Fast Multi-GPU communication over PCI Express

Sivert Andresen Cubedo

Thesis submitted for the degree of
Master in Distributed systems and networks
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2021

Fast Multi-GPU communication over PCI Express

Sivert Andresen Cubedo

© 2021 Sivert Andresen Cubedo

Fast Multi-GPU communication over PCI Express

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

Today the demand for large-scale Machine Learning (ML) models is increasing. Training such models require more and more hardware resources. Distributing ML training is a way to reduce training time. However, this depends on the ability of machines to work together.

In this thesis, we have developed a proof of concept plugin for the NVIDIA Collective Communication Library (NCCL), enabling inter-machine PCIe communication. NCCL is a state-of-the-art Collective Operations library for Nvidia GPUs. Our plugin is implemented using Dolphin NTB adapters, allowing for inter-machine PCIe communication.

We are able to show that network interconnects do affect distributed ML training time. Our plugin is able to make the Collective Operation time insignificant compared to the computation time when training ML models.

Contents

1 Introduction **1**
1.1 Background 1
1.2 Problem Definition 4
1.3 Main Contributions 4
1.4 Limitations 4
1.5 Research Method 5
1.6 Chapter Summary 5

2 Technologies **7**
2.1 PCIe 7
2.2 Non-Transparent Bridging 8
2.2.1 SISCI API 8
2.3 NCCL 9
2.4 TensorFlow 10
2.5 Chapter Summary 10

3 NCCL Implementation **11**
3.1 NCCL 11
3.2 Terminology 11
3.3 Bootstrap 12
3.3.1 Initial Communication 13
3.3.2 Topology 14
3.3.3 Communication 15
3.4 Collective Calls 16
3.4.1 Broadcast 17
3.4.2 Reduce 17
3.4.3 All Gather 17
3.4.4 All Reduce 17
3.4.5 Reduce Scatter 18
3.5 Async Interface 18
3.6 Chapter Summary 18

4 SISCI-NCCL Implementation **19**
4.1 Other NCCL plugins 20
4.1.1 AWS OFI NCCL 20
4.1.2 Mellanox Plugins 20
4.2 Creating the Plugin 20

4.2.1 Error Management 20
4.2.2 Plugin Interface 21
4.2.3 Device Properties 24
4.2.4 Allocating Segments 26
4.2.5 Channels 27
4.2.6 Mailbox 27
4.2.7 Establishing Connections 28
4.2.8 Register Memory 32
4.2.9 Transfer Data 33
4.2.10 Cleanup 34
4.3 IOMMU and GPUDirect Problems 34
4.4 Emergency Solution 36
4.5 Future work 37
4.6 Chapter Summary 37

5 NCCL Benchmark **39**
5.1 NCCL tests 39
5.2 Variables 39
5.3 Metrics 40
5.4 Benchmark Technique 41
5.5 Hardware 41
5.6 Benchmark Results 41
5.7 Chapter Summary 44

6 Tensorflow Benchmark **47**
6.1 Installation 47
6.2 Metrics 47
6.3 Model Configuration 48
6.4 Tensorflow Profiler 48
6.5 Messuring System Usage 49
6.6 Execution Enviroment 49
6.7 Benchmark Results 50
6.8 Chapter Summary 51

7 Conclusion **53**
7.1 Summary 53
7.2 Main Contributions 54
7.3 Future Work 54
7.3.1 Compare to Other High Performance Interconnects . 54
7.3.2 Larger Benchmarks 54

A Source Code **55**

List of Figures

3.1 Example of how identifiers are assigned to hardware. 12

5.1 Algorithm Bandwidth formula 40
5.2 All Reduce algorithm bandwidth SISCI vs socket. 42
5.3 All Reduce bus bandwidth SISCI vs socket. 43
5.4 All Reduce time SISCI vs socket. 43
5.5 All Reduce average algorithm bandwidth for all collective calls. 44
5.6 All Reduce bus bandwidth for all collective calls. 45

6.1 Average Batch Time SISCI vs socket. 50
6.2 Average Device Collective Communication Time SISCI vs socket. 51

List of Tables

2.1 Transfer performance for PCIe version 1.0 to 5.0 [24] 8

5.1 Table of hardware installed in each node. 42

6.1 Average system usage recorded during model execution. . . 51

Chapter 1

Introduction

1.1 Background

During the past decade, the general interest in Machine Learning (ML) and Deep Learning has been rising. ML, a field that used only to be developed and researched by narrow academic science communities, is now becoming mainstream. Both academic and commercial actors are pushing the field forward in both development and application.

In the context of computing, Artificial Intelligence (AI) is a collective term for systems that perform tasks that are usually dependent on human intelligence. ML is a subfield of AI, where the model learns based on input data. The input data usually represents observations. The classic example of a problem where ML is required is image classification, where we want to assign a set of known labels to a set of unknown images. Now, if the decision borders for the labels are mathematically well defined - for example, if our image is represented as a 2-dimensional matrix of pixels. We have two labels. An image is assigned label 1 if the sum of the pixels is less than N. If not, the image is assigned label 2. Then the problem is trivial to solve. However, if the decision borders are not well defined, which is usually the case for image classification, then we need our machine to perform some "human-like" evaluation.

Usually, the labels represent some human idea/construct. A simplified example of this is if we want to classify if there is a cat present in the image or not. In this case, there is no trivial way to define a pattern solving the problem. The definition of what is a cat or not becomes a philosophical question as there is an infinite number images possibly containing a cat. However, humans can still, at a quick glance, determine if a picture contains a cat or not. While there is a possibility, some people will disagree on some edge cases. The vast majority of human-made classifications will reach a consensus. As humans, we acquire the knowledge of what a cat is and what is not based on experience. Our human senses provide input that we can learn from in our environment, allowing us to perform such abstract classification based on experience. We want to perform such a classification on a computer.

In ML, we emulate the human learning process to then predict based

on experience. Instead of solving the problem by applying an algorithm directly to draw a decision border, ML uses a set of known observations to derive a border. In this example, the training set would contain pairs of images and labels. These labels are usually manually labeled by humans, and then the goal is to make the system imitate human labeling on unknown images. This process is called training and is a key mechanic of most AI methods.

Non-computer science research fields like biology, physics, medicine and mathematics are exploring the use of ML as a tool in their research arsenal. In the general case, ML can be used to automatically explore search spaces, relieving resources that otherwise would have been used on time-consuming pattern matching. An example of this is in cancer research, where researchers are training ML algorithms to detect tumors based on images, and medical data [14].

ML has also been adopted by various commercial actors both inside and outside the technology industry. Companies are able to automate tasks that used to be manually performed by humans using AI and ML. Video streaming services are employing ML algorithms to decide what users are shown when browsing content. Advertising companies are using ML to map out demographics to improve advertisement accuracy. Car companies are using ML to develop driver assistance tools to improve car safety [7].

The size and number of data sets have been increasing over the past decade [5], and it will most likely continue to increase. A consequence of this is that there is a demand for more efficient and faster ML training capabilities. To improve performance in ML algorithms, the scale, and complexity of the system increase. Deep Learning is used to describe an ML system that is using many layers of ML techniques to perform an evaluation. The goal of having many networks is for the system to extract features from the data set that are not accessible directly. While a Deep Learning system may increase the algorithm's performance, it comes at the cost of being more computationally expensive.

The computational cost of training and using a machine learning algorithm depends on the problem. Variables such as the number of features and observations in a data set, and the complexity of the algorithm dictates the computational cost. However, some of the developed ML algorithms can be distributed across multiple computing instances, allowing for horizontal scaling.

Graphics Processing Units (GPUs) were first developed for the video game industry. The original motivation behind GPUs was to hardware accelerate resource expensive video rendering. In general terms, a GPU performs homogeneous instructions in large-scale parallelism. A modern CPU will usually support parallelism having more than one physical core, where each core can execute OS threads in parallel. In addition, a modern CPU will also support parallelism per physical core using Single Instruction Multiple Data (SIMD). A GPU uses a combination of many physical cores and SIMD to achieve high bandwidth computation. This performance advantage does, however, assume the data is accessible in a vectorized form. GPU performance heavily relies on input data being

discrete. A CPU comes with features such as branch prediction to optimize branches in code. This makes CPUs ideal for computing code that heavily relies on branches. On GPUs, branches potentially come at a high cost because this means that the GPU can not handle data homogeneously. If the data can be handled with homogeneous instructions, then a GPU will probably yield a speedup compared to a CPU.

Modern ML models are usually implemented on GPUs. Most computation executed in an ML model is versions of matrix multiplication. As the order of operation is not strict when performing a matrix multiplication, it is ideal to be performed on a GPU. Usually, all samples are treated the same way in the model as well, making the computation symmetric. The size of matrices is determined by the number of tunable parameters (weights) and the number of input parameters in a layer. A weight is usually represented with a floating-point number, usually 32 bit. State-of-the-art ML models use a lot of tunable parameters, as this has shown to give better accuracy for models. EfficientNet [27] is a image classification model, the smallest version (EfficientNetB0) uses around 11 million trainable parameters. The trainable parameters will then use 0.33 GB of memory, but the structures to train the parameters uses significantly more space.

There is a demand for applying ML on larger and larger datasets. Therefore it is necessary to increase training speed. This can be done by acquiring a more capable GPU. However, this is limited by what GPU vendors are able to offer. Another way is to distribute training across many GPUs. The most common intra-machine communication method is Peripheral Component Interconnect Express (PCIe). Most GPUs are installed in a machine using PCIe, allowing CPU, GPUs, and other PCIe devices to communicate. An ML model can be distributed across multiple GPUs on a machine, allowing for higher capacity model training. It is also possible to distribute the training across multiple machines, using inter-machine communication. Both Ethernet and InfiniBand are common interconnects for inter-machine communication. This allows for GPUs located on many machines to work together.

Distributing training allows for higher bandwidth training [4]. However, the distribution comes at a cost. All execution units need to be synchronized, so they are able to work together. It is not trivial to invent a communication protocol that is able to utilize the available hardware at peak performance. So a common way to implement communication is to use an existing solution. Collective Operations are a set of communication patterns that can be used to build such an efficient protocol easily. Collective Operations are usually provided with a library such as Message Passing Interface (MPI) [9] or NVIDIA Collective Communication Library (NCCL) [21]. In this thesis, we want to look at how distributed ML training may benefit from using a PCIe interconnect for inter-machine communication.

1.2 Problem Definition

The current implementation of multi-machine multi GPU computation (NCCL) in the Nvidia ecosystem either uses RDMA (Remote Direct Memory Access) over InfiniBand or TCP over Ethernet to communicate between machines. The existing solutions can theoretically achieve as high bandwidth as a PCIe network, but these solutions heavily depend on CPU IO to communicate between machines, possibly bottlenecking the system.

In the thesis, we want to implement a proof of concept of using PCIe for inter-machine communication in NCCL. To realize this, Dolphin Interconnect Solutions (Dolphin) provides hardware that exposes PCIe capabilities for multi-machine communication. The PCIe hardware will allow for direct RDMA between CPU/GPUs (and other PCIe devices) via the PCIe hardware. NCCL supports a plugin system for third-party interconnects. We want to use this plugin system to implement support for Dolphin hardware in NCCL.

Then we want to compare the PCIe interconnect to the existing NCCL solutions. We want to benchmark how NCCL is performing in a vacuum, where only NCCL operations are benchmarked alone. As well as benchmarking NCCL in a larger context, comparing how different interconnects affect performance in a real-world ML environment.

1.3 Main Contributions

High-Performance Computing (HPC) usually relies on fast interconnects, such as InfiniBand or high-performance Ethernet solutions. Most systems today use PCIe for intra-system communication between CPU and I/O devices. There needs to be a protocol translation between the internal PCIe interconnect and exterior interconnect to transfer data between machines.

Dolphin provides hardware and software that is able to use PCIe for inter-machine communication. This allows data to be transferred only using PCIe as a protocol. Any PCIe device on one system can potentially transfer data to another PCIe device on another machine.

Our proof of concept implementation provides such functionality, allowing for direct PCIe transfers between GPUs between machines in NCCL. As long as machines are connected to the same PCIe network, NCCL is able to perform collective operations over PCIe. We provide support for both RDMA and GPU Direct RDMA in NCCL, enabling potentially more efficient hardware usage.

1.4 Limitations

Due to resource constraints, our benchmarks do not reflect a state-of-the-art HPC environment. NCCL supports InfiniBand and TCP over Ethernet by default, and these depend on the capability of hardware located on the system. We were not able to benchmark with InfiniBand due to a lack of hardware availability. As for Ethernet, our machines only support a 1

Gigabit Ethernet interface, while 40/100 Gigabit Ethernet interfaces exist. Our PCIe interconnect only supports PCIe 3.0 speeds at 8x width, giving us a theoretical bandwidth of 63 Gb/s. While PCIe 4.0 16x is capable of 252 Gb/s bandwidth.

Because we are benchmarking interconnects, the available bandwidth in both transfer and compute resources heavily affects the results. Ideally, we want a hardware configuration that outperforms the interconnect, such that the interconnect is the weakest link. Then we can measure the performance difference between interconnects. However, we were not able to acquire hardware that was able to outperform all interconnects tested.

While we implemented support for GPU Direct RDMA, the benchmark machines had faulty Input Output Memory Management Units (IOMMU). This made it impossible to enable GPU Direct RDMA for our benchmarks, so we had to fall back to using RDMA between system RAM instead. This caused at least two extra intermediate stops when transferring between 2 GPUs on remote machines.

1.5 Research Method

To implement and benchmark our implementation, we used the paradigm defined by Association for Computing Machinery [6]. The main goal of implementation was to develop a proof of concept. Then to test and validate the system. In our investigation to benchmark the implementation, we used the experimental scientific method defined in the paper. Coming up with a hypothesis. Construct a model to make predictions about our hypothesis. Then designing an experiment to test the hypothesis and collect data. And analyze the collected data.

