NCCL

<https://github.com/NVIDIA/nccl>

NCCL 2.x

Key Features

Multi-gpu and multi-node communication collectives such as all-gather, all-reduce, broadcast, reduce, reduce-scatter

Automatic topology detection to determine optimal communication path

Optimized to achieve high bandwidth over PCIe and NVLink high-speed interconnect

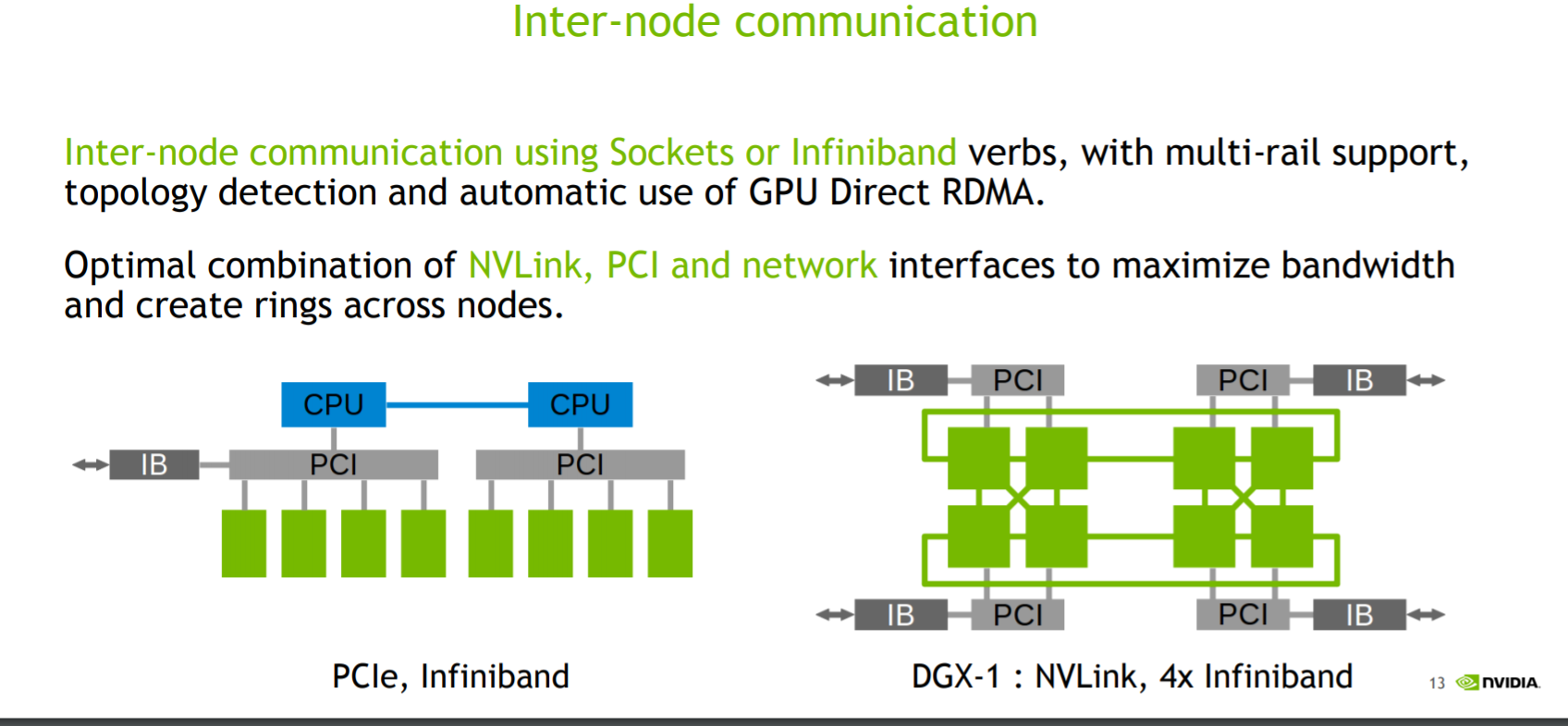
Support multi-threaded and multiprocess applications

