Fast Multi-GPU communication over PCI Express

Benchmarking PCIe transport with the NVIDIA Collective Communications Library (NCCL) using legacy GPUs

Audun Kühne Johansen

Thesis submitted for the degree of
Master in Programming and System Architecture -
Distributed systems and networks
60 credits

Department of Informatics
The Faculty of Mathematics and Natural Sciences

UNIVERSITY OF OSLO

Spring 2023

Fast Multi-GPU communication over PCI Express

Benchmarking PCIe transport with the NVIDIA Collective Communications Library (NCCL) using legacy GPUs

Audun Kühne Johansen

Fast Multi-GPU communication over PCI Express

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

Printed: Reprosentralen, University of Oslo

Abstract

GPUs and PCIe are core components in distributed ML and HPC environments, and the software and hardware development within ML and HPC have been rapid. In this thesis, we look into how distributed communication with NCCL performs on legacy GPUs interconnected locally with QPI and P2P and externally with gigabit Ethernet, IPoPCIe and SmartIO device lending. We discover that many software packages lack backward compatibility.

Acknowledgments

The research presented in this thesis has benefited from the Experimental Infrastructure for Exploration of Exascale Computing (eX3).

A big thanks to the supervisors Håkon Kvale Stensland, Michael Riegler, and Jonas Markussen, as well as Pål Halvorsen for their knowledge and advice, the University of Oslo for supplying the servers and GPUs, Morten Werner Forsbring for helping me get time off work to focus on the thesis, and most importantly, my fiancée, who never stopped cheering me on, despite many delays.

Contents

Abstract i

Acknowledgments ii

1 Introduction 3
- 1.1 Background & Motivation . . . 3
- 1.2 Problem Statement . . . 4
- 1.3 Scope and limitations . . . 5
- 1.4 Research Method . . . 6
- 1.5 Ethical considerations . . . 6
- 1.6 Main contributions . . . 6
- 1.7 Thesis outline . . . 7

2 Background 8
- 2.1 PCIe . . . 8
  - 2.1.1 Root complex - the message bringer . . . 9
  - 2.1.2 PCIe layers . . . 10
  - 2.1.3 Transparent Bridging vs Non-Transparent Bridging (NTB) . . . 12
  - 2.1.4 Input–Output Memory Management Unit (IOMMU) 12
- 2.2 Intel QuickPath Interconnect (QPI) . . . 12
- 2.3 Remote Direct Memory Access (RDMA) . . . 13
  - 2.3.1 Dolphin Interconnect Solutions (Dolphin ICS) . . . 13
- 2.4 NVIDIA Collective Communications Library (NCCL) . . . 14
  - 2.4.1 Message Passing Interface (MPI) . . . 16
  - 2.4.2 Distributed machine learning techniques in NCCL . 16
- 2.5 Multimachine machine learning . . . 17
- 2.6 Summary . . . 18

3 System setup and challenges 19
- 3.1 Hardware . . . 19
- 3.2 Software . . . 23
  - 3.2.1 Operating system . . . 23
  - 3.2.2 NVIDIA drivers & tools . . . 24
  - 3.2.3 NCCL Tests . . . 26
  - 3.2.4 Dolphin eXpressWare . . . 29
  - 3.2.5 IPoPCIe . . . 30
  - 3.2.6 SmartIO . . . 30

4 Experiments and results **35**
4.1 Base-line reference . . . 35
4.2 NCCL . . . 36
4.2.1 Broadcast . . . 38
4.2.2 All-Reduce . . . 44
4.2.3 All-to-All . . . 50
4.2.4 All-Gather . . . 56
4.3 Discussion . . . 62
4.3.1 QPI vs P2P . . . 62
4.3.2 Gigabit Ethernet . . . 62
4.3.3 IPoPCIe . . . 62
4.3.4 SmartIO . . . 63
4.3.5 Closing thoughts . . . 64

5 Conclusions **66**
5.1 Summary . . . 66
5.2 Main Contributions . . . 67
5.3 Future Work . . . 68

A Captains log - The adventure of setting up our system **69**
A.1 Initial system . . . 69
A.1.1 Old setup . . . 69
A.1.2 Defective IOMMU . . . 69
A.2 Final system . . . 70
A.2.1 New setup . . . 70
A.3 Server configuration . . . 71
A.3.1 Firmware & BMC/IPMI . . . 71
A.3.2 Console access . . . 71
A.3.3 Operating system . . . 72
A.3.4 Snapshot tool . . . 73
A.3.5 Network . . . 73
A.3.6 Hosts . . . 75
A.3.7 SSH . . . 75
A.3.8 NVIDIA drivers & tools . . . 76
A.3.9 NCCL Tests . . . 80
A.3.10 Dolphin eXpressWare . . . 84
A.3.11 Tensorflow . . . 92
A.3.12 Misc notes . . . 93

List of Figures

2.1 Various PCI slots on a computer motherboard 8
2.2 Example of a PCIe topology 9
2.3 Illustration of bytes multiplexed across available PCIe lanes 10
2.4 Illustration of QPI interconnecting a NUMA and IO configuration 12
2.5 Accessing remote resources using RDMA 14
2.6 Illustration displaying the development of NCCL workload distribution 14
2.7 Illustration showing the unidirectional ring data travels in . 15
2.8 Illustration detailing the unidirectional data travel from each GPUs point of view 15
2.9 Illustration displaying the various interfaces that NCCL can use, both within and between systems (nodes) 16
2.10 Collective communication with multiple senders and/or receivers with NCCL & MPI 17

3.1 Supermicro X9DRG-HF System Block Diagram 20
3.2 Inside view of the Supermicro X9DRG-HF servers 22
3.3 Outside view behind the Supermicro X9DRG-HF servers . . 23

4.1 Low-level benchmark of the PXH810, displaying how different segment sizes affect performance. 35
4.2 Broadcast illustration 38
4.3 *Broadcast performance with GPUs over QPI (abel1), vs direct P2P (abel2)* 38
4.4 *Broadcast performance over gigabit Ethernet* 39
4.5 *Broadcast performance over IPoPCIe* 40
4.6 *Broadcast performance using SmartIO* 41
4.7 *Comparing multi-node interconnects 2 GPU broadcast performance, 1 in each node* 42
4.8 *Comparing multi-node interconnects 4 GPU broadcast performance, 2 in each node* 43
4.9 All-Reduce illustration 44
4.10 *All-reduce performance with GPUs over QPI (abel1), vs direct P2P (abel2)* 44
4.11 *All-reduce performance over gigabit Ethernet* 45
4.12 *All-reduce performance over IPoPCIe* 46
4.13 *All-reduce performance using SmartIO* 47

4.14 *Comparing multi-node interconnects 2 GPU all-reduce performance, 1 in each node* 48
4.15 *Comparing multi-node interconnects 4 GPU all-reduce performance, 2 in each node* 49
4.16 All-to-All illustration 50
4.17 *All-to-All performance with GPUs over QPI (abel1), vs direct P2P (abel2)* 50
4.18 *All-to-All performance over gigabit Ethernet* 51
4.19 *All-to-All performance over IPoPCIe* 52
4.20 *All-to-All performance using SmartIO* 53
4.21 *Comparing multi-node interconnects 2 GPU All-to-All performance, 1 in each node* 54
4.22 *Comparing multi-node interconnects 4 GPU All-to-All performance, 2 in each node* 55
4.23 All-Gather illustration 56
4.24 *All-Gather performance with GPUs over QPI (abel1), vs direct P2P (abel2)* 56
4.25 *All-Gather performance over gigabit Ethernet* 57
4.26 *All-Gather performance over IPoPCIe* 58
4.27 *All-Gather performance using SmartIO* 59
4.28 *Comparing multi-node interconnects 2 GPU All-Gather performance, 1 in each node* 60
4.29 *Comparing multi-node interconnects 4 GPU All-Gather performance, 2 in each node* 61
4.30 *All-to-All performance running 2 GPUs locally on abel2, or borrowed to abel1 with SmartIO* 64

A.1 IPMIView on an iPad 72

List of Tables

2.1 PCI Express link performance 11

3.1 Hardware specifications 19
3.2 Software specifications 23

4.1 Peak base-line performance 35
4.2 Performance and price comparison of some GPUs used in HPC environments 65

List of abbreviations

AGP	Accelerated Graphics Port
API	Application Programming Interface
BAR	Base Address Register
BMC	Baseboard Management Controller
CPU	Central Processing Unit
EISA/ISA	Extended ISA / Industry Standard Architecture
EMI	Electromagnetic Interference
FCP	Flow Control Packets
FMA	Fused Multiply–Add
GART	Graphics Address Remapping Table
GFLOPS	Giga Floating Point Operations Per Second
GPU	Graphics Processing Unit
HDD	Hard Disk Drive
HPC	High-Performance Computing
IOMMU	Input–Output Memory Management Unit
IPMI	Intelligent Platform Management Interface
IPoPCIe	IP driver for PCIe
LLM	Large Language Model
ML	Machine Learning
MPI	Message Passing Interface
NCCL	NVIDIA Collective Communications Library
NTB	Non-Transparent Bridging
NUMA	Non-Uniform Memory Access

P2P	Peer-To-Peer
PCIe	Peripheral Component Interconnect Express
QPI	Intel QuickPath Interconnect
RDMA	Remote Direct Memory Access
SSD	Solid State Drive
TCP/IP	Transmission Control Protocol / Internet Protocol
TDP	Thermal Design Power
TLP	Transaction Layer Packets
VESA	Video Electronics Standards Association

Chapter 1

Introduction

1.1 Background & Motivation

The computing demand for machine learning (ML) workloads has grown and is expected to grow. This demand is tackled in two ways; rapid development of specialized hardware and distribution of the workload.

The matrix-heavy nature of machine learning means the workloads are mostly offloaded from the CPU to accelerator cards, such as NVIDIA GPUs (graphics processing units). These GPUs contain large amounts of CUDA cores to execute operations on the matrix in parallel. Since late 2017, NVIDIA GPUs also began to offer Tensor cores[9] that of which each performs 64 fused multiply–add (FMA) mixed-precision operations. Primarily for deep learning, accelerating operations that would otherwise be performed by multiple CUDA cores. VRAM has also significantly increased to meet the demands of ML workloads.

Yet, some workloads are still to time and compute expensive for a single computer to finish within a reasonable time. To work around these issues, many ML algorithms allow splitting the workload and utilizing distributed computing. In computers or clusters with multiple NVIDIA GPUs, ML workloads can utilize the NVIDIA Collective Communications Library (NCCL) for a high-level distribution of the workload. By clustering multiple compute nodes to tackle the workload, we can use horizontal scaling to speed up training time, lowering the time-to-completion (strong scaling). Or we can use the extra computing power to increase the training set (weak scaling).

A downside of distributed computing is that some performance is lost in communication overhead between each GPU and compute node. How much of this overhead depends on the latency added, how much data transfer is needed, and available bandwidth. The cost of overhead, therefore, depends on the types of interconnect between the GPUs inside each node and the interconnect between the nodes themselves.

Scaling up and out with more GPUs and nodes is also costly, both monetary and performance-wise. Demand, availability, and thus prices for new GPUs and servers are high, partly due to semiconductor shortage[23]. And yet, in many data centers, existing GPUs and servers are taken

out of service, not because they're faulty, but because the warranty and service agreement has run out. New replacements are often fronted positively from an environmental perspective, as newer hardware yields higher performance per watt. While at the same time conflicting with the environment by generating e-waste. Still, some data centers must replace old hardware with new to handle higher requirements, such as enough VRAM. For example, running one of the trained models from OpenAIs Whisper project can require a minimum of 10 GB of VRAM[15]. And for training large language models (LLM), NVIDIA is releasing cards with 188GB of VRAM[6]. And with tensor cores, mixed precision (half and single) performance has increased significantly.

However, not all ML models demand as much VRAM to justify purchasing the latest and greatest. Nor does many HPC applications. Despite lower VRAM and mixed precision performance, older cards can offer strong double-precision performance that is on par with even the latest GPUs. Could expanding existing clusters with older hardware be a cost-effective alternative to replacing them with new hardware? Help reduce pressure on semiconductor shortage and the environment. Reuse and redistribute do come before recycling in the list of end-of-life scenarios for electronics[16]. A look at NVIDIAs driver support of their legacy line of GPUs shows that older models can serve far beyond the product warranty[7]. The question remains if there is anything else in the software stack that won't play well with older hardware.

With this in mind, we have our hands on two 10-year-old decommissioned GPU servers. We explore what it takes to get them back into production and the hardware and software limitations. Then we benchmark the performance of NCCL communication between GPUs across the various interconnects within and between the two servers. Among these, we will look at the NCCL performance when lending GPUs between the servers using SmartIO, a zero-overhead device-sharing method over PCIe Network.

1.2 Problem Statement

Modern NVIDIA GPUs for use in ML and HPC are expensive and hard to get. Availability and price have been affected by semiconductor shortages, supply chain issues, and high demand. GPU functionality, such as mixed precision performance, has increased over the years, but double precision performance has yet to get any better than models ten years older. This leads us to ask the following research question:

> *Do GPUs now considered legacy have a place in modern ML and HPC environments?*

The research question has been broken down into the following three objectives:

Objective 1

Find if there are challenges concerning software support, packages, and drivers when using legacy GPUs.

