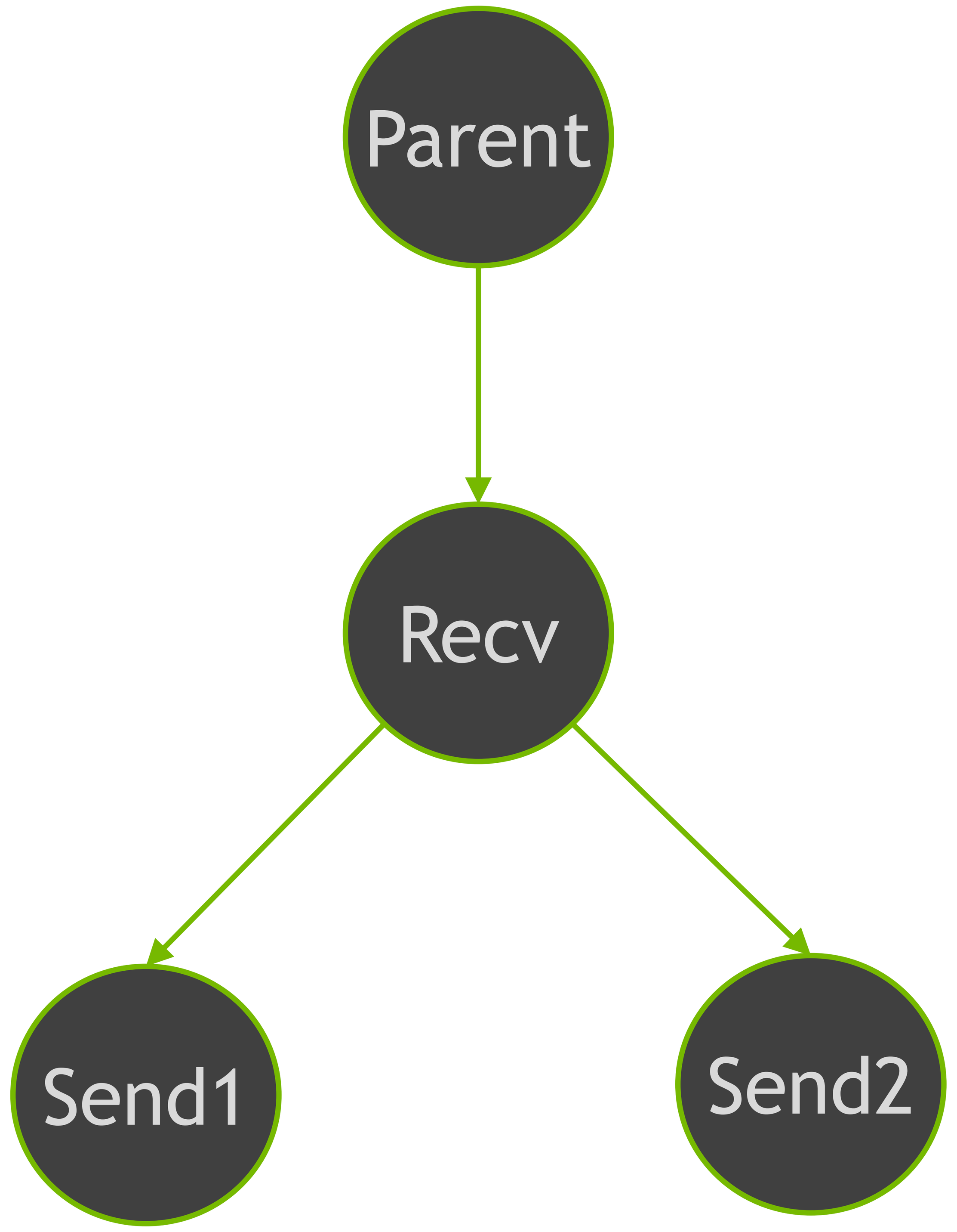
Stream Synchronous Send NBX

```
ucs_status_ptr_t stream_send(..., cudaStream_t *cuda_stream)
{
    ucp_condition_h condition;
    ucp_condition_param_t cond_param = {
        .category = UCP_CONDITION_CATEGORY_STREAM,
        .context  = cuda_stream
    };

    status = ucp_create_condition(&cond_param, &condition);

    ucp_request_param_t param = {
        .op_attr_mask = ... | UCP_OP_ATTR_CONDITION,
        .condition    = condition,
        ...
    };

    status = ucp_tag_send_nbx(..., &param);
    status = ucp_destroy_condition(&condition);
    ...
}
```