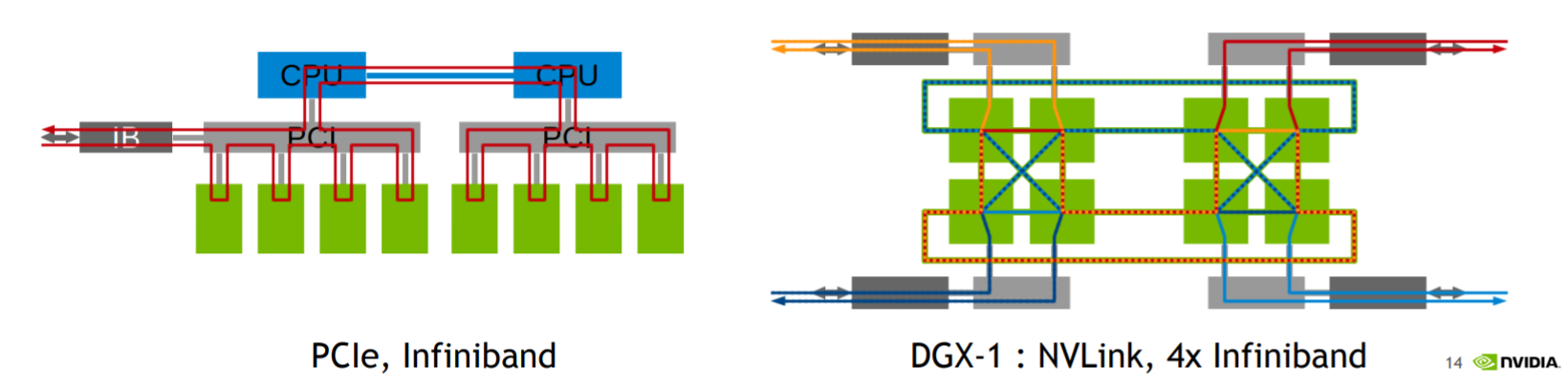
Multiple ring formations for high bus utilization

Support for InfiniBand verbs, RoCE and IP Socket internode communication

Note: NCCL support for internode communication.. (the node here means CPU node???, it works for multi CPU (each CPU with multi-GPU) ??)

NCCL 2.0 Inter-node communication using Sockets or Infiniband verbs, with multi-rail support, topology detection and automatic use of GPU Direct RDMA. Optimal combination of NVLink, PCI and network interfaces to maximize bandwidth and create rings across nodes.





Per: <http://on-demand.gputechconf.com/gtc/2017/presentation/s7155-jeaugey-nccl.pdf>

# NCCL history

**Q4 2015**: NCCL 1.x Open-source research project on github, helping Deep Learning frameworks compute on multiple GPUs with efficient collective operations. Limited to intra-node.

**Q2 2017**: NCCL 2.x and beyond NVIDIA Library, multi-node support and improved API.

For NCCL 2.0

## [NCCL Release 2.0.2](https://docs.nvidia.com/deeplearning/sdk/nccl-release-notes/rel_2.0.2.html#rel_2.0.2)

## **Key Features and Enhancements**

This NCCL release includes the following key features and enhancements.

* NCCL 2.0.2 provides support for intra-node and inter-node communication.
* NCCL optimizes intra-node communication using NVLink, PCI express, and shared memory.
* Between nodes, NCCL implements fast transfers over sockets or InfiniBand verbs.
* GPU-to-GPU and GPU-to-Network direct transfers, using the GPU Direct technology, is extensively used when the hardware topology permits it.

## [NCCL Release 2.1.2](https://docs.nvidia.com/deeplearning/sdk/nccl-release-notes/rel_2.1.2.html#rel_2.1.2)

## **Key Features and Enhancements**

This NCCL release includes the following key features and enhancements.