1.6 Chapter Summary

The demand for high-performance ML and Deep Learning applications is increasing. Scientific and commercial actors are developing more complex and large ML models. Larger datasets are becoming available due to modern sensor usage and data collection methods. Training is the most resource-expensive part of developing an ML model. Researchers are usually bound by the capabilities of their hardware. Distributed training is one way to increase training capacity for an ML model. The Nvidia ecosystem provides a library created for distributed communication. We want to extend this library to use PCIe for inter-machine communication, then benchmark its performance. In the next chapter, we introduce more detailed the technology used in this thesis.

Chapter 2

Technologies

In this chapter, we introduce and discuss the technologies we use in the project.

2.1 PCIe

Peripheral Component Interconnect Express (PCIe) [24] is a serial data transfer protocol and hardware specification created in 2003. The main purpose of PCIe was to replace and improve preceding specifications such as PCI, PCI-X, and AGP. As the name implies, PCIe is a direct descendant of PCI, and the two standards are software compatible. This makes it possible to use a PCI driver on a PCIe system as long as the hardware is compatible.

The main difference between PCI and PCIe is the physical data transfer mechanism used. PCI (and PCI-X) use parallel communication to transfer data between devices. For each clock cycle, the transferring device can send 32 bits in PCI. Usually, a single 32-bit parallel bus is used to connect PCI devices together, meaning all data sent over the bus is broadcasted to all devices, only allowing for one sender at the time on the bus. Having more parallel lanes sending bits at the same time gives more bandwidth per clock cycle. However, the maximum clock speed of the transfer is decreased as more parallel lanes are added. When lanes are transferring at the same time, the interference between lanes is increasing with the number of lanes due to capacitance, making transfers unreliable. The cross-lane interference can be reduced by changing the length of different lanes on the bus or by slightly alter the send time for each lane, but this comes as a performance cost in a parallel system. Transfer time for each lane on the bus can differ, meaning the transfer always has to wait for the slowest lane in a clock cycle.

PCIe achieves a higher bandwidth than PCI (and PCI-X) even though the hardware transfers bits in a serial manner due to significantly higher clock speed. While PCIe is a serial protocol, it supports more than 1 lane going to each device. Each lane is a full-duplex connection, meaning two endpoints can transfer to and from each other at the same time. When there is more than one lane going to a device, each lane will transfer data serially in its own context.

As mentioned earlier, PCI connects all devices to a single parallel

PCIe Version	Release Year	Transfer Rate	Throughput/Lane	x16 Throughput
1.0	2003	2.5 GT/sec	250 MB/sec	4.0 GB/sec
2.0	2007	5.0 GT/sec	500 MB/sec	8.0 GB/sec
3.0	2010	8.0 GT/sec	1.0 GB/sec	16.0 GB/sec
4.0	2017	16.0 GT/sec	2.0 GB/sec	32.0 GB/sec
5.0	2019	32.0 GT/sec	4.0 GB/sec	64.0 GB/sec

Table 2.1: Transfer performance for PCIe version 1.0 to 5.0 [24]

bus. On the hardware level, PCIe is rather a point-to-point network, where all PCIe lanes are connected to a controller located at the system's root complex. The controller acts as a switch in the network, routing PCIe packets between PCIe lanes. Because every PCIe lane is connected individually to the controller, a higher throughput on the network is possible compared to PCI. More than one device can, for example, read/write to RAM at the same time on the network, whereas only one device can communicate at any moment on a PCI bus. However, the PCIe network topology limits the number of possible PCIe lanes in a given system to the number of supported lanes in the controller architecture.

As of writing this, the PCI Special Interest Group (PCI-SIG) is actively developing the PCIe standard, releasing new revisions of the standard. Table 2.1 shows the performance spec of different PCIe revisions.

2.2 Non-Transparent Bridging

As mentioned in the PCIe section, all PCIe lanes are connected to a switch located at the root complex of the system. The standard assumes there is only one root complex in a PCIe network. However, it is possible to connect more than one root complex together using a Non-Transparent Bride (NTP) [10]. An NTB acts as a PCIe device in each system, creating a communication channel between the root complexes. As the name implies, applications need to be aware of the NTB and its interface to communicate with other root complexes.

Dolphin Interconnect Solutions [25] is a hardware vendor specializing in high performance PCIe interconnect solutions. They provide PCIe NTB cards that allow for easy multi-computer PCIe networks.

2.2.1 SISCI API

To use the Dolphin NTB hardware capabilities in an application, Dolphin provides the SISCI API [26]. The goal of the API is to be an easy-to-use and safe environment for developers to use Dolphin PCIe hardware. SISCI is designed to be both architecture-independent and operating system-independent, allowing for flexible solutions. The API exposes PCIe transfer mechanisms to the user, such as Direct Memory Access (DMA), Programmed Input Output (PIO), and PCIe interrupts. In addition, the API comes with utilities for managing memory mapping and error checking.

A key abstraction in the SISCI API is what is called segments. Segments represent memory in a process that has been mapped in such a way that it can be transmitted using the SISCI API over an NTB. There are two main types of segments, local segments, and remote segments. Local segments represent memory that is mapped locally to a machine, while remote segments represent memory that can be accessed via the NTB from a remote machine. The API provides functions that can be applied to segments to initiate/handle data transfers. In particular, for this project, the SISCI API supports mapping PCIe device memory to segments, allowing direct transfer to/from devices.

SISCI also provides multiple data transport mechanisms. Interrupts allow a program to wait for a remote node to signal an event. Data interrupts do the same as regular interrupts, but the signal can carry a small extra payload when notifying the listener.

Process Input Output (PIO) allows the CPU to read/write directly to a remote segment. PIO is ideal for writing small messages to remote segments, as there is no cost in starting the transfer. However, when reading with PIO, the CPU has to make a round trip request to retrieve the data. It may be costly to use PIO for large messages, as it requires CPU resources, and the CPU write speed may bottleneck the transfer.

For larger messages, the Dolphin hardware provides a more efficient mechanism to transfer data. The NTB cards come with a DMA engine that can transfer data directly from any registered SISCI segment. While there is some cost in engaging the DMA engine, it is more efficient for large messages. As with PIO, DMA supports both read and write transfers, and the same property for reading apply. In general, it is always faster to write, given that both ends of the transfer got the same hardware capabilities. If the receiving end got hardware that supports a higher read bandwidth than the sender's write bandwidth, then a read transfer might be more performant.

2.3 NCCL

NVIDIA Collective Communication Library (NCCL) [18, 19] is a library for multi GPU processing. The goal of the library is to provide collective primitives that can be distributed across multiple machines with multiple devices(GPUs). NCCL is similar to the open Message Passing Interface (MPI) [8, 9] standard. The main difference between NCCL and MPI is that NCCL is only targeting the Nvidia HPC ecosystem, while MPI is independent of specific hardware vendors. Although NCCL only targets Nvidia hardware, it is still possible to use in conjunction with an MPI application. In this thesis, we will only focus on NCCL and the Nvidia ecosystem.

As mentioned, NCCL provides primitive building blocks that can be distributed across a GPU cluster. NCCL calls these primitives Collective Operations. Currently, the library implements five operations; AllReduce, Broadcast, Reduce, AllGather and ReduceScatter. Section 3.4 provide a

detailed description of each supported collective call.

NCCL maintains communication between individual devices on a single machine, in addition to communication between many machines. Between devices on a single machine, NCCL currently supports PCIe and NVLINK. While between machines, it supports InfiniBand and TCP/IP sockets by default. In addition, NCCL implements a plugin system for custom interconnect solutions. The plugin system allows third-party developers to implement and load different interconnect solutions easily. Section 4.1 explores some existing 3rd party plugins.

2.4 TensorFlow

TensorFlow (TF) [15, 29, 32] is an AI platform originally developed internally by Google. The platform was later open-sourced in 2015 under the Apache License 2.0 [1], granting the public access to the platform. TensorFlow provides a wide selection of algorithms and utilities to create AI models, such as Neural Networks and Deep Neural Networks.

In particular interest for this thesis, TensorFlow supports hardware-accelerated processing using Nvidia CUDA [17]. More specifically, the platform support distributed computing using NCCL. These features allow us to use TensorFlow as a drop-in benchmarking tool for our project.

2.5 Chapter Summary

There are many types of HPC hardware/systems/libraries. In this chapter, we explored some technologies we want to use to improve ML training. PCIe is a hardware and protocol specification that is almost ubiquitous for intra-machine communication. However, PCIe is not commonly used for inter-machine communication. NTB is a technique for connecting multiple PCIe networks together, like connecting multiple nodes together. Dolphin provides NTB adapters that allow us to perform inter-node communication over PCIe. SISCI is an API provided by Dolphin that simplifies implementing PCIe communication for an application. NCCL is a Collective Operations library made for GPUs in the Nvidia ecosystem. It supports Collective Operations for both inter and intra-node communication. Tensorflow is an ML framework provided by Google. It can be used to implement distributed training for ML models. Tensorflow implements distributed training using NCCL. In the next chapter, we discuss how some parts of NCCL are implemented.

Chapter 3

NCCL Implementation

In this chapter, we discuss how NCCL is implemented. Before we implement PCIe support for inter-node communication in NCCL, we explore how NCCL is implemented in general and examine how some specific features in NCCL are implemented.

3.1 NCCL

NCCL is open source, and the code is hosted on Github. This allows us to read and analyze the code and make changes if needed. For this project, NCCL supports a plugin feature that allows us to implement a custom interconnect without changing the original NCCL code. The plugin feature is further discussed in the SISCI-NCCL section.

NCCL is a bleeding-edge library and is continually being developed, publishing new releases with new features and improvements. For our analysis and implementation, we use NCCL v2.8.4-1. While the user API has stayed mostly the same over the previous releases, internal structures have changed. For example, the plugin API we use in this project has changed.

3.2 Terminology

NCCL is only focusing on using GPUs to do computation. The CPU is only used to manage memory and connections. Here is a list of words that are used to explain the system.

- *Node/Host* refers to a system in a network.
- *Device* in PCIe terns refers to hardware that is accessible to a host via PCIe. In the context of CUDA, a device refers to a GPU and resources accessible to a GPU. Devices are enumerated from 0 to n for each node. For example, if a host has 2 devices, they are accessed by index 0 and 1, respectively. NCCL does also refer to other PCIe devices to model the topology of a network, such as NICs (Network Interface Cards).

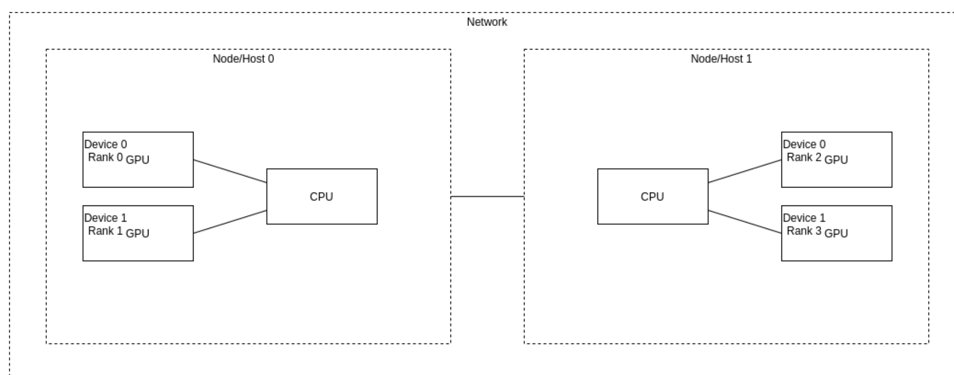

Figure 3.1: Example of how identifiers are assigned to hardware.

- *Rank* in NCCL refers to a device. Each device in an NCCL instance is assigned a unique rank. Ranks are enumerated from 0 to n.
- *Communicator/Comm* is an abstraction NCCL uses to map devices and ranks in a program.
- *Host Memory/RAM* refers to a memory that is directly accessible by a CPU. It is possible to have many CPUs in a system and Non-Uniform Memory Access (NUMA), however for simplicity, a host refers to all CPUs in a system as the same actor.
- *Device Memory/VRAM* refers to memory that is located on a GPU, and each device has its own memory. If we were to copy memory from RAM to device memory, we call it "host to device" copy. The location of memory plays a big part in the performance of an algorithm.

Figure 3.2 shows how NCCL identifies and represents hardware.

3.3 Bootstrap

In order for NCCL to perform collective operations, it needs to establish connections to all nodes first. In the simple case where we only have one node, this is trivial as CUDA has access to all devices on a system. However, NCCL needs to make the systems work together under one instance if we have more than one node. The *Bootstrap* phase is responsible for doing this.

NCCL treats all ranks as discrete units. While we can have more than one rank on a node, each rank has to negotiate and know about every other rank in the instance. Briefly summarized, the bootstrap process relies on using TCP/IP connections to distribute initial connection, topology, and hardware data. When the initial data has been distributed, each rank uses the data to select parameters for the real connections.

3.3.1 Initial Communication

NCCL assigns rank 0 as root rank. If there is more than one node, a user will call ncclGetUniqueId to get a ncclUniqueId object. The user is responsible for distributing the unique id object to all other ranks. This is trivial if there is only one node. If not, the data has to be copied with an external tool. As of NCCL v2.8.4-1, the ncclUniqueId only contains an IPv4/IPv6 address union, so NCCL can only bootstrap using an IP network. The unique id is pointing to the root rank. Then every other rank will connect to the root rank. The root rank will wait until $nranks - 1$ ranks have connected. Now the root rank knows the location (address) of every rank, and it can create an "all gather" ring of the ranks. The bootstrap process uses this ring to configure the system, as mentioned earlier. The ring is created by ordering the ranks, having each rank listen for a connection from rank rank + 1 % nranks and connect to rank rank + 1 % nranks. Then to perform an all-gather operation, each node iterates $ranks - 1$ times, storing and sending the received slice via its connection. Code listing 3.1 shows how the ring is created, and listing 3.2 shows how the ring performs an all-gather call.

While the bootstrap ring can be used to perform some collective operations, it is not ideal for High-Performance Applications. The ring created in the bootstrap phase does not take the network topology into account. The Topology phase will enable NCCL to set up a more efficient structure for collective operations.

