Accelerators and Accelerated Systems - 046278
Spring 2020

Homework assignment 3

Submission via moodle only.
Due: Thursday 25.6.2020, 23:55
Make sure to submit a zip/tar.gz archive with all necessary files: `ex3.cu`, `ex2.cu` and additional header files (.h) you created if you have done so.
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.

Homework package

The archive includes the following files:

Filename	Description
`homework3.pdf`	This file.
`ex3.cu`	A template for you to implement the exercise and submit. **You may edit this file.**
`ex2.cu`	Example solution of the second part of homework 2. **You may edit this file.**
`ex3-cpu.cu`	A CPU implementation of the algorithm that is used to check the results.
`server.cu`	An RDMA server that processes images on the GPU (using your code in `ex2/3.cu`).
`client.cu`	An RDMA client that sends images to be processed to the server and verifies its output (using your code in `ex3.cu`).
`common.cu`	Common RDMA code used by both the client and the server.
`Makefile`	Allows building the client and the server using `make`. This makefile adds the `-maxrregcount=32` argument to the `nvcc` command in order to control the amount of registers used and the `-arch=sm_75` to compile for Turing GPUs and support C++ atomics.

`ex2.h`	Header file with declarations of `ex2.cu` functions and structs needed by `client.cu/server.cu/ex2.cu/ex3.cu`.
`ex3.h`	Header file with declarations of `ex3.cu` functions and structs needed by `client.cu/server.cu/ex2.cu/ex3.cu`.

Submission Instructions

You will be submitting an archive (`id1_id2.tar.gz` or `id1_id2.zip`) with at least 2 files:

1. `ex3.cu`, `ex2.cu`, and additional files you have created:
    - 1.1. The `ex3.cu` contains the client and the server classes you will implement.
    - 1.2. The `ex2.cu` file contains the GPU queues code. You may use the file provided in the homework package or modify it.
    - 1.3. If you need to, create additional .h header files to allow shared declarations between `ex2.cu` and `ex3.cu`. Note that due to a bug in CUDA 10.2, only a single object file may include <cuda/atomic>. As a workaround, ex3.cu has been updated to include ex2.cu, instead of compiling them separately.
    - 1.4. Do not modify other files in the package.
    - 1.5. Make sure to check for errors for all RDMA verbs calls and CUDA calls.
    - 1.6. Make sure your code compiles **without warnings** and runs correctly before submitting.
    - 1.7. Free all memory and release resources once you have finished using them.

Neighborliness

Machines in this course are being shared among several student teams. Please be sure to clean up after yourselves when you are done with a machine:

1. GPU resources, RDMA resources, and TCP sockets are released when the process terminates. Do not leave processes running or stopped.
2. Do not leave open debugging sessions such as nsight or cuda-gdb with running or stopped processes.

General Description of the problem

In this assignment, we will implement a network client-server version of the image quantization server from homework 2 using RDMA 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 learn how to implement the second part. **No need to implement this part yourselves**.
The main RPC protocol is:

1. The client sends a request to the server using a **Send** operation. Each request contains a unique identifier chosen by the client, 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 quantization) 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 in step 1.
5. The client receives a completion, notices 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 2-3, replying with an empty RDMA Write with Immediate operation (step 4) to let the client know it is terminating.

You can find the server code for this protocol in the `server_rpc_context` class and the client code in the `client_rpc_context` class.