UCP Example

Graph Synchronous Broadcast NBX

```
ucs_status_ptr_t graph_bcast(cudaGraphNode_t *parent_node, cudaGraph_t *cuda_graph)
{
    ucp_condition_h recv_cond, send1_cond, send2_cond;
    ucp_condition_param_t recv_cparam = send1_cparam = send2_cparam = {
         // Node will be returned through the context field
        .category = UCP_CONDITION_CATEGORY_GRAPH
    };

    status = ucp_create_condition(&recv_cparam, &recv_cond);
    status = ucp_create_condition(&send1_cparam, &send1_cond);
    status = ucp_create_condition(&send2_cparam, &send2_cond);

    ucp_request_param_t recv_param = {
        .op_attr_mask = ... | UCP_OP_ATTR_CONDITION, .condition = recv_cond; };
    ucp_request_param_t send1_param = {
        .op_attr_mask = ... | UCP_OP_ATTR_CONDITION, .condition = send1_cond; };
    ucp_request_param_t send2_param = {
        .op_attr_mask = ... | UCP_OP_ATTR_CONDITION, .condition = send2_cond; };

    status = ucp_tag_recv_nbx(..., &recv_param);
    status = ucp_tag_send_nbx(..., &send1_param);
    status = ucp_tag_send_nbx(..., &send2_param);

    cudaGraphAddDependencies(*cuda_graph, send1_cparam.context, recv_cparam.context, 1);
    cudaGraphAddDependencies(*cuda_graph, send2_cparam.context, recv_cparam.context, 1);
    cudaGraphAddDependencies(graph, recv_cparam.context, parent_node, 1);
    ...
}
```