ML frameworks have developed rapidly. Newer GPU architectures offer higher compute capability, with features not found in legacy hardware. Support for legacy NVIDIA GPUs depends on the backward compatibility of the operating system, GPU driver, NVIDIA CUDA Compiler, and CUDA code.

Objective 2

Find if PCIe can be used as an efficient interconnect in an ML communication framework, such as NCCL.

Efficient GPU-to-GPU communication relies on the underlying PCIe interconnect. PCIe topologies can get messy as we add NUMA configurations and multimachine NTB interconnects. These can introduce barriers and bottlenecks for efficient communication between PCIe slots.

Objective 3

Find if it is possible to use PCIe device lending on GPUs and if it will help ML communication performance.

PCIe-based NTB interconnects offer functionality to connect multiple PCIe topologies to lend PCIe devices from one machine to another. It adds flexibility but is limited to the bandwidth of the NTB interconnecting card and the bandwidth of the PCIe slot it is connected.

1.3 Scope and limitations

While the servers used to be part of an HPC environment, they are from around 2013 and are no longer state-of-the-art but have PCIe Gen 3 capabilities. The NVIDIA Tesla K20X GPUs are from the Kepler generation with PCIe Gen 2 connectors at x16 width (8GB/s). The single available PCIe Gen 3 port for the Dolphin NTB interconnect is limited to x8 in width (7,877GB/s). Therefore, the bandwidth between the GPUs and the Dolphin interconnect is relatively similar. However, it becomes a theoretical bottleneck when transferring data from more than one GPU.

We do synthetic tests of NCCL performance across two internal PCIe interconnects, QPI and P2P, and two external interfaces, 1Gb Ethernet and Dolphin PXH810 interconnect. We explore two protocols with the Dolphin interconnect, TCP IP over PCIe (IPoPCIe) and device sharing with SmartIO. NCCL also supports 10/25/50+G Ethernet and InfiniBand with GPU Direct RDMA. However, we did not have the necessary components to test these scenarios.

This thesis also does not go into ML frameworks such as TensorFlow and PyTorch. They do, however, extensively use the NCCL API.

1.4 Research Method

Based on the paradigms for the discipline in Computing[8], our research has been within the third major paradigm; design. Rooted in engineering. Of the many subareas in computing, we are focusing on the architecture. Implementing machines for various computational models such as ML and high-performance computing (HPC). We started with a set of hardware limitations and the goal to benchmark a functional and as up-to-date as possible system within these limitations. We hypothesized and tested the newest operating system, drivers, CUDA libraries, and benchmark tools available. Then iterated multiple times on each of these as we discovered their limitations and faults, as well as methods to circumvent or fix them, in order to reach our goal.

1.5 Ethical considerations

Part of this thesis regards the performance of hardware and software that have been supplied by Dolphin Interconnect Solutions AS (Dolphin). Specifically the PXH810 cards, PCIe cable, drivers for IPoPCIe and SmartIO, and the benchmark tool for PIO and DMA performance. One of the two external supervisors of this thesis has employment at Dolphin. We have considered the possibility of conflicting interests and ensured to benchmark each interconnects with the exact same code and configuration. No benchmark methods, poor results, or errors are emitted or modified to favor one interconnect over the other.

The author of this thesis is independent and not affiliated with Dolphin. The assignment is supplied by the Department of Informatics (IFI) at the University of Oslo and Simula Research Laboratory AS (Simula). The main supervisor has employment at IFI and Simula. The second external supervisor has employment at Simula.

1.6 Main contributions

We found challenges concerning software support, packages, and drivers when using legacy GPUs. We confirmed that OS and GPU drivers still had support, but not with the newest versions. We found that the GPUs age hindered the use of the latest compiler, breaking compatibility with the newer CUDA code in NCCL tests.

We found that PCIe can be used efficiently in an ML communication framework such as NCCL, and we benchmarked internal and external PCIe interconnects. We found that with our legacy GPUs, in NUMA setups where GPU-to-GPU communication crosses QPI, it had little to no bottleneck in performance compared to a P2P configuration.

We confirmed that SmartIO increased bandwidth and lowered latency in multi-node communication. But the more GPUs added, the lower the average bandwidth went. We see there is potential for improvements with RDMA support. We found that IPoPCIe did not improve NCCL communication performance over built-in gigabit Ethernet.

We confirmed using PCIe device lending with SmartIO on legacy GPUs is possible. It did surprisingly help ML communication performance, such as borrowed GPUs performing better than internal GPUs. The software is, however, bleeding edge, with a few odd behaviors and a setup process that can easily cause the server to freeze.

With that, we answered our research question with the following:
The value of legacy GPUs in a modern ML and HPC environment is limited as they age, mainly because of software support. However, for HPC environments that do not mind supporting older software packages, they offer cost-effective double-precision performance.

1.7 Thesis outline

This thesis is divided into five chapters and one Appendix.

Chapter 1 introduces what and why we find it interesting to work on legacy GPUs and the potential of modern PCIe interconnects.

Chapter 2 gives the reader domain knowledge to help them understand the context of the rest of the thesis.

Chapter 3 presents the main steps on how we configured our system and explains some of the caveats that may appear for anyone deploying something similar.

Chapter 4 presents our test data and graphs, sorted by the type of collective communication.

Chapter 5 concludes the findings and discusses some of the potential.

Appendix A is an informal log of how the systems used in this thesis were configured. It is a longer and more in-depth version of Chapter 3, including tips and tricks, longer terminal outputs, and other things that are hopefully helpful to anyone replicating a similar solution.

Chapter 2

Background

This chapter details the underlying hardware and software that lay our benchmark's foundation.

2.1 PCIe

Adding peripherals such as accelerators and network cards is usually done by connecting to the "Peripheral Component Interconnect Express" (PCIe - 2003). Most modern CPUs offer point-to-point links ("bus lanes") following the PCIe standard, allowing for expansions beyond its built-in functions. Some of these lanes may be directly soldered to built-in peripherals such as USB controllers on a motherboard. Manufacturers may also run lanes to physical ports allowing us to insert our peripherals. These are the ports we connect our accelerator cards, such as GPUs, or network cards, such as fiber-optic Ethernet and Dolphin interconnects.

While the CPUs today have a lot of built-in functions, historically, everything from disk controllers, memory controllers, graphics, and networking were handled externally from the CPU die. Preceding PCIe, there was, for example, the "Industry Standard Architecture" (ISA - 1981), an extended version (EISA - 1988), or the "VESA Local Bus" (VLB 1992). However, "Peripheral Component Interconnect" (PCI - 1992) became the industry standard of the 90s. As accelerator

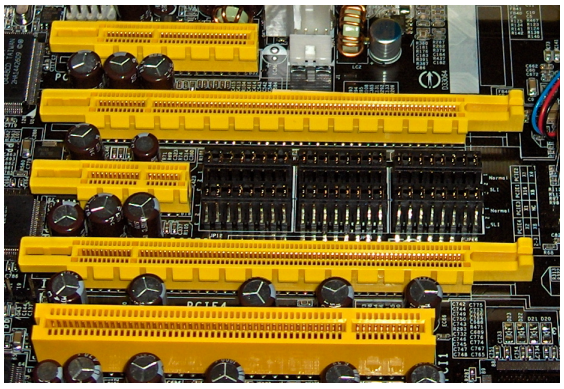


Figure 2.1: Various PCI slots on a computer motherboard[1]

[27]

[1]Slot type from top to bottom: PCI Express x4, PCI Express x16, PCI Express x1, PCI Express x16, Conventional PCI

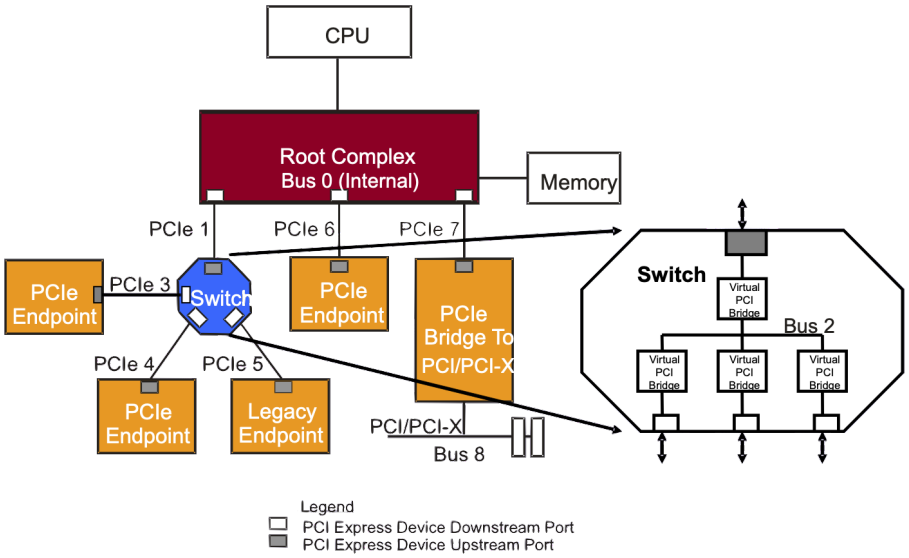


Figure 2.2: Example of a PCIe topology [17]

cards such as GPUs grew to be bottlenecked by the PCI standard, a superset of the conventional PCI bus named "Accelerated Graphics Port" (AGP) existed in parallel with PCI until the express version of PCI got introduced in 2003. The use of parallel links was common at the time for PCI and AGP, as well as the mentioned earlier standards. They send several bits as a whole down the link at the same time using several parallel channels. Meaning an 8-bit message demands eight dedicated channels (wires). Furthermore, the channels must be the same length, or signals arrive offset. With the introduction of PCIe, link communication is changed from parallel to serial, sending data one bit at a time, sequentially over one or more communication channels. The more channels added to PCIe, the higher the bandwidth. And increasing clock speed enables higher throughput. By the end of the decade after its introduction, PCIe took over PCI and AGP as the industry standard for adding peripherals.

2.1.1 Root complex - the message bringer

The root complex is the interchange that connects the CPU and memory to the PCIe switch fabric. Historically this part used to be implemented as a discrete motherboard component (north bridge chip); however, it is now mostly integrated into the CPU die. The root complex is the unit that generates transaction requests to the PCIe devices on behalf of the CPU. For examples of how various devices are connected, see topology figure 2.2. The operating system will set up the memory table for each PCIe end-point (Type 0 Configuration Table). It will give the root complex a master record of memory spaces accessible by each end-point (Type 1 Configuration Table).

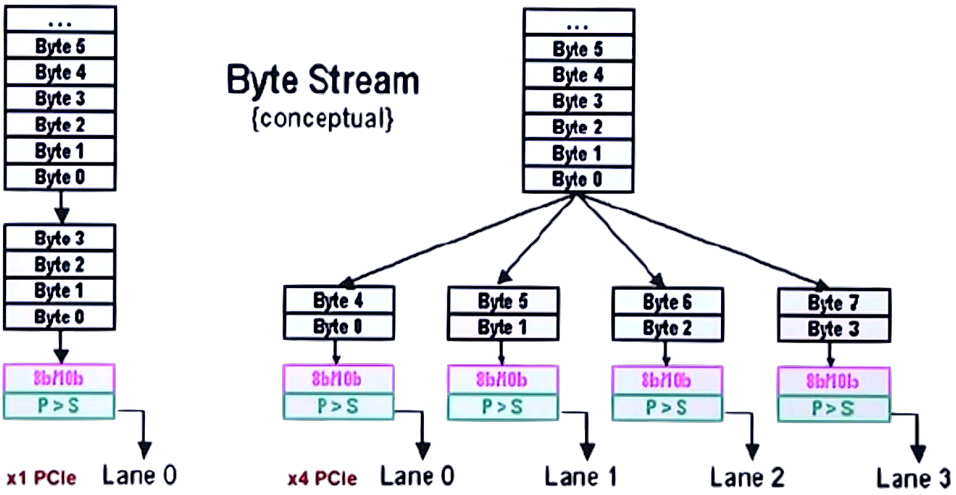


Figure 2.3: Illustration of bytes multiplexed across available PCIe lanes [10]

2.1.2 PCIe layers

Three layers make up the protocol of PCIe, the Physical layer, the Data link layer, and the Transaction layer.

Physical Layer - links and lanes

A lane consists of four physical wires to make two differential signaling pairs. One pair for transfer and one pair for receiving means lanes can operate at full duplex. A minimum PCIe connection needs only one lane, commonly abbreviated to "x1". A link can be made by one or more lanes, up to 32 at most. The link size is often marked next to the port in a conventional computer. For example, a port to use for a GPU might be marked "PCIe x16", meaning it contains enough channels to support 16 lanes.

Although a PCIe device might have pinouts to support 16 lanes, we don't have to use them all for it to be functional. A PCIe card with 16 lanes will still work when connected to a 4-lane port. Albeit with throttled throughput. And vice versa, a card with four lanes can still be connected to a 16-lane port. The extra lanes are then simply not used ("wasted").

If a link is made using multiple lanes, broadening the bandwidth, data will be multiplexed by striping it across the lanes as illustrated in figure 2.3.

