TensorPipe

20230503
박현우

Table of Contents

1. Interface

1. Classes
2. Procedure of sending/receiving message

1.1. Classes

1. tensorpipe::Context
2. tensorpipe::Listener
3. tensorpipe::Pipe
4. tensorpipe::Message

1.1.1. Context, Listener

Context

Contain and keep track of the global state of the system, such as

1. thread pools,
2. open file descriptors, etc.

Listener

An entry point for connection to which other processes can connect.

1.1.2. Pipe

Communication primitive.

Pipe can be obtained either

1. by connecting to the listener of another process
2. or having such a listener of own.

Once you have a pipe, you can send messages on it.

1.1.3. Message

Data structure that pipes read and write in.

Pipes are streams of structured messages.

A message is composed of

1. a Core payload
   (memory living on CPU)
2. a list of Tensors
   (memory living on any device, like GPUs).

1.2.1. Sending message

Using the **write** method of Pipe class, which takes

1. a message (with the data to send)
2. and a callback
   which will be invoked once the sending has completed.

Callback is required because the pipe is asynchronous.

This callback will be invoked

1. with an error (if one happened)
2. and with the message.

1.2.2 Receiving message

On an incoming message, with two steps:

1. first the pipe asks you to provide some memory for buffer to hold the message in,
2. and then you ask the pipe to read the data into that memory.

To do so, first you must register a callback that will be notified for incoming messages.

This is because Pipe is asynchronous.

1.2.2 Receiving message

Registering callback can be done by calling the readDescriptor method with said callback.

This readDescriptor is a message with no buffers;
i.e. set to null pointers.

Note : Do not confuse with callbacks for read or write methods;
callbacks for notifying incoming messages MUST be for readDescriptor.

1.2.2 Receiving message

The job of this callback is to acquire memory space for those buffers;
e.g.

1. directly allocating the required memory
2. or obtaining as a slice of a batch that’s being assembled (alongside other pipes).

Also, readDescriptor contains some metadata,
which can be used to provide allocation hints or any other information that can help the receiver determine where to store the data.

1.2.2 Receiving message

Once callback makes the message’s buffers ready,

user can call the pipe to fill the buffers with the incoming data
by passing the message to the read method (together with a callback).

2. Transports and Channels

TensorPipe aims to be "backend-agnostic".

To select the fastest medium available, the two processes will automatically negotiate during setup to determine:

1. which of the backends can be used
2. and how well they would perform.

Two features for backend selection:

1. Transports
2. Channels

2.1. Transports

Transports are the connections used by the pipes to transfer

1. control messages
2. and the (smallish) core payloads. (for CPU)

They are meant to be lightweight and low-latency.

Examples:

1. TCP, which is the basic transport.
2. Shared Memory with Ring buffer.

2.2. Channels

Channels take care of copying the (larger) tensor data. (for Device)
c.f. Core payloads for CPU.

High bandwidths are a requirement.

Examples:

1. Multiple TCP sockets and processes group
   c.f. Single TCP socket for Transports

2. CUDA memcpy call via NVLink.

2.3. Basis for Negotiation

Note that selection of Transport and Channel may conflict;

For example,

1. in perspective of Transports,
   a TCP-based transport is best implemented using a single connection (due to handshake overhead),

2. in perspective of Channels,
   whereas a TCP-based channel should multiplex the payload over multiple connections, in order to saturate the bandwidth.

2.3. Basis Negotiation

As a rule of thumb, we require more from the transports:

this is because transports are for bootstrapping and controlling the connection; e.g. signaling or “waking up” the opponent machine to have connection.

Channels, on the other hand, have much looser requirements:

they basically just need to implement a memcpy, and use transport for anything beyond capability, which TensorPipe recommends and supports.

3.1. IB is for setup data path, not hot data path

Registration and deregistration of memory with InfiniBand is considered a "setup" step, as is very slow,

and should thus be avoided as much as possible during the "hot" data path;

for example, by

1. using a staging (hierarchial) area
2. or caching these registrations.

3.1. IB is for setup data path, not hot data path

IB is considered “slow”, compared to NVLink (or PCIe), which is for Device to Device Communication.
c.f. Infiniband for Inter-machine network communication.