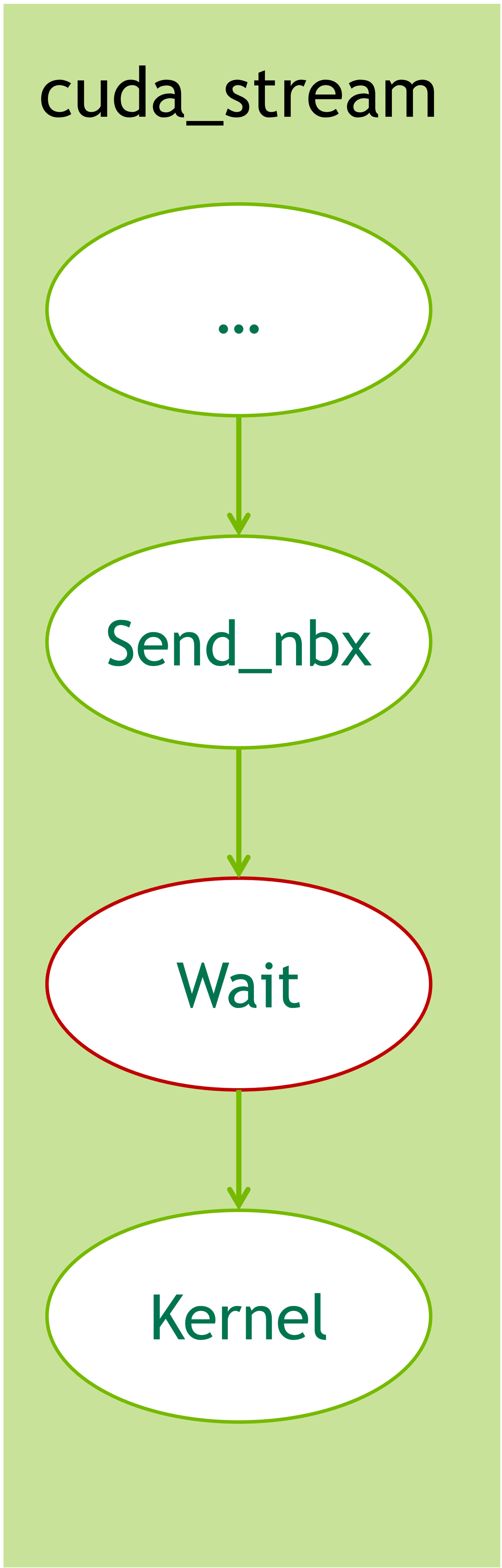
Stream Synchronous Wait Operation

- Existing ucp_request_query API is non-blocking
- Need blocking equivalent like MPI_Wait/MPI_Waitall to enforce dependencies

- Introduce request completion operations:
  - Add “condition” field to ucp_request_attr_t

ucs_status_t ucp_request_wait(void * request, ucp_request_attr_t * attr);

ucs_status_t ucp_request_waitall(size_t nreq, void *requests, ucp_request_attr_t *attrs);



UCT API Extension

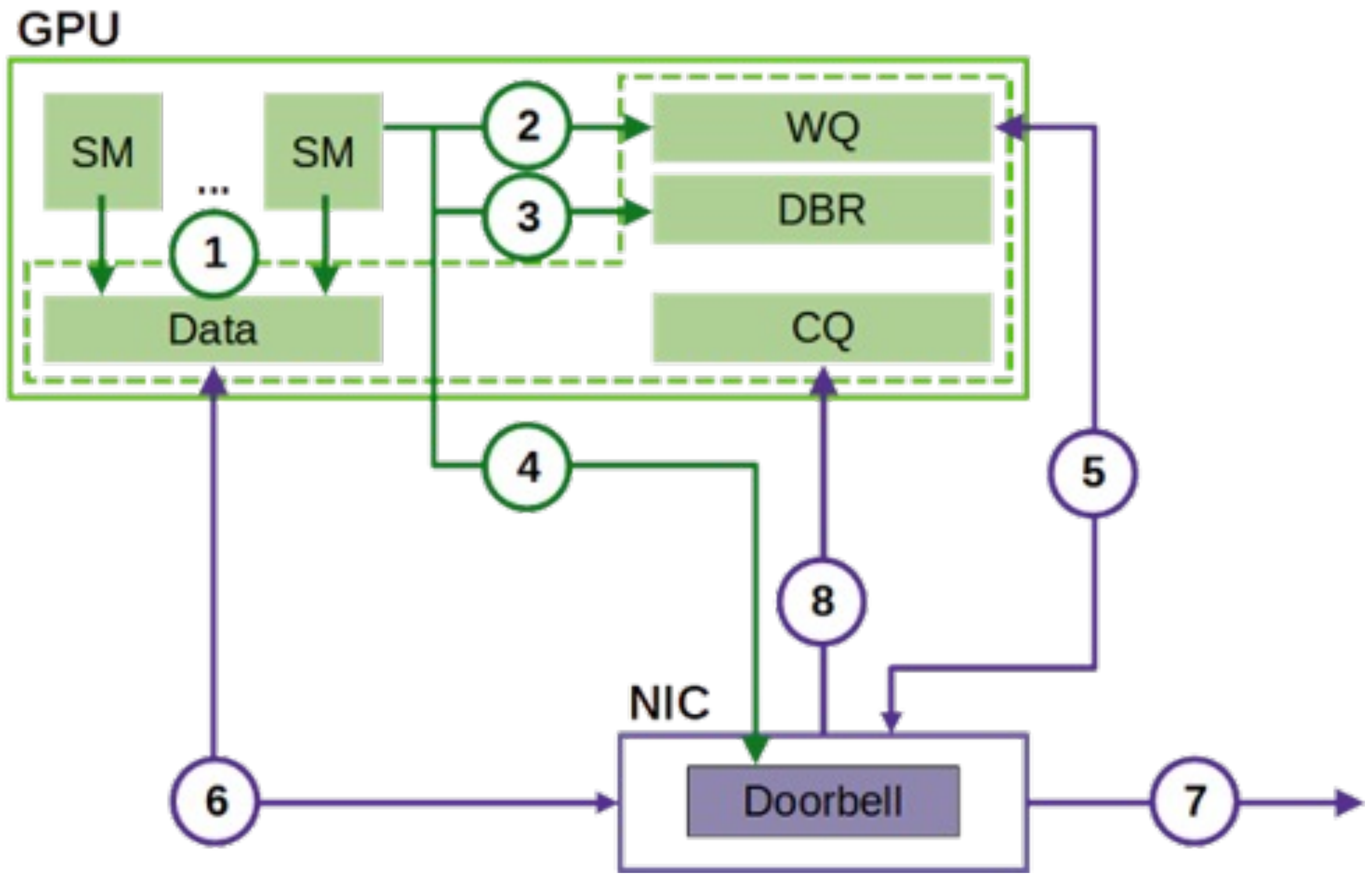
Add Request Parameters to Support Condition

```
typedef struct {
    ...
    ucs_cpu_set_t local_cpus;

    /* Condition types that the MD can process */
    uint64_t condition_types;

} uct_md_attr_v2_t;


UCT_INLINE_API ucs_status_t uct_ep_put_zcopy_nbx(uct_ep_h ep, const uct_iov_t *iov, size_t iovcnt,
                                                 uint64_t remote_addr, uct_rkey_t rkey,
                                                 uct_completion_t *comp, const uct_request_param_t *param)
{
    return ep->iface->ops.ep_put_zcopy_nbx(ep, iov, iovcnt, remote_addr, rkey, comp, param);
}
```