* New algorithms for improved latency communication
* RoCE support

## [NCCL Release 2.1.4](https://docs.nvidia.com/deeplearning/sdk/nccl-release-notes/rel_2.1.4.html#rel_2.1.4)

## **Key Features and Enhancements**

This NCCL release includes the following key features and enhancements.

* Added support for InfiniBand GID selection, enabling the use of RoCE v2.
* Added support for InfiniBand Service Level (SL) selection.

Using NCCL to do reduction

<https://devblogs.nvidia.com/fast-multi-gpu-collectives-nccl/>

And in the nccl-dev-guide

<https://docs.nvidia.com/deeplearning/sdk/nccl-developer-guide/index.html>

NCCL conveniently removes the need for developers to optimize their applications for specific machines. NCCL provides fast collectives over multiple GPUs both within and across nodes. It supports a variety of interconnect technologies including PCIe, NVLink™ , InfiniBand Verbs, and IP sockets. NCCL also automatically patterns its communication strategy to match the system’s underlying GPU interconnect topology.

<https://arxiv.org/pdf/1802.05799.pdf>

Horovod: fast and easy distributed deep learning in TensorFlow

from Uber

. We replaced the Baidu ring-allreduce implementation with NCCL [13]. NCCL is NVIDIA’s

library for collective communication that provides a highly optimized version of ringallreduce.

NCCL 2 introduced the ability to run ring-allreduce across multiple machines,

enabling us to take advantage of its many performance boosting optimizations

<https://eng.uber.com/horovod/>

<https://github.com/uber/horovod>

Pytorch is also using NCCL

<https://pytorch.org/tutorials/intermediate/dist_tuto.html>

Gloo Backend

The Gloo backend provides an optimized implementation of collective communication procedures, both for CPUs and GPUs. It particularly shines on GPUs as it can perform communication without transferring data to the CPU’s memory using GPUDirect. It is also capable of using NCCL to perform fast intra-node communication and implements its own algorithms for inter-node routines.

And for Gloo

<https://github.com/facebookincubator/gloo>

Gloo is a collective communications library. It comes with a number of collective algorithms useful for machine learning applications. These include a barrier, broadcast, and allreduce.

Transport of data between participating machines is abstracted so that IP can be used at all times, or InifiniBand (or RoCE) when available. In the latter case, if the InfiniBand transport is used, GPUDirect can be used to accelerate cross machine GPU-to-GPU memory transfers.

If NCCL 2.x can do reduce across machine, then Gloo is completely valueless....

And another post:

Sylvain Jeaugey

Nvidia

sjeaugey commented on 4 Jan 2017

NCCL does not support inter-node communication. Also, it launches CUDA kernels on the user-provided stream, so that communication can be performed asynchronously and concurrently with other CUDA kernels.

sjeaugey commented on 5 Aug 2017

All,

NCCL2 supports multi-node communication and is now available for download at <https://developer.nvidia.com/nccl>.

It does not depend on MPI but indeed if you want to run on multiple nodes, you will need some mechanism to manage the multiple processes as well as CPU-to-CPU communication, which is why MPI is a good complement to NCCL2 when running across nodes.

According to <https://images.nvidia.com/events/sc15/pdfs/NCCL-Woolley.pdf>

INTRODUCING NCCL Accelerating multi-GPU collective communications

GOAL:

• Build a research library of accelerated collectives that is easily integrated and topology-aware so as to improve the scalability of multi-GPU applications

APPROACH:

• Pattern the library after MPI’s collectives

• Handle the intra-node communication in an optimal way

• **Provide the necessary functionality for MPI to build on top to handle inter-node**

So, NCCL should depends on MPI to do the inter-node stuff.

# Performance improvement

Accroding to this article, NCCL can NOT bring overall performance improvement, though it can boost the agg gradients.