Listing 3.1: Bootstrap create ring pseudo code

```
// This runs on all ranks

nranks = number of ranks
rank = rank this thread is executing, in range 0 to nranks − 1
root_address = address of root rank read from ncclUniqueId

next_rank = NULL // connection to next rank in the ring
prev_rank = NULL // connection to previous rank in the ring

if rank == 0 { // rank 0 is root
  // gather all ranks
  conn_list = array[nranks]
  for i = 1; i < nranks; ++i {
    listen and accept connection from rank i
    place connection data in conn_list[i]
  }
  // create all gather ring
  for i = 1; i < nranks; ++i {
    data = conn_list[i + 1 % nranks] // get next connection for rank i
    send data to rank i
  }
  next_rank = conn_list[rank + 1 % nranks] // store rank 0 next_rank
```

```
else {
  connect to root
  receive next connection data from root and store it in next_rank
}
// listen and connect must not block each other
listen for (rank - 1 % nranks) while connecting to (rank + 1 % nranks)
store connections in prev_rank and next_rank
```

Listing 3.2: Bootstrap all gather pseudo code

```
nranks = number of ranks
rank = rank this thread is executing, in range 0 to nranks - 1
next_rank = connection to next rank in the ring
slice = data to send
slices = array[nranks] // array to store data for all ranks

slices[rank] = slice
for i = 0; i < nranks - 1; ++i {
  // send and receive must not block each other
  send slices[rank - i % nranks] to next_rank
  receive data from prev_rank store in slices[rank - i - 1 % nranks]
}
// now slices in all ranks contains data from all ranks in the same ord
```

When the all-gather ring is set up, the instance will share all rank addresses. This enables each rank to create an individual connection between any other rank and is used later when setting up the proper connections.

3.3.2 Topology

The next step is to detect the topology for each rank in the system to decide the best configuration for collective calls. One way this can be done is with a user-defined configuration file. NCCL does, in addition, support automatic topology detection and configuration. This is done by having each rank detect hardware and links. The links are used to build a graph of the system based on what a rank can detect. The topology maps out the intra-node location of hardware on a machine, where devices are placed, and interconnects' speed. The topology does only model intra-machine. Inter-node routes (network topology) are not considered, only the speed of the interfaces connecting to the external network.

NCCLs network topology graph has 6 node types:

- *GPU* (Device/rank)
- *PCI* (PCIe bus) Usually the way hardware is connected
- *NVS* (NVLink) Can connect CPU and GPUs without an PCIe connection

- *CPU* (NUMA domain) A machine can have multiple CPUs, where the ram has different access times
- *NIC* (Network Interface Card)
- *NET* (Network Connection) The network topology itself is not modeled, just the endpoints

Listing 3.3 shows a XML dump of a topology.

```
<system version="1">
  <cpu numaid="-1" arch="x86_64" vendor="GenuineIntel" familyid="6
    " modelid="60">
    <pci busid="0000:04:00.0" class="0x030000" link_speed="5 GT/s"
     link_width="4">
      <gpu dev="0" sm="50" rank="0" gdr="0"/>
    </pci>
    <pci busid="0000:05:00.0" class="0x030000" link_speed="5 GT/s"
     link_width="4">
      <gpu dev="1" sm="50" rank="1" gdr="0"/>
    </pci>
    <pci busid="0000:09:00.0" class="0x020000" link_speed="2.5 GT/
    s" link_width="1">
      <nic>
        <net name="enp9s0" dev="0" speed="1000" port="0" guid="0x0
    " maxconn="65536" gdr="0"/>
      </nic>
    </pci>
  </cpu>
</system>
```

Listing 3.3: NCCL topology XML dump

While each GPU is its own rank in the whole NCCL instance, each machine selects intra-machine ranks. The intra-node rank is responsible for selecting how data is moved internally in the machine.

During the topology phase, NCCL will also determine how it can best utilize the detected hardware. This is done by testing and tuning the graph.

3.3.3 Communication

When the topology is analyzed and communication patterns have been built, the nodes will set up the actual connections between ranks.

Each rank will set up a send to manage communication and a receive proxy abstraction for each connection. Send/Receive proxies are responsible for dispatching transfers. A connection supports multiple overlapping transfers, so the proxies implement a slot system to manage multiple messages. Depending on the available bandwidth and topology, NCCL may also create multiple connections between each pair of ranks. The proxy is also responsible for scheduling on what connection a message is transmitted on.

In order for ranks to keep track of data transfers, NCCL defines some simple protocols. The system requires efficient hardware usage, but it also needs low transfer latency. To achieve this, NCCL uses protocols that are

optimized for different message sizes. As of NCCL v2.8.4-1, there are three supported protocols: Simple, LL (Low Latency) and LL128.

The Simple protocol is optimized for large message sizes. It is a trivial FIFO queue where the sender sends messages in order, and the receiver receives messages in order. The implementation still allows for out-of-order completion of transfers, but the point is that each message is ordered in the FIFO queue. Each message is treated as an atomic object, so a full message must have been transferred before it is completed. This means that the receiving end must create a memory barrier and wait until the underlying transfer layer signals a complete message transfer. The receive proxy is executed at the CPU, so for a GPU to clear the memory barrier, it has to be notified by the CPU when using the Simple protocol.

LL is optimized for small messages. It works by encoding the data such that it is divided into 8-byte atomic chunks. A chunk consists of 4-byte data, and a 4-byte flag. The encoding is done to remove the memory barrier from the Simple protocol. It relies on the GPU being able to read 8 bytes atomically, so when the flag section is set, it knows the data received for that chunk is complete. Instead of waiting for the complete message to arrive, the GPU can wait for individual chunks of a message. This also means that a GPU can receive a message directly without waiting for a completion signal from the CPU. In the Simple protocol, direct DMA is possible because the memory does not change. However, LL requires encoding and decoding of messages. This adds extra cost when sending messages using this protocol. In addition, the message size becomes twice as big because half of each chunk is used for the flag. However, it also has some speed advantages because it divides the data into discrete atomic chunks. These chunks can be pipelined on the GPU, starting processing before all chunks have arrived. This protocol is only beneficial if the encoding/decoding process cost is low and potential speed gain is sufficient.

LL128 is also made for small messages like LL. It is similar to LL, except the encoding is different. 128-byte chunks are used instead, where the first 120 bytes are data, and the last 8 bytes are the flag. However, LL128 also requires data to be received in order. This is because the GPU can not read 128 bytes atomically, so there needs to be a guarantee that the payload has arrived before the flag is set. LL128 has potentially better bandwidth usage than LL because the flag encoding only uses 8 of 128 bytes of each chunk. However, it can not always be used given the order constraint.

The exact threshold for deciding what protocol to use, given the message size, is calculated/tuned during the topology phase and can differ for communicators. According to the pre-computed thresholds, the Send/Receive proxies are also responsible for dispatching a transfer with a given protocol.

3.4 Collective Calls

Collective calls are the core functionality of NCCL. The code implementing the collective calls is highly optimized for execution speed and efficient

hardware utilization. A collective operation is executed both on the CPU and on the GPU, like in a normal CUDA program. Special parts of the device code are implemented in Nvidia Parallel Thread Execution [22] (PTX). The device code is generic to support parameters that are set in the bootstrap phase, in addition to environment variables. C++/CUDA meta template programming is used to make the code generic, but it also makes it hard to read. The host code uses send/receive proxies that are set up during the bootstrap phase to send data. However, the device code needs to be able to encode/decode communication protocols used.

The collective calls NCCL offers are:

- Broadcast
- All Gather
- Reduce
- All Reduce
- Reduce Scatter

3.4.1 Broadcast

Broadcast distributes data from one rank (called the root rank) to all other ranks. All ranks will end up with the array send from the root rank.

3.4.2 Reduce

Reduce gathers data from all ranks to a single rank, and perform a specified reduction operation on the data. All ranks must send data of equal size. In addition, the data must be aligned in such a way that the reduce-operation can be performed. The data type specifies the alignment. NCCL supports four operations: Sum, Product, Max, Min. The root rank will end up in an array composed of reduced data from all ranks. Reduction is performed across ranks, so in the output array, the element at index i will be the reduction of index i at all ranks.

3.4.3 All Gather

All Gather gathers different data from all ranks to all ranks. Ranks must send data of equal size. All ranks will end up with equal arrays composed of data from each rank. The data is sorted by rank index and separated by send size.

3.4.4 All Reduce

All Reduce does the same as Reduce, but for all ranks. Ranks must send data of equal size, and the data must be aligned. All ranks will end up with equal output arrays with reduced elements. The reduction is performed the same way as in Reduce.

3.4.5 Reduce Scatter

Reduce Scatter does the same as All Reduce. However, the output array is scattered for each rank. Each rank receives an equally sized part of the complete reduced output array.

3.5 Async Interface

By default, NCCL supports non-blocking asynchronous library calls. This enables a user to dispatch more than one collective operation, much like a CUDA stream. *ncclGroupStart* and *ncclGroupEnd* are used to control the dispatch. *ncclGroupStart* allocates a queue for NCCL calls, then every NCCL call from the calling thread will be stored in the queue. Then to dispatch the operations, *ncclGroupEnd* is called. NCCL will dispatch calls to their designated CUDA stream like in a normal asynchronous CUDA program.

3.6 Chapter Summary

In this chapter, we explored how some features of NCCL are implemented.

The bootstrap phase initiates all nodes and then all ranks in a collective. It makes sure all ranks know about all other ranks, so any rank is able to communicate with any other rank.

When all ranks are initialized, the topology phase starts. Here each node detects and maps out what hardware is located on the machine. Hardware includes CPUs, GPUs, NICs and interconnect types. Each node then creates an intra-machine graph, connects hardware with PCIe or NVLink interconnect, and evaluates the graph. When the intra-node topology is decided, the system will decide what pattern to use for the whole system. The two main patterns are a tree or a ring. While the topology is evaluated, NCCL is also tuning it by performing tests. This allows each rank to pre-compute thresholds for message sizes.

When the topology is evaluated and tuned, the collective can be used. A user can dispatch many collective operations using the same topology.

In the next chapter, we discuss our plugin design and show the implementation of our PCIe plugin.

Chapter 4

SISCI-NCCL Implementation

In this chapter, we discuss and explain how we implemented support for Dolphin PCIe support in NCCL. First, an acknowledgment to the original author of SISCI-NCCL, Eivind Alexander Bergem at Dolphin Interconnect Solutions. His work laid the groundwork for this project, designing and implementing the first version of SISCI-NCCL. Eivinds code implemented a fully functional SISCI plugin for NCCL version v2.4.*, and it served as a great starting point for this project. The plugin implemented in this project makes changes to the original version to be compatible with NCCL version v2.8.4-1. However, it uses the same abstractions, and the general program flow is the same. The main reason for using a more recent version of NCCL was to access some newer features of Tensorflow. This is discussed more in chapter 6.

As of NCCL version v2.8.4-1, two plugin interfaces for network interconnects are provided. The plugins are dynamically loaded on application startup, and therefore do not require NCCL to be recompiled in order to use a plugin.

We call the first interface NCCL-NET (Network), it is a simple interface connecting two nodes with a one-directional connection. It provides a mechanism to register/deregister memory for RDMA transfer, and it has optional support for CUDA pointers. This makes it simple to implement new interconnects because it is only necessary to implement a sender and receiver abstraction. However, the one-directional connection abstraction does not allow for more complex network features such as multicast.

We call the second interface NCCL-COLL-NET (Collective Network). This is not a complete replacement for the NET plugin but an optimization for collective calls. Instead of NCCL composing communicators with many one-directional connections, the plugin can create its own multi-connection communicator. It is more complex and assumes the plugin is able to perform data reduction en route. The Dolphin hardware does not support such reduction capability, so it is not possible to use the NCCL-COLL-NET interface directly. A way to make this interface work would be to call the reduction kernels from the plugin itself. However, this would require the plugin to dispatch CUDA kernels which would add major complexity to the code. NCCL plugins are dynamically loaded, so the reduction kernels

located in the NCCL source code would have to be linked separately to the plugin.

4.1 Other NCCL plugins

As mentioned, NCCL features a plugin system for implementing custom transports. NCCL calls this plugin interface NET and is also used to implement NCCLs internal interconnect support. By default, NCCL supports TCP and InfiniBand via Linux. These implementations are located in the NCCL source code and provide a reference for how to implement the plugin. In addition, there are open source third-party plugin implementations.

4.1.1 AWS OFI NCCL

AWS OFI NCCL is an NCCL plugin developed by Amazon Web Services. It allows NCCL applications to use libfabric [23] as a proxy instead of implementing the interconnect directly. Libfabric is a project that tries to create a generic middle layer for fabric communication. The goal is that applications only need to know about Libfabric, and hardware APIs only need to interface with libfabric to be used by an application. This plugin is implemented with the NCCL-NET interface.

4.1.2 Mellanox Plugins

Mellanox also supports NCCL plugins for their RDMA and Switch technology. Mellanox Scalable Hierarchical Aggregation and Reduction Protocol (SHARP) [16] is a protocol that allows for hardware located in a network, such as switches, to perform collective operations. The SHARP plugin is implemented using the more complex NCCL-COLL-NET interface. Mellanox also provides a plugin for their InfiniBand hardware, and this is implemented using NCCL-NET.

4.2 Creating the Plugin

As mentioned earlier, for this project, we use the NCCL-NET plugin to provide support for Dolphin PCIe hardware in NCCL.

The code listed in this section is from the NCCL repository version v2.8.4-1 and from the SISCI-NCCL repository.

4.2.1 Error Management

When creating a plugin, it is good practice to report or handle all errors when working with an API. Because the plugin is called from inside NCCL instead of by a library user, NCCL needs to reason about potential errors. Both NCCL and SISCI provide their own error managing patterns.

NCCL uses a custom return (*ncclResult_t*) type to signal if a function was successful or not. All functions in the plugin interface return this type. To make the code more ergonomic to read and write, we use a macro to unwrap errors. Functions called inside *NCCLCHECK* is automatically unwrapped as long as the function returns a *ncclResult_t* type.

