THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
RELAXED SYSTEM LAB

Nvidia Collective Communication Library

COMP4901Y

Binhang Yuan

What is NCCL?

Collective Communication

RELAXED
SYSTEM LAB

- **Process group**: all communication is within a group of processes, and the collective communication is over all of the processes in that group.
- **World**: defines all of the processes for the parallel job.
- **World size**: number of processes in the world.
- **Rank**: the unique process ID in the world.

NCCL Overview

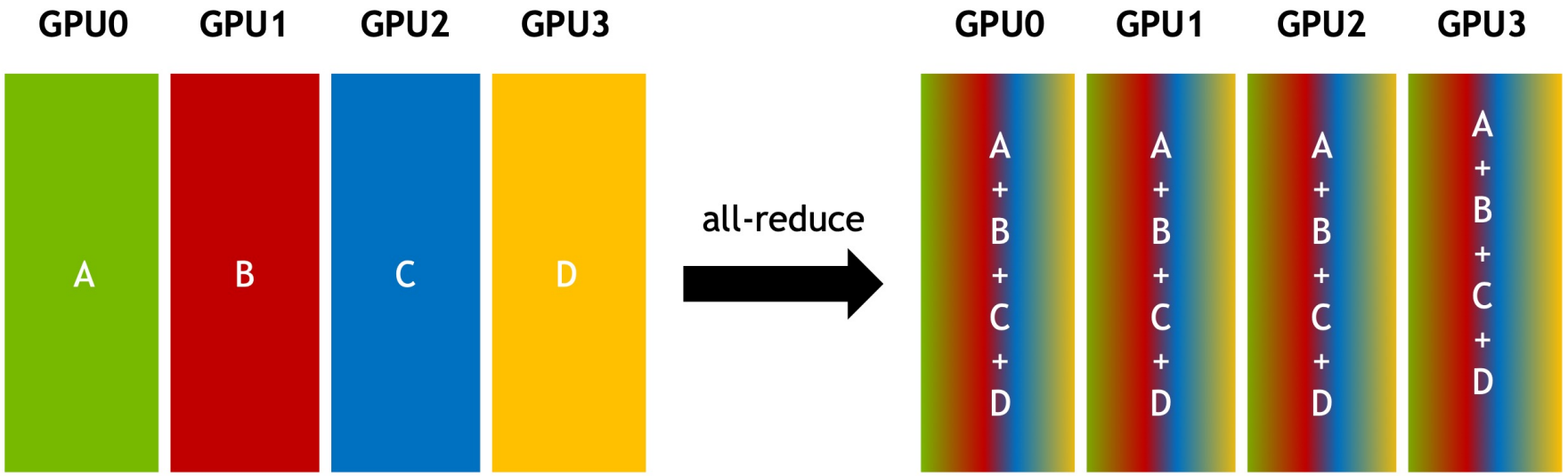
RELAXED
SYSTEM LAB

- Optimized collective communication library from Nvidia to enable high-performance communication between CUDA devices.
- NCCL Implements:
  - **AllReduce**;
  - **Broadcast;**
  - **Reduce;**
  - **AllGather;**
  - **Scatter;**
  - **Gather;**
  - **ReduceScatter;**
  - **AlltoAll.**
- Easy to integrate into DL framework (e.g., PyTorch).

AllReduce

RELAXED SYSTEM LAB

- The **AllReduce** operation performs reductions on data (for example, sum, min, max) across devices and writes the result in the receive buffers of **every** rank.



AllReduce in PyTorch

RELAXED
SYSTEM LAB

torch.distributed.all_reduce(*tensor*, *op=<RedOpType.SUM: 0>*, *group=None*, *async_op=False*) [SOURCE]

Reduces the tensor data across all machines in a way that all get the final result.

After the call `tensor` is going to be bitwise identical in all processes.

Complex tensors are supported.

Parameters

- **tensor** (*Tensor*) – Input and output of the collective. The function operates in-place.
- **op** (*optional*) – One of the values from `torch.distributed.ReduceOp` enum. Specifies an operation used for element-wise reductions.
- **group** (*ProcessGroup, optional*) – The process group to work on. If None, the default process group will be used.
- **async_op** (*bool, optional*) – Whether this op should be an async op

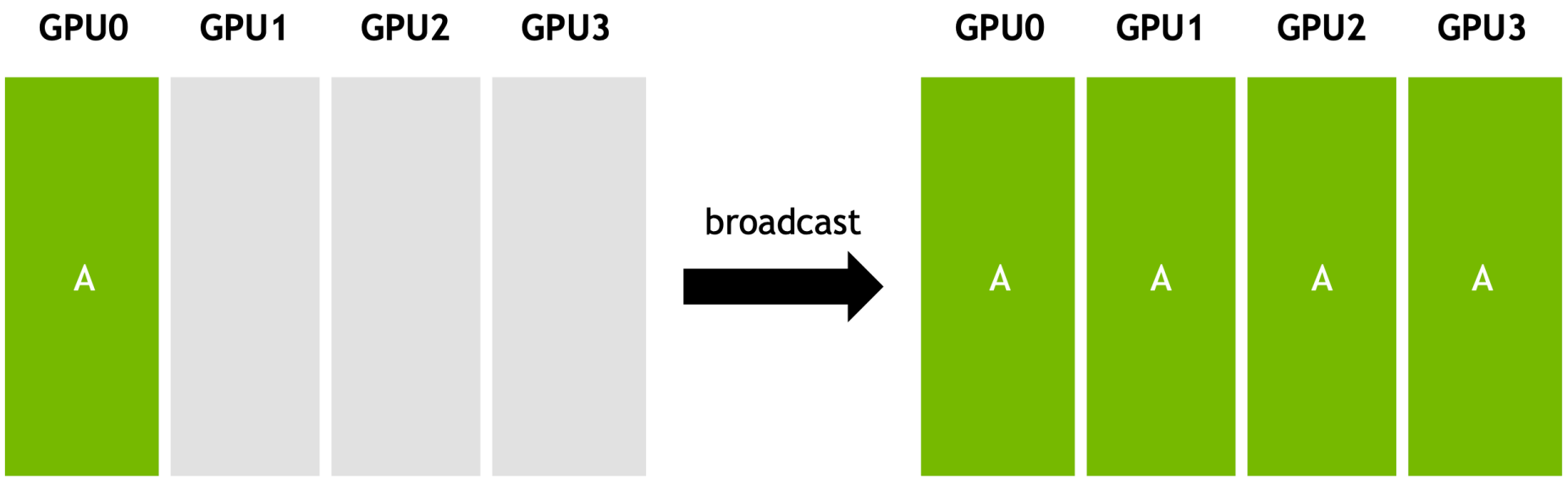
Returns

Async work handle, if async_op is set to True. None, if not async_op or if not part of the group

Broadcast

RELAXED
SYSTEM LAB

- The **Broadcast** operation copies an *N*-element buffer on the root rank to all ranks.



Broadcast in PyTorch

RELAXED
SYSTEM LAB

torch.distributed.broadcast(*tensor*, *src*, *group=None*, *async_op=False*) [SOURCE]

Broadcasts the tensor to the whole group.

`tensor` must have the same number of elements in all processes participating in the collective.

Parameters

- **tensor** (*Tensor*) – Data to be sent if `src` is the rank of current process, and tensor to be used to save received data otherwise.
- **src** (*int*) – Source rank.
- **group** (*ProcessGroup, optional*) – The process group to work on. If None, the default process group will be used.
- **async_op** (*bool, optional*) – Whether this op should be an async op

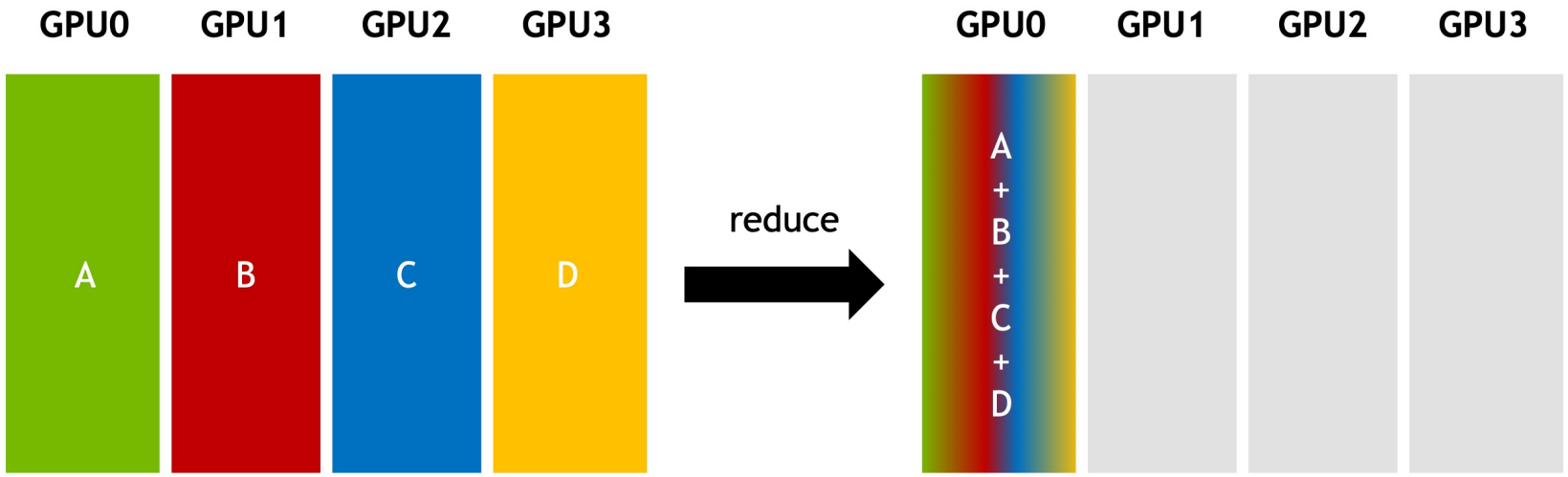
Returns

Async work handle, if async_op is set to True. None, if not async_op or if not part of the group