Initial communication consists of link training. When a link comes up, both peers start at PCIe version 1 speed (2.5 Gbps). Each lane has its own clock, but the two devices' clocks may be offset. In order for any signals to be correctly interpreted, the clock on the receiver end will synchronize itself to the sender by sampling arrivals of rising edges (as 0 turns to 1). A common 'Training set' is transmitted across each lane. Both peers may then negotiate up to the highest common speeds supported by both peers. This enables backward support and means any PCIe devices, regardless of the version supported, can still communicate at high speeds with each other.

Table 2.1: PCI Express link performance [27]

Version	Intro-duced	Line code	Transfer rate per lane	Throughput				
				×1	×2	×4	×8	×16
1.0	2003	8b/10b	2.5 GT/s	0.250 GB/s	0.500 GB/s	1.000 GB/s	2.000 GB/s	4.000 GB/s
2.0	2007	8b/10b	5.0 GT/s	0.500 GB/s	1.000 GB/s	2.000 GB/s	4.000 GB/s	8.000 GB/s
3.0	2010	128b/130b	8.0 GT/s	0.985 GB/s	1.969 GB/s	3.938 GB/s	7.877 GB/s	15.754 GB/s
4.0	2017	128b/130b	16.0 GT/s	1.969 GB/s	3.938 GB/s	7.877 GB/s	15.754 GB/s	31.508 GB/s
5.0	2019	128b/130b	32.0 GT/s	3.938 GB/s	7.877 GB/s	15.754 GB/s	31.508 GB/s	63.015 GB/s
6.0 (planned)	2021	PAM-4 + 256B FLIT + FEC	64.0 GT/s	8.000 GB/s	16.000 GB/s	32.000 GB/s	64.000 GB/s	128.000 GB/s

Link training also consists of the peers agreeing on polarity inversion, calibrating out delays between lanes, i.e., compensating differences in lane length and changing lane numbers if the wiring is out of order (eases routing of the PCB layout).

Line coding is done to reduce electromagnetic interference (EMI) during transfer. The sender and receiver contain an identical pseudo-random data scrambler. Line encoding and decoding of the signal are done to keep the electric disparity as close to zero as possible. Ensuring the number of 1s and 0s is even. Without this, a series of binary 1s would slowly charge up the DC voltage in the circuit. Then when a 0 is sent, the high charge in the circuit could cause it to be interpreted as a 1, corrupting the message. Or if there were a long enough stream of 0s, the receiver won't have any rising edges to synchronize its clock to. Line coding in earlier PCIe versions was 8b/10b, and as of version 3.0 to 5.0, 128b/130b is used (essentially 64b/66b with double the block size). For a complete chart of the various versions, see table 2.1.

Data link layer - Dealing transaction layer packets

PCI Express communication is encapsulated in transaction layer packets (TLPs) packets. The data link layer sequences the TLPs generated by the transaction layer. In the header of outgoing TLPs, the data link layer inserts an incremental sequence number. A redundancy check code is also added for the receiver to check against unexpected corruptions. Similar to network protocols like TCP, an acknowledgment protocol consisting of ACK and NAK signaling ensures reliable delivery of TLPs between endpoints. If a NAK signal is received, or no response is heard from the receiver after a certain time, then something goes wrong along the way, and the same TLP is sent again. On the receiving end of the data link layer, after receiving a valid TLP, the packet is forwarded to the transaction layer.

Transaction layer

Buffers between PCIe devices vary, so a credit-based flow control is used not to overflow the receiver buffer. The devices communicate their credit (buffer size), and the transmitter only sends new TLPs if there's credit left. On the receiver end, it'll process the TLPs in its buffer and, after that's done, respond to the sender with a special packet named Flow Control Packets

(FCP) to update the sender that the receiver is ready for more data (aka return credit).

2.1.3 Transparent Bridging vs Non-Transparent Bridging (NTB)

[4] The PCIe architectural model allows only one root in the root complex. And all connected devices ("endpoints") must share a common address space. A transparent bridge means the root complex can see all endpoints in the system. All allocated memory for a PCI-e device is synonymously used by only one end-point (one PCIe device and not a switch or some other middleman).

Non-Transparent Bridging (NTB) circumvents this limitation to connecting multiple root complexes together. An "end-point" and its memory area are thus not the actual endpoint but a middleman to interconnect the root complex onto another root complex. This means one host (CPU) or PCIe device on one switch partition can initiate transactions with other hosts and their devices. Software maps the "middle-mans" allocated memory and sees beyond the NTB bridge and what devices are available outside the host's root complex. NTB enables multi-machine interconnects, and in this paper, we'll work with the interconnecting "middlemen"-devices Dolphin Interconnect Solutions offer.

2.1.4 Input–Output Memory Management Unit (IOMMU)

IOMMU enables virtual addressing of the main memory to PCIe devices [24]. Similar to MMU, that does virtual addressing of physical memory addresses for the CPU. Among other things, IOMMU allows PCIe devices to map continuous virtual memory addresses, even if the underlying physical address space is fragmented. Graphics Address Remapping Table (GART) is an example of IOMMU used by PCI Express GPUs to read and write from main memory.

2.2 Intel QuickPath Interconnect (QPI)

QPI is an Intel processor interconnect between CPU sockets and IO hubs (root complex) [1]. In illustration 3.1, we can see how a Supermicro X9DRG-HF server has traces on the motherboard to use QPI as the interconnect between the CPU sockets. QPI

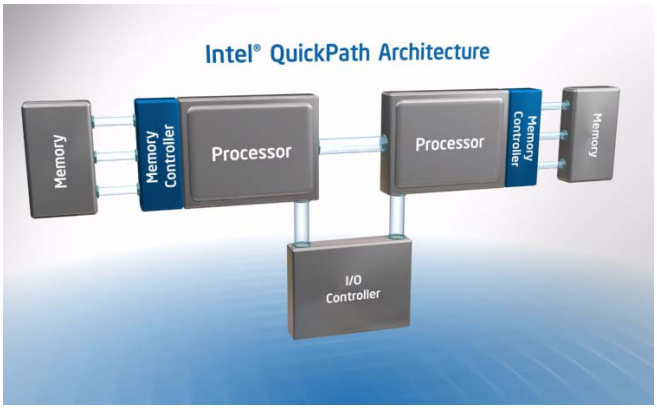


Figure 2.4: Illustration of QPI interconnecting a NUMA and IO configuration [1]

enables non-uniform
memory access (NUMA)
architecture in such
dual-socketed configurations.

From 2017 QPI was replaced with Intel Ultra Path Interconnect (UPI) [25].

2.3 Remote Direct Memory Access (RDMA)

RDMA allows the direct writing of one PCIe device's memory into another without involving the CPU. A co-processor, such as the GPU, can then do a "zero-copy" operation, circumventing the central processor and not performing any local memory copying between application memory and kernel memory. Processes running on the CPU can continue to operate in parallel while data is transferred, increasing performance and enabling high-throughput, low-latency data transfers between devices.

Support for RDMA depends on the interconnect and topology. If one PCIe device wants to write to another connected to a different NUMA node over QPI or via an NTB solution like Dolphin ICS, RDMA is not supported out of the box. A system designer must be conscious of the position of PCIe devices in a system to utilize RDMA.

2.3.1 Dolphin Interconnect Solutions (Dolphin ICS)

Network-oriented interconnects such as Ethernet and Fibre Channel are common for interconnecting hosts and sharing data. However, the overhead introduced by routable protocols may be undesirable, and a lower-level interconnect is needed. Some examples are InfiniBand, RapidIO, or NUMAlink. However, these are different standards than PCIe, adding a translation layer since the interconnecting hardware still operates over PCIe on each host.

The PCIe local-bus standard by itself, however, can also be used for interconnecting hosts and creating clusters. Dolphin's NTB PCIe hardware uses the existing protocol without translating and repackaging data. We can use this to expose PCIe capabilities for multi-machine communication. Allow direct RDMA between different hosts/root complexes, including peer-to-peer data transfer between PCIe devices (such as GPUs) on different fabrics without involving either CPU. With the bandwidth of PCIe and features such as reflective memory, multicast, and PCIe peer-to-peer communication, we can quickly move large amounts of data between hosts at low latency. Such interconnects can help distribute large data sets for computation. The host CPU and GPU can focus on doing calculations while the distribution of data is offloaded and handled by the dolphin cards.

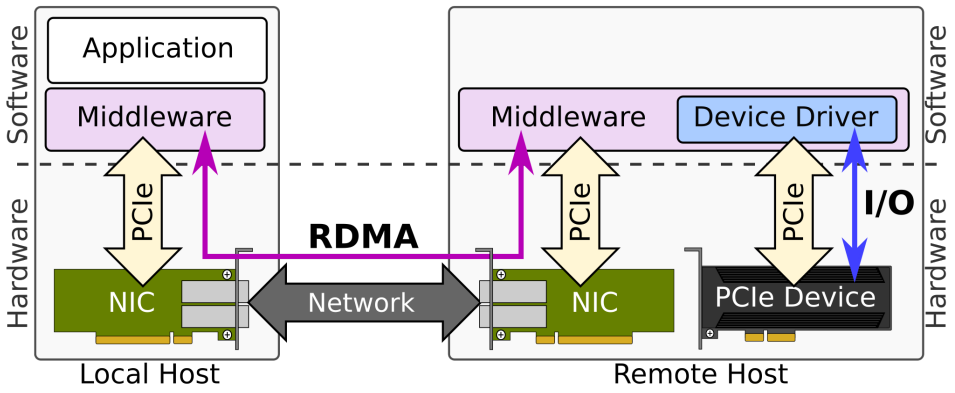


Figure 2.5: Accessing remote resources using RDMA [14]

IPoPCIe

With Dolphin's TCP IP driver for PCIe, the NTB cards can appear in the OS as a network interface, allowing quicker transfers with lower latency than built-in gigabit+ Ethernet [18]. It uses both SISCI PIO and RDMA operations depending on message sizes.

SmartIO

SmartIO is a framework that enables the lending and borrowing of PCIe Devices over a PCIe network without any software overhead [20]. Device Lending can be used to reconfigure systems and reallocate resources. GPUs, NVMe drives, or FPGAs can be added or removed without being physically installed in a particular system on the network.

2.4 NVIDIA Collective Communications Library (NCCL)

[5]

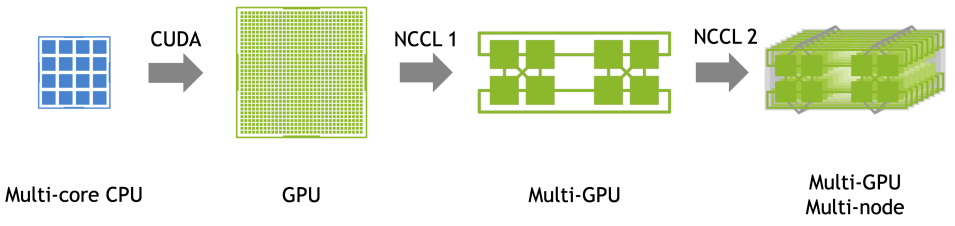


Figure 2.6: Illustration displaying the development of NCCL workload distribution [12]

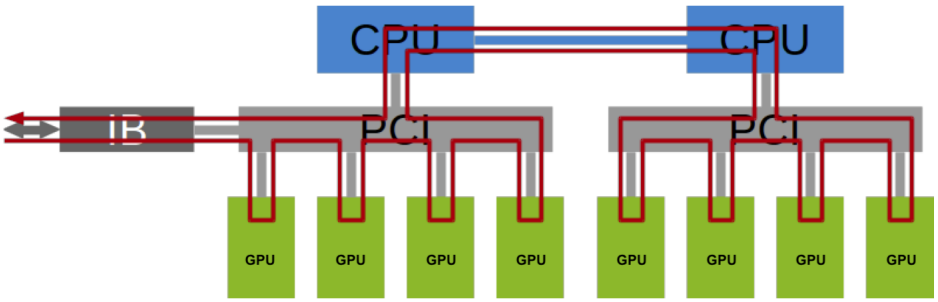


Figure 2.7: Illustration[2]showing the unidirectional ring data travels in [12]

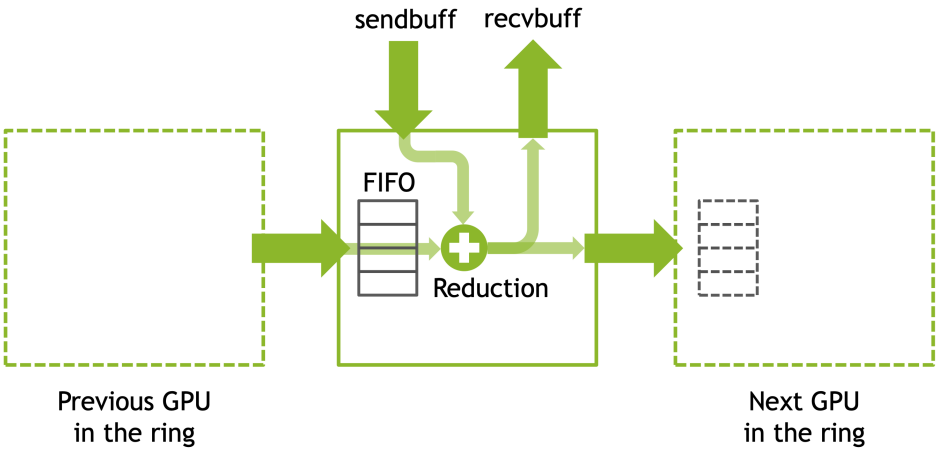


Figure 2.8: Illustration detailing the unidirectional data travel from each GPUs point of view [12]

The NVIDIA Collective Communications Library (NCCL, pronounced "Nickel") is a library providing inter-GPU communication primitives that are topology-aware and can be easily integrated into applications. NCCL implements both collective communication and point-to-point send/receive primitives. It is not a full-blown parallel programming framework but a library focused on accelerating inter-GPU communication. NCCL provides (among others) communication primitives such as AllReduce collective, which is heavily used for neural network training. It allows for point-to-point send/receive communication which allows for scatter, gather, or all-to-all operations. CUDA-based collectives would traditionally be realized through a combination of CUDA memory copy operations and CUDA kernels for local reductions. NCCL, on the other hand, implements each collective in a single kernel handling both communication and computation operations. This allows for fast synchronization and minimizes the resources needed to reach peak bandwidth. NCCL conveniently removes the need for developers to optimize their applications for specific

[2]Network interface named "IB" would in our setup be Gigabit Ethernet or the Dolphin card

machines. NCCL provides fast collectives over multiple GPUs and also across nodes since NCCL version 2. It supports a variety of interconnect technologies, including PCIe, NVLINK, InfiniBand Verbs, and IP sockets, as illustrated in 2.9. NCCL uses a simple C API, which can be easily accessed from a variety of programming languages.

