Accelerators and Accelerated Systems - 046274
Spring 2019

Homework assignment 3

Submission via moodle only.
Due: Sunday 30.6, 23:55
Make sure to submit a zip/tar.gz archive with 4 files: `client.c`, `server.cu`, `common.c` & `common.h`.
Archive name should be IDs of the students, separated with underscore (e.g. `123456789_123571113.tar.gz`)

In this assignment, we will implement an RDMA-based client application performing the algorithm from homework 1 using the producer-consumer queue you developed in homework 2.

Please make sure that in all versions of your GPU implementation the average distance your implementation produces is equal to what the CPU implementation produces.

Submission Instructions

You will be submitting an archive (id1_id2.tar.gz or id1_id2.zip) with 4 file:

1. client.c, server.cu, common.c, common.h:
   - 1.1. Contains your implementation. Do not add additional prints aside from the ones that are already in the file. Do not modify anything except where instructed.
   - 1.2. Make sure to check for errors for all InfiniBand verbs calls.
   - 1.3.. Make sure your code compiles **without warnings** and runs correctly before submitting.
   - 1.4. Free all memory and release resources once you have finished using them.

General Description of the problem

In this assignment, we will implement a network client-server version of the image histogram equalization server from homework 2 using InfiniBand Verbs. We will control the CPU-GPU queues remotely using RDMA from the client.
As an example, you are provided with an RPC-based implementation. Your task will be to implement a second version in which the client uses RDMA to control the GPU.

RPC Protocol

This version is given as an example you can use to to implement your code. No need to implement it yourselves. The main protocol is:

1. The client sends a request to the server using a **Send** operation. Each request contains a unique identifier and the parameters the server needs to access the input and output images remotely (Key, Address).
2. The server performs **RDMA Read** operation to read the input image from the address specified by the client.
3. The server performs its task (image histogram equalization) using the GPU.
4. The server uses an **RDMA Write with Immediate** operation to send the output image back to the client at the requested address. The immediate value is the unique identifier chosen by the client for this request.
5. The client is notified the task has completed and continues to the next task.

We use a special RPC request to indicate that the server needs to terminate. Such a request causes the server to terminate without doing steps 2-4.

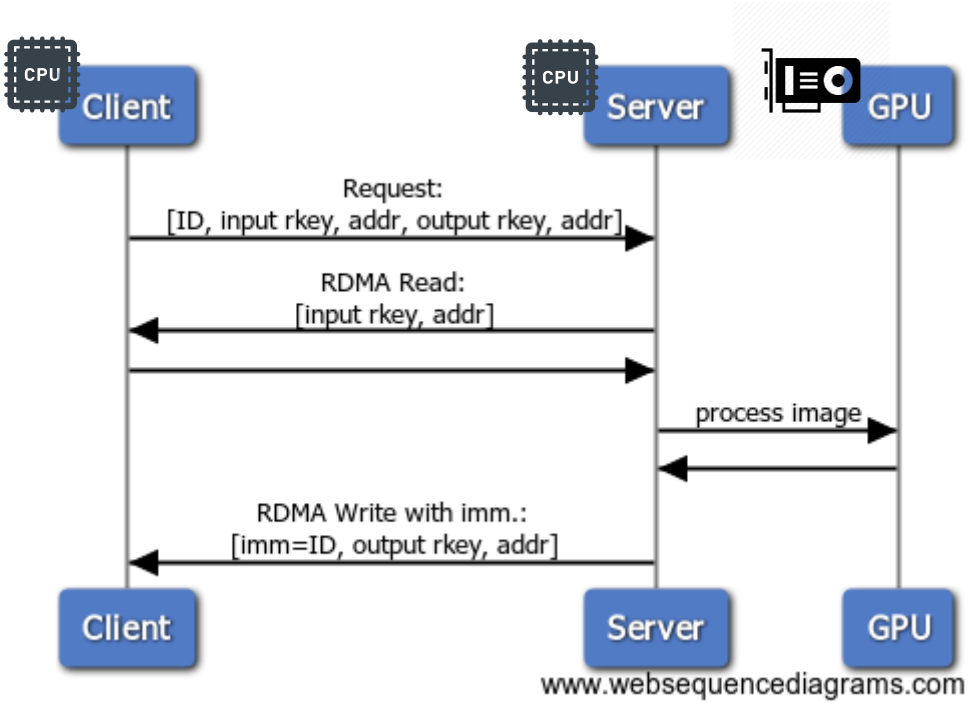

Figure 1 RPC protocol illustration

RPC protocol connection establishment

The client and server need to establish an InfiniBand connection and to exchange parameters for the above protocol to work. They use the following steps to do that.