In the SISCI-NCCL code, all SISCI functions are prefixed with *nccl*. To simplify the error unwrapping code when using SISCI, a macro is created to wrap around each SISCI API function. The macro automates the error unwrapping and converts the error into an NCCL error. By doing this, it is possible to call SISCI functions with the *NCCLCHECK* macro.

By converting errors, all errors should be reported to NCCL, causing it to abort if we encounter a problem.

4.2.2 Plugin Interface

To implement a plugin, NCCL provides a declaration of a set of functions that the plugin needs to implement. Listing 4.1 defines a struct with function pointers. The struct is used to locate where in the plugin each function is located. This is necessary to load the plugin dynamically.

```
typedef struct {
  // Name of the network (mainly for logs)
  const char* name;
  // Initialize the network.
  ncclResult_t (*init)(ncclDebugLogger_t logFunction);
  // Return the number of adapters.
  ncclResult_t (*devices)(int* ndev);
  // Get various device properties.
  ncclResult_t (*getProperties)(int dev, ncclNetProperties_v4_t* props);
  // Create a receiving object and provide a handle to connect to it. The
  // handle can be up to NCCL_NET_HANDLE_MAXSIZE bytes and will be exchanged
  // between ranks to create a connection.
  ncclResult_t (*listen)(int dev, void* handle, void** listenComm);
  // Connect to a handle and return a sending comm object for that peer.
  ncclResult_t (*connect)(int dev, void* handle, void** sendComm);
  // Finalize connection establishment after remote peer has called connectHandle
  ncclResult_t (*accept)(void* listenComm, void** recvComm);
  // Register/Deregister memory. Comm can be either a sendComm or a recvComm.
  // Type is either NCCL_PTR_HOST or NCCL_PTR_CUDA.
  ncclResult_t (*regMr)(void* comm, void* data, int size, int type, void** mhandle);
  ncclResult_t (*deregMr)(void* comm, void* mhandle);
  // Asynchronous send to a peer.
  // May return request == NULL if the call cannot be performed (or would block)
  ncclResult_t (*isend)(void* sendComm, void* data, int size, void* mhandle, void** request);
  // Asynchronous recv from a peer.
```

```
26   // May return request == NULL if the call cannot be performed (
       or would block)
27   ncclResult_t (*irecv)(void* recvComm, void* data, int size, void
       * mhandle, void** request);
28   // Perform a flush/fence to make sure all data received with
       NCCL_PTR_CUDA is
29   // visible to the GPU
30   ncclResult_t (*iflush)(void* recvComm, void* data, int size,
       void* mhandle, void** request);
31   // Test whether a request is complete. If size is not NULL, it
       returns the
32   // number of bytes sent/received.
33   ncclResult_t (*test)(void* request, int* done, int* size);
34   // Close and free send/recv comm objects
35   ncclResult_t (*closeSend)(void* sendComm);
36   ncclResult_t (*closeRecv)(void* recvComm);
37   ncclResult_t (*closeListen)(void* listenComm);
38 } ncclNet_v4_t;
39 typedef ncclNet_v4_t ncclNet_t;
40 #define NCCL_PLUGIN_SYMBOL ncclNetPlugin_v4
```

Listing 4.1: NET plugin interface from `nccl:src/include/nccl_net.h`

When the plugin is started, NCCL calls the *init* function. As the name suggests, it is meant to initialize the plugin and network structures. For SISCI, this allows us to initialize the SISCI library with *ncclSCIInitialize*, which is necessary to use the SISCI API. Because the plugin is loaded as a Linux shared object, it is important to keep track of where and how memory is allocated. Memory can be owned by the shared object or by the application using the shared object. If a global variable is declared with *extern*, it can cause a situation where the shared object memory can be accessed by all applications using the shared library. Therefore it is important to make sure the memory access is handled correctly when implementing a shared object. If not, the code might produce non-obvious race conditions because two or more separate processes are writing to shared object memory at the same time. Even though this project does not need to have more than one NCCL application running on each machine, it is still useful to be strict when accessing shared memory.

```
1  // Initialize the network.
2  ncclResult_t ncclSisciInit(ncclDebugLogger_t logFunction) {
3      ncclDebugLog = logFunction;
4      pthread_mutex_lock(&ncclSisciLock);
5      if (ncclSisciNDevs == -1) {
6          INFO(NCCL_NET|NCCL_INIT, "Trying to load SISCI");
7          NCCLCHECK(ncclSCIInitialize(NO_FLAGS));
8          ncclSisciNDevs = 0;
9          for (int i = 0; i < MAX_SCI_DEVS; i++) {
10             struct ncclSisciDev *dev = &ncclSisciDevs[i];
11             dev->adapter_no = i;
12             if (ncclSCIGetLocalNodeId(dev->adapter_no, &dev->
   node_id, NO_FLAGS) ==
13                 ncclSuccess) {
14                 INFO(NCCL_INIT|NCCL_NET, "NET/SISCI : adapter %u,
   node id %u",
15                      dev->adapter_no, dev->node_id);
16                 dev->node_offset = (dev->node_id >> 2) - 1;
```

```c
                ncclSisciNDevs++;
            }
            else {
                break;
            }
        }
    }
    pthread_mutex_unlock(&ncclSisciLock);
    if (ncclSisciNDevs == 0) {
        INFO(NCCL_INIT|NCCL_NET, "NET/SISCI : No devices found.");
    }
    return ncclSuccess;
}
```

Listing 4.2: SISCI-NCCL init function from `sisci-nccl:src/sisci_nccl.c`

Listing 4.2 shows the implemented init function for SISCI-NCCL. The *logFunction* parameter is a function pointer to NCCLs log function. It allows the plugin to forward log messages to the NCCL log system. Log messages are reported with the *WARN* and *INFO* macros defined in `sisci-nccl:src/sisci_nccl.h`. The log system provides an interface for filtering messages for testing and development. To ensure the network structures are only initialized once, the initialization is wrapped in a mutex.

The NET abstraction is modeled after a system using NICs (Network Interface Cards), and it calls each NIC a device. As mentioned in section 3.3.2, each NIC is treated as a node in the topology. This enables NCCL to select the best NIC or even send via more than one NIC to achieve higher bandwidth. The Dolphin hardware is similar to a NIC, so it can operate under the same abstraction. It is a PCIe device, just like most NICs, and there are no fundamental differences between the two domains.

To initialize the network structures, the init function tries to detect all SISCI capable devices available. When the Dolphin driver is installed and configured properly, it knows about connected devices. The SISCI API provides functions to detect and query Dolphin devices. Dolphin devices are numbered by an adapter number. We assume that devices always are numbered from 0 upwards and that there is no gap between adapters. Listing 4.3 shows the *ncclSisciDev* struct. *adapter_no* member is necessary to specify what adapter we use when calling other SISCI functions. *node_id* will be distributed later to other nodes to create connections. *node_offset* is used to assign unique memory segment ids (this is not used).

```c
struct ncclSisciDev {
    unsigned int adapter_no;
    unsigned int node_id;
    unsigned int node_offset;
};
```

Listing 4.3: SISCI-NCCL device structure from `sisci-nccl:src/sisci_nccl.c`

The plugin interface assumes each device is enumerated from 0 to N. NCCL can indirectly reference a plugin device without knowing about the structure itself using the index. The *ncclSisciDevs* array is used to map device indexes to device structs, as it is trivial with array indexing. Listing

4.4 implements the *devices* call. It returns the number of devices the plugin found after initializing, so NCCL knows the range it can query devices.

```
// Return the number of adapters.
ncclResult_t ncclSisciDevices(int* ndev) {
    *ndev = ncclSisciNDevs;
    return ncclSuccess;
}
```

Listing 4.4: NET plugin ncclSisciDevices from `nccl:src/include/nccl_net.h`

4.2.3 Device Properties

In order for NCCL to reason about plugin devices without knowing about the actual implementation, it has a standardized properties field. Listing 4.5 shows the *ncclNetProperties_t* struct. The *getProperties* function allows NCCL to query each device for properties, so the plugin needs to produce the necessary data.

name is only used for logging and debugging, so the name does not matter for the functionality of the plugin, so we only assign a unique name based on the device/adapter index.

```
typedef struct {
  char* name;      // Used mostly for logging.
  char* pciPath;   // Path to the PCI device in /sys.
  uint64_t guid;   // Unique identifier for the NIC chip. Important
     for
                   // cards with multiple PCI functions (Physical
    or virtual).
  int ptrSupport;  // NCCL_PTR_HOST or NCCL_PTR_HOST|NCCL_PTR_CUDA
  int speed;       // Port speed in Mbps.
  int port;        // Port number.
  int maxComms;    // Maximum number of comms we can create
}ncclNetProperties_v4_t;
typedef ncclNetProperties_v4_t ncclNetProperties_t;
```

Listing 4.5: NET plugin properties struct from `nccl:src/include/nccl_net.h`

pciPath is a regular path string to a location in the file system (sysfs). Sysfs is standardized in the Linux kernel, giving NCCL the ability to locate all types of PCIe devices on a Linux system. The path describes where the device is located in the system's PCIe topology. NCCL uses this in the topology phase to model and evaluates the hardware topology, so it is important that this path is correct. To get this path, the SISCI API provides functionality to query the location of a device with a Bus:Device.Function (BDF) format. The SCIQuery function fills a 16-bit field with the BDF of the requested device. In the sysfs hierarchy, PCIe devices are projected under `/sys/class/pcie_bus` according to their BDF. So it is trivial to construct the complete path to the device when the values are extracted from the bit field. Listing 4.6 shows how the path is created.

```
// Return the device path in /sys. NCCL will call free on this
    path.
```

```
2  ncclResult_t ncclSisciPciPath(int dev, char** path) {
3      struct ncclSisciDev *devp = &ncclSisciDevs[dev];
4      char devicepath[PATH_MAX];
5      sci_query_adapter_t query;
6      uint16_t bdf;
7      uint8_t bus;
8      uint8_t device;
9
10     query.subcommand = SCI_Q_ADAPTER_BDF;
11     query.localAdapterNo = devp->adapter_no;
12     query.data = &bdf;
13     NCCLCHECK(ncclSCIQuery(SCI_Q_ADAPTER, &query,
14                            NO_FLAGS));
15     bus = bdf >> 8;
16     device = bdf & 0x00ff;
17
18     snprintf(devicepath, PATH_MAX,
19              "/sys/class/pci_bus/0000:%02x/device/0000:%02x:%02x
       .0/",
20              bus, bus, device);
21     *path = realpath(devicepath, NULL);
22
23     return ncclSuccess;
24 }
```

Listing 4.6: SISCI PCIe path function from `sisci-nccl:src/sisci_nccl.c`

The *guid* field is not used by NCCL directly, so it does not affect the functionality of the plugin.

ptrSupport describes the supported transport features of the device. As discussed earlier, NCCL supports RDMA, where memory can be directly read/written to a remote host. This is usually referred to in the context of RAM, but NCCL also supports *GPUDirect RDMA* (GDRDMA). GDRDMA allows a remote host to access device memory directly, bypassing intermediate write and read steps. By default, NCCL is configured to always store data in RAM before sending it to another node. However, with GDRDMA, it is possible to remove the intermediate step, sending data located on a GPU directly to a GPU or RAM on a remote host. In order to enable GDRDMA, we need an interconnect that is able to perform RDMA directly on GPUs. The Dolphin hardware and SISCI API does have this capability, so we can set *ptrSupport* to support both host and CUDA pointers. Section 4.2.8 discusses how this is implemented.

The *speed* parameter should be set to the bandwidth capability of the device. NCCL uses this parameter in the topology and tuning steps to compute the most efficient paths and topology patterns. In addition, it is used to guide how much resources should be allocated, such as the number of communicators and buffers sizes. The interface expects a speed for the device. However, the Dolphin hardware allows for multiple links for each device. NCCL does not model the inter-host network topology, so the plugin device is treated as a node with a speed attribute in the topology. To get a reliable speed, we query all links of the device then select the lowest speed.

port is only for logging as the interface was originally intended for NICs.

It is not relevant for SISCI and does not affect the behavior of NCCL.

maxComms is the maximum number of communicators NCCL is allowed to create for each plugin device. Communicators require resources on the device, so to provide reliable operation, this should be set so the device will not run out of resources. Each communicator only connects two ranks in one direction. NCCL will also create more than one communicators between the same ranks to enable overlapping transfers if the bandwidth allows it. The distribution strategy/pattern selected dictates how many communicators NCCL will use. The number of GPUs on each host and interconnect bandwidth determine the pattern. For this version, the *maxComms* parameter is hardcoded to 256.

Listing 4.7 shows an NCCL topology dump of a system with our plugin installed. NCCL is able to detect and reason about the Dolphin adapter.

```
<system version="1">
  <cpu numaid="-1" arch="x86_64" vendor="GenuineIntel" familyid="6" modelid="60">
    <pci busid="0000:04:00.0" class="0x030000" link_speed="5 GT/s" link_width="4">
      <gpu dev="0" sm="50" rank="0" gdr="1"/>
    </pci>
    <pci busid="0000:05:00.0" class="0x030000" link_speed="5 GT/s" link_width="4">
      <gpu dev="1" sm="50" rank="1" gdr="1"/>
    </pci>
    <pci busid="0000:01:00.0" class="0x060400" link_speed="8 GT/s" link_width="8">
      <pci busid="0000:03:00.0" class="0x068000" link_speed="8 GT/s" link_width="8">
        <nic>
          <net name="sisci_adapter_0_node_4" dev="0" speed="63040" port="0" guid="0x0" maxconn="256" gdr="1"/>
        </nic>
      </pci>
    </pci>
  </cpu>
</system>
```

Listing 4.7: NCCL topology XML dump detecting a Dolphin NTB adapter

4.2.4 Allocating Segments

In order for ranks to send data to remote ranks, they need to know where to send it. SISCI abstracts memory to segments. Segments have their own machine unique id and memory to be referenced with the id instead of a system global address.