<https://www.tensorflow.org/performance/performance_models>

NCCL

In order to broadcast variables and aggregate gradients across different GPUs within the same host machine, we can use the default TensorFlow implicit copy mechanism.

However, we can instead use the optional NCCL (tf.contrib.nccl) support. NCCL is an NVIDIA® library that can efficiently broadcast and aggregate data across different GPUs. It schedules a cooperating kernel on each GPU that knows how to best utilize the underlying hardware topology; this kernel uses a single SM of the GPU.

In our experiment, we demonstrate that although NCCL often leads to much faster data aggregation by itself, it doesn't necessarily lead to faster training. Our hypothesis is that the implicit copies are essentially free since they go to the copy engine on GPU, as long as its latency can be hidden by the main computation itself. Although NCCL can transfer data faster, it takes one SM away, and adds more pressure to the underlying L2 cache. Our results show that for 8-GPUs, NCCL often leads to better performance. However, for fewer GPUs, the implicit copies often perform better.

NCCL API

Code:

The ncclAllReduce function reduces data arrays of length count in sendbuff using op operation and leaves identical copies of the result on each recvbuff.

ncclResult\_t ncclAllReduce(const void\* sendbuff, void\* recvbuff, size\_t

count,

ncclDataType\_t datatype, ncclRedOp\_t op, ncclComm\_t comm, cudaStream\_t

stream);

In distributed environment, how the sendbuff and recvbuff be mapped to remote nodes???

2.3. Data Pointers

In general NCCL will accept any CUDA pointers that are accessible from the CUDA device associated to the communicator object. This includes:

device memory local to the CUDA device

host memory registered using CUDA SDK APIs cudaHostRegister or cudaGetDevicePointer

managed and unified memory

The only exception is device memory located on another device but accessible from the current device using peer access. NCCL will return an error in that case to avoid programming errors (only when NCCL\_CHECK\_POINTERS=1 since 2.2.12).

!!! important !!!

So, the data pointers can be The only exception is device memory located on another device but accessible from the current device using peer access

3.2.6.4. Peer-to-Peer Memory Access

When the application is run as a 64-bit process, devices of compute capability 2.0 and higher from the Tesla series may address each other's memory (i.e., a kernel executing on one device can dereference a pointer to the memory of the other device). This peer-to-peer memory access feature is supported between two devices if cudaDeviceCanAccessPeer() returns true for these two devices.

Peer-to-peer memory access must be enabled between two devices by calling cudaDeviceEnablePeerAccess() as illustrated in the following code sample. Each device can support a system-wide maximum of eight peer connections.

A unified address space is used for both devices (see Unified Virtual Address Space), so the same pointer can be used to address memory from both devices as shown in the code sample below.

cudaSetDevice(0); // Set device 0 as current

float\* p0;

size\_t size = 1024 \* sizeof(float);

cudaMalloc(&p0, size); // Allocate memory on device 0

MyKernel<<<1000, 128>>>(p0); // Launch kernel on device 0

cudaSetDevice(1); // Set device 1 as current

cudaDeviceEnablePeerAccess(0, 0); // Enable peer-to-peer access

// with device 0

// Launch kernel on device 1

// This kernel launch can access memory on device 0 at address p0

MyKernel<<<1000, 128>>>(p0);

Peer-to-peer requires that the two GPUs are connected to the same PCIe root complex. These days, CPUs incorporate a PCIe root complex, so only GPUs connected to the same CPU socket can do peer-to-peer.

For managed and unified memory, and the peer-to-peer, they need to be find by the host!!!!

cudaDeviceCanAccessPeer()

cudaDeviceEnablePeerAccess()

...

cudaSetDevice(gpuid\_0);

cudaDeviceDisablePeerAccess(gpuid\_1);

cudaSetDevice(gpuid\_1);

cudaDeviceDisablePeerAccess(gpuid\_0);

NCCL in TF

<https://www.tensorflow.org/performance/performance_models>

Gradient aggregation across the server can be done in different ways:

Using standard TensorFlow operations to accumulate the total on a single device (CPU or GPU) and then copy it back to all GPUs.