1. Both client and server allocate required InfiniBand resources, such as PD, QP, CQ, etc.
2. Server allocates an array of requests, to receive requests from the client over an InfiniBand QP. The client allocates 3 buffers: a buffer for input images to send to the

server, a buffer for output images to receive from the server, and an array of requests to send to the server.

3. The server and the client register the relevant buffers with the HCA's memory system. The buffers of the requests are registered for local access, as they are only used as send or receive buffers. The images buffers are registered for RDMA access, to allow the server to read/write them.
4. Server and client briefly use a TCP socket to exchange information about their InfiniBand address (LID, QP number), as well as information about the RDMA buffers (Key, Address). Once they exchange this information we close this socket and never use TCP again.
5. Now we can do real work. We start performing RPC calls as described above.

Client-side RDMA protocol

An alternative protocol we will examine through this assignment uses your CPU-GPU producer-consumer queues from homework 2 remotely from the client using RDMA. You may use your GPU-side code as is. However, rather than having the server process write tasks directly to the GPU queues and poll the GPU queues, the client process will do that through RDMA.

The server tasks are:

1. Initialize CUDA environment, allocate CPU buffers mapped for the GPU, and instantiate the CUDA kernel (taken from homework 2).
2. Register the memory that the GPU accesses also for remote access through verbs.
3. Establish connection with the client and sends the client all necessary information to access these buffers over TCP, as well as other information needed by the client (e.g., the number of GPU queues).

After connection is established, the server CPU is not involved. It only waits for an RPC message that indicates termination.

The client main loop is like the main loop in homework 2 but uses RDMA to access the GPU queues remotely. For each image the client:

1. Polls the GPU queues using RDMA read operations for any tasks that have completed. You may wait for the RDMA read operation to complete, but you must not wait for the GPU to complete each task before moving on to the next step.
2. Check if there is room on the GPU queues to submit new tasks using RDMA read.
3. If there is room, post the next task to the GPU using RDMA write operations.