Reduce

RELAXED
SYSTEM LAB

- The **Reduce** operation is performing the same operation as **AllReduce**, but writes the result only in the receive buffers of a specified root rank.



Reduce in PyTorch

RELAXED
SYSTEM LAB

torch.distributed.reduce(*tensor*, *dst*, *op=<RedOpType.SUM: 0>*, *group=None*, *async_op=False*) [SOURCE]

Reduces the tensor data across all machines.

Only the process with rank `dst` is going to receive the final result.

Parameters

- **tensor** (*Tensor*) – Input and output of the collective. The function operates in-place.
- **dst** (*int*) – Destination rank
- **op** (*optional*) – One of the values from `torch.distributed.ReduceOp` enum. Specifies an operation used for element-wise reductions.
- **group** (*ProcessGroup, optional*) – The process group to work on. If None, the default process group will be used.
- **async_op** (*bool, optional*) – Whether this op should be an async op

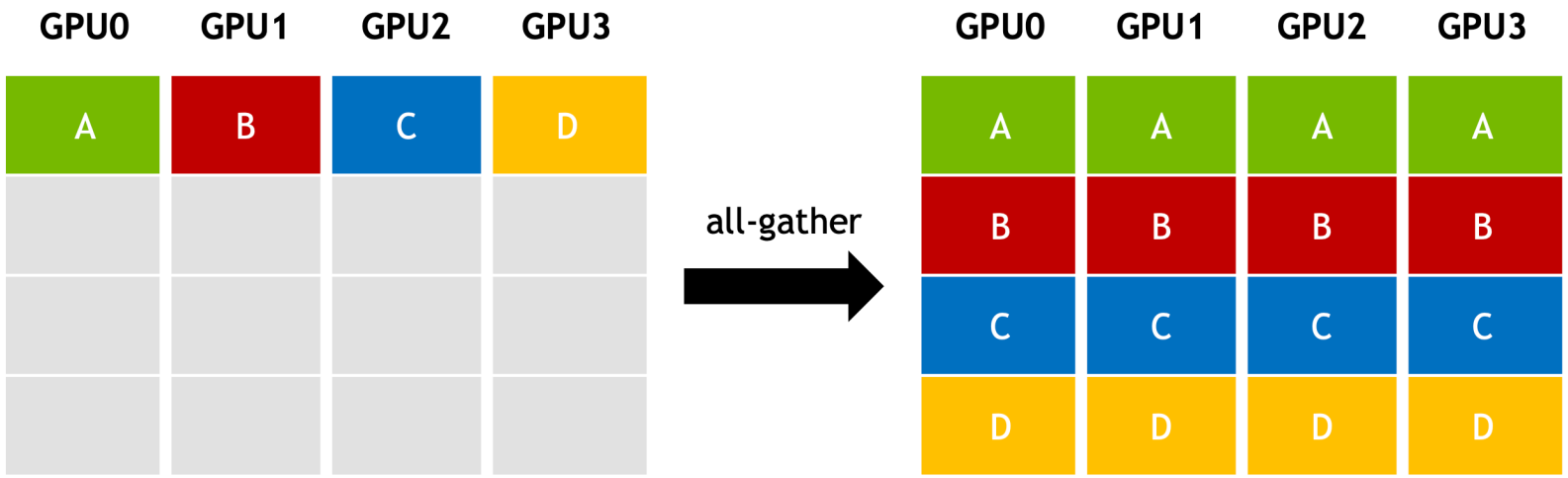
Returns

Async work handle, if async_op is set to True. None, if not async_op or if not part of the group

AllGather

RELAXED
SYSTEM LAB

- The **AllGather** operation gathers N values from k ranks into an output of size kN, and distributes that result to all ranks.
- The output is ordered by rank index. The **AllGather** operation is therefore impacted by a different rank or device mapping.



AllGather in PyTorch

RELAXED
SYSTEM LAB

torch.distributed.all_gather(*tensor_list*, *tensor*, *group=None*, *async_op=False*) [SOURCE]

Gathers tensors from the whole group in a list.

Complex tensors are supported.

Parameters

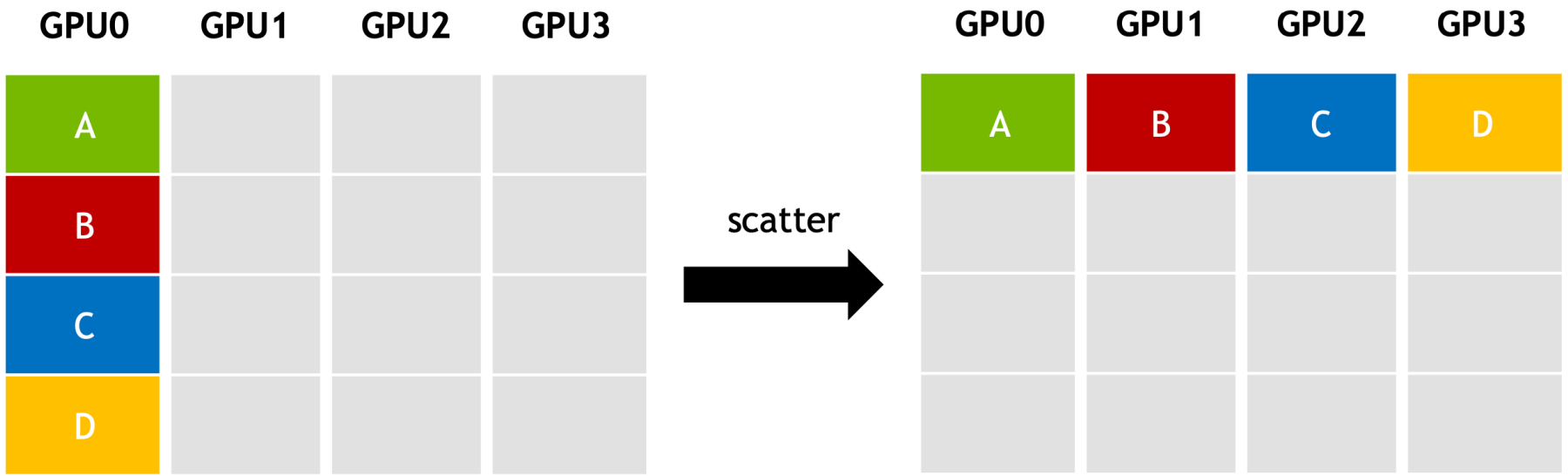
- **tensor_list** (*list*[*Tensor*]) – Output list. It should contain correctly-sized tensors to be used for output of the collective.
- **tensor** (*Tensor*) – Tensor to be broadcast from current process.
- **group** (*ProcessGroup, optional*) – The process group to work on. If None, the default process group will be used.
- **async_op** (*bool, optional*) – Whether this op should be an async op

Returns

Async work handle, if async_op is set to True. None, if not async_op or if not part of the group

Scatter

- **Scatter** is a collective routine similar to **Broadcast**.
- **Scatter** involves a designated root process sending data to all processes.
- **Broadcast** sends the same piece of data to all processes while **Scatter** sends chunks of an array to different processes.



Scatter in PyTorch

RELAXED
SYSTEM LAB

torch.distributed.scatter(*tensor*, *scatter_list=None*, *src=0*, *group=None*, *async_op=False*) [SOURCE]

Scatters a list of tensors to all processes in a group.

Each process will receive exactly one tensor and store its data in the `tensor` argument.

Complex tensors are supported.