Using NVIDIA® NCCL, described below in the NCCL section.

So, NCCL is used only in Gradient agg???

NCCL performance improvement:

Our results show that for 8-GPUs, NCCL often leads to better performance. However, for fewer GPUs, the implicit copies often perform better.

<https://www.tensorflow.org/api_docs/python/tf/contrib/nccl>

Functions

all\_max(...): Returns a list of tensors with the all-reduce max across tensors.

all\_min(...): Returns a list of tensors with the all-reduce min across tensors.

all\_prod(...): Returns a list of tensors with the all-reduce product across tensors.

all\_sum(...): Returns a list of tensors with the all-reduce sum across tensors.

broadcast(...): Returns a tensor that can be efficiently transferred to other devices.

reduce\_sum(...): Returns a tensor with the reduce sum across tensors.

<https://www.tensorflow.org/api_docs/python/tf/contrib/all_reduce>

all\_sum uses \_apply\_all\_reduce to do the all sum reduce.

from tensorflow.contrib.nccl.ops import gen\_nccl\_ops

def \_apply\_all\_reduce(reduction, tensors):

"""Helper function for all\_\* functions."""

if not tensors:

raise ValueError('Must pass >0 tensors to all reduce operations')

\_validate\_and\_load\_nccl\_so()

shared\_name = \_get\_shared\_name()

res = []

for t in tensors:

\_check\_device(t)

with ops.device(t.device):

res.append(

gen\_nccl\_ops.nccl\_all\_reduce(

input=t,

reduction=reduction,

num\_devices=len(tensors),

shared\_name=shared\_name))

return res

from tensorflow.contrib import nccl

def build\_nccl\_all\_reduce(input\_tensors, red\_op, un\_op=None):

output\_tensors = nccl.all\_sum(input\_tensors)

!!!! Construct hybrid of NCCL within workers, Ring across workers.

<https://www.tensorflow.org/api_docs/python/tf/contrib/all_reduce/build_nccl_then_ring>

tf.contrib.all\_reduce.build\_nccl\_then\_ring(

input\_tensors,

subdiv,

red\_op,

un\_op=None

)

def build\_nccl\_then\_ring(input\_tensors, subdiv, red\_op, un\_op=None):

"""Construct hybrid of NCCL within workers, Ring across workers."""

def upper\_builder(y):

return build\_ring\_all\_reduce(y, len(y), subdiv, [0], red\_op, un\_op)

def upper\_level\_f(x):

return \_reduce\_non\_singleton(x, upper\_builder, un\_op)

return \_build\_nccl\_hybrid(input\_tensors, red\_op, upper\_level\_f)

!!!!! there is NCCL version of ring all reduce....

<https://www.tensorflow.org/api_docs/python/tf/contrib/all_reduce/build_ring_all_reduce>

tensorflow/contrib/nccl/ops/nccl\_ops.cc

REGISTER\_OP("NcclAllReduce")

REGISTER\_OP("NcclReduce")

REGISTER\_OP("\_NcclReduceSend")

REGISTER\_OP("\_NcclReduceRecv")

REGISTER\_OP("NcclBroadcast")

REGISTER\_OP("\_NcclBroadcastSend")

REGISTER\_OP("\_NcclBroadcastRecv")

Questions:

1. How to use NCCL in distributed environment?

<https://devtalk.nvidia.com/default/topic/1029922/?comment=5239141>

<https://devtalk.nvidia.com/default/topic/1032266>.

These two posts did not answer the question either.

# References

<https://images.nvidia.com/events/sc15/pdfs/NCCL-Woolley.pdf>

<https://docs.nvidia.com/deeplearning/sdk/nccl-developer-guide/index.html#overview>