Because it can be more than one connection located on a machine, we need to make sure all connections have unique segment ids. Listing 4.8 shows how the segment id allocator is implemented. It is just a simple ticker at its core. However, it does allocate a range of ids. This is because some connections may need more than one segment depending on the configuration (RDMA vs. GDRDMA). It is possible that this

implementation can run out of segment ids. However, it is sufficient for the scope of this thesis.

```c
// make room for 256 memory segments for each comm
static unsigned int alloc_segment_id() {
    pthread_mutex_lock(&ncclSisciLock);
    unsigned int id = (SEGMENT_PREFIX | (ncclSisciSegmentIdCount
    << 8));
    ncclSisciSegmentIdCount++;
    pthread_mutex_unlock(&ncclSisciLock);
    return id;
}
```

Listing 4.8: Segment allocator function from `sisci-nccl:src/sisci_nccl.c`

4.2.5 Channels

The plugin interface is designed as an asynchronous interface. This allows for overlapping send operations on a single connection. To support this, we created a channel abstraction for the connection, where each connection has multiple channels. Each send call is scheduled on a channel, allowing for multiple transfers at the same time.

Data transfers are implemented using the DMA engine provided by the SISCI API. To implement overlapping transfers in SISCI, we created a DMA queue for each channel. This allowed us to start DMA transfers independently. To schedule messages, round-robin is used, assigning messages to channels in order.

4.2.6 Mailbox

In order to synchronize communication between the send and receive peers, it is necessary to implement a protocol. We created a mailbox abstraction to maintain this protocol. A send peer needs to know when the corresponding receive peer is ready to receive, and the receive peer need to know when the corresponding send peer is done sending.

The protocol is as follows:

1. Receive peer is ready to receive, it sends RECV_REQUESTED command to send peer, and waits.

2. Send peer receives RECV_REQUESTED from receiver, it starts a DMA transfer to receiver segment.

3. Send peer completes DMA transfer, then sends SEND_NOTIFIED command to receive peer, send operation is complete.

4. Receive peer receives SEND_NOTIFIED command, receive operation is complete.

The mailbox is a bidirectional communication channel. Both peers have a local segment and connect to the remote peers' segment. Local segments

are initialized to NULL. To send, a peer writes to the remote segment. To receive, a peer reads its local segment. If the local segment is not NULL, a message has arrived. Importantly the receiving peer must read the message, then reset the local segment. Protocol messages are not big, so it makes sense to use PIO to send messages.

Listing 4.9 shows the implementation of the read and write functions. The functions take a channel parameter instead of a mailbox. This is so we can use the same mailbox for multiple channels, as one connection may have more than one channel. Each channel has its own space on the segments, so they don't overwrite each other. The mailbox also supports attaching a 64-bit payload to each message. This is used by the receive end to notify data offsets, and the send peer to notify how much data was sendt in a message.

```
static void mailbox_write(struct ncclSisciChannel *channel,
    uint64_t command, uint64_t value) {
    uint64_t *remote_command = (uint64_t*)channel->mailbox->
    remote_addr + channel->id*MAILBOX_SEGMENT_SIZE;
    uint64_t *remote_value = remote_command + 1;

    *remote_value = value;
    *remote_command = command;
}

static int mailbox_read(struct ncclSisciChannel *channel, uint64_t
     command, uint64_t *value) {
    uint64_t *local_command = (uint64_t*)channel->mailbox->
    local_addr + channel->id*MAILBOX_SEGMENT_SIZE;
    uint64_t *local_value = local_command + 1;

    if (*local_command == command) {
        if (value != NULL) {
            *value = *local_value;
        }
        *local_command = NO_CMD;
        return 1;
    }
    return 0;
}
```

Listing 4.9: Mailbox read/write from `sisci-nccl:src/sisci_nccl.c`

4.2.7 Establishing Connections

When the plugin is initialized, and NCCL has evaluated the device parameters, it can start creating connections. The plugin interface provides a simple handshake protocol to implement this, *listen*, *connect* and *accept* functions are intended for this.

A simplified workflow is as follows: We want to establish a connection from rank 1 to 2. Rank 2 will call *listen*. Rank 1 has to know the address of rank 2 and call *connect* with rank 2s address as a parameter. Rank 2 will call *accept* and the function will block until the connect message from rank 1 arrives. After the accept function is done, both ranks should have enough

information about each other to maintain a connection. Rank 1 will be a send communicator, and rank 2 will be a receiver communicator.

Listing 4.10 shows how the connect call is implemented. The call creates a comm object and stores it in the *listenComm* parameter. To establish a connection between the two ranks, we opted to use data interrupts provided by the SISCI API. Data interrupts allow us to listen for triggers on a specific id. The listen call will create a data interrupt, then send the interrupt id to the connect rank. To send to interrupt id, NCCL provides the *opaqueHandle* parameter. This is a pointer to a data block that can be used by the plugin to exchange information. We store the interrupt id together with some other parameters from the listen rank. When the *ncclSisciListen* function returns, NCCL will send the data stored at the opaqueHandle to the connect rank.

```c
// Create a receiving object and provide a handle to connect to it. The
// handle can be up to NCCL_NET_HANDLE_MAXSIZE bytes and will be exchanged
// between ranks to create a connection.
ncclResult_t ncclSisciListen(int dev, void* opaqueHandle, void** listenComm) {
    struct ncclSisciListenComm *comm;

    NCCLCHECK(ncclCalloc(&comm, 1));
    comm->dev = &ncclSisciDevs[dev];

    comm->local_mailbox_segment_id = alloc_segment_id();
    comm->local_memory_segment_id = alloc_segment_id();

    NCCLCHECK(ncclSCIOpen(&comm->sd, NO_FLAGS));
    *listenComm = comm;
    NCCLCHECK(cuda_get_device(&comm->gpu));

    struct ncclSisciHandle* handle = (struct ncclSisciHandle*) opaqueHandle;
    static_assert(sizeof(struct ncclSisciHandle) < NCCL_NET_HANDLE_MAXSIZE,
                  "ncclSisciHandle size too large");
    handle->node_id = comm->dev->node_id;
    handle->gpu = comm->gpu;
    handle->mailbox_segment_id = comm->local_mailbox_segment_id;
    handle->memory_segment_id = comm->local_memory_segment_id;

    NCCLCHECK(ncclSCICreateDataInterrupt(comm->sd, &comm->ir, comm->dev->adapter_no, &handle->irno,
                                         NO_CALLBACK, NO_ARG, NO_FLAGS))
    INFO(NCCL_INIT|NCCL_NET, "Listening on %u", handle->irno);

    return ncclSuccess;
}
```

Listing 4.10: Listen function from `sisci-nccl:src/sisci_nccl.c`

Listing 4.11 shows the connect implementation. The connect rank will then call connect when it receives data pointed to by *opaqueHandle*. A *sendComm* object is then created by the call and given back to NCCL. Unlike

the listen command, NCCL does not provide a bootstrap mechanism in the reverse direction (connect to accept). This is why we initialized the interrupt on the listen side, opening a communication channel. Local peer data is added to a data block, and then the data interrupt is triggered together with the data block. After the data interrupt is triggered, the send peer has received enough information to initialize. The interrupt is disconnected and freed as we do not use the interrupt for operation. Then the mailbox and channel abstractions are initialized, and the *sendComm* parameter is pointed to the communicator object.

```
// Connect to a handle and return a sending comm object for that peer.
ncclResult_t ncclSisciConnect(int dev, void* opaqueHandle, void** sendComm) {
    struct ncclSisciSendComm *comm;
    struct ncclSisciHandle* handle = (struct ncclSisciHandle*)opaqueHandle;

    NCCLCHECK(ncclCalloc(&comm, 1));
    comm->dev = &ncclSisciDevs[dev];
    comm->remote_node_id = handle->node_id;
    comm->type = SEND_COMM;
    NCCLCHECK(cuda_get_device(&comm->local_gpu));
    comm->remote_gpu = handle->gpu;
    comm->local_memory_segment_id = alloc_segment_id();
    comm->remote_memory_segment_id = handle->memory_segment_id;

    INFO(NCCL_INIT|NCCL_NET, "Connecting to node %d on %d", handle->node_id, handle->irno);

    unsigned int local_mailbox_segment_id = alloc_segment_id();

    sci_desc_t sd;
    sci_remote_data_interrupt_t ir;
    uint32_t data[IR_DATA_SIZE];
    data[0] = htonl(comm->dev->node_id);
    data[1] = htonl(comm->local_gpu);
    data[2] = htonl(local_mailbox_segment_id);
    data[3] = htonl(comm->local_memory_segment_id);


    NCCLCHECK(ncclSCIOpen(&sd, NO_FLAGS));

    while (ncclSCIConnectDataInterrupt(sd, &ir, handle->node_id, comm->dev->adapter_no, handle->irno, 0, NO_FLAGS) != ncclSuccess) {
        sleep(1);
    }

    NCCLCHECK(ncclSCITriggerDataInterrupt(ir, &data, sizeof(*data)*IR_DATA_SIZE, NO_FLAGS));

    NCCLCHECK(ncclSCIDisconnectDataInterrupt(ir, NO_FLAGS));
    NCCLCHECK(ncclSCIClose(sd, NO_FLAGS));

    NCCLCHECK(ncclSisciCreateMailbox(comm, &comm->mailbox, local_mailbox_segment_id, handle->mailbox_segment_id));
```

```
40     NCCLCHECK(ncclSisciInitChannels(comm->channels, comm->mailbox,
41                                    comm->type));
42
43     *sendComm = comm;
44
45     return ncclSuccess;
46 }
```

Listing 4.11: Connect function from `sisci-nccl:src/sisci_nccl.c`

Listing 4.12 shows the accept function. The accept function is called by the receive peer of a connection. Parameters are the *listenComm* that was initilized in the listen function, and a *recvComm* pointer that the function is responsible for initilizing. The function will block until the send peer is triggering the data interrupt. When the interrupt arrives, the receive peer knows that the send peer is initialized, so the handshake is complete. The *recvComm* structure and its mailbox and channel structures are initialized. Now the *listenComm* object is no longer needed. However, it got its own cleanup function that NCCL is calling seperatly to free its resources. Given all the handshake functions executed successfully, both the send and receive peer should be initialized.

```
1 // Finalize connection establishment after remote peer has called
    connectHandel
2 ncclResult_t ncclSisciAccept(void* listenComm, void** recvComm) {
3     struct ncclSisciListenComm *lcomm = (struct
    ncclSisciListenComm*)listenComm;
4
5     struct ncclSisciRecvComm *rcomm;
6
7     uint32_t data[IR_DATA_SIZE];
8     unsigned int size = IR_DATA_SIZE*sizeof(*data);
9
10    NCCLCHECK(ncclCalloc(&rcomm, 1));
11    rcomm->dev = lcomm->dev;
12    rcomm->type = RECV_COMM;
13    rcomm->local_gpu = lcomm->gpu;
14
15    NCCLCHECK(ncclSCIWaitForDataInterrupt(lcomm->ir, &data, &size,
     SCI_INFINITE_TIMEOUT,
16                                            NO_FLAGS));
17
18    rcomm->remote_node_id = ntohl(data[0]);
19    rcomm->remote_gpu = ntohl(data[1]);
20    const unsigned int remote_mailbox_segment_id = ntohl(data[2]);
21    rcomm->local_memory_segment_id = lcomm->
    local_memory_segment_id;
22    rcomm->remote_memory_segment_id = ntohl(data[3]);
23
24    NCCLCHECK(ncclSisciCreateMailbox(rcomm, &rcomm->mailbox, lcomm
    ->local_mailbox_segment_id, remote_mailbox_segment_id));
25    NCCLCHECK(ncclSisciInitChannels(rcomm->channels, rcomm->
    mailbox,
26                                   rcomm->type));
27
28    INFO(NCCL_NET, "Accepted connection from node %d", rcomm->
    remote_node_id);
29
```

```
30     *recvComm = rcomm;
31 
32     return ncclSuccess;
33 }
```

Listing 4.12: Accept function from `sisci-nccl:src/sisci_nccl.c`

4.2.8 Register Memory

Now that the send and receive peer has performed a handshake and is initialized, one step is necessary before transmission can start. Most DMA APIs require memory to be registered before it is possible to perform DMA transfers. NCCL is aware of this and provides an API call to register memory (*regMr*). For our plugin, this allows us to register memory as SISCI segments.

NCCL will always register memory in a symmetrical fashion, where any memory registered on the send peer, must have a corresponding area on the receive peer. When memory is registered, NCCL expects the plugin to create a memory handle. The memory handle acts as a connection between the two corresponding memory areas that are registered. NCCL assumes it is possible to register multiple memory areas for each communicator object. This is why the segment allocator function allocates a range of segment ids for each communicator (section 4.2.4).

As a user, NCCL only accepts CUDA pointers of memory that are located on a device, but in the context of our plugin, memory can be located in the host or device memory. Listing 4.13 shows the function signature of the register function. The *comm* parameter is a pointer to the communcator object that was either created in the *connect* or *accept* functions. Parameters *data*, *size* and *type* defines the memory area and if it is located on host or device.

SISCI makes it relatively easy to register memory, but it has some constraints that must be considered. SISCI requires memory to be page-aligned in order to register it. The data pointer passed by the *data* parameter is not always page-aligned, so it is necessary to adjust the pointer, so we always pass a page-aligned pointer to SISCI. To register host memory in SISCI, we first need to create an empty SISCI segment, then attach memory to it using *SCIRegisterSegmentMemory*. CUDA pointers (device memory) are registered in a similar way as host memory, except setting some CUDA attributes and calling *SCIAttachPhysicalMemory*. While this is usually a simple process, we had some problems with the IOMMU on our machines, resulting in us having to implement a workaround (more in section 4.3).

```
1 ncclResult_t ncclSisciRegMr(void* comm, void* data, int size, int
      type, void** mhandle);
```

Listing 4.13: Register memory function signature from `sisci-nccl:src/sisci_nccl.c`

4.2.9 Transfer Data

When memory has been registered, data can be transmitted from sender peers to corresponding receiver peers. Listing 4.14 shows the function signatures of the transfer functions. NCCL will call these to perform a transfer. The interface is designed in an asynchronous way, so NCCL expects none of the functions to block but rather to signal if a call can be performed or not. NCCL will poll these functions periodically to perform transfers. *ncclSisciIsend* and *ncclSisciIrecv* is called from peers respectively. If a function is able to be performed, it will return a request. Then NCCL will poll *ncclSisciTest* on the request. The function will signal if a request is completed. Meaning the send peer will know if a send operation is complete, and the receive peer will know if a full message has arrived.