Parameters

- **tensor** (*Tensor*) – Output tensor.
- **scatter_list** (*list*[*Tensor*]) – List of tensors to scatter (default is None, must be specified on the source rank)
- **src** (*int*) – Source rank (default is 0)
- **group** (*ProcessGroup, optional*) – The process group to work on. If None, the default process group will be used.
- **async_op** (*bool, optional*) – Whether this op should be an async op

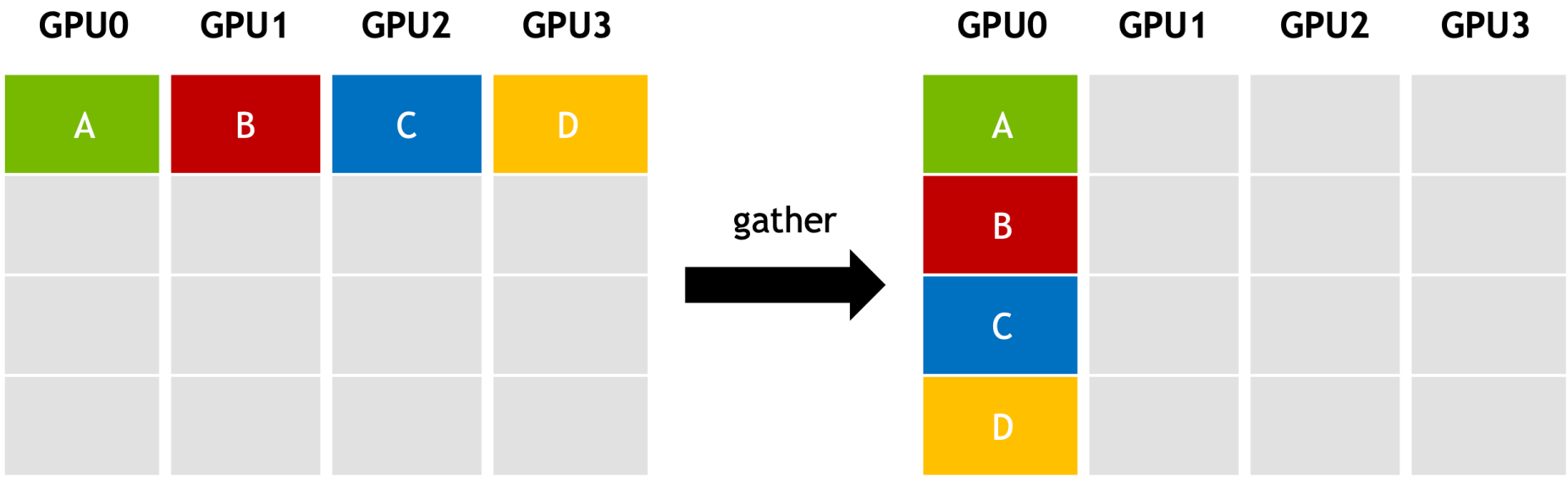
Returns

Async work handle, if async_op is set to True. None, if not async_op or if not part of the group

Gather

RELAXED
SYSTEM LAB

- Instead of spreading elements from one process to many processes, **Gather** takes elements from many processes and gathers them to one single process.



Gather in PyTorch

RELAXED
SYSTEM LAB

torch.distributed.gather(*tensor*, *gather_list=None*, *dst=0*, *group=None*, *async_op=False*) [SOURCE]

Gathers a list of tensors in a single process.

Parameters

- **tensor** (*Tensor*) – Input tensor.
- **gather_list** (*list*[*Tensor*], *optional*) – List of appropriately-sized tensors to use for gathered data (default is None, must be specified on the destination rank)
- **dst** (*int*, *optional*) – Destination rank (default is 0)
- **group** (*ProcessGroup*, *optional*) – The process group to work on. If None, the default process group will be used.
- **async_op** (*bool*, *optional*) – Whether this op should be an async op

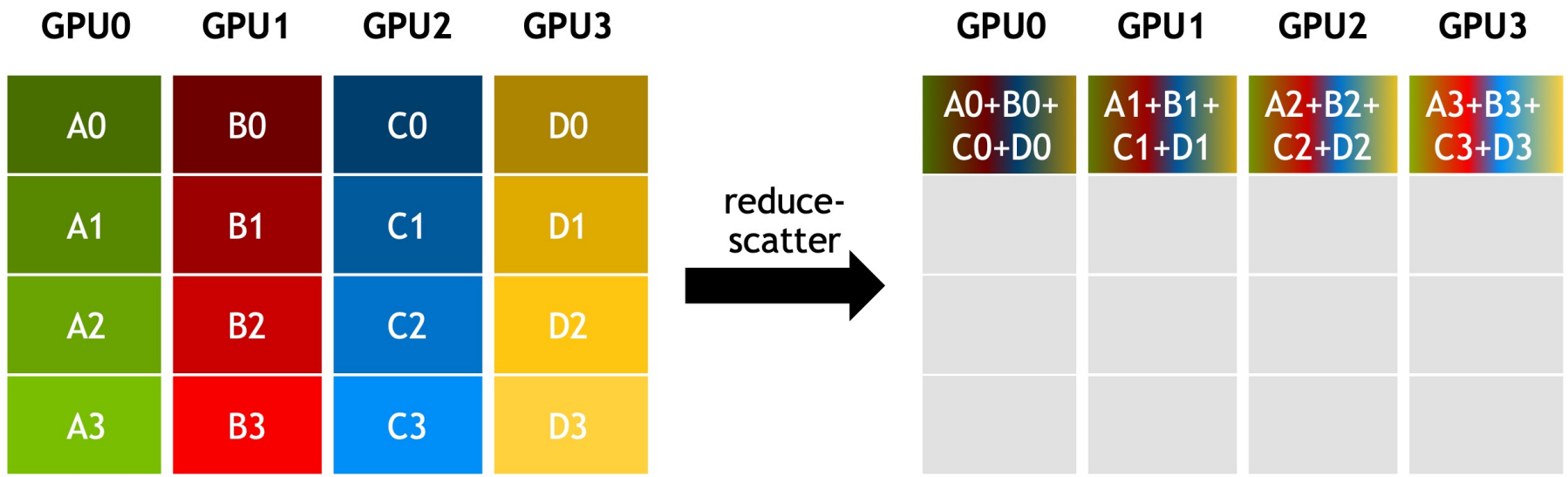
Returns

Async work handle, if async_op is set to True. None, if not async_op or if not part of the group

ReduceScatter

RELAXED SYSTEM LAB

- The **ReduceScatter** operation performs the same operation as the Reduce operation, except the result is scattered in equal blocks among ranks, each rank getting a chunk of data based on its rank index.



ReduceScatter

RELAXED
SYSTEM LAB

torch.distributed.reduce_scatter(*output*, *input_list*, *op=<RedOpType.SUM: 0>*, *group=None*, *async_op=False*) [SOURCE]

Reduces, then scatters a list of tensors to all processes in a group.

Parameters

- **output** (*Tensor*) – Output tensor.
- **input_list** (*list*[*Tensor*]) – List of tensors to reduce and scatter.
- **op** (*optional*) – One of the values from `torch.distributed.ReduceOp` enum. Specifies an operation used for element-wise reductions.
- **group** (*ProcessGroup, optional*) – The process group to work on. If None, the default process group will be used.
- **async_op** (*bool, optional*) – Whether this op should be an async op.

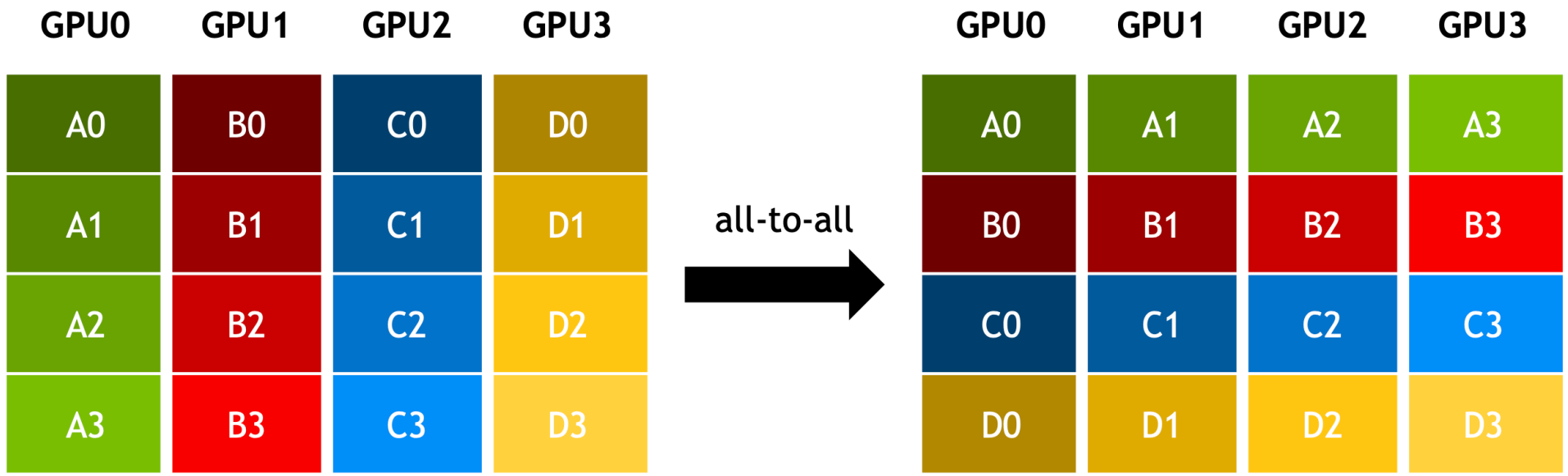
Returns

Async work handle, if async_op is set to True. None, if not async_op or if not part of the group.