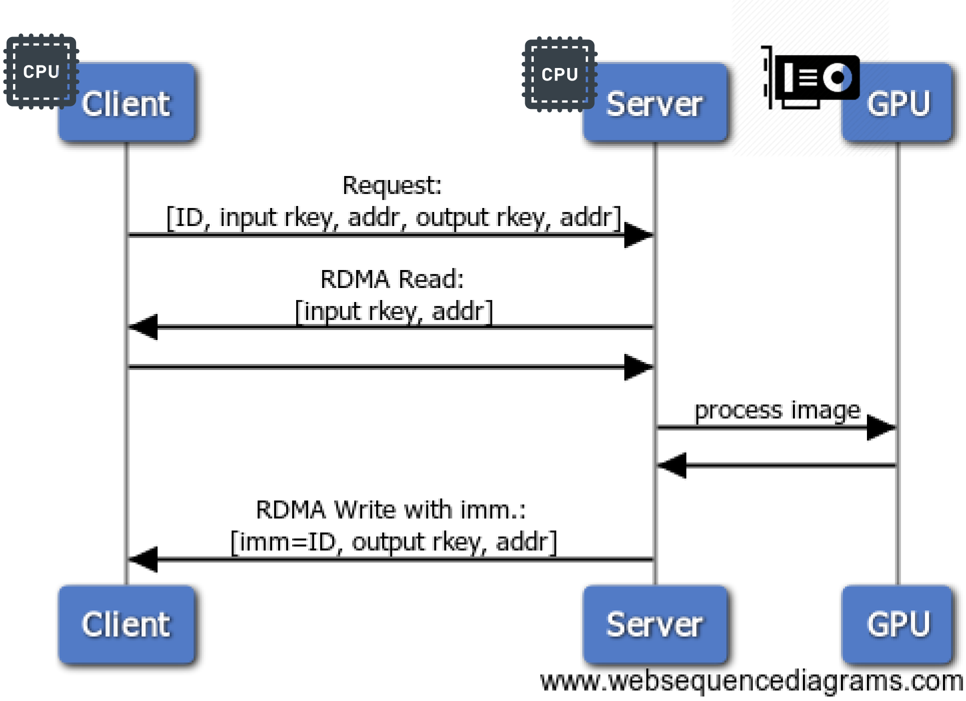

Figure 1 RPC protocol illustration

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 RDMA resources, such as PD, QP, CQ, etc (in the `rdma_context::initialize_verbs` method).
2. Both allocate an array of requests (part of the `rdma_context` parent class), to allow the client to send requests to the server over an RDMA QP. The `initialize_verbs` method also registers these arrays with the NIC's memory system and creates an MR.
3. The client class receives pointers to the input and output buffers to register them as MRs as well (`set_input_images/set_output_images`). These buffers are registered with RDMA access, allowing the server to read/write them remotely.
4. Server and client briefly use a TCP socket (`rdma_context::tcp_connection`) to exchange information about their RDMA address (LID/GID/IP address, QP number), as well as information about the RDMA buffers (Key, Address), using the `send_connection_establishment_data` and `recv_connection_establishment_data`.
5. Now we can do real work over the established RDMA connection. We start performing RPC calls as described above. The server does that in its `server_rpc_context::event_loop` method, while the client uses its `client_rpc_context::enqueue` and `client_rpc_context::dequeue` methods from a main loop in client.cu.

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 send 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 to indicate termination.

The client enqueue and dequeue operations are like those used in homework 2, but here they use RDMA to access the GPU queues remotely. To enqueue, the client:

1. Checks if there is room on the GPU queues to submit new tasks, using an RDMA read operation.
2. If there is room, copy the image to the server and enqueue the next task on the GPU's queue using RDMA write operations.

For dequeuing, 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. If a task has complete, use RDMA read operations to read the image ID and image content back to the client machine.