Inter-node transfers are performed as a FIFO queue. On the send side, NCCL will divide send data into chunks, then call *ncclSisciIsend* on each chunk. The receive side does not know the size of each chunk, but the total size of a transmission sent over a connection, so our plugin must forward the size of each message. NCCL will call *ncclSisciIrecv* until the sum of bytes received is equal to the expected transfer size. This is where the mailbox abstraction comes in (section 4.2.6).

The number of channels created for each connection (section 4.2.5) determines the number of concurrent transfers possible. Each chunk is designated to a channel using a round-robin scheduling scheme.

NCCL provide a flush function (*ncclSisciIflush*). This function is intended to block until a requested message has arrived. Flush is required for systems that do not have cache coherence, where caches are not automatically updated. For our implementation, we did not find an issue with not implementing this function, as our program was still able to perform correctly. However, due to our IOMMU issue, this may not be the case if GDRDMA was working (4.3), as GPU memory does not necessarily provide the same memory guarantees as x86 chips.

```
1  // Asynchronous send to a peer.
2  // May return request == NULL if the call cannot be performed (or
       would block)
3  ncclResult_t ncclSisciIsend(void* sendComm, void* data, int size,
       void* mhandle, void** request);
4
5  // Asynchronous recv from a peer.
6  // May return request == NULL if the call cannot be performed (or
       would block)
7  ncclResult_t ncclSisciIrecv(void* recvComm, void* data, int size,
       void* mhandle, void** request);
8
9  ncclResult_t ncclSisciIflush(void* recvComm, void* data, int size,
        void* mhandle, void** request);
10
11 ncclResult_t ncclSisciTest(void* request, int* done, int* size);
```

Listing 4.14: Transfer function signatures from `sisci-nccl:src/sisci_nccl.c`

4.2.10 Cleanup

All types of objects created by the plugin (*listenComm*, *sendComm*, *recvComm* and *memhandle*), has its own destructor function. NCCL will call these in such an order that all "sub" objects are freed before any super objects are freed. For example, all *memhandle* objects are deregistered before a comm object can be freed. In addition, the interrupt, mailbox, and channel structures are freed when their corresponding comm objects are freed.

4.3 IOMMU and GPUDirect Problems

While the original version of SISCI-NCCL created for version v2.4.* was tested and fully working on a certain system. When we started working on this thesis, we wanted to test NCCL using a newer version of Tensorflow. This required us to update our plugin to a newer version of NCCL. Now this transition was not as smooth as we would have liked. Focusing on upgrading the plugin, we did not do our due diligence verifying that the hardware was working correctly.

The SISCI API was not able to detect if the IOMMU was operational or not. To register memory, we were able to create SISCI segments using *SCIRegisterSegmentMemory* and *SCIAttachPhysicalMemory* without any errors reported. This made us assume transmission was possible because we were able to map remote segments into user memory without any errors as well. And our mailbox abstraction was working. We were able to perform PIO transfers over the segments allocated for the mailboxes.

When we were trying to perform DMA transfers between segments, no data was transferred. For segments registered with *SCIAttachPhysicalMemory* (device memory), this manifested in a SISCI error when trying to dispatch a DMA transfer. This problem was not apparent until very late in the implementation phase, as we were testing without GDRDMA enabled. For segments registered with *SCIRegisterSegmentMemory*, we were able to dispatch DMA transfers, and there were no reported errors from SISCI. However, there was no data transmitted, so the receive chunks were not equal to send chunks. As it turned out, the mailbox segments were working because they were allocated by the SISCI driver itself, which does not require a working IOMMU. The CUDA runtime allocates registered memory, so to register the memory in SISCI, we needed a working IOMMU. A large amount of time was spent trying to figure this out, and our development environment did not help to determine this was an issue.

As all ranks in a collective operation are aware of the input parameters used, no extra data is added to payloads by NCCL. By default, NCCL does not take the transferred payload into account when performing collective operations. This means that NCCL does not detect if a transfer is correct or not, and it does not matter for executing the operation. The collective operation algorithm will not halt in this configuration because our plugin will report that messages were transmitted (because SISCI did

not report errors when dispatching DMA transfers). This causes the output of collective operations to not be correct, something that we would be able to detect. However, it was not trivial to detect due to the complexity of NCCL.

One particular feature of NCCL made it very hard to detect that no data was actually transmitted between registered memory. The low latency protocols described in section 3.3.3 made it very hard to determine where the fault was originating from. Our naive thought was that the Low Latency issue would only manifest if we used small data sizes in a collective operation. However, NCCL has to initialize before it can perform operations, and during the topology phase (section 3.3.2), it will test and tune connections. To do this, NCCL will have to use our plugin, registering and transferring data. It will test many transfer sizes, and crucially all transfer protocols.

Using the Simple protocol, a receiver peer will not process a data chunk until our plugin reports that a chunk has arrived. If an error would occur during a transfer, and it was reported, NCCL will throw an error during the initializing phase. In our case, when the Simple protocol was tested, no error was thrown as our plugin was not able to detect it. The mailbox mechanism was working, so the receiver was told that a message had arrived, even though the data was incorrect. This is not correct, but it did not halt the execution. If this was the only way the IOMMU problem manifested, we would have been able to detect that our collective operation output was incorrect, but there was an additional way.

The LL protocols are implemented using busy waiting, where a CUDA kernel will be dispatched before a chunk has arrived. The kernel does not wait for our plugin to signal when a chunk has arrived. Instead, it will continually look at its memory, checking if flags have been set. NCCL assumes the chunk will arrive in the device at some point, so it does not implement a way to cancel the busy waiting mechanism when our plugin signals completion. The kernels would enter an infinite loop as a result of flags never arriving. This caused our test programs to halt, waiting for CUDA kernels to complete until NCCL would terminate, reporting a timeout.

It became very difficult to debug this problem. Our application is a distributed system. It is executing on two machines, each machine contains 2 GPUs each, and we are controlling Dolphin adapters as well. Using regular debugging tools like GDB and Valgrind was not sufficient, as we had to deal with CUDA kernels. Nvidia provides its own debugger tool called CUDA-GDB. However, it was hard to make it debug our plugin. NCCL is also able to report a stack trace if it encounters an error. However, because our plugin did not detect the issue itself, this was not of much use. The way this LL issue was manifesting, these tools did not give us a good picture of what was happening.

After a lot of time and struggle, we did determine that the IOMMU on both test machines was broken. The obvious solution was to replace the broken hardware, but due to external factors, it became very hard to source new hardware. We determined that we had to find a solution that worked

on this hardware. DMA and PIO transfers were still possible using the Dolphin adapters. It was just not working on segments that SISCI did not allocate.

4.4 Emergency Solution

The Dolphin adapters do support systems without an IOMMU, so it is still possible to perform transfers using SISCI. Because the mailbox abstraction was working, we investigated if it was still possible to perform DMA transfers via segments allocated by SISCI. This was working, so it seemed like our solution would be to not use SISCI calls that register memory not allocated by SISCI.

In our plugin, the register memory function either use *SCIRegisterSegmentMemory* or *SCIAttachPhysicalMemory* to attach memory to our segments. The SISCI documentation does state that these functions require an enabled IOMMU, so it makes sense they are not working. To replace these, we had to register memory the same way we registered memory in the mailbox abstraction by having SISCI do the allocation. One downside to this is that the plugin interface does not support a way for our plugin to return memory back to NCCL. The register memory function will always receive a buffer from NCCL.

The SISCI driver is responsible for allocating segment memory. Segments allocated without an IOMMU also has some constraints. They require the memory pages making up the area to be contiguous. As memory becomes more fractured, there will be less space to allocate large contiguous areas. This increases the possibility that allocation will fail, in effect reducing the maximum collective operation size.

Our first solution was to create intermediate buffers in both the send and receive peer. The intermediate buffers would have the same size as the requested registered memory. To send, the send peer must copy from its registered memory to its intermediate buffer. Then it can dispatch a DMA transfer to the receiver peer intermediate buffer. When the DMA transfer is done, the receiver peer must copy the message from its intermediate buffer to its registered memory area. This solution does work. However, it comes at the cost of increasing the memory consumption by two times. Due to the contiguous page restriction, the effective possible buffer size would be reduced by more than two times, so it would be beneficial to reduce memory consumption.

A more advanced second solution would be to only allocate enough memory for the number of concurrent chunks (number of channels) supported by our plugin. However, the plugin interface does not signal the maximum chunk size necessary. The chunk size is determined during the tuning phase, so the plugin must be dynamic in that sense. This solution would not be possible given the plugin interface, but it would save a lot of memory, not allocating a full-size contiguous buffer.

Our third solution was to look at the SISCI API. There is an additional DMA call *SCIStartDmaTransferMem*. The SISCI documentation states that it

is possible to transfer directly from user memory, and there is no IOMMU requirement. After testing this call, we did confirm that this API call was working without an IOMMU. This allowed us to transfer from user memory, i.e., directly from the requested registered memory on the send peer, to a SISCI segment on the receive peer. By doing this, we do not have to allocate an intermediate buffer on the send peer, in effect reducing the required contiguous memory by half compared to the first solution. This solution did give use good and reliable performance given the circumstances.

Our solution is not the most efficient in terms of hardware capabilities. It does incur extra costs in terms of memory usage and CPU resources. GDRDMA does not work without a working IOMMU, so all transfers are intermediately stored in RAM before RDMA operations. However, we still have a working system, and it is working reliably for collective operations with buffer sizes under 1 Gigabyte (buffer per rank).

4.5 Future work

This implementation adapted an already existing plugin interface to enable NCCL applications to use Dolphin hardware. However, the NCCL-NET plugin is designed for a simple 1-directional connection between 2 nodes. PCIe, Dolphin hardware, and SISCI support more complex connection configurations. Such as Reflective Memory / PCIe Multicast, but the plugin interface does not allow for such features to be used. A solution to this would be to implement an additional plugin interface in NCCL to support this. The SHARP [16] plugin from Mellanox uses a different plugin interface from the standard NCCL-NET plugin.

4.6 Chapter Summary

In this chapter, we showed how we implemented the SISCI-NCCL plugin. The segment abstraction provided by the SISCI API makes it simple to perform RDMA operations, regardless of where memory is located. Having broken IOMMUs, made it hard to verify that our implementation was working correctly. The SISCI API did not report that there was something wrong with transmissions in certain configurations, making it hard to determine where the fault was located. However, we were able to find a workaround that allowed us to run benchmarks without using GDRDMA. In the next chapter, we benchmark our SISCI plugin and compare it to the default socket implementation.

Chapter 5

NCCL Benchmark

In this chapter, we present and discuss methods we use to measure the performance of NCCL. We use two different machine-to-machine interconnects (TCP/IP and PCIe). Here we are specifically profiling the performance of NCCL itself by only executing collective operations in a vacuum.

5.1 NCCL tests

NVIDIA provides a separate test repository for NCCL called *nccl-tests* [20]. This repository provides test programs that can validate the correctness of a build. There is one program for each NCCL collective operation. In addition, the test programs come with profiling functionality. The profiling features can be used to test different hardware configurations. This can be useful to verify if a configuration is working optimally, identifying bottlenecks. Profile data can, for example, be used to configure Tensorflow to work optimally together with NCCL [33].

Specifically for this thesis, we can use the profiling programs to compare different interconnects (TCP/IP against PCIe). These benchmarks will give concrete benchmarks for each collective operation. However, for this thesis, we want to measure the performance of an ML application using the library. In section 6 we discuss how to benchmark using Tensorflow.

5.2 Variables

To make a comprehensive benchmark, we need to determine the scale. The number of variables and resolution we select determines the number of benchmark runs we need to perform.

Hardware configuration plays a major role in the behavior of NCCL. GPUs, CPUs, network interconnect, memory, ranks, and nodes all play a role in performing the benchmark. Benchmarking on many hardware configurations is expensive, both monetary and in terms of time. While we considered it is interesting to explore how all hardware is affecting the performance, it is not strictly necessary for our benchmark. We wanted to

$$B_{\text{algorithm bandwidth}} = \frac{S_{\text{size}}}{t_{\text{time}}} \quad (5.1)$$

Figure 5.1: Algorithm Bandwidth formula

look at how the network interconnects are performing, so we only changed that type of hardware.

Data input size was also of interest to see how the performance developed. As the input size increases, execution time should increase. By running the benchmark at multiple sizes, it may be possible map out where a configuration is running optimally. As well as comparing how well configurations are running for certain sizes.

5.3 Metrics

In order to compare different benchmark runs, it is necessary to define metrics, so we are able to compare performance. We are trying to measure how hardware is able to perform algorithms and not how algorithms are able to solve a task. So we want to gather metrics that are able to give insight into how the hardware is performing.

The simplest metric to reason about is the *(*time (t)) it takes an algorithm to perform an operation. If one benchmark configuration is spending less time completing an operation than another, it is performing better. Time is ideal for measuring the latency of a system in small-scale tests. However, when the data size is increasing, time will usually map with the complexity of the algorithm. Only measuring time does not give us insight into how hardware is performing for a configuration.

Algorithm Bandwidth (equation 5.1) measures how much data an algorithm is able process over time. This is simple to measure and compute, as there is only 2 parameters to collect. By measuring the bandwidth of an algorithm, it gives insight into the capacity of the hardware. If the bandwidth is measured for a hardware configuration, then it is possible to estimate how long an operation takes by dividing the operation size by the bandwidth. However, this metric does not take the number of ranks into account. The complexity of collective operations increases as the number of ranks increase. For example, in *All Reduce*, the number of elements to reduce for each row increase by the number of ranks. While the algorithm bandwidth is good for a certain number of ranks, it does not give insight into how a system would perform with a different number of ranks.