AlltoAll

RELAXED
SYSTEM LAB

- Scatter/Gather distinct messages from each participant to every other.



AlltoAll in PyTorch

RELAXED
SYSTEM LAB

torch.distributed.all_to_all(*output_tensor_list*, *input_tensor_list*, *group=None*, *async_op=False*) [SOURCE]

Scatters list of input tensors to all processes in a group and return gathered list of tensors in output list.

Complex tensors are supported.

Parameters

- **output_tensor_list** (*list*[*Tensor*]) – List of tensors to be gathered one per rank.
- **input_tensor_list** (*list*[*Tensor*]) – List of tensors to scatter one per rank.
- **group** (*ProcessGroup, optional*) – The process group to work on. If None, the default process group will be used.
- **async_op** (*bool, optional*) – Whether this op should be an async op.

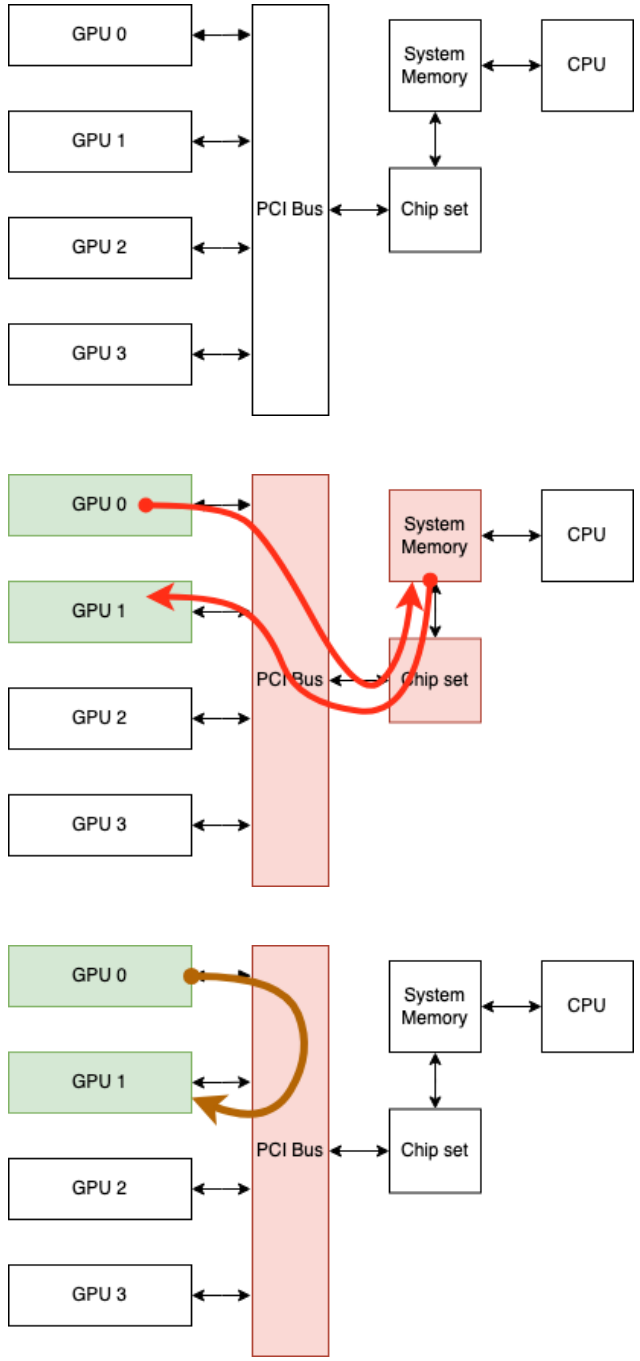
Returns

Async work handle, if async_op is set to True. None, if not async_op or if not part of the group.

NVLink and NVSwitch

Base System PCIe

- GPUs connected to a motherboard via the PCIe;
- If there is **no peer access**, the data would first be copied from GPU 0 to the system memory, and then copied from the system memory to GPU 1.
- If the system provides **peer access** over PCIe, then the data would only be copied once. Note that the data still flows over a relatively slow PCIe interface.

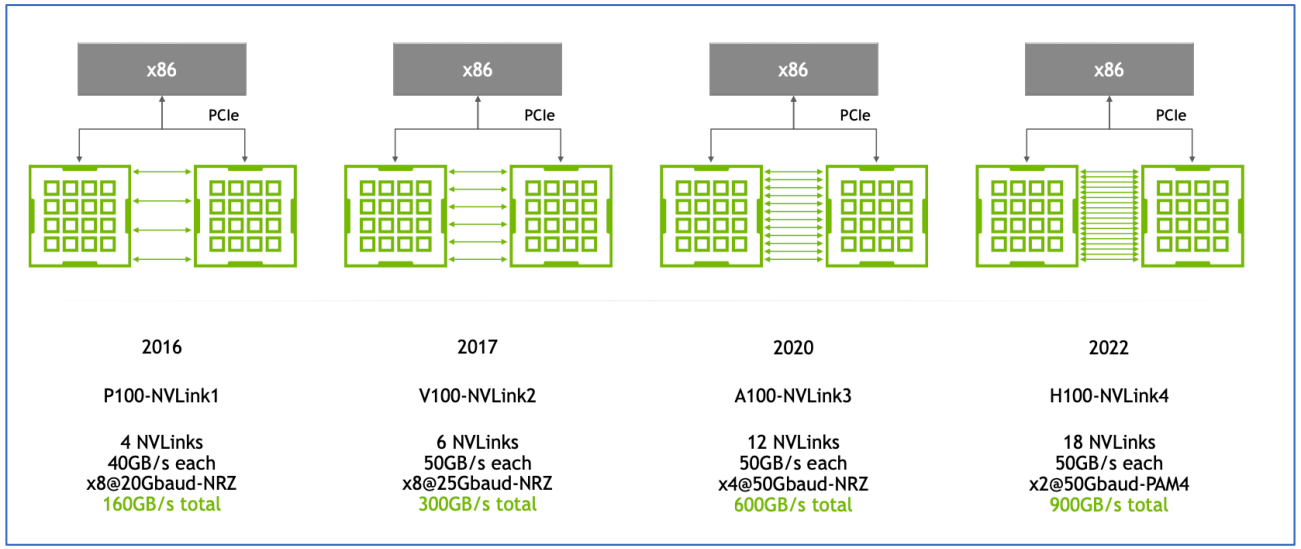


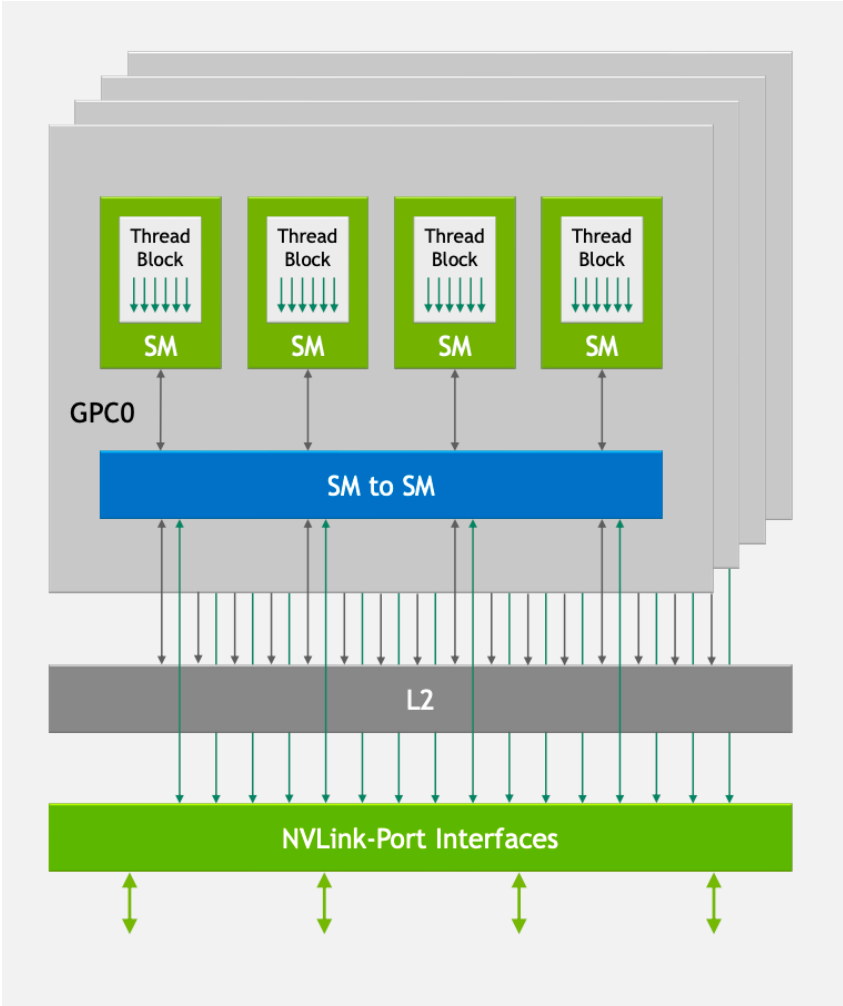
RELAXED SYSTEM LAB

NVlink

RELAXED SYSTEM LAB

- Much faster connection:
  - 100 Gbps-per-lane (NVLink4) vs 32Gbps-per-lane (PCIe Gen5);
  - Multiple NVLinks can be "ganged" to realize higher aggregate lane counts.