Implementation Considerations

Higher Performance, Lower Flexibility

- Proxy Thread
  - Can progress internal UCX state
  - E.g. protocol selection, pipelines, etc.
  - Cannot submit CUDA work while CUDA is blocked on a task
- CUDA Host Callbacks
  - Executed in arbitrary order
  - Even more limited ability to make CUDA calls
- GPU “Verbs”
  - GPU posts WQEs, rings DB, and polls the CQ
  - Can reuse the same QPs for multiple streams/graphs
  - Sharing between CPU/GPU comes with tradeoffs
- GPUDirect Async
  - CPU posts WQEs, GPU rings doorbell and polls CQ
  - Requires separate QPs per stream to deal with head of line blocking on the QP
  - Requires a serialization of graph into available QPs



Protocol Simplification

Simplify/Resolve Control Plane to Enable Offloading

- Challenge: Protocol selection must be completed to enable optimizations and offloading
- Proposed “MPI_Prepare” function
  - Resolve matching (first iteration)
  - Resolve receiver ready (every iteration)
  - Enables MPI_Pready to be implemented as RDMA write
- We could apply similar ideas in UCX:

```
ucs_status_t ucp_prepare_transfers(
        ucp_ep_h ep, void *prepared_memh,
        const ucp_buffer_param_t *param);

ucs_status_t ucp_release_preparations(
        ucp_ep_h ep, void *prepared_memh);
```

```
MPI_Request req[2];
MPI_Prequest preq;
MPI_Psend_init(..., &req[0]);
MPI_Precv_init(..., &req[1]);
MPI_Prequest_create(req[0], MPI_INFO_NULL, &preq);
while (...) {
  MPI_Startall(2, req);
  MPI_Prepare_all(req, 2);
  kernel<<<..., s>>>(..., preq);
  MPI_Waitall(2, req);
}
MPI_Prequest_free(&preq);
MPI_Request_free(&req[0]);
MPI_Request_free(&req[1]);
```

Conclusions

Conclusions

Work in Progress, Feedback Appreciated

Benefits from stream/graph synchronous communication:

1. Eliminate overhead of GPU-CPU synchronization when CPU drives communication
   - Better overlap of communication with computation
   - Better ability to hide offloading overheads
   - Can improve strong scaling efficiency
2. Improve programmability by including communication dependencies in the stream or graph