NCCL Tests [20] implements *Bus Bandwidth* as an alternative to general algorithm bandwidth. This metric is designed to be independent of the number of ranks that are used in the measured operation. Each collective operation has its own Bus Bandwidth Formula as the complexity is different between the algorithms. It also takes the topology into account when correcting for ranks (ring or tree).

General system resource metrics are also of interest to measure. The socket plugin implementation relies on the CPU performing the inter-node

Name	Description
CPU	Intel i5-4590 CPU 4 core 3.30GHz
RAM	8 GB
GPU	2x Nvidia Quadro K2200 4GB VRAM
NIC	1Gb/s
Dolphin PXH810	Dolphin PCIe 3.0 NTB adapter

Table 5.1: Table of hardware installed in each node.

communication, while SISCI-NCCL is using the DMA engine located on the Dolphin adapter. It is interesting to see if this can contribute to the worse performance of the system. CPU and memory usage is reported by the Linux kernel and can be recorded during the execution of the benchmark. The Nvidia driver also exposes GPU metrics that can be recorded. We were not able to measure system usage for the strict NCCL benchmarks.

5.4 Benchmark Technique

To perform a benchmark, we needed to make sure our measurements were recorded reliably and without interference. Because we were running our benchmarks on an OS (Linux), we needed to make sure there were no background processes affecting measurements. As long as we made sure all benchmarks were run with the same background state, the measurements should not be affected.

Running each benchmark configuration more than once is also important. Even if we are running on a system only dedicated to benchmarking, it is still expected to have minor differences in measurements. There might be certain subsystems that are not initialized before the benchmark is executed. Dynamic objects, for example, are loaded on demand by processes. SISCI NCCL is a dynamic object and is loaded on demand by NCCL. If we were to install a new version of the plugin, the kernel would have to do a first-time load of the plugin. However, the second time the kernel has cached the dynamic object, reducing the startup time. So to protect against statistical outliers in our data, we made sure to run each configuration more than once and run "warm-up" runs before collecting data.

5.5 Hardware

To perform these benchmarks, we had two machines. They contained identical hardware (listed in table 5.1). Both machines ran Ubuntu 20.04.1 LTS. Unfortunately, these were the same machines that we used in the implementation phase. So the hardware suffered from the IOMMU problems described in section 4.3. While we wanted to perform benchmarks on different hardware, due to time and resource constraints, we were not able to do so.

Figure 5.2: All Reduce algorithm bandwidth SISCI vs socket.

5.6 Benchmark Results

These benchmarks compare the performance of the SISCI and socket implementation.

Figure 5.6 compares the algorithm bandwidth. The chart is constructed of multiple benchmark runs of different operation sizes. Samples are collected in power of 2s, starting at 8 and ending at 1GB. This benchmark is reducing floats, so each element is 4 bytes large. Between 2 machines, we have 4 GPUs, so the reduction has 4 ranks.

Figure 5.6 shows the computed bus bandwidth. The shape is almost identical to figure 5.6. However, the values are corrected to give insight into the link bandwidth in the system.

Figure 5.6 and 5.6 show that for operation sizes under 8000 elements, the performance is similar. For larger sizes, the SISCI version is able to achieve higher bandwidth and peeks at sizes above 1100000 elements. The socket version also peeks around the size of the SISCI version but at a much lower bandwidth. Also, the SISCI version was able to handle a larger capacity than the socket version. We were able to run the SISCI version on a 1GB large all reduce operation, while the socket was only able to reach 0.5 GB.

Figure 5.6 compares the time used for each version. The SISCI version is generally faster except for small sizes under 10000 elements. This may be caused by the cost of starting a DMA transfer.

Figure 5.6 and 5.6 shows the average algorithm and bus bandwidth of all supported operations. These benchmarks are sampled the same way as in the previous charts by profiling for many-sized operations. The charts show that every collective operation benefits from a faster interconnect. It

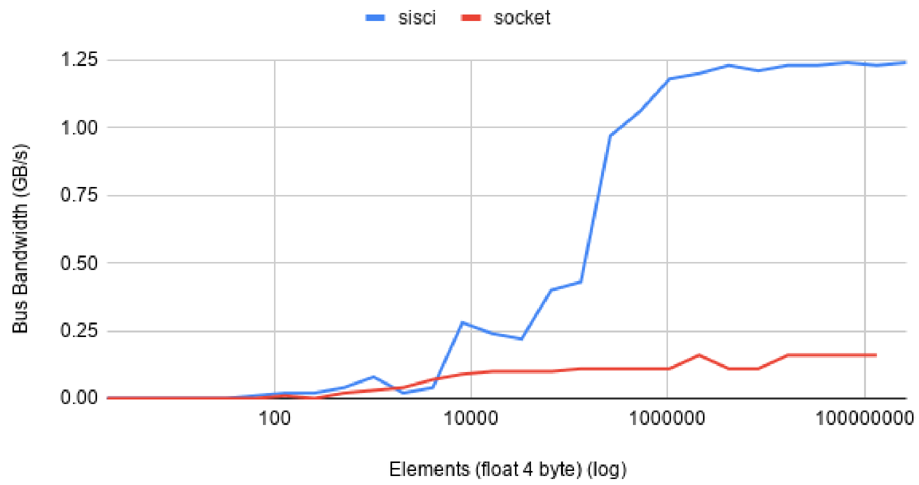

Figure 5.3: All Reduce bus bandwidth SISCI vs socket.

Figure 5.4: All Reduce time SISCI vs socket.

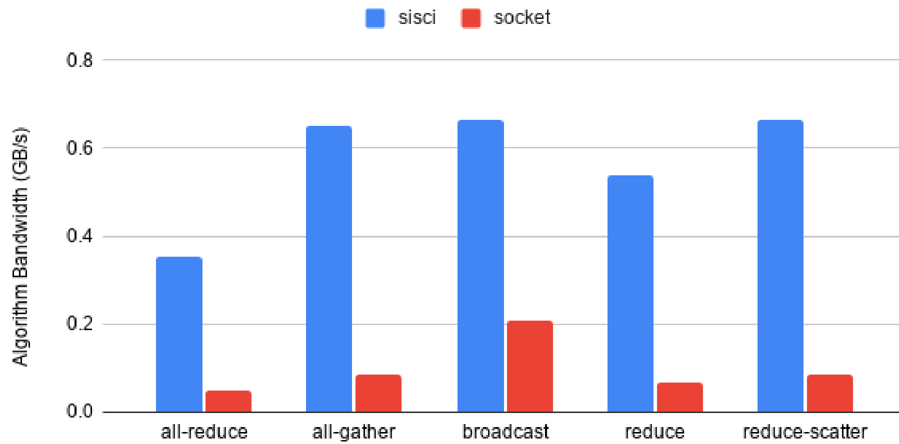

Figure 5.5: All Reduce average algorithm bandwidth for all collective calls.

also shows that a 1 Gigabit Ethernet interconnect is not sufficient to reach optimal performance for this hardware configuration.

5.7 Chapter Summary

In this chapter, we benchmarked NCCL alone, only performing collective operations. Our benchmarks show that our plugin is able to utilize hardware resources provided by the Dolphin adapters. The benchmarks show that small-size operations do not benefit from our plugin. This may be due to the cost of dispatching DMA transfers. While large operations do benefit, compared to the socket version. However, in the scope of this benchmark, the available bandwidth for each interconnect is very different, so this is not a fair comparison.

In the next chapter, we look at how our SISCI-NCCL plugin is performing when used in a Tensorflow application.

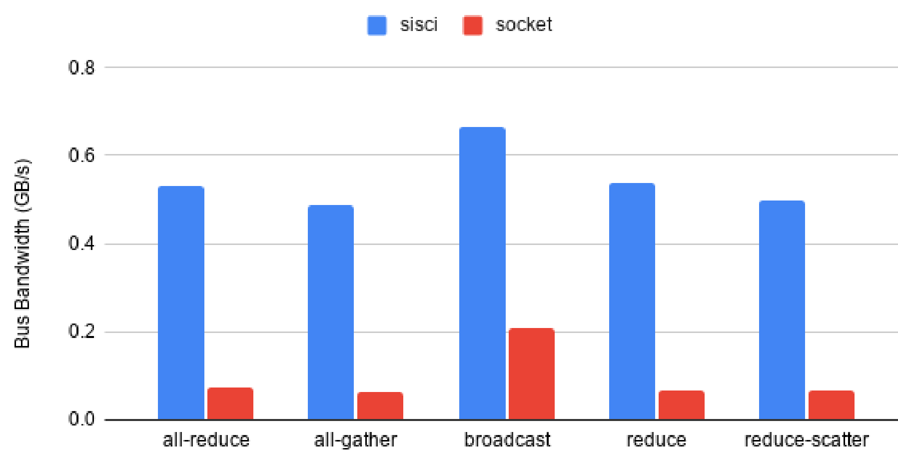

Figure 5.6: All Reduce bus bandwidth for all collective calls.

Chapter 6

Tensorflow Benchmark

NCCL is a library, so its main purpose is to provide the functionality to other applications. While we are observing speedup when benchmarking NCCL by itself, we also want to investigate if this speedup is also observed when using Tensorflow.

6.1 Installation

We decided to use a fairly new version of Tensorflow (version 2.4.1) for this project. This was to access some experimental features of Tensorflow. While NCCL is usually installed as a dynamic library on a system, Tensorflow uses NCCL as a static library. This required us to implement our plugin for NCCL v2.7.* or later, which we did.

6.2 Metrics

Like in chapter 5, we need to select metrics to measure our benchmarks with.

There are many ways to measure the performance of an ML application. A common way to evaluate an ML application is to measure how fast it converges to a given accuracy. Many variables affect the speed of convergence, such as dataset and batch size. In our case, we want to measure how the machine interconnects is performing when computing an ML model and not how a particular model is performing. Given the assumption that the model is deterministic, i.e., the output will always be the same for a given configuration, the interconnect should not matter for the model performance. However, it does matter for the execution speed in a multi-node training configuration.

One way to measure execution speed is to measure the time to complete a full training job. This is usually constructed of many multiple epochs that are again constructed of multiple batches. Batch time only measures the time it takes to complete a batch.

6.3 Model Configuration

At the core of any Tensorflow application, there is a model. We are trying to measure how Tensorflow applications are performing with our SISCI NCCL plugin, so we have to create some models that are able to run on our system. ML models vary in configuration. The number of features and model size dictate the memory footprint of the model. We need to select models that are able to run on our hardware configuration. In addition, we need to configure our Tensorflow applications to be distributed over multiple nodes.

Luckily Tensorflow features many state-of-the-art implementations of image classification models [3, 30, 31]. This made it easy to create a Tensorflow application to test NCCL with. We wanted to use models that are commonly used in the ML community to provide a realistic ML scenario. We selected three models for our benchmark, they are:

- EfficientNet [27]
- MobileNet [13]
- ResNet [12]

While these three models are able to take any images as input, they were originally created to be trained on the imagenet dataset [5]. Imagenet is commonly used as an image classification benchmark. The models are optimized to be executed with GPU acceleration, which is what we want for our benchmark.

When configuring our models, we had to make sure they were able to fit on our hardware. In our case, the main limiting factor was available VRAM on our GPUs. Simplified, the way Tensorflow distributes training across multiple GPUs is to create a copy of the model on all ranks. Each ranks processes an independent part of the dataset and applies it to the model. Then using collective operations (all reduce), all ranks reach a consensus on how to update the model. Because every rank has to store the complete model, we are limited to the amount VRAM available on each rank.

6.4 Tensorflow Profiler

TensorFlow provides a large selection of tools to create ML applications. TensorBoard [28] is a data visualization tool distributed together with Tensorflow. It allows a user to easily visualize and analyze logs generated by Tensorflow. While the most common usage of TensorBoard is to analyze the performance of ML models. It is possible to use TensorBoard to analyze how models are performing on hardware.

Tensorflow Profiler is a tool that measures how a TensorFlow computation is executed on a system. The profiler is able to log what computation is done, where the computation is done in a system, and how many resources a computation is using. Specifically useful for this thesis, the profiler is able

to log when and where data is moved in the system. This can give very specific and accurate insight into how the underlying NCCL implementation is performing. As with the NCCL profiler, the TensorFlow profiler can be used to tune a system for optimal throughput. The profiler does use a significant amount of resources, and it does affect the performance compared to not enabling the profiler. Tensorflow Profiler will account for this in its own measurements. However, this makes external measurements inaccurate.

For the version we were using, Tensorflow Profiler does not implement a full feature set for distributed training. This meant it was hard for us to get a complete profile of our system using the built-in profiler. While the profiler is intended to be able to profile a distributed system, we found it unreliable in its current state to do so. We were able to use the profiler to collect aggregate metrics such as average batch time and average collective call time per batch. However, we were not able to gather trace data for the whole system using the profiler.

6.5 Messuring System Usage

Because we were not able to collect a performance trace using TF Profiler, we decided to create our own tracer. The Linux kernel does expose a mechanism to record system performance. Certain files in the sysfs structure are continually printing hardware metrics, such as CPU time, memory usage, and process time. For the GPUs, we are also able to record metrics via the Nvidia driver. We can record these metrics over time while executing a benchmark and get a trace of how hardware is performing over time.

We wrote a script to poll these metrics at time intervals continually. This script had a small resource footprint, as we did not do any computation, only virtual file IO. We also set the sample interval to 200 milliseconds to minimize interference. This provided some extra system data we were not able to record with TF Profiler. One downside to this program was that we had to run every benchmark twice, one with TF profiler enabled and one without. Because TF Profiler does affect performance significantly, we did not want that to poison our hardware trace.

6.6 Execution Enviroment

Tensorflow does not provide a system for bootstrapping distributed training. While there are existing generic third-party solutions for this, but these are usually intended for a normal Tensorflow workflow. As we intended to record benchmarks, we decided to create our own system to bootstrap our benchmark. By creating our own system, we were able to tailor it to our needs easily.