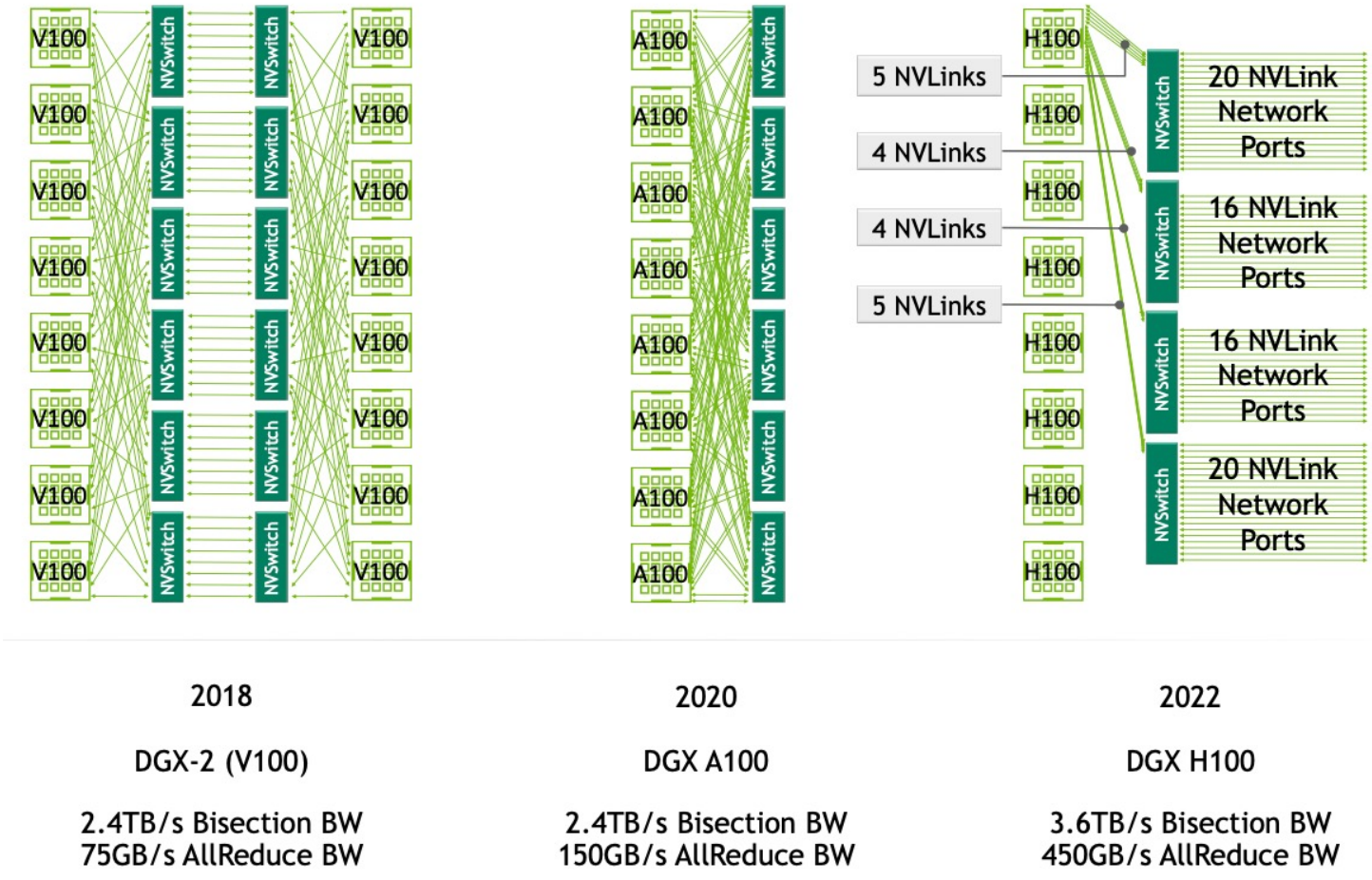




NVSwitch

RELAXED
SYSTEM LAB

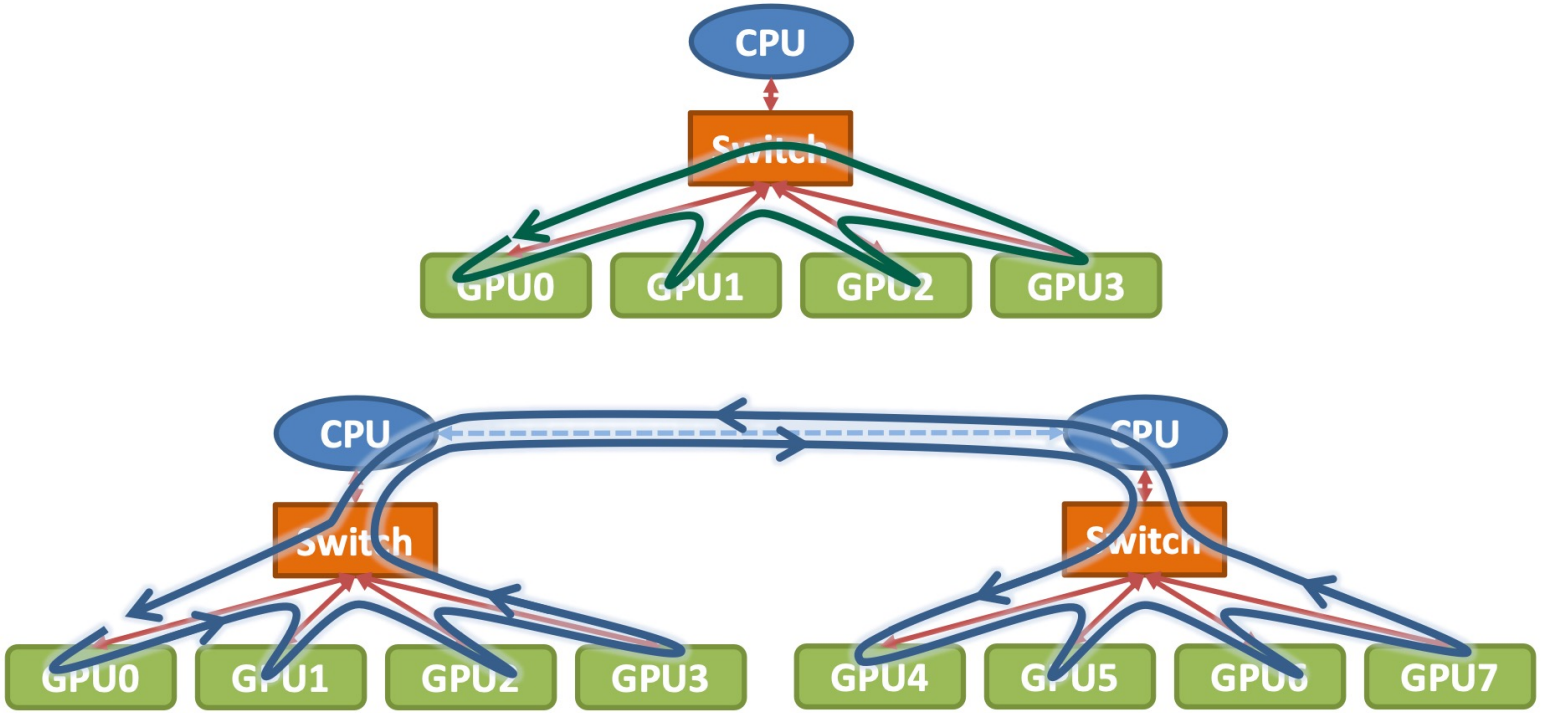
- NVSwitch is an NVLink switch chip;
- NVSwitch facilitate s seamless, high-bandwidth communication between multiple GPUs by interconnecting GPUs through NVLink interfaces.