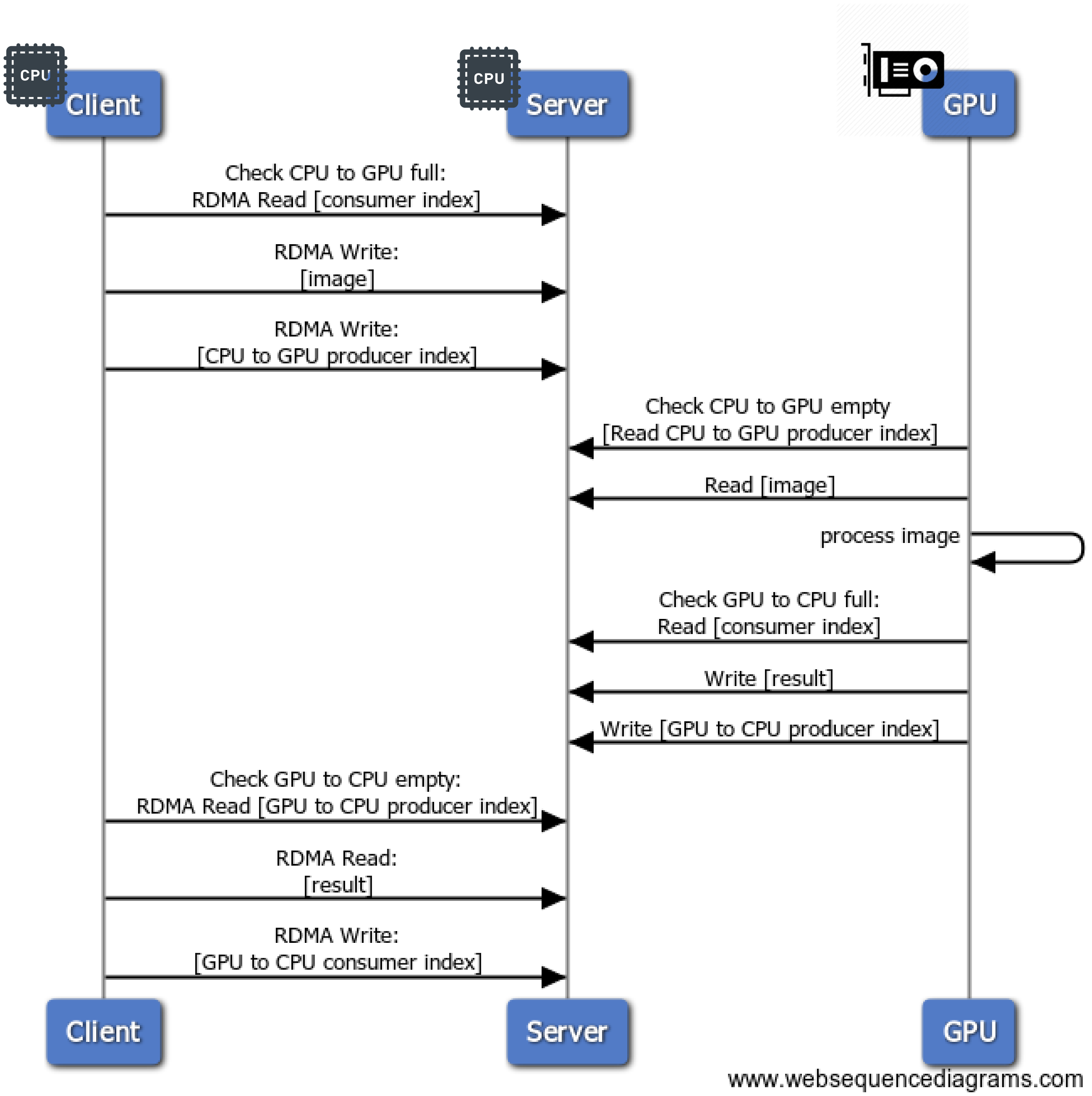

Figure 2 Client RDMA protocol example

Setup

We'll be running both the server and the client on the same machine. The RDMA card in the machine (ConnectX-4 Lx) has two physical ports, connected to each other via an Ethernet cable. Each port has a separate network device. The server and the client will both use port 1 by default, communicating via NIC loopback.

Building and running

Compile both the server and the client using the `make` command.

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.

Run the server, and after it starts run the client with the same port and mode of operation.

Your tasks

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, implement the changes in the `server_queues_context` class:

1. In the constructor:
    a. register the host memory accessible by the GPU through RDMA verbs, to create one or more memory regions that are accessible remotely.
    b. Exchange the parameters of the GPU queues (Memory region keys, addresses, number of queues, etc.) with the client. You may use the `send_over_socket` and `recv_over_socket` methods for that.
    c. Start the persistent kernel on the GPU.
2. In the destructor:
    a. Make sure the GPU kernel terminates and teardown the added memory regions.
3. In the event_loop method:
    a. There is no need to handle messages here. Just wait for termination message from the client, return a reply and exit the method.

The client should be implemented as part of the `client_queues_context` class:

1. Add necessary state to track the CPU-GPU queues on the client side to the class as member variables (e.g. client's / CPU's producer indices and consumer indices), as well as information about the server such as number of queues, rkeys and addresses.
2. Add memory regions the client may use (e.g. for RDMA operations toward the server).
3. In the constructor:
    a. Communicate with the server to receive the necessary parameters.
    b. Register memory regions defined in (2).
4. In the `set_input_images/set_output_images` register memory regions for the client's input and output buffers. Pay attention to the necessary access permissions.
5. In the destructor:
    a. Send a termination message to the server and wait for a response.
    b. Release registered memory regions.
6. Implement the enqueue method:
    a. Use RDMA Read operations to check whether the destination queue is full.
        i. Return false for a full queue.
    b. Use RDMA Write operations to write the data to the queue.
    c. Use RDMA Write to increment the producer index.
7. Implement the dequeue method:
    a. Use RDMA Read operations to check whether the desired queue is empty.
        i. Return false if so.
    b. Use RDMA Read operations to read the data from the queue.
    c. Use RDMA Write to increment the consumer index.

You'll be using these verbs: `ibv_post_send(), ibv_reg_mr(), ibv_poll_cq()`.
You can see their usage in the provided code in the RPC server example, 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/