We build a generic bash [11] script to execute and record statistics about our benchmarks. The script is using Secure Shell Protocol (SSH) [2] to control multiple nodes. Simplified the system works like this for each node:

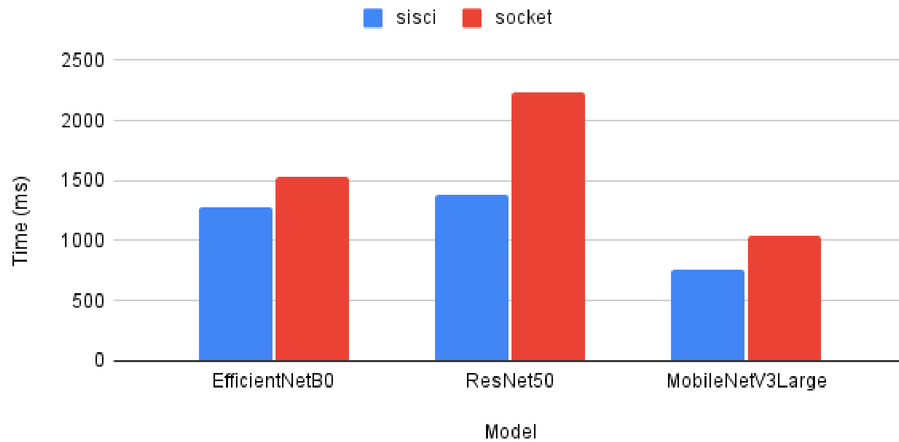

Figure 6.1: Average Batch Time SISCI vs socket.

1. Copy benchmark program to node
2. Set up enviroment variables, and start background recording programs (hardware trace)
3. Execute benchmark program
4. Stop background recording programs
5. Copy recorded logs and output back to source machine

After we made this system work, it was simple to record benchmarks.

6.7 Benchmark Results

Figure 6.7 compares the average time used to execute a batch. A batch is constructed of multiple collective operations depending on the number and type of layers in the model. The size of collective operations also varies depending on the number of weights used in each layer of the model. Figure 6.7 shows that the SISCI version is faster, but it does not give a speedup equal to the difference in available bandwidth.

The most time during a batch execution is spent performing non-collective operations on GPUs. To get a more accurate view of how the collective calls are performing, figure 6.7 extracts the time spent executing NCCL kernels. Here the SISCI version is performing much better, giving us a 12x speedup for the collective operations for the *EfficientNetB0* model.

Table 6.1 shows the average of recorded system usage when executing an epoch for a given model.

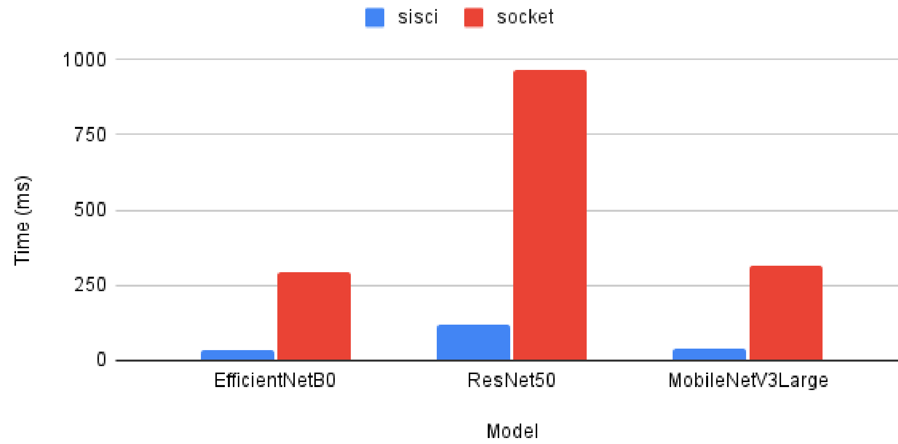

Figure 6.2: Average Device Collective Communication Time SISCI vs socket.

Model	Interconnect	average cpu usage (%)	average memory usage (%)	average gpu usage (%)	average gpu memory usage (%)
EfficientNetB0	sisci	0.6392495741	0.9546551428	0.9897945942	0.4483567118
ResNet50	sisci	0.6438623702	0.9673391203	0.9913877798	0.546127255
MobileNetV3Large	sisci	0.677733989	0.9713289193	0.9798745917	0.3826588625
EfficientNetB0	socket	0.6264324063	0.8496968488	0.9890931916	0.3701553092
ResNet50	socket	0.3974370299	0.9684871325	0.9927373446	0.3371228901
MobileNetV3Large	socket	0.6365228468	0.9390030276	0.9803619229	0.2783296208

Table 6.1: Average system usage recorded during model execution.

6.8 Chapter Summary

In this chapter, we benchmarked our SISCI-NCCL plugin in a "real-world scenario". While distributed training depends on collective operations to work, the benchmark machines spent the majority of execution time on compute, and not performing the collective operation. For the collective operation time, our plugin was able to outperform the default socket version. As for system resources, it seems like there is not a significant difference in aggregate resource consumption between the plugins. This might be due to the broken IOMMU, forcing us to perform extra copy operations on the CPU when transferring data.

Chapter 7

Conclusion

7.1 Summary

The goal of this thesis was to investigate if a PCIe interconnect may benefit distributed ML training. Our hypothesis was that using PCIe for both intra- and inter-communication may be beneficial in a distributed ML training application.

We started out by exploring relevant technologies used for extending PCIe for inter-machine communication by creating an NTB. Then we introduced NCCL, a Collective Operation library created to provide state-of-the-art collective communication for Nvidia GPUs. Tensorflow is an ML framework that can use NCCL to perform distributed ML training.

To enable inter-machine PCIe support, we had to implement a plugin for NCCL. During the implementation phase, we discovered our plugin was not operating as expected. Due to the complexity of NCCL, we had a hard time determining the cause of this disruption. After analyzing the NCCL implementation, we did narrow it down to a faulty IOMMU. Due to resource restrictions, we were not able to get new hardware, so we devised an emergency solution. With our solution, we were able to produce a working plugin.

With our working plugin, we performed some benchmarks. First, we tested NCCL in a vacuum, profiling Collective Operations at many configurations. We compared the performance of our plugin to the performance of the default NCCL socket plugin.

Then we wanted to test NCCL with our plugin in a real-world scenario. We set up a Tensorflow installation that was able to use our SISCI-NCCL plugin. We were able to test 3 common image classification models using SISCI-NCCL and socket NCCL. Our results showed that our NCCL plugin was able to reduce the time used in collective operations to an insignificant amount. While our plugin did not utilize the available hardware resources perfectly, we were able to yield a significant speedup in terms of collective operations in an ML training application.

7.2 Main Contributions

A large part of this thesis was used debugging systems. While NCCL supports a plugin interface, they provide no explicit documentation on how to implement a plugin. We did a deep investigation into how NCCL is implemented, exploring the implementation of key features in the library.

We were able to implement a proof of concept plugin, enabling NCCL to use Dolphin NTB adapters. Our benchmarks demonstrate that our plugin is able to utilize Dolphin adapters. While we were not able to demonstrate all intended features (due to the IOMMU issue), we were still able to demonstrate a working and reliable system.

Benchmarking Tensorflow, we were able to show how network interconnects affect model training performance. We were able to show that our plugin reduced the time spent performing collective operations to an insignificant amount compared to the default socket plugin.

7.3 Future Work

In this thesis, we implemented and benchmarked a proof of concept. While our implementation is working, it is not working optimally, and there is much room for improvement. In addition, our benchmarks are not complete due to limited hardware resources.

7.3.1 Compare to Other High Performance Interconnects

It would be interesting to see how our plugin is performing compared to the other supported NCCL plugins. In this thesis, we were only able to test the socket default version with a 1 Gigabit Ethernet interface. Comparing our plugin to InfiniBand, 40/100 Gigabit Ethernet or Mellanox SHARP capable networks would be of interest.

7.3.2 Larger Benchmarks

Our benchmarks only used 2 nodes with 2 GPUs each, giving us 4 ranks. However, the Dolphin PCIe interconnect supports larger systems. Due to GPU capacity, the models used when benchmarking with Tensorflow were also of reduced size. Given a larger benchmark, we would probably observe a greater speedup than what we were able to in this thesis.

Appendix A

Source Code

The source code produced in this thesis can be accessed at https://github.uio.no/sivertac/sisci-nccl-tf-benchmark.

The plugin code is located under the *sisci-nccl* folder. It is a fork from the original SISCI NCCL repository created by Eivind Alexander Bergem (https://github.com/Dolphinics/sisci-nccl).

The *tf-benchmark* folder contains code scripts that were used to benchmark Tensorflow applications.

Listings

3.1 Bootstrap create ring pseudo code 13
3.2 Bootstrap all gather pseudo code 14
3.3 NCCL topology XML dump 15
4.1 NET plugin interface from `nccl:src/include/nccl_net.h` . 21
4.2 SISCI-NCCL init function from `sisci-nccl:src/sisci_nccl.c` 22
4.3 SISCI-NCCL device structure from `sisci-nccl:src/sisci_nccl.c` 23
4.4 NET plugin ncclSisciDevices from `nccl:src/include/nccl_net.h` 24
4.5 NET plugin properties struct from `nccl:src/include/nccl_net.h` 24
4.6 SISCI PCIe path function from `sisci-nccl:src/sisci_nccl.c` 24
4.7 NCCL topology XML dump detecting a Dolphin NTB adapter 26
4.8 Segment allocator function from `sisci-nccl:src/sisci_nccl.c` 27
4.9 Mailbox read/write from `sisci-nccl:src/sisci_nccl.c` . . 28
4.10 Listen function from `sisci-nccl:src/sisci_nccl.c` 29
4.11 Connect function from `sisci-nccl:src/sisci_nccl.c` . . . 30
4.12 Accept function from `sisci-nccl:src/sisci_nccl.c` 31
4.13 Register memory function signature from `sisci-nccl:src/sisci_nccl.c` 32
4.14 Transfer function signatures from `sisci-nccl:src/sisci_nccl.c` 33

Bibliography

[1] *Apache License, Version 2.0*. URL: https://www.apache.org/licenses/LICENSE-2.0.

[2] Daniel J. Barrett, Richard E. Silverman and Robert G. Byrnes. *SSH, the Secure Shell: The Definitive Guide*. O'Reilly Media, Inc., 2005. ISBN: 0596008953.

[3] François Chollet et al. *Keras*. https://keras.io. 2015.

[4] Jeffrey Dean et al. 'Large Scale Distributed Deep Networks'. In: *NIPS*. 2012.

[5] Jia Deng et al. 'ImageNet: A large-scale hierarchical image database'. In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.

[6] P.J. Denning et al. 'Computing as a discipline'. In: *Computer* 22.2 (1989), pp. 63–70. DOI: 10.1109/2.19833.

[7] Mica R. Endsley. 'Autonomous Driving Systems: A Preliminary Naturalistic Study of the Tesla Model S'. In: *Journal of Cognitive Engineering and Decision Making* 11.3 (2017), pp. 225–238. DOI: 10.1177/1555343417695197. eprint: https://doi.org/10.1177/1555343417695197. URL: https://doi.org/10.1177/1555343417695197.

[8] MPI Forum. *A Message-Passing Interface Standard Version 3.1*. 2015. URL: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[9] MPI Forum. *MPI Forum*. URL: https://www.mpi-forum.org/.

[10] Linux Foundation. *NTB Drivers*. URL: https://www.kernel.org/doc/Documentation/ntb.txt.

[11] P GNU. *Free Software Foundation. Bash (3.2. 48)[Unix shell program]*. 2007.

[12] Kaiming He et al. 'Deep Residual Learning for Image Recognition'. In: (2015). URL: https://arxiv.org/abs/1512.03385.

[13] Andrew G. Howard et al. 'MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications'. In: (2017). URL: https://arxiv.org/abs/1704.04861.

[14] Esther Landhuis. 'Deep learning takes on tumours'. In: (). DOI: 10.1038/d41586-020-01128-8. URL: https://doi.org/10.1038/d41586-020-01128-8.

[15] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[16] Mellanox. *Mellanox SHARP website*. URL: https://www.mellanox.com/products/sharp.

[17] NVIDIA. *CUDA Toolkit Documentation*. URL: https://docs.nvidia.com/cuda/.

[18] NVIDIA. *NCCL documentation*. URL: https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/index.html.

[19] NVIDIA. *NCCL git repository*. URL: https://github.com/NVIDIA/nccl.

[20] NVIDIA. *NCCL tests git repository*. URL: https://github.com/NVIDIA/nccl-tests.

[21] NVIDIA. *NCCL website*. URL: https://developer.nvidia.com/nccl.

[22] NVIDIA. *Parallel Thread Execution ISA*. URL: https://docs.nvidia.com/cuda/parallel-thread-execution/.

[23] OpenFabrics. *Libfabric website*. URL: https://ofiwg.github.io/libfabric/.

[24] PCI-SIG. *PCI Special Interest Group*. URL: https://pcisig.com/.

[25] Dolphin Interconnect Solutions. *Dolphin Interconnect Solutions website*. URL: https://www.dolphinics.com/.

[26] Dolphin Interconnect Solutions. *Dolphon SISCI API Users Guide*. URL: https://www.dolphinics.com/download/SISCI/OPEN_DOC/SISCI_API_2_users_guide.pdf.

[27] Mingxing Tan and Quoc V. Le. 'EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks'. In: (2019). URL: https://arxiv.org/abs/1905.11946.

[28] tensorflow. *TensorBoard website*. URL: https://www.tensorflow.org/tensorboard.

[29] tensorflow. *tensorflow git repository*. URL: https://github.com/tensorflow/tensorflow.

[30] tensorflow. *TensorFlow Models Datasets*. URL: https://www.tensorflow.org/resources/models-datasets.

[31] tensorflow. *TensorFlow Models Github*. URL: https://github.com/tensorflow/models.

[32] tensorflow. *tensorflow website*. URL: https://www.tensorflow.org/.

[33] Zongwei Zhou. *Scaling TensorFlow 2 models to multi-worker GPUs (TF Dev Summit '20)*. URL: https://www.youtube.com/watch?v=6ovfZW8pepo.