NCCL Implementation & Optimization

NCCL Implementation

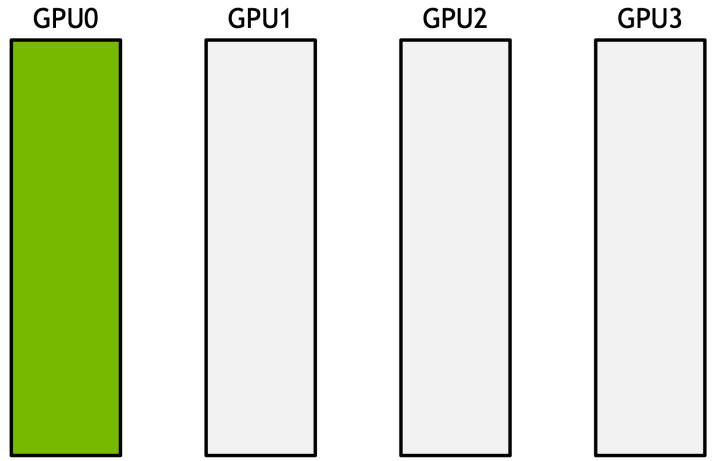
RELAXED
SYSTEM LAB



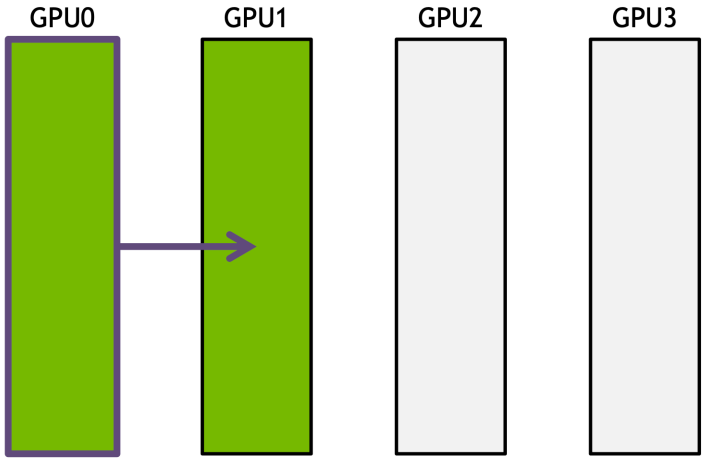
NCCL uses ***rings*** to move data across all GPUs and perform reductions.

Naïve Implementation of Broadcast

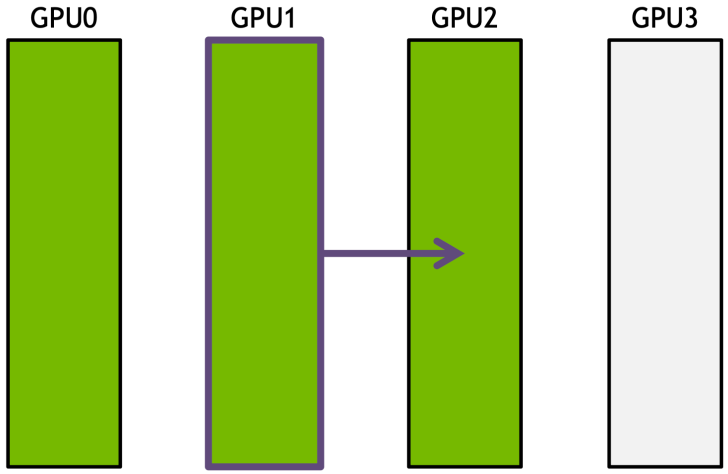
RELAXED SYSTEM LAB



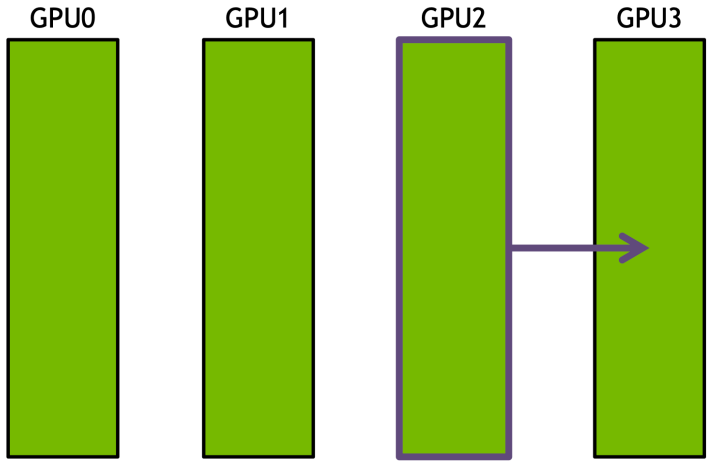
Step 0



Step 1



Step 2

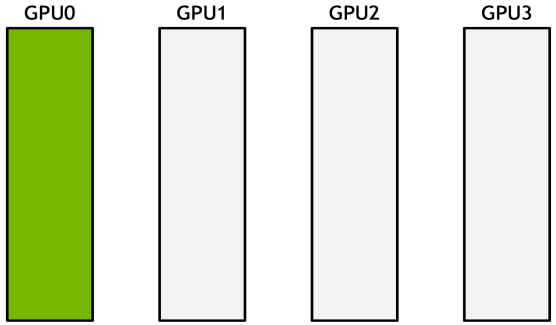


Step 3

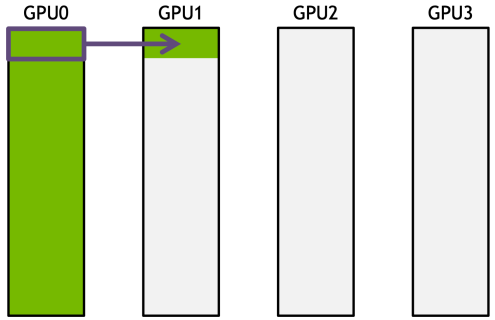
- N: bytes to broadcast
- B: bandwidth of each link
- k: number of GPUs
- Total time: $\mathrm{T} = \frac{(k-1)N}{B}$.

Optimized Implementation of Broadcast

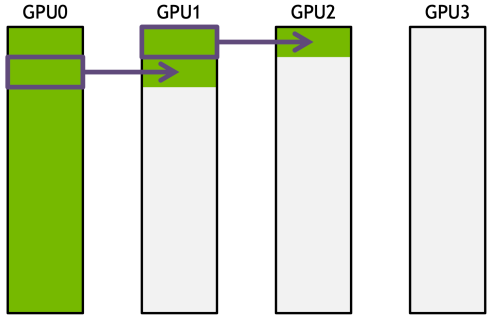
RELAXED SYSTEM LAB



Step 0

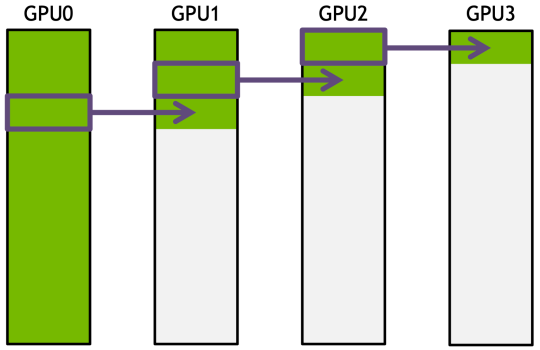


Step 1

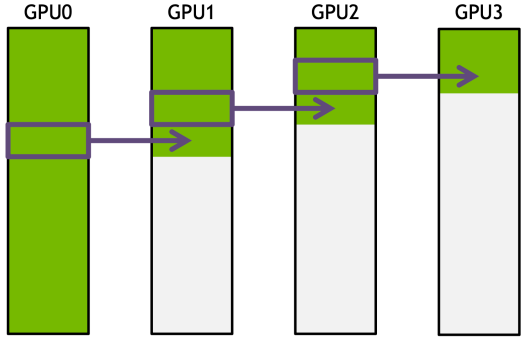


Step 2

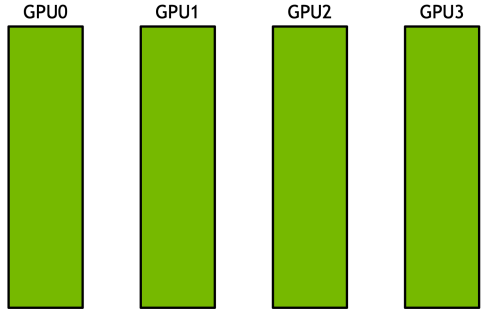
- N: bytes to broadcast
- B: bandwidth of each link
- k: number of GPUs
- Split data to s message.
- Each step $t = \frac{N}{SB}$
- Total time:
  $\frac{(S+k-2)N}{SB} \rightarrow \frac{N}{B}$.