Example in FSDP:

1. Each layers are computed through NVLink within each machine
2. Gradients are propagated on all machines for synchronization through IB.

3.2. Extra NVIDIA kernel module for IB is required

When we pass a pointer to InfiniBand for registration, InfiniBand needs to acknowledge that this virtual address points to CUDA device memory, not to some CPU memory.

For this, it requires so-called "peer memory client",
which can be queried by IB, thus properly recognized as CUDA device memory.

3.2. Extra NVIDIA kernel module for IB is required

NVIDIA provides this peer memory client feature through a separate kernel module and is only available in Mellanox’s InfiniBand distribution called OFED, OpenFabrics Enterprise Distribution.

OFED is not provided in vanilla upstream InfiniBand.

3.3. GPUs need to be matched with the right IB NIC

On some machine types, there may be

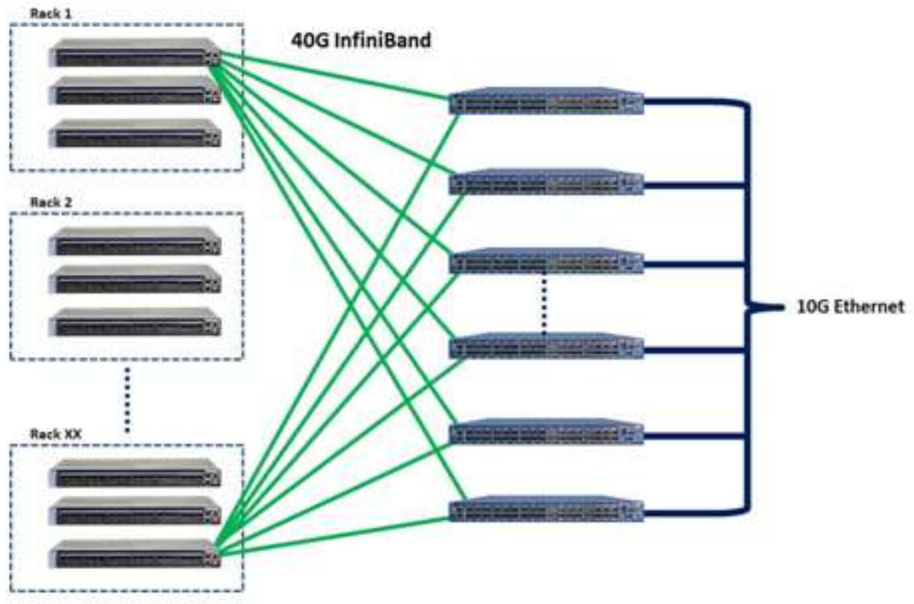
1. multiple GPUs
2. and multiple InfiniBand devices (e.g. IB NIC/Switches, etc)

and they need to be carefully matched.

Using the same IB NIC will introduce a bottleneck for all GPUs while leaving all other NICs unused.

This may be because IB network is not on strict 1:1 matching, but N:N matching.

3.3. GPUs need to be matched with the right IB NIC



Although all devices (GPU) are connected to all IB NICs,

the devices only utilize few IB NICs, weighing heavy overload on them, while leaving other NICs unused.

(@ Isn’t there any load balancer in IB network ?)

3.3. GPUs need to be matched with the right IB NIC

Matching them up "randomly" means that the data paths over PCIe of different GPU-NIC pairs

1. might cross each other
   (thus, again, causing a bottleneck),
2. might traverse the host,
   i.e. useless D2H and H2D copies occur
3. or otherwise interfere.

(@ this “interfere” is on physically ?)

3.3. GPUs need to be matched with the right IB NIC

These machines are usually set up as-a-whole, so that each GPU has one NIC that it’s "naturally" closest to. (@ physically?)

In such case, they usually share the same PCIe switch, thus we need a logic to be able to detect and implement this manually: to match or load balancing each GPU with appropriate NIC.

3.4. IB messages have a maximum size

Each send/recv operation over InfiniBand can only handle up to a certain amount of data, usually at best 1GB, and will fail for larger amounts.

Thus, chunking must be used for larger sizes.

There's also a "minimum size" of 32 bytes, with messages failing with odd errors for smaller sizes.

It's still unclear whether it's a bug.