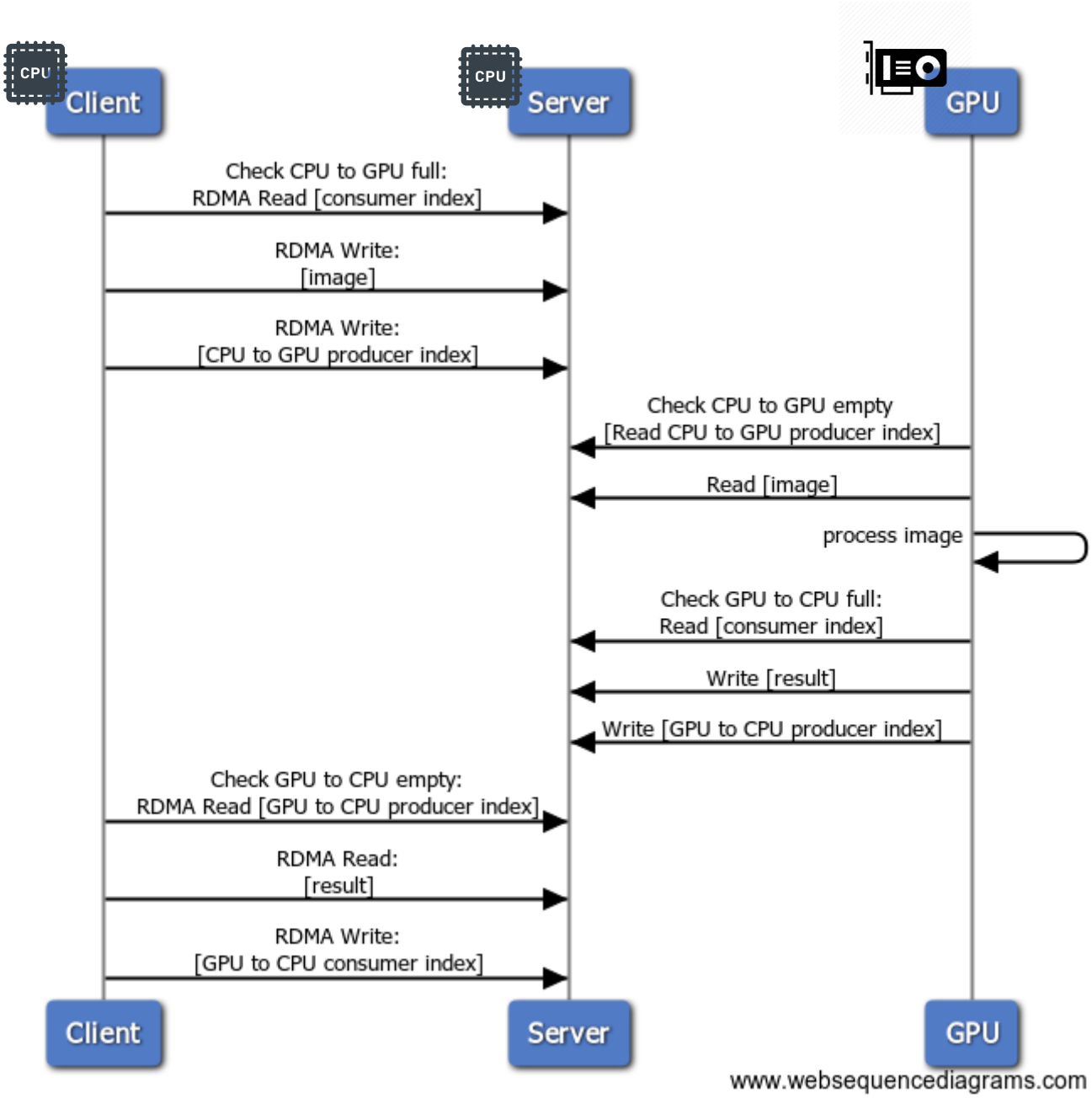

Figure 2 Client RDMA protocol example

Setup

We'll be running both the server and the client on the same machine. The InfiniBand card in the machine (ConnectX-3) has two physical ports, connected to each other via an InfiniBand cable. Think of each port as a separate network device. The server will be using InfiniBand port 1, while the client will use InfiniBand port 2. So, although we use the same machine for the client and the server, we'll be generating real InfiniBand network traffic between the client and the server.

Your task

In hw3.tar.gz you'll find 5 files:

1. `Makefile` for compiling the assignment.
2. `common.h` – a header file with common definitions shared by the client and the server.
3. `common.c` – implementation of the common functions.
4. `server.cu` – server implementation.
5. `client.c` – client implementation.

Both the server and the client accept two command line arguments:
1. Type of the run: **rpc** or **queue**.
2. TCP port to use. If the server's TCP port argument is omitted, a random port is chosen.

The client and server already implement the RPC protocol. Your task is to implement the queue version. The missing parts are marked with TODO comments with instructions of what must be done. Specifically, you would need to perform the following steps.
For the server:
1. Copy the GPU kernel code that uses producer-consumer queues from homework 1 into `server.cu`. You may use any version of the code, not necessarily your own, as we will not going to check the implementation of this part in this assignment.
2. Register the host memory accessible by the GPU through RDMA verbs, to create one or more memory regions that are accessible remotely.
3. Exchange the parameters of the GPU queues (Memory region keys, addresses, number of queues, etc.) with the client. You may edit the `ib_info_t` struct in `common.h` to do that.
4. Run the GPU kernel before starting the main event loop.
5. Teardown the added memory regions during server termination.

On the client:
1. Copy struct definitions of your GPU queues from homework 2, to allow address calculation.
2. Allocate local state on the client to track its side of the queues (e.g., producer index or consumer index).
3. Edit the client event loop to implement the protocol as described above.

You'll be using these verbs: `ibv_post_send(), ibv_reg_mr(), ibv_poll_cq()`.
You can see their usage in the provided code (`server.cu, client.c`), in the man pages (`man ibv_post_send`), or in the RDMAmojo website, which has excellent explanations and examples for these verbs:
http://www.rdmamojo.com/2013/01/26/ibv_post_send/
http://www.rdmamojo.com/2013/02/15/ibv_poll_cq/
http://www.rdmamojo.com/2012/09/07/ibv_reg_mr/