Step 3



Step 4



By the end

Ring based AllReduce

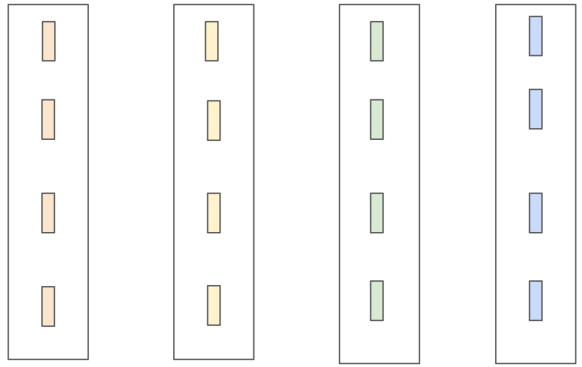
RELAXED
SYSTEM LAB

- In ring based AllReduce, we assume:
  - N: bytes to aggregate
  - B: bandwidth of each link
  - k: number of GPUs
  - The original tensor is equally split into k chunks.
- Ring based AllReduce implementation has two phases:
  - Reduction phrase (Aggregation phrase);
  - AllGather Phrase.
  - Total time: $\frac{2(k-1)N}{kB}$.

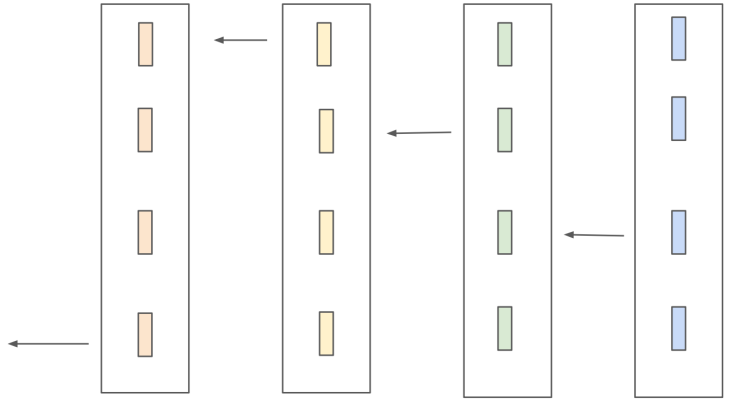
Ring based AllReduce- Reduction Phrase

RELAXED SYSTEM LAB

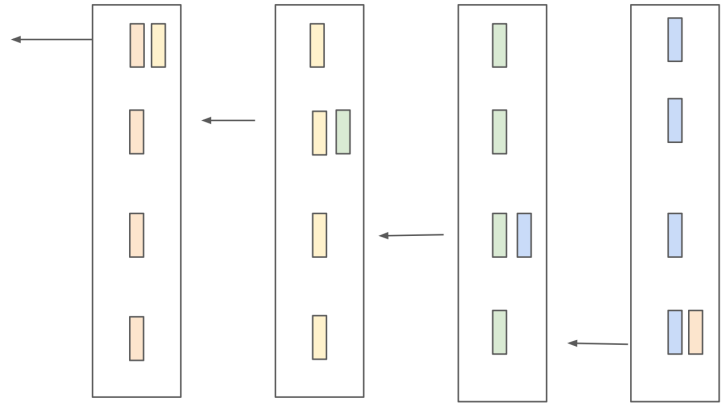
In this visualization, two or more blocks mean the aggregation results of two or more blocks by the same shape as the original block.



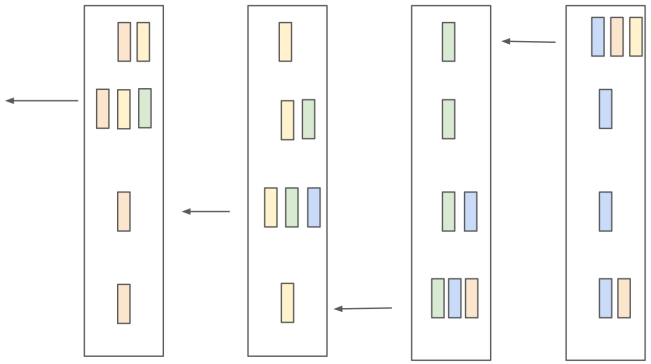
Initial State



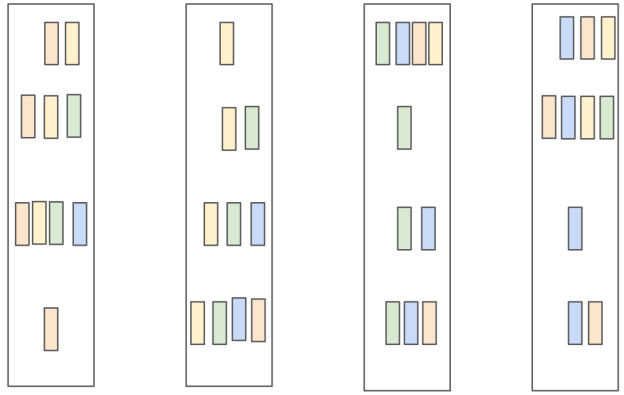
Step 0



Step 1



Step 2



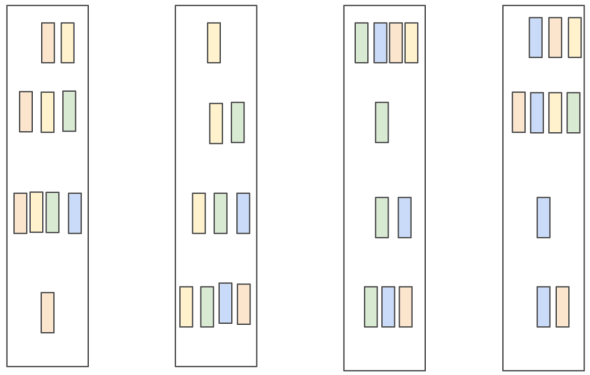
Step 3

- N: bytes to aggregate
- B: bandwidth of each link
- k: number of GPUs
- Time in reduction phase: $\frac{(k-1)N}{kB}$.

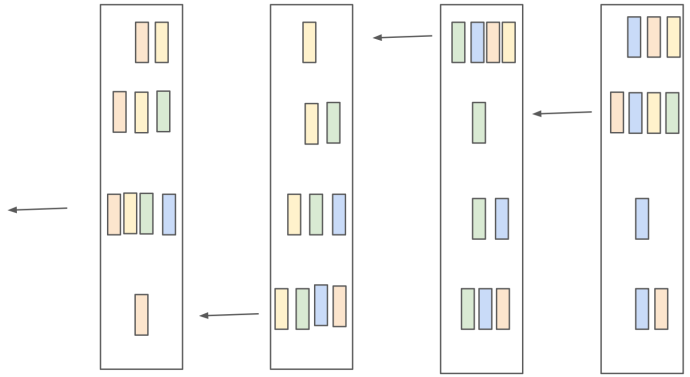
Ring based AllReduce

RELAXED SYSTEM LAB

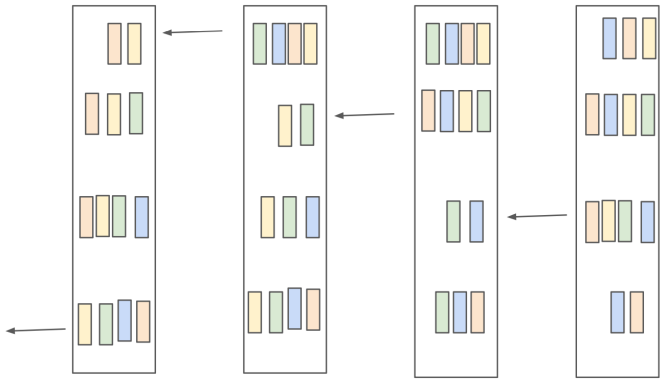
In this visualization, two or more blocks mean the aggregation results of two or more blocks by the same shape as the original block.



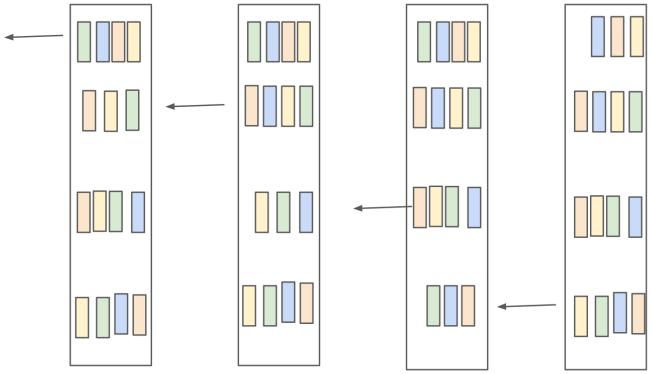
Initial State



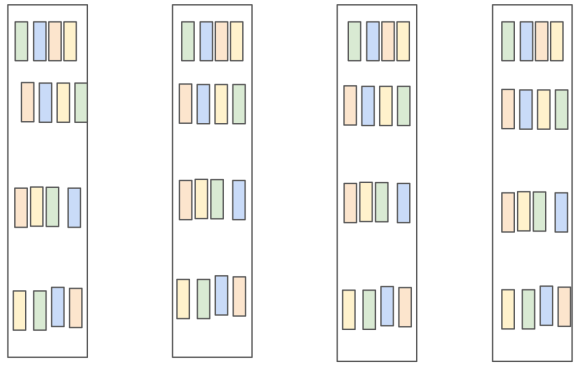
Step 0



Step 1



Step 2



Step 3

- N: bytes to broadcast
- B: bandwidth of each link
- k: number of GPUs
- Time in AllGather phase: $\frac{(k-1)N}{kB}$.