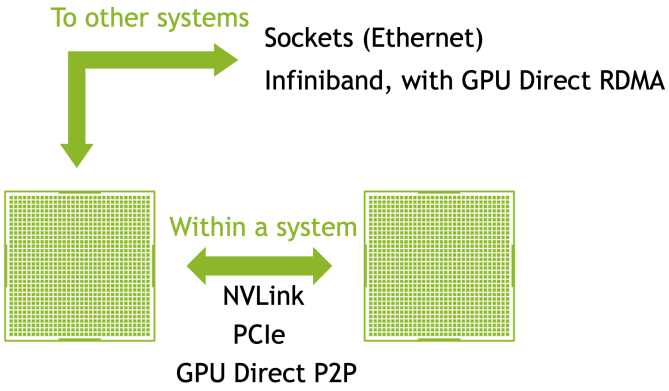


Figure 2.9: Illustration[3]displaying the various interfaces that NCCL can use, both within and between systems (nodes) [12]

Efficient scaling of neural network training is possible with the multi-GPU and multi-node communication provided by NCCL. And we'll be looking at the implementation of IP communication over PCIe using the Dolphin interconnect solutions.

Worth noting is that NCCL closely follows the popular collective API defined by MPI (Message Passing Interface).

[12] NCCL 1.0 introduced Multi-GPU (one node/computer with many GPUs) NCCL 2.0 introduced Multi-node on top of Multi-GPU (Many nodes/computers with many GPUs).

2.4.1 Message Passing Interface (MPI)

The NCCL API and usage are similar to MPI [2], but many minor differences exist. However, NCCL can be easily used in conjunction with MPI. NCCL collectives are similar to MPI collectives. Therefore, creating an NCCL communicator out of an MPI communicator is straightforward. It is, therefore, easy to use MPI for CPU-to-CPU communication and NCCL for GPU-to-GPU communication.

2.4.2 Distributed machine learning techniques in NCCL

Data is moved across all GPUs using a ring model. Communication collectives available as shown in figure 2.10 are: Broadcast. Scatter. Gather. All-Gather. All-to-All. Reduce. All-reduce. For machine learning, All-reduce, broadcast and gather are often used.

[3]In the list of interfaces to other systems, Dolphin interconnect can be added, with IPoPCIe(sockets) and SmartIO. Better support for Direct RDMA might come, and currently exist in the form of an outdated SISCI NCCL plugin [21]

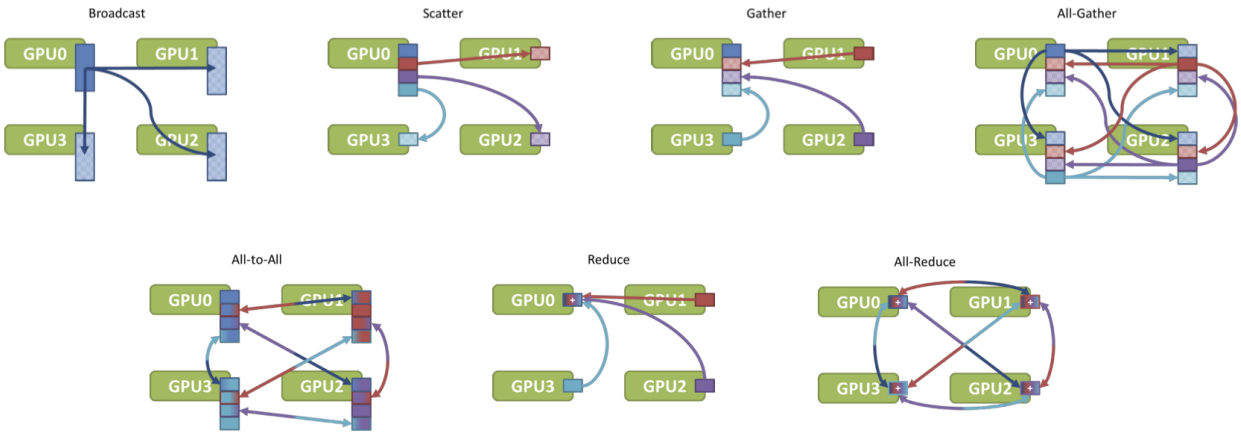


Figure 2.10: Collective communication with multiple senders and/or receivers with NCCL & MPI [29]

For machine learning applications, data can be passed across all GPUs, and they'll perform reductions as data is passed from one GPU onto the next GPU. Machine learning tasks may be split and distributed in various ways using NCCL. With multiple machines, each with multiple GPUs and threads, you may distribute the workload as such: Multiple processes per node. Each process has one GPU. Multiple processes per node, each process with multiple threads, and each thread has one GPU. One process per node, all of the node's GPUs for that process.

2.5 Multimachine machine learning

A single machine has a finite amount of computational resources as it is bound by the capabilities of its processor, memory, and accelerator cards (such as graphics cards). Computational problems such as machine learning, however, may grow so large the amount of data and computational demand surpasses what a single machine can perform or complete in a reasonable time; despite multiple cores and accelerator cards. To overcome this limit and increase the computational resource, we can add multiple machines and connect them together, pooling the resources.

How do we utilize this computational power? It starts with splitting the workload to utilize multiple threads on multi-threaded processors. And further, split to use an accelerator card (such as NVIDIA GPUs when implementing the CUDA API). We can utilize multiple GPUs using NVIDIA Collective Communications Library (NCCL) to further distribute the machine learning workload. At this point, we have a multi-GPU distribution of the workload. NCCL also empowers us to further distribute the workload outside of just one node. Pooling multiple machines, we can do multi-GPU and multi-node distribution of the workload. Thus we have multimachine machine learning.

2.6 Summary

In this chapter, we talked about how PCIe us built and how it enables expansions for use with, e.g. accelerator cards and Network interface controllers. Then we looked at QPI; an internal interconnects for intel CPUs. Then how RDMA enable direct copy operations initiated by PCIe devices instead of the CPU, with a mention of Dolphin interconnect tools that we will use in this thesis. Then we looked at NCCL and how this helps us distribute ML workloads across GPUs within one or more computers. We also mention MPI, the underlying interface NCCL is based on. And we ended talking a little about what it means to do multimachine machine learning.

Chapter 3

System setup and challenges

This chapter briefly explains how we set up our pair of legacy Supermicro GPU servers to run NCCL tests and the challenges we encounter when modern software meets legacy GPUs. For a complete detailed version of how we set up the cluster from scratch, including more fleshed-out terminal outputs, see the appendix A.

3.1 Hardware

We are working with two identical 1U Supermicro x9 SuperServers, type 1027GR-TRF, chassis Model 118-18, Motherboard X9DRG-HF with a NUMA configuration (aka two CPU sockets). With that, we can create a two-node cluster.

We named the nodes *Abel1* and *Abel2* as a homage to the decommissioned Abel cluster at the University of Oslo, where they originated.

They both contain the following:

CPU	2xIntel Xeon E5-2609 @ 2.40GHz
Memory	64GB DDR3 split between the NUMA nodes
GPU	2xNVIDIA Tesla K20X, 6GB GDDR5, 16x PCIe gen2
PCIe NTB interconnect	Dolphin PXH810, 8x PCIe gen3
LAN	Gigabit Ethernet

Table 3.1: Hardware specifications

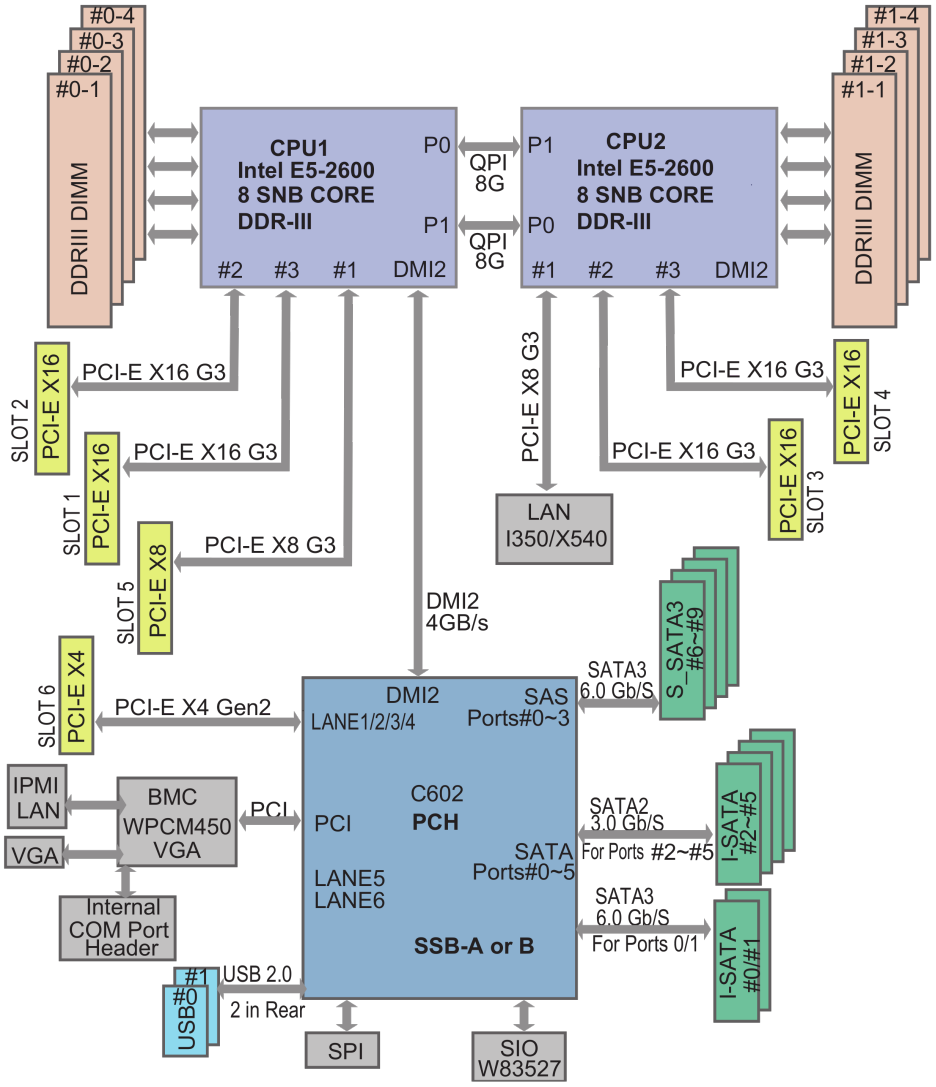


Figure 3.1: Supermicro X9DRG-HF System Block Diagram [11]

The Tesla K20X GPUs are of the Kepler generation, released in November 2012.

The GPUs are arranged on the *Abel1* node so that each GPU is connected to a different CPU (different NUMA node). We can benchmark GPU-to-GPU communication performance in this configuration while crossing the QPI interconnect.

The GPUs are arranged on the *Abel2* node, so both are connected to the same CPU (same NUMA node). Shown in the System Block Diagram 3.1 as `SLOT1` and `SLOT2`. In this configuration, we can benchmark direct P2P communication between the GPUs as they share the same PCIe controller and utilize RDMA.

In both nodes, we have the Dolphin PXH810 card connected to `SLOT5`, the same CPU as the two GPUs in *Abel2*, and one of the GPUs in *Abel1*. The

NUMA location of the PXH810 is crucial, as we later found that SmartIO does not support device lending beyond the same NUMA node. The PXH810 cards on each node are interconnected with a single x8 PCIe cable, as shown in image 3.3.