Several success stories, including NCCL, NVSHMEM, and LibMP

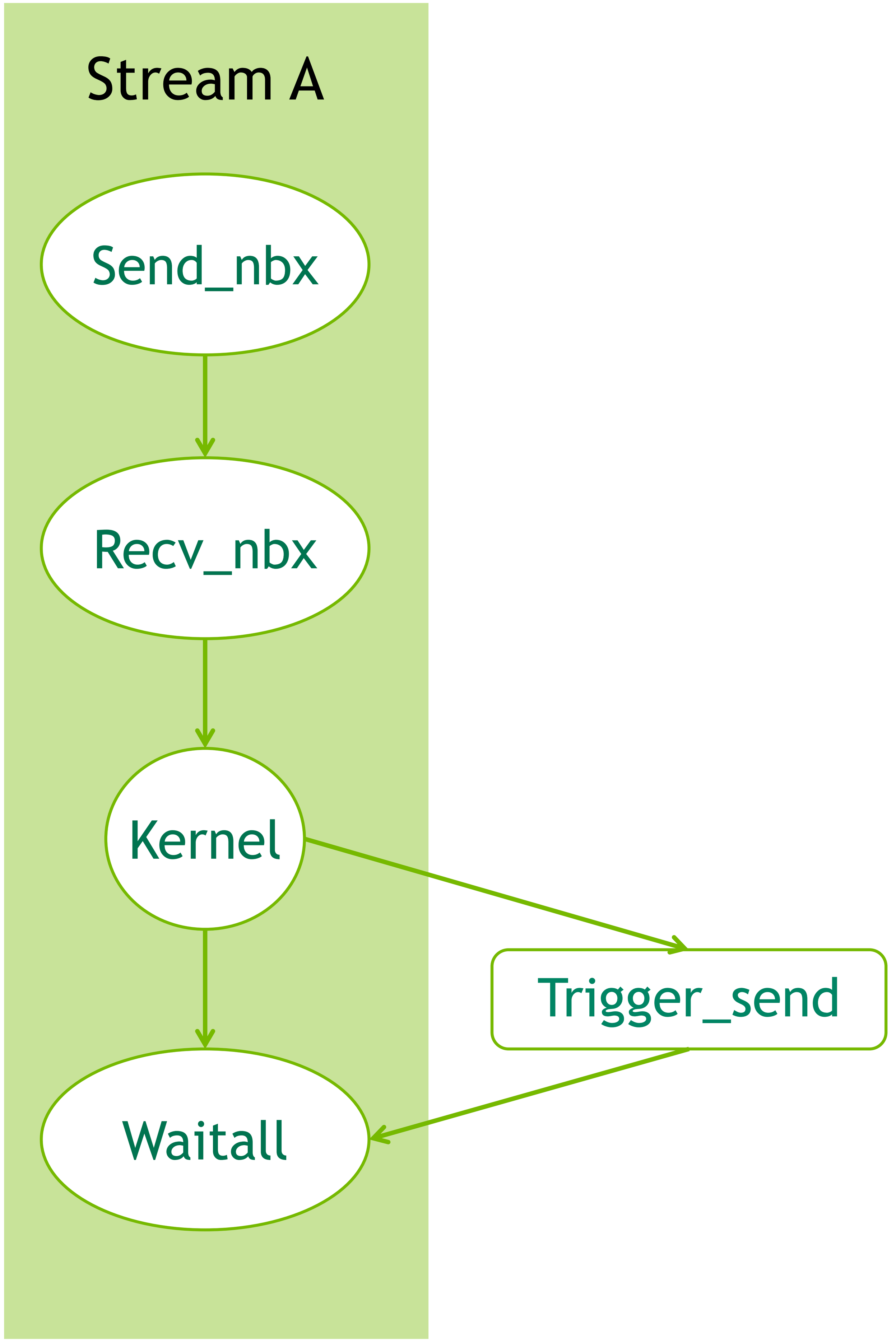
- CUDA graphs provide efficiency improvements over streams

MPI Forum investigating accelerator-integrated communication

- Stream/graph synchronous and kernel triggered (partitioned APIs)

Proposed UCX extension adds "condition" object

- Allows operations to be enqueued and managed by the CUDA runtime
- Efficient implementation requires separation of control and data planes
- Work in progress, feedback is greatly appreciated!



NVIDIA®