RELAXED
SYSTEM LAB

NCCL Practice in PyTorch

PyTorch Distributed NCCL Backend

RELAXED SYSTEM LAB

- PyTorch Distributed Process Groups:
  - Process groups (PG) take care of communications across processes. It is up to users to decide how to place processes, e.g., on the same machine or across machines. PG exposes a set of communication APIs, e.g., send, recv, and the collective communication operators.
- Process Group Backends:
  - Gloo: for distributed CPU training;
  - NCCL: for distributed GPU training;
  - MPI.

Backend	`gloo`		`mpi`		`nccl`	
Device	CPU	GPU	CPU	GPU	CPU	GPU
send	✓	✘	✓	?	✘	✓
recv	✓	✘	✓	?	✘	✓
broadcast	✓	✓	✓	?	✘	✓
all_reduce	✓	✓	✓	?	✘	✓
reduce	✓	✘	✓	?	✘	✓
all_gather	✓	✘	✓	?	✘	✓
gather	✓	✘	✓	?	✘	✓
scatter	✓	✘	✓	?	✘	✓
reduce_scatter	✘	✘	✘	✘	✘	✓
all_to_all	✘	✘	✓	?	✘	✓
barrier	✓	✘	✓	?	✘	✓

Initialize the Process Group.

RELAXED SYSTEM LAB

- The PyTorch distributed package needs to be initialized using the torch.distributed.init_process_group() function before calling any other methods.
- The process will block until all processes have joined.
- By default collectives operate on the default group (also called the world) and require all processes to enter the distributed function call.
- However, some workloads can benefit from more fine-grained communication.
- new_group() function can be used to create new groups, with arbitrary subsets of all processes. It returns an opaque group handle that can be given as a group argument to all collectives.

torch.distributed.init_process_group(*backend=None, init_method=None, timeout=None, world_size=-1, rank=-1, store=None, group_name='', pg_options=None*) [SOURCE]

Initialize the default distributed process group.

This will also initialize the distributed package.

There are 2 main ways to initialize a process group:

1. Specify `store`, `rank`, and `world_size` explicitly.
2. Specify `init_method` (a URL string) which indicates where/how to discover peers. Optionally specify `rank` and `world_size`, or encode all required parameters in the URL and omit them.

If neither is specified, `init_method` is assumed to be "env://".

Parameters

- **backend** (*str or Backend, optional*) – The backend to use. Depending on build-time configurations, valid values include `mpi`, `gloo`, `nccl`, and `ucc`. If the backend is not provided, then both a `gloo` and `nccl` backend will be created, see notes below for how multiple backends are managed. This field can be given as a lowercase string (e.g., `"gloo"`), which can also be accessed via `Backend` attributes (e.g., `Backend.GLOO`). If using multiple processes per machine with `nccl` backend, each process must have exclusive access to every GPU it uses, as sharing GPUs between processes can result in deadlocks. `ucc` backend is experimental.
- **init_method** (*str, optional*) – URL specifying how to initialize the process group. Default is "env://" if no `init_method` or `store` is specified. Mutually exclusive with `store`.
- **world_size** (*int, optional*) – Number of processes participating in the job. Required if `store` is specified.
- **rank** (*int, optional*) – Rank of the current process (it should be a number between 0 and `world_size`-1). Required if `store` is specified.
- **store** (*Store, optional*) – Key/value store accessible to all workers, used to exchange connection/address information. Mutually exclusive with `init_method`.
- **timeout** (*timedelta, optional*) – Timeout for operations executed against the process group. Default value is 10 minutes for NCCL and 30 minutes for other backends. This is the duration after which collectives will be aborted asynchronously and the process will crash. This is done since CUDA execution is async and it is no longer safe to continue executing user code since failed async NCCL operations might result in subsequent CUDA operations running on corrupted data. When TORCH_NCCL_BLOCKING_WAIT is set, the process will block and wait for this timeout.
- **group_name** (*str, optional, deprecated*) – Group name. This argument is ignored
- **pg_options** (*ProcessGroupOptions, optional*) – process group options specifying what additional options need to be passed in during the construction of specific process groups. As of now, the only options we support is `ProcessGroupNCCL.Options` for the `nccl` backend, `is_high_priority_stream` can be specified so that the nccl backend can pick up high priority cuda streams when there're compute kernels waiting.

Blocking Mode VS. Non-Blocking Mode

RELAXED SYSTEM LAB

- Blocking mode: all processes stop until the communication is completed.
- Non-blocking: the script continues its execution and the methods return a Work object upon which we can choose to wait().

Blocking Mode

```
def init_process(rank, backend='nccl'):
    """ Initialize the distributed environment. """
    os.environ['MASTER_ADDR'] = '127.0.0.1'
    os.environ['MASTER_PORT'] = '29500'
    dist.init_process_group(backend, rank=rank, world_size=2)

def run(rank, size, device):
    tensor = torch.zeros(10).to(device)
    if rank == 0:
        tensor += 1
        # Send the tensor to process 1
        dist.send(tensor=tensor, dst=1)
    else:
        # Receive tensor from process 0
        dist.recv(tensor=tensor, src=0)
    print('Rank ', rank, ' has data ', tensor[0])
```

Blocking Mode VS. Non-Blocking Mode

RELAXED SYSTEM LAB

- Blocking mode: all processes stop until the communication is completed.
- Non-blocking: the script continues its execution and the methods return a Work object upon which we can choose to wait().

Non-Blocking Mode

```
def init_process(rank, backend='nccl'):
    """ Initialize the distributed environment. """
    os.environ['MASTER_ADDR'] = '127.0.0.1'
    os.environ['MASTER_PORT'] = '29500'
    dist.init_process_group(backend, rank=rank, world_size=2)

def run(rank, size):
    tensor = torch.zeros(10).to(device)
    req = None
    if rank == 0:
        tensor += 1
        # Send the tensor to process 1
        req = dist.isend(tensor=tensor, dst=1)
        print('Rank 0 started sending')
    else:
        # Receive tensor from process 0
        req = dist.irecv(tensor=tensor, src=0)
        print('Rank 1 started receiving')
    # Call print('Rank ', rank, ' has data ', tensor[0]) here may present incorrect results.
    req.wait()
    print('Rank ', rank, ' has data ', tensor[0])
```

Blocking Mode VS. Non-Blocking Mode

RELAXED SYSTEM LAB

- For collective communications, you should set the **async_op** to determine whether to run it in blocking mode (**async_op =False** by default) or Non-blocking mode (**async_op =True**).

Blocking Mode

```
def init_process(rank, backend='nccl'):
    """ Initialize the distributed environment. """
    os.environ['MASTER_ADDR'] = '127.0.0.1'
    os.environ['MASTER_PORT'] = '29500'
    dist.init_process_group(backend, rank=rank, world_size=4)

def run(rank, size, device):
    tensor = torch.ones(10).to(device)
    dist.all_reduce(tensor, op=dist.ReduceOp.SUM)
    print('Rank ', rank, ' has data ', tensor[0])
```

Blocking Mode VS. Non-Blocking Mode

RELAXED SYSTEM LAB

- For collective communications, you should set the **async_op** to determine whether to run it in blocking mode (**async_op =False** by default) or Non-blocking mode (**async_op =True**).

Non-Blocking Mode

```
def init_process(rank, backend='nccl'):
    """ Initialize the distributed environment. """
    os.environ['MASTER_ADDR'] = '127.0.0.1'
    os.environ['MASTER_PORT'] = '29500'
    dist.init_process_group(backend, rank=rank, world_size=4)

def run(rank, size, device):
    tensor = torch.ones(10).to(device)
    handle = dist.all_reduce(tensor, op=dist.ReduceOp.SUM, async_op=True)
    # Call print('Rank ', rank, ' has data ', tensor[0]) here may present incorrect results.
    handle.wait()
    print('Rank ', rank, ' has data ', tensor[0])
```

References

- https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html#understand-perf
- https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/
- https://www.alcf.anl.gov/sites/default/files/2021-07/ALCF_A100_20210728%5B80%5D.pdf
- https://hc34.hotchips.org/assets/program/conference/day2/Network%20and%20Switches/NVSwitch%20HotChips%202022%20r5.pdf
- https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/collectives.html#collective-operations
- https://pytorch.org/tutorials/intermediate/dist_tuto.html
- https://images.nvidia.com/events/sc15/pdfs/NCCL-Woolley.pdf
- https://dlsys.cs.washington.edu/pdf/lecture11.pdf
- https://docs.mstarcfd.com/KB/peer-access.html