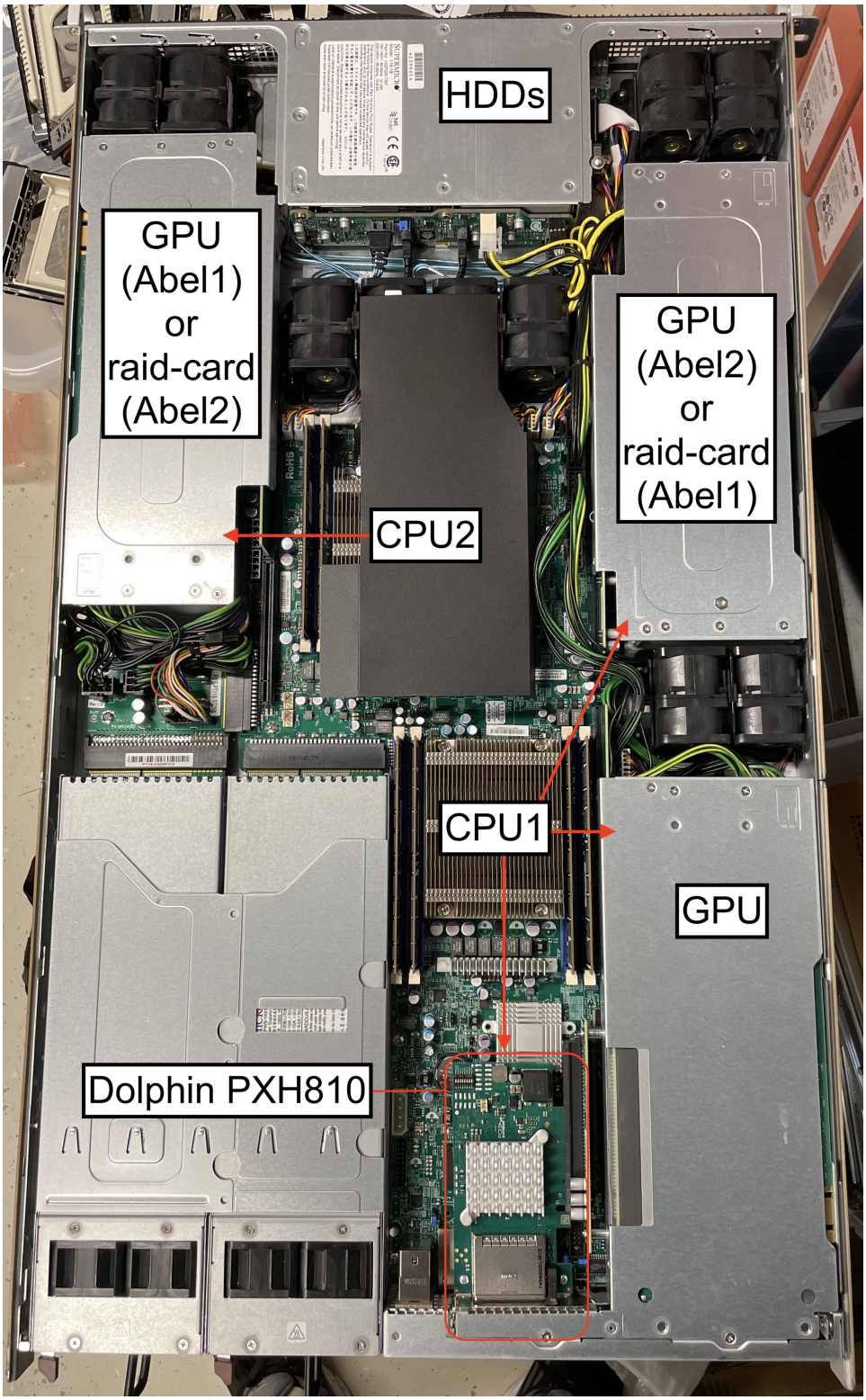


Figure 3.2: Inside view of the Supermicro X9DRG-HF servers

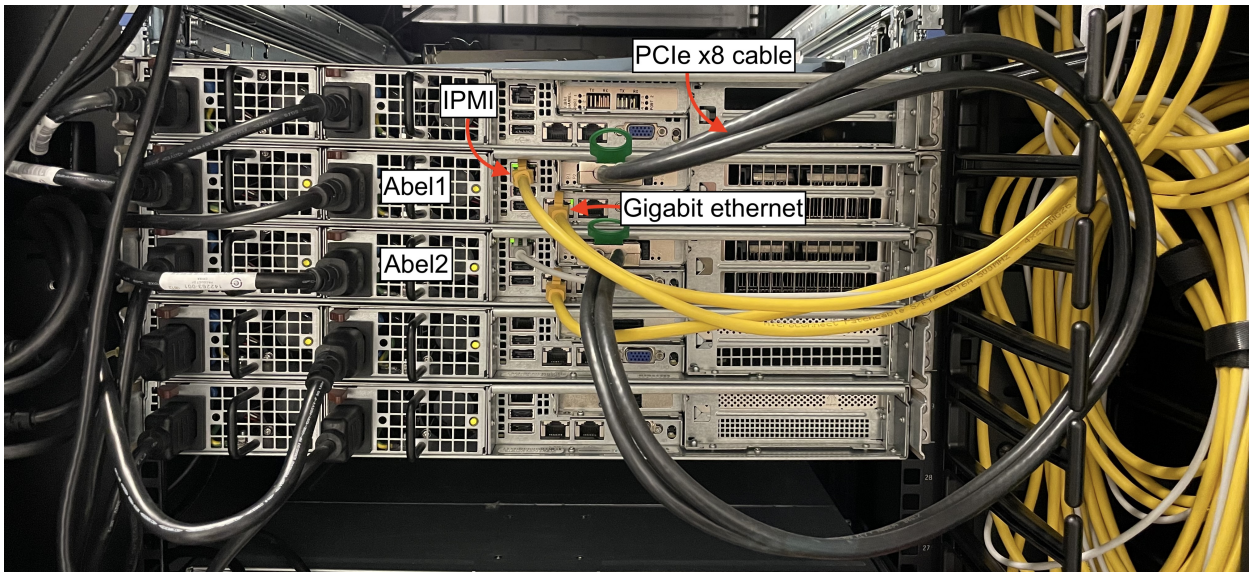


Figure 3.3: Outside view behind the Supermicro X9DRG-HF servers

3.2 Software

The following covers the main parts of how we set up the software in our system and what the most up-to-date software and drivers compatible with our system were at the time of writing.

OS	Ubuntu 20.04 LTS server
GPU	NVIDIA-driver-470-server NVIDIA HPC SDK 21.9 CUDA Version 11.4
PCIe NTB interconnect	Dolphin eXpressWare pipeline 22529
Benchmark tools	iperf3 scibench2 dma_bench nccl-tests, commit 8274cb4 (27 May 2022)

Table 3.2: Software specifications

3.2.1 Operating system

We went for Ubuntu as the operating system by recommendation from our main supervisor. It's a well-supported OS for HPC environments with good GPU support and is easy to download without registration. Another common alternative in HPC is Red Hat Enterprise Linux (RHEL). However, acquiring an RHEL license is a more strict process as they demand user registration for both download and installation.

The initial trial was with Ubuntu server 22.04 LTS. It proved incompatible with the older NVIDIA HPC SDK we needed for our legacy GPUs

(more on this in A.3.8). Compiling compatible version of NVIDIAs NCCL-test code using NVCC for CUDA 11.4 would throw the following error

```
"/usr/include/stdio.h(189): error: attribute __malloc__" does
↪  not take arguments
```

Since stdio.h is distributed as part of the OS, we installed Ubuntu 20.04 LTS on another server to compare against 22.04 LTS. The error disappeared on the older version, so we downgraded both servers to 20.04 LTS. That means hardware and maintenance updates from Ubuntu are only guaranteed until April 2025, and Extended Security Maintenance (ESM) until 2030 at a cost [13].

IOMMU

In Ubuntu, IOMMU is not enabled by default. On our system, we enabled it by modifying `/etc/default/grub` to add

```
GRUB_CMDLINE_LINUX_DEFAULT="intel_iommu=on"
```

then enable the change to take effect after the next reboot with

```
$ sudo update-grub
```

3.2.2 NVIDIA drivers & tools

Tesla K20X is currently on the R470 Long Term Support Branch (LTSB) for NVIDIA drivers. That means NVIDIA has set the End of Life to July 2024, on which they will stop supplying bug and security releases.

We installed the drivers, diagnostic tools, and other tools needed as such

```
# apt install nvidia-headless-470-server
# apt install nvidia-utils-470-server
# apt install environment-modules
```

We need CUDA, NVCC, NCCL, and MPI to perform our benchmarks. NVIDIAs HPC SDK includes all this as well as environment modules that make setting it up across nodes consistent and stable. The latest working SDK version for our Tesla K20X GPUs is 21.9, with CUDA 11.4. Instructions for adding and downloading their repository were found on the NVIDIA HPC SDK 21.9 site. Installation was done as follows:

```
$ curl https://developer.download.nvidia.com/hpc-sdk/ubuntu/⌋
↪  DEB-GPG-KEY-NVIDIA-HPC-SDK | sudo gpg --dearmor -o
↪  /usr/share/keyrings/nvidia-hpcsdk-archive-keyring.gpg
$ echo 'deb [signed-by=/usr/share/keyrings/⌋
↪  nvidia-hpcsdk-archive-keyring.gpg]
↪  https://developer.download.nvidia.com/hpc-sdk/ubuntu/amd64
↪  /' | sudo tee /etc/apt/sources.list.d/nvhpc.list
```

```
$ sudo apt update -y
$ sudo apt install -y nvhpc-21-9
```

After the installation, we load in the environment module as such

```
$ module use /opt/nvidia/hpc_sdk/modulefiles
$ module load nvhpc/21.9
```

This way, the location of the various SDK libraries, such as CUDA, NCCL, and MPI, are added to the `LD_LIBRARY_PATH` for all software to find. As well as relevant programs such as NVCC and mpirun are added to PATH, we can call and run them without prompting the executable's path.

A downside of using NVIDIAs HPC SDK is that the CUDA, NCCL, and MPI libraries locations are not in the default location that many CUDA programs expect. We experienced that makefiles tend to be configured with hard-coded CUDA paths. Even in NVIDIAs own code, such as `CUDA samples` and `NCCL-tests`. They look for the CUDA library in `/usr/local/cuda`, while the HPC SDK 21.9 puts it at `/opt/nvidia/hpc_sdk/Linux_x86_64/21.9/cuda/`. This generally means all non-standard locations must be explicitly defined as an argument during compilation.

As an example, we tested various CUDA capabilities using CUDA samples, part of the CUDA toolkit. The various code samples include makefiles for easy building and running. On our setup, we must pass through arguments for the non-standard compiler and library locations as well as compute architecture. Example:

```
$ make NVCC=/opt/nvidia/hpc_sdk/Linux_x86_64/21.9/compilers/
↪ bin/nvcc
↪ CUDA_PATH=/opt/nvidia/hpc_sdk/Linux_x86_64/21.9/cuda/11.4/
↪ SMS="35"
```

Complications

The K20X GPUs support most CUDA capabilities 3.5 and CUDA 11.4. Initially, we tried to use the newest SDK, 22.7, and then later, 22.9, as it was released. Specifically the multipack version, including CUDA 10.2, 11.0, and 11.7. However, they are missing environment module files for anything but CUDA 11.7. So when we did

```
$ module load nvhpc/22.9
```

We'd get nvcc for CUDA 11.7

This should theoretically be fine, as we can point the compiler to the relevant CUDA library during compilation. However, when we loaded module nvhpc/22.9, and tried to compile NCCL-test while pointing to CUDA 11.0 included in 22.9, we'd get pgc++ errors

```
$ make
 ↪  CUDA_HOME=/opt/nvidia/hpc_sdk/Linux_x86_64/22.9/cuda/11.0/
 ↪  NCCL_HOME=/opt/nvidia/hpc_sdk/Linux_x86_64/22.9/
 ↪  comm_libs/nccl
 ↪  NVCC_GENCODE="-gencode=arch=compute_35,code=sm_35"
...
"/opt/nvidia/hpc_sdk/Linux_x86_64/22.9/cuda/11.0//bin/../
 ↪  targets/x86_64-linux/include/crt/host_config.h", line 118:
 ↪  catastrophic error: #error directive: -- unsupported pgc++
 ↪  configuration! Only pgc++ 18, 19 and 20 are supported!
...
```

Even if we try pointing to the known working CUDA 11.4 from 21.9 while using nvcc loaded from 22.9, We'd get the following

```
$ make
 ↪  CUDA_HOME=/opt/nvidia/hpc_sdk/Linux_x86_64/21.9/cuda/11.4/
 ↪  NCCL_HOME=/opt/nvidia/hpc_sdk/Linux_x86_64/21.9/
 ↪  comm_libs/nccl
 ↪  NVCC_GENCODE="-gencode=arch=compute_35,code=sm_35"
...
"/opt/nvidia/hpc_sdk/Linux_x86_64/21.9/cuda/11.4//bin/../
 ↪  targets/x86_64-linux/include/crt/host_config.h", line 118:
 ↪  catastrophic error: #error directive: -- unsupported pgc++
 ↪  configuration! Only pgc++ 18, 19, 20 and 21 are supported!
 ↪  The nvcc flag '-allow-unsupported-compiler' can be used to
 ↪  override this version check; however, using an unsupported
 ↪  host compiler may cause compilation failure or incorrect
 ↪  run time execution. Use at your own risk.
...
```

If we use SDK 21.9, we don't get pgc++ errors. With this, we see newer NVIDIA C compilers have already removed compiler support for our Tesla K20X. Nor renamed `pgc++` to `nvc++` despite NVIDIA rebranding and integrating PGI compilers into the Nvidia HPC SDK [28]. The compiler in both SDK 22.9 and 21.9 include a warning about deprecated architecture:

```
nvcc warning : The 'compute_35', 'compute_37', 'compute_50',
'sm_35', 'sm_37' and 'sm_50' architectures are deprecated,
and may be removed in a future release
```

informing that our Tesla K20X is living on borrowed time.

3.2.3 NCCL Tests

The code to benchmark NCCL performance and correctness is distributed by NVIDIA and called NCCL Tests. In order to compile to our legacy

GPUs using NVCC for CUDA 11.4, we found the newest release was incompatible.

The following is an example of errors that arrive when trying to compile the latest NCCL test version (commit 365b92a as of writing) with NVCC from HPC SDK 21.9 (`nvcc release 11.4, V11.4.100`).

```
$ make
 ↪  CUDA_HOME=/opt/nvidia/hpc_sdk/Linux_x86_64/21.9/cuda/11.4/
 ↪  NCCL_HOME=/opt/nvidia/hpc_sdk/Linux_x86_64/21.9/comm_libs/nccl
 ↪  NVCC_GENCODE="-gencode=arch=compute_35,code=sm_35"
...
../verifiable/verifiable.cu(124): warning: function
 ↪  "<unnamed>::castTo<Y>(float) [with Y=__nv_bfloat16]" was
 ↪  declared but never referenced

../verifiable/verifiable.cu(119): warning: function
 ↪  "<unnamed>::castTo<Y>(float) [with Y=half]" was declared
 ↪  but never referenced

../verifiable/verifiable.cu(147): warning: function
 ↪  "<unnamed>::ReduceSum::operator()(half, half) const" was
 ↪  declared but never referenced

../verifiable/verifiable.cu(155): warning: function
 ↪  "<unnamed>::ReduceSum::operator()(__nv_bfloat16,
 ↪  __nv_bfloat16) const" was declared but never referenced

"../verifiable/verifiable.cu", line 353: error: expected a ")"
  return (uint64_t)((((unsigned __int128)a) * ((unsigned
 ↪  __int128)b)) >> 64);
                                ^

"../verifiable/verifiable.cu", line 353: error: expected a ")"
  return (uint64_t)((((unsigned __int128)a) * ((unsigned
 ↪  __int128)b)) >> 64);
                                                        ^

"../verifiable/verifiable.cu", line 353: warning: shift count
 ↪  is too large
  return (uint64_t)((((unsigned __int128)a) * ((unsigned
 ↪  __int128)b)) >> 64);

 ↪  ^

2 errors detected in the compilation of
 ↪  "/tmp/tmpxft_00001276_00000000-6_verifiable.cudafe1.cpp".
```

```
...
```

The code includes newer commands and changes that would not compile on the slightly older NVIDIA C compiler. To circumvent the issue, we utilized the git history. The newest verified git-release that compiled and worked on our system was git-commit 8274cb4 (27 May 2022), so we went ahead and used that as such:

```
$ git checkout 8274cb4
```

This means that bug fixes in the benchmark tool and testing of new features in NCCL beyond git-commit 8274cb4 have become unavailable for Tesla K20X users unless backward compatibility in the code is addressed in a future commit.

We also found in the makefile that NVIDIA added a check for CUDA capability to define NVCC_GENCODE to reduce compile time. We see in the git history that after the release of CUDA 11, the makefile was modified to check the CUDA version of the NVCC compiler. If it is CUDA 11, it will only compile to GPUs with CUDA capability 6.0 and newer:

```
ifeq ($(shell test "0$(CUDA_MAJOR)" -ge 11; echo $$?),0)
NVCC_GENCODE ?= -gencode=arch=compute_60,code=sm_60 \
                -gencode=arch=compute_61,code=sm_61 \
                -gencode=arch=compute_70,code=sm_70 \
                -gencode=arch=compute_80,code=sm_80 \
                -gencode=arch=compute_80,code=compute_80
else
NVCC_GENCODE ?= -gencode=arch=compute_35,code=sm_35 \
                -gencode=arch=compute_50,code=sm_50 \
                -gencode=arch=compute_60,code=sm_60 \
                -gencode=arch=compute_61,code=sm_61 \
                -gencode=arch=compute_70,code=sm_70 \
                -gencode=arch=compute_70,code=compute_70
endif
```

Our GPUs were at CUDA capability 3.5, so in order to compile the code using CUDA 11, we could either modify the makefile or manually override by adding the compiler argument:

```
NVCC_GENCODE="-gencode=arch=compute_35,code=sm_35"
```

We chose to manually override.

Thus to compile the NCCL test with MPI for multi-node testing using the NVIDIA HPC SDK, we ran the following

```
$ make MPI=1 MPI_HOME=/opt/nvidia/hpc_sdk/Linux_x86_64/21.9/
↪  comm_libs/mpi
↪  CUDA_HOME=/opt/nvidia/hpc_sdk/Linux_x86_64/21.9/cuda/11.4/
↪  NCCL_HOME=/opt/nvidia/hpc_sdk/Linux_x86_64/21.9/
↪  comm_libs/nccl
↪  NVCC_GENCODE="-gencode=arch=compute_35,code=sm_35"
```

To speed up the compilation time, `-j8` can be added to compile using all eight threads in our system.

Compiling with MPI also means we have to run the program with mpirun. To run two MPI processes on a single node:

```
$ mpirun -np 2 -host abel1:2
↪  ~/cudastuff/nccl-tests/build/all_reduce_perf -b 8 -e 128M
↪  -f 2 -g 1
```

To run four MPI processes across our two nodes

```
$ mpirun -np 4 -host abel1:2,abel2:2 -x LD_LIBRARY_PATH
↪  ~/cudastuff/nccl-tests/build/all_reduce_perf -b 8 -e 128M
↪  -f 2 -g 1
```

`-np` are number of MPI processes.
`-x` export our specified environment variables to the remote nodes before executing the program. In this case `"LD_LIBRARY_PATH"`. This is critical since MPI SSH onto each node in a non-interactive way. Moreover, it means the HPC SDK environment modules will not be loaded; thus, the program will not find NCCL or CUDA in `"LD_LIBRARY_PATH"` on the remote nodes. This method only works if the libraries are installed at identical locations on all nodes.

Each of our nodes has only two GPUs. Therefore, each node can run two MPI processes, each having one GPU: `"-np 2 -g 1"`, or one MPI process having two GPUs: `"np 1 -g 2"`. We tested both scenarios and saw no difference in performance in our system. In theory, the NUMA-locality of MPI processes can be relevant for performance on systems like Abel1, where the GPUs are connected to separate CPUs. So of the two, we chose to stick with two MPI processes per node, each having one GPU.

3.2.4 Dolphin eXpressWare

Driver and software for the Dolphin PXH810 were supplied to us by Dolphin in a package called eXpressWare.

We installed and enabled features like SmartIO and SuperSockets (needed for IPoPCIe) as such:

```
$ sudo bash
 ↪  Dolphin_eXpressWare-Linux-x86_64-PX-66aa356545_c0e0d090cc.
 ↪  ubuntu20.04.sh --disable-gui --enable-smartio
 ↪  --enable-supersockets

Explanation:
--disable-gui           #Graphical interface. Disabled since
 ↪  the server is headless.
--enable-smartio        #Enables smartIO functionality.
--enable-supersockets   #Enables SuperSockets and IPoPCIe.
```

We noticed quickly that the Dolphin driver is closely tethered to the kernel version in the OS and would break if the kernel were updated. The solution we received from Dolphin was to rerun the install script and build against the new kernel after each update—alternatively, block kernel updates in the OS. By default, Ubuntu automatically installs available kernel and security updates during reboot. For security reasons, we chose not to freeze the kernel version and instead reran the install script when it happened.

3.2.5 IPoPCIe

With Dolphin eXpressWare installed and supersockets enabled, as shown in 3.2.4, a new network interface called `dis0` will appear for the operating system. With netplan, we assign static IP addresses for both ethernet and dis0. Then made hostnames for a more human-readable experience instead of using IPs. Examples are shown in A.3.6.

3.2.6 SmartIO

After installing the necessary drivers and frameworks from Dolphin eXpressWare as shown in 3.2.4, we start by running the Dolphin tool `dis_config` to find the prefetch size on the PXH810 card. The default prefetch size is shown as 512MB. The prefetch size for the GPUs can be found running `lspci -vs [pci device ID]`, and we observe that one Tesla K20X needs 256M prefetched memory space. In theory, with a 512MB prefetch set for the PXH810 card, we will have just enough for two GPUs. However, when we tried device lending two GPUs using SmartIO, the second GPU failed to be lent. A look into `dmesg` revealed:

```
BAR X: no space for [mem size 0x10000000 64bit pref]
BAR X: failed to assign [mem size 0x10000000 64bit pref]
```

A look into the SmartIO manual informs that, quote:

> The sum of PCIe BAR sizes + natural alignment for all added devices must be smaller than the prefetch space allocated by the Dolphin NTB board.

To give us some ample headroom, we increase the PX cards prefetch allocation to 4096MB. This would, for older systems, be the limit. However, our systems could also handle more if we wanted, as it supports "Above 4G Decoding". That enables 64-bit capable devices to be decoded in the Above 4G Address Space. An example of that is newer GPUs that support resizable BAR (Base Address Register).

Now we can use SmartIO to borrow GPUs between the nodes. First, we check with `nvidia-smi` that each of our two nodes contains two GPUs. abel1:

```
abel1$ nvidia-smi
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.182.03   Driver Version: 470.182.03   CUDA Version: 11.4     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla K20Xm         Off  | 00000000:05:00.0 Off |                    0 |
| N/A   32C    P0    55W / 235W |      0MiB /  5700MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   1  Tesla K20Xm         Off  | 00000000:85:00.0 Off |                    0 |
| N/A   29C    P0    58W / 235W |      0MiB /  5700MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
```

abel2:

```
abel2$ nvidia-smi
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.182.03   Driver Version: 470.182.03   CUDA Version: 11.4     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla K20Xm         Off  | 00000000:04:00.0 Off |                    0 |
| N/A   26C    P0    55W / 235W |      0MiB /  5700MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   1  Tesla K20Xm         Off  | 00000000:05:00.0 Off |                    0 |
| N/A   31C    P0    57W / 235W |      0MiB /  5700MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
```

Then we do the following, slowly, one by one, to borrow and lend two GPUs from abel2 to abel1.

```
// Lender side, if it's abel2:
// Connect to abel1
$ sudo smartio_tool connect 4
// Get PCI-addresses to the GPUs
$ lspci | grep NVIDIA
// Add them to the lender list
$ sudo smartio_tool add 04:00.0
$ sudo smartio_tool add 05:00.0
// Make them available for borrowers
```

```
$ sudo smartio_tool available 04:00.0
$ sudo smartio_tool available 05:00.0

// Borrower side:
// We begin by stopping the sisci service
$ sudo systemctl stop dis_sisci
// Then we list available GPUs to borrow
$ sudo smartio_tool list
// We borrow the two GPUs with their ID and the DMA window
 ↪  size
$ sudo smartio_tool borrow 80400 1024
$ sudo smartio_tool borrow 80500 1024
// Then we enable p2p between the remote GPUs so they locally
 ↪  can talk directly with each other
$ sudo smartio_tool enable-p2p 80400 80500
$ sudo smartio_tool enable-p2p 80500 80400
// Now for us to use the remote GPUs, we must reload the
 ↪  NVIDIA kernel module. We must unload in the order of its
 ↪  dependencies as seen by running "$ lsmod | grep nvidia"
$ sudo modprobe --remove nvidia_uvm nvidia_drm nvidia_modeset
 ↪  nvidia
// Then reload it back in again. Dependencies will follow
 ↪  along
$ sudo modprobe nvidia
// And finally, start the sisci service again
$ sudo systemctl start dis_sisci
// To confirm that the lending was sucessfull, we check that
 ↪  the remote GPUs are listed
$ nvidia-smi
```

Abel1 now displays four GPUs:

```
abel1:~$ nvidia-smi
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.182.03   Driver Version: 470.182.03   CUDA Version: 11.4     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla K20Xm         Off  | 00000000:03:04.0 Off |                    0 |
| N/A   28C    P0    54W / 235W |      0MiB /  5700MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   1  Tesla K20Xm         Off  | 00000000:03:05.0 Off |                    0 |
| N/A   33C    P0    56W / 235W |      0MiB /  5700MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   2  Tesla K20Xm         Off  | 00000000:05:00.0 Off |                    0 |
| N/A   32C    P0    56W / 235W |      0MiB /  5700MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   3  Tesla K20Xm         Off  | 00000000:85:00.0 Off |                    0 |
| N/A   29C    P0    57W / 235W |      0MiB /  5700MiB |      0%      Default |
|                               |                      |                  N/A |
```

```
+-----------------------------+---------------------+---------------------+
```

SmartIO complications

If the series of smartio_tool commands for lender and borrower are executed too rapidly, for example, by pasting a block of them into the terminal, lending may appear to have worked. `nvidia-smi` may show external GPUs as expected. However, CUDA-programs that try to run on them will quickly fail. `p2pBandwidthLatencyTest` from CUDA-samples will return:

```
Cuda failure p2pBandwidthLatencyTest.cu:610: 'unknown error'
```

NCCL tests will report 'unknown error' and segmentation faults:

```
...
abel1: Test CUDA failure common.cu:1045 'unknown error'
 .. abel1 pid 3326: Test failure common.cu:1007
abel1: Test CUDA failure common.cu:1045 'unknown error'
 .. abel1 pid 3325: Test failure common.cu:1007
...
[abel1:03374] *** Process received signal ***
[abel1:03374] Signal: Segmentation fault (11)
[abel1:03374] Signal code: Address not mapped (1)
[abel1:03374] Failing at address: 0x30
[abel1:03374] *** End of error message ***
/opt/nvidia/hpc_sdk/Linux_x86_64/21.9/comm_libs/mpi/bin/
↪  mpirun: line 15:  3374 Segmentation fault      (core
↪  dumped) $MY_DIR/.bin/$EXE "$@"
```

Our "solution" was to wait for a second or two between executing each `smartio_tool` command. We suspect `smartio_tool` is exposed to race conditions.

We initially believed that the NVIDIA kernel module on the lender side needed to be unloaded before `smartio_tool` took over GPUs for lending. However, doing so would cause the server to freeze once a GPU is added to SmartIO before making it available for borrowers.

```
// Freeze reproduction #1:
$ sudo modprobe --remove nvidia_uvm nvidia_drm nvidia_modeset
↪  nvidia
$ sudo smartio_tool add 05:00.0
(server freezes here)

// Freeze reproduction #2:
$ sudo smartio_tool add 05:00.0
```

```
$ sudo modprobe --remove nvidia_uvm nvidia_drm nvidia_modeset
 ↪  nvidia
$ sudo smartio_tool remove 05:00.0
$ sudo smartio_tool add 05:00.0
(server freezes here)
```

We circumvented the problem by not unloading the NVIDIA kernel module on the lender side, as it appeared not to cause any issues leaving it. We are unsure if this is expected behavior and what the connection is between the lack of an NVIDIA kernel module and adding a GPU device `smartio_tool` that causes the operating system to freeze.

Chapter 4

Experiments and results

4.1 Base-line reference

Table 4.1: Peak base-line performance

Interface	Tool	Protocol	Bandwidth
Gigabit Ethernet	iperf3	UDP/TCP	112 MBytes/sec
PXH810 IPoPCIe	iperf3	UDP/TCP	857 MBytes/sec
PXH810 PIO	scibench2	none	4134 MBytes/sec
PXH810 DMA	dma_bench	none	5449 MBytes/sec

With `iPerf3`, we ran both UDP and TCP benchmarks and received the same result, as shown in table 4.1.

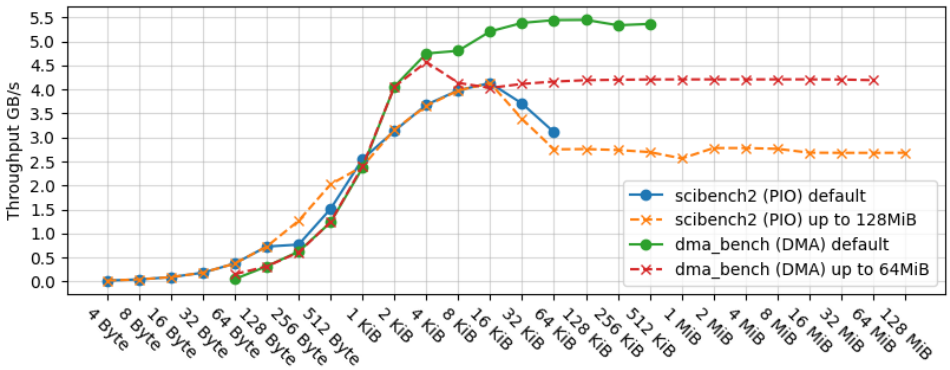


Figure 4.1: Low-level benchmark of the PXH810, displaying how different segment sizes affect performance.

For unknown reasons `scibench2` and `dma_bench` return different performance numbers when increasing the test area of segment sizes beyond the default. We showed this to the vendor of the software (Dolphin), but we couldn't quickly find the cause. Therefore, we have included all results.

The reason the graphs in 4.1 go up to at most 128MiB is because `scibench2` and `dma_bench` return the following error if we choose larger

segment sizes:

```
SCICreateSegment failed: Out of local resources (0x40000904)
```

The reason and solution can be found in the Dolphin eXpressWare Installation and Reference Guide, quote [19]:

> `SCICreateSegment()` will fail if the system can't allocate a large enough physical contiguous memory. The ability to do this will be reduced over time as the physical memory will often be fragmented.
>
> To overcome the problem caused by memory fragmentation, the eXpressWare software supports "Memory preallocation" to allow the driver to allocate the required memory during the initial boot and driver load.

We chose not to do memory preallocation in case it could invalidate our existing benchmark results.

4.2 NCCL

We have chosen four collective communications to benchmark, Broadcast, All-Reduce, All-to-All, and All-Gather.
The graphs include three data types, explained from NCCL tests documentation as such: [3]:

Algorithm bandwidth:

> The most commonly used formula for bandwidth : `size (S) / time (t)`. It is useful to compute how much time any large operation would take by simply dividing the size of the operation by the algorithm bandwidth.
>
> ```
> algbw = S/t
> ```

Bus bandwidth:

> Applying a formula to the algorithm bandwidth to reflect the speed of the inter-GPU communication. Using this bus bandwidth, we can compare it with the hardware peak bandwidth, independently of the number of ranks used.

Time:

> To measure the constant overhead (or latency) associated with operations. On large sizes, the time becomes linear with the size (since it is roughly equal to overhead + size / bw) and is no longer measuring the latency but also the bandwidth multiplied by the size.

We ran the following default setup of NCCL-tests, with parallel init enabled [3]:
5 warmup iterations, then 20 iterations.
Number of operations to aggregate together in each iteration is 1.
Reduce operation is sum.
Datatype is Float.
Reported performance is the average across all ranks (GPUs)

4.2.1 Broadcast

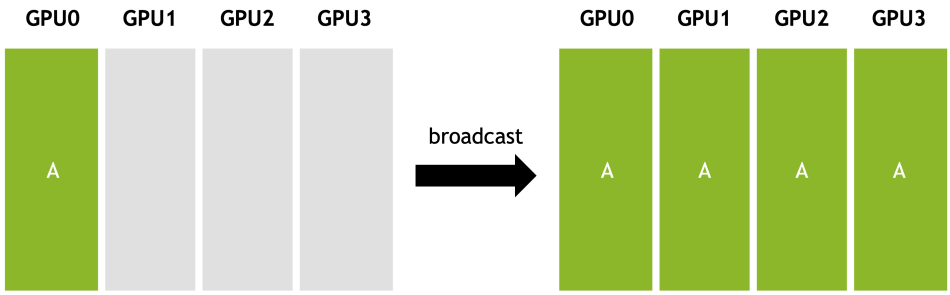
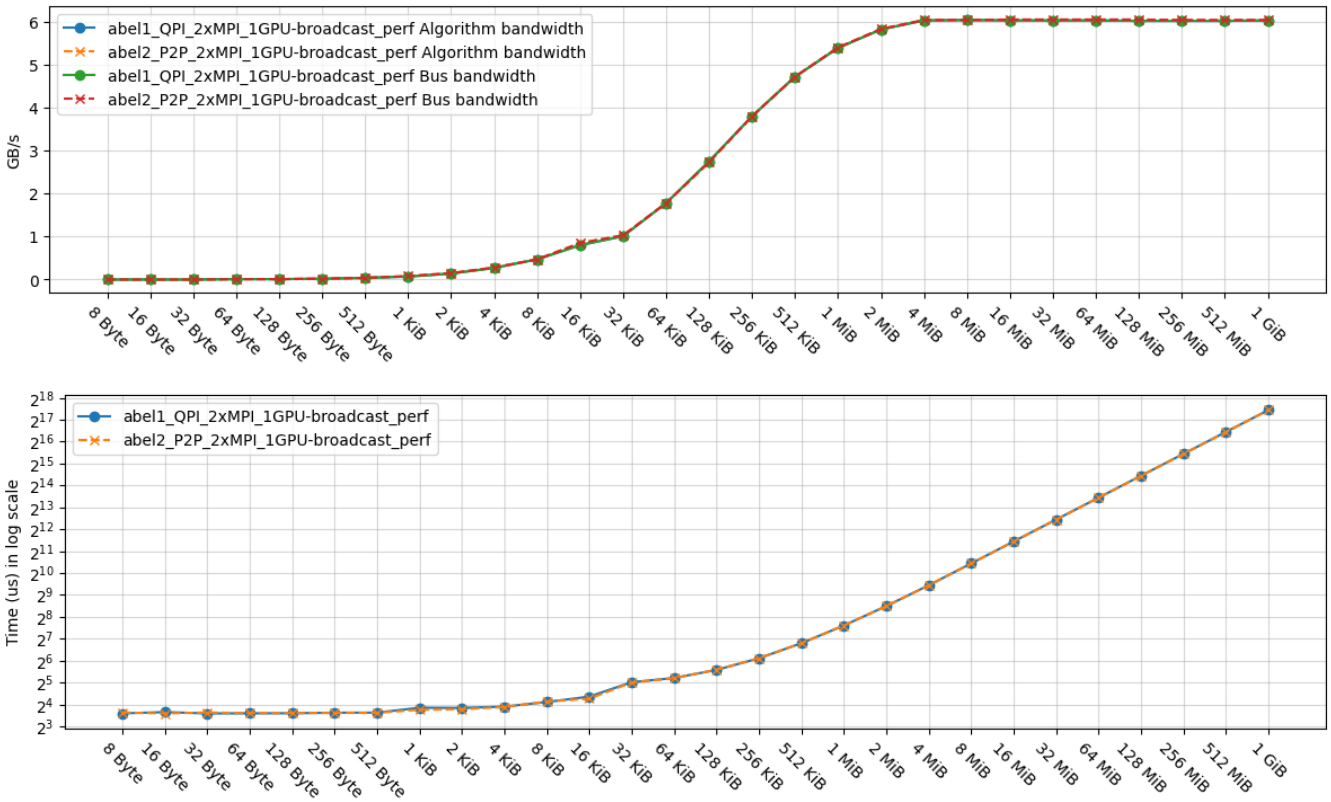


Figure 4.2: Broadcast illustration [29]

Broadcast is a one-to-all transfer, and it is used in, e.g. Deep Learning [29].

QPI vs P2P

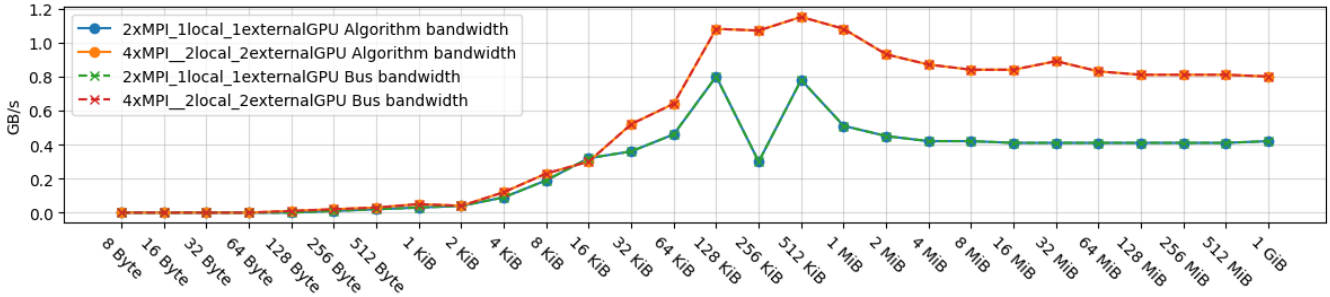
Figure 4.3: *Broadcast performance with GPUs over QPI (abel1), vs direct P2P (abel2)*



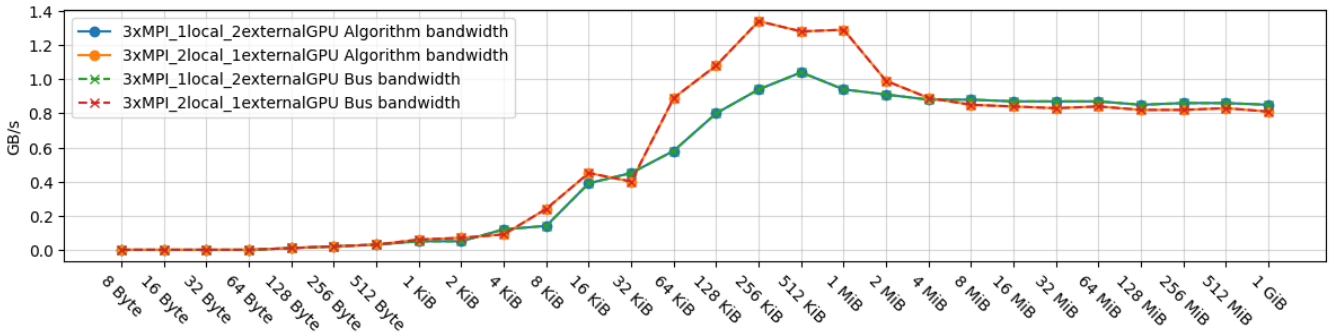
Gigabit Ethernet

Figure 4.4: *Broadcast performance over gigabit Ethernet*

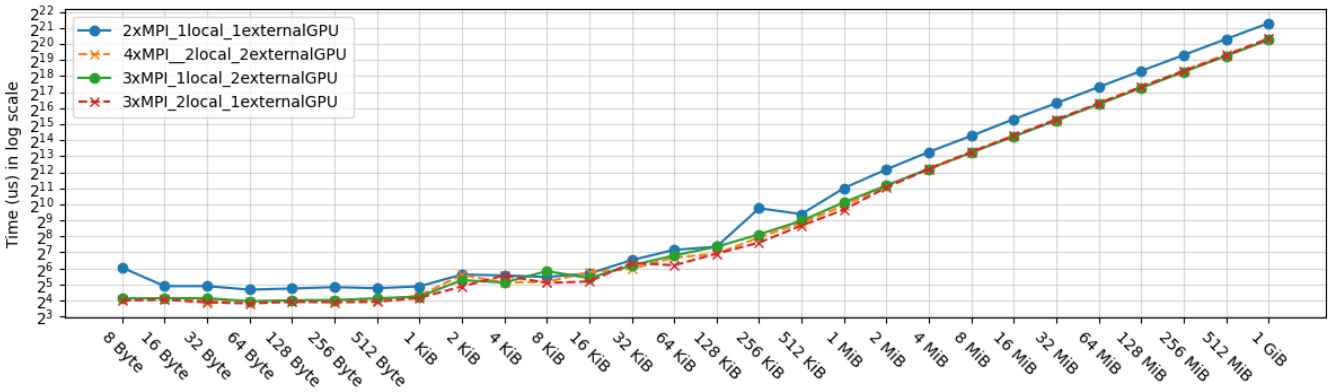
(a) *Difference running one vs two GPUs per node*



(b) *Difference in running two of three GPUs on local, vs remote node*



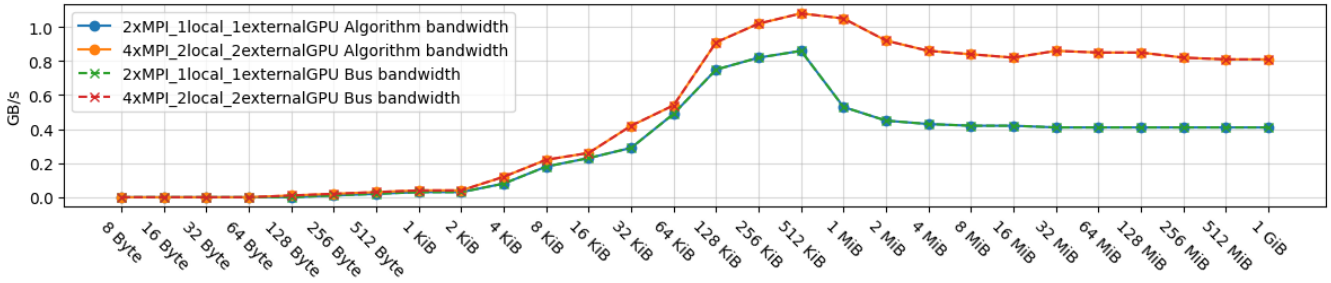
(c) *Time*



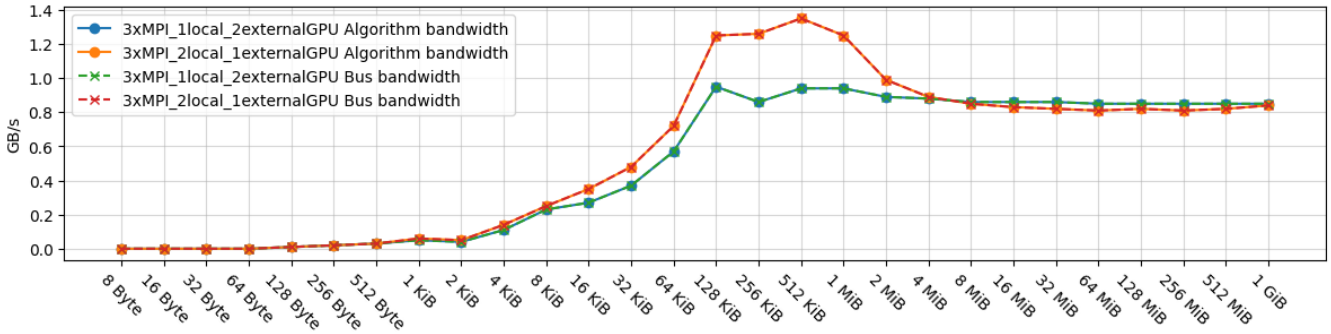
IPoPCIe

Figure 4.5: *Broadcast performance over IPoPCIe*

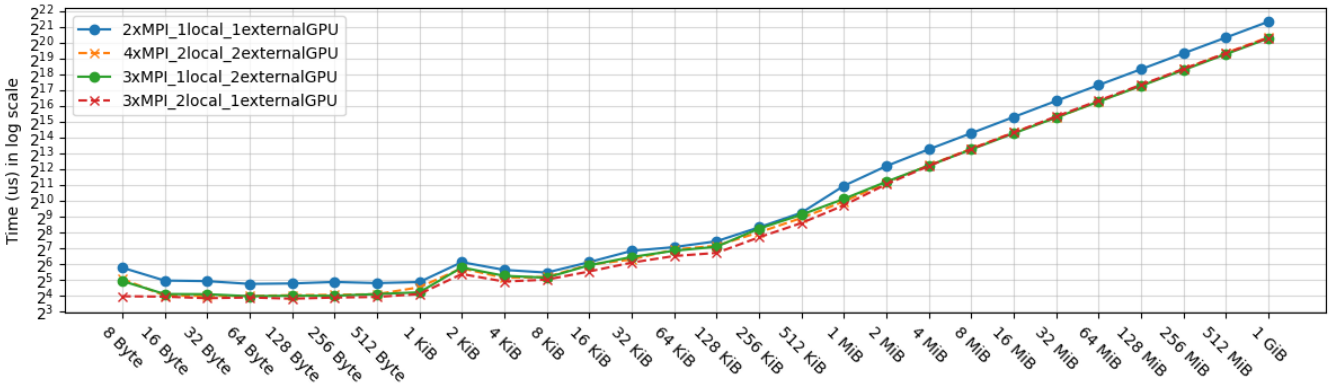
(a) *Difference running one vs two GPUs per node*



(b) *Difference in running two of three GPUs on local, vs remote node*

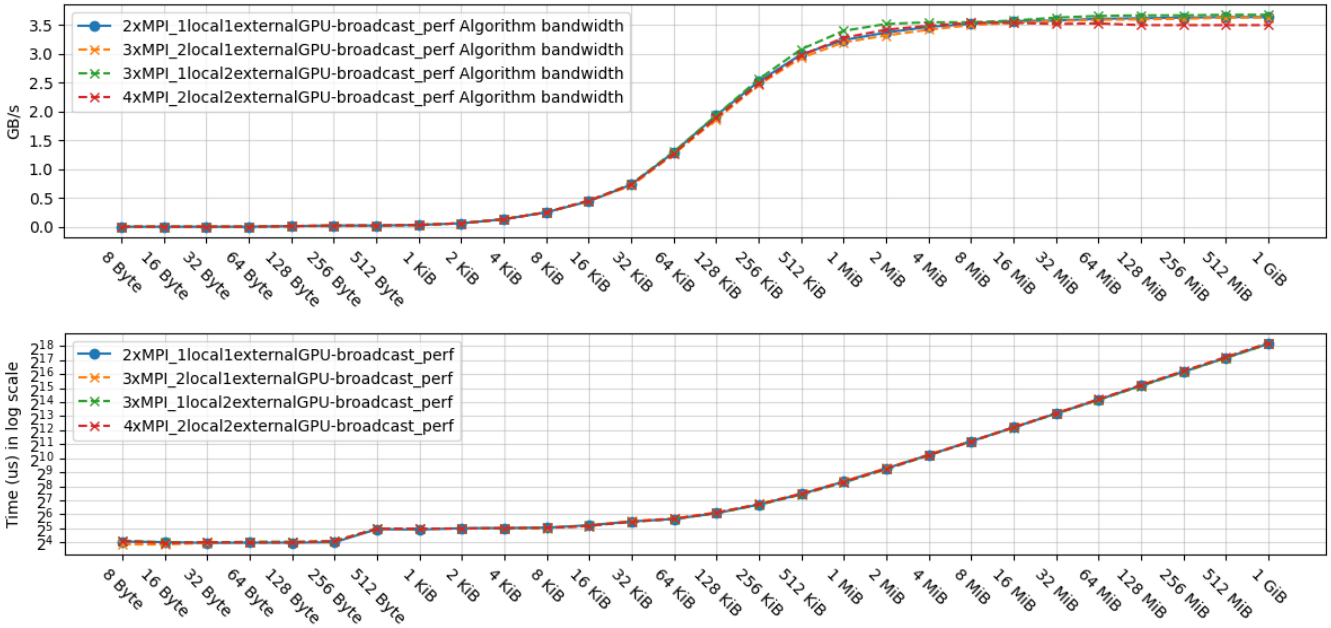


(c) *Time*



SmartIO

Figure 4.6: *Broadcast performance using SmartIO*



The graph is, unfortunately, missing bus bandwidth. Bus bandwidth was, in this case, identical to the algorithm bandwidth.

Stacked summary (multi-node)

Figure 4.7: *Comparing multi-node interconnects*
2 GPU broadcast performance, 1 in each node

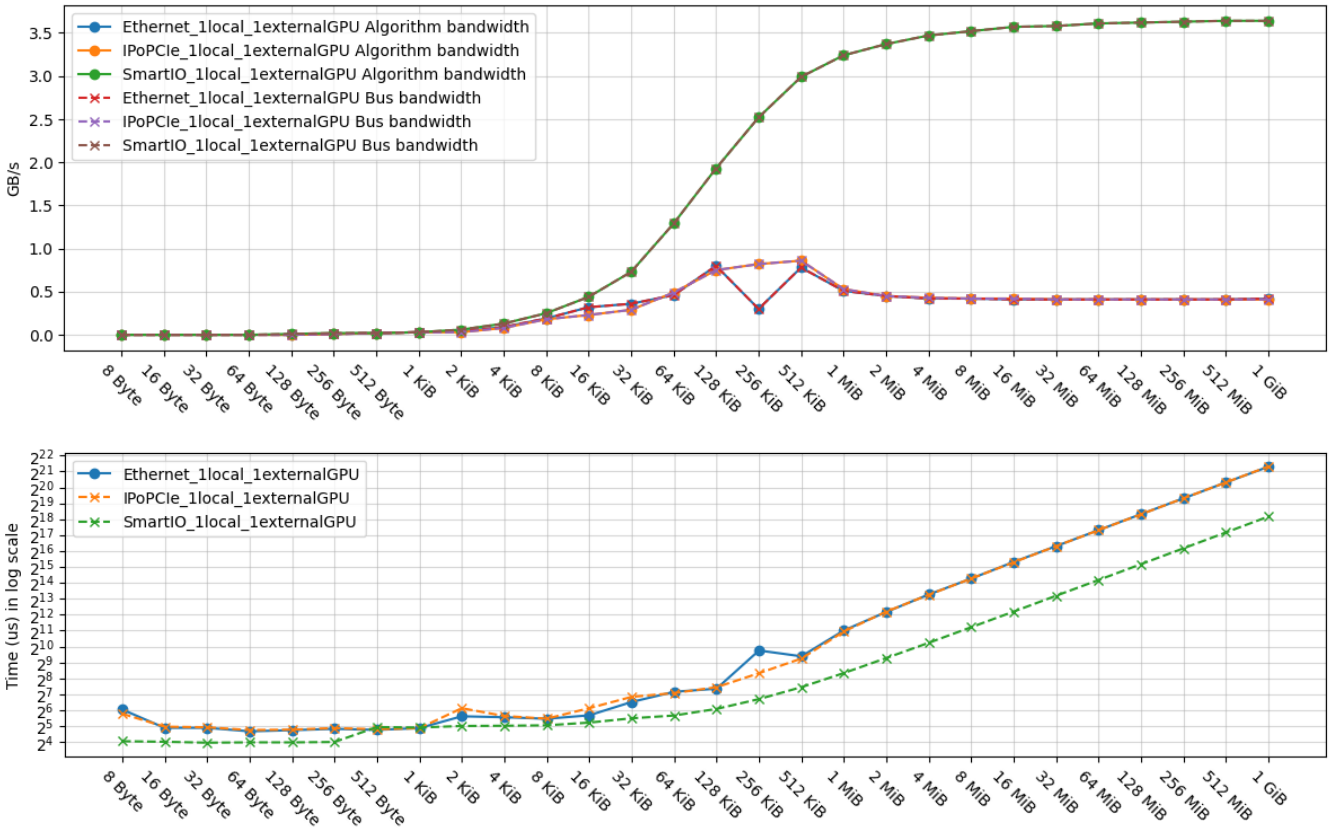
