GPU Synchronous Communication in UCX

Jim Dinan

Acknowledgement: Akshay Venkatesh, Sreeram Potluri, Hessam Mirsadeghi, Yossi Itigin, and Others

GPU Synchronous Communication Libraries

Lessons to Apply to UCX/MPI

	Stream Triggered	Graph Triggered	Kernel Triggered	Kernel Initiated	Implementation
NCCL	X	X			Proxy
NVSHMEM	X	X		X	Proxy or GDA-KI
LibMP	X	X	X		GPUDirect Async
MPI	Proposed	Proposed	Partitioned		TBD

1. Simple protocols enable efficient integration of communication with CUDA
   - Memory registration, matching, protocol progression, etc.
2. Overheads from setting up communication must be low, and
   - Minimized by enqueueing in batches (e.g. batch memOps)
   - Hidden by overlapping with computation (e.g. kernel)
3. CUDA MemOp parallelism is limited by the number of FIFOs assigned to the CUDA context
   - CUDA_DEVICE_MAX_CONNECTIONS - 1 to 32 (default is 8)

CUDA 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 – Graphs resolve dependencies close to GPU where greater parallelism is possible

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 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_param_t *param,
    ucp_condition_h *condition);

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

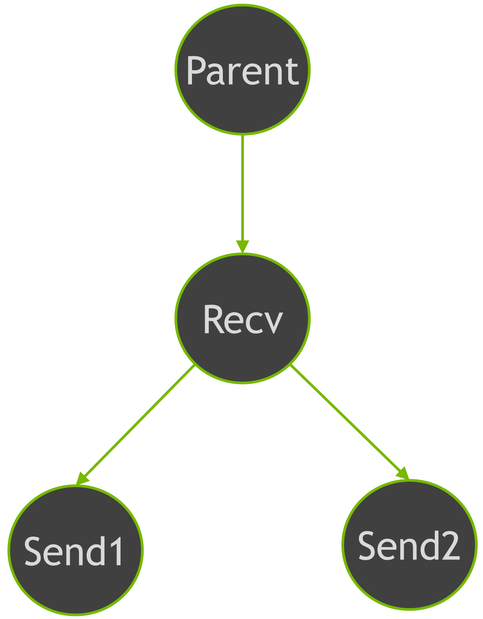
```
ucs_status_ptr_t graph_bcast(cudaGraphNode_t *parent_node, cudaGraph_t *cuda_graph)
{
    ucp_condition_h recv_cond, send1_cond, send2_cond;
    ucp_condition_param_t recv_cparam = send1_cparam = send2_cparam = {
         // Node will be returned through the context field
        .category = UCP_CONDITION_CATEGORY_GRAPH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

    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(*cuda_graph, recv_cparam.context, parent_node, 1);
    ...
}
```



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 perform 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);
```

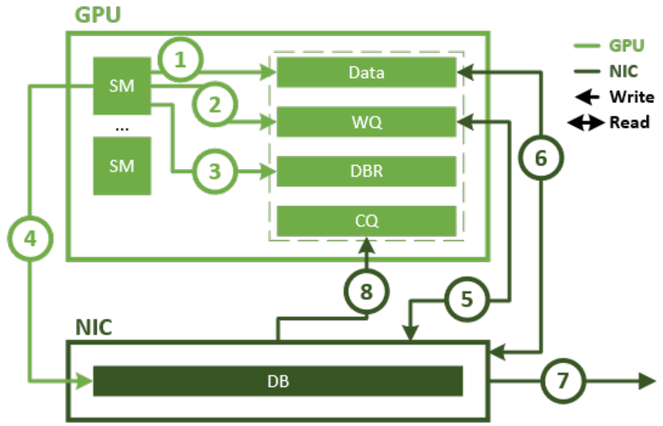
```
MPI_Request req[2];
MPI_Prequest preq;
MPI_Psend_init(..., &req[0]);
MPI_Precv_init(..., &req[1]);
MPI_Prequest_create(req[0], MPI_INFO_NULL, &preq);
while (...) {
  MPI_Startall(2, req);
  MPI_Prepare_all(req, 2);
  kernel<<<..., s>>>(..., preq);
  MPI_Waitall(2, req);
}
MPI_Prequest_free(&preq);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

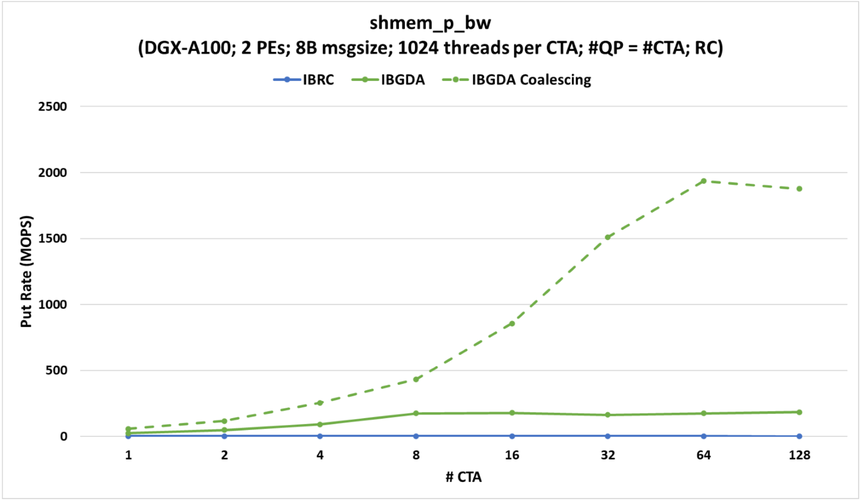
Implementation Considerations

NVIDIA Mellanox ConnectX HCAs

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
- GPUDirect Async – Stream Triggered
  - 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
- GPUDirect Async – Kernel Initiated
  - 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






NVIDIA

CPU Versus Stream Synchronous Communication

GPU Coordinates Data Dependencies Without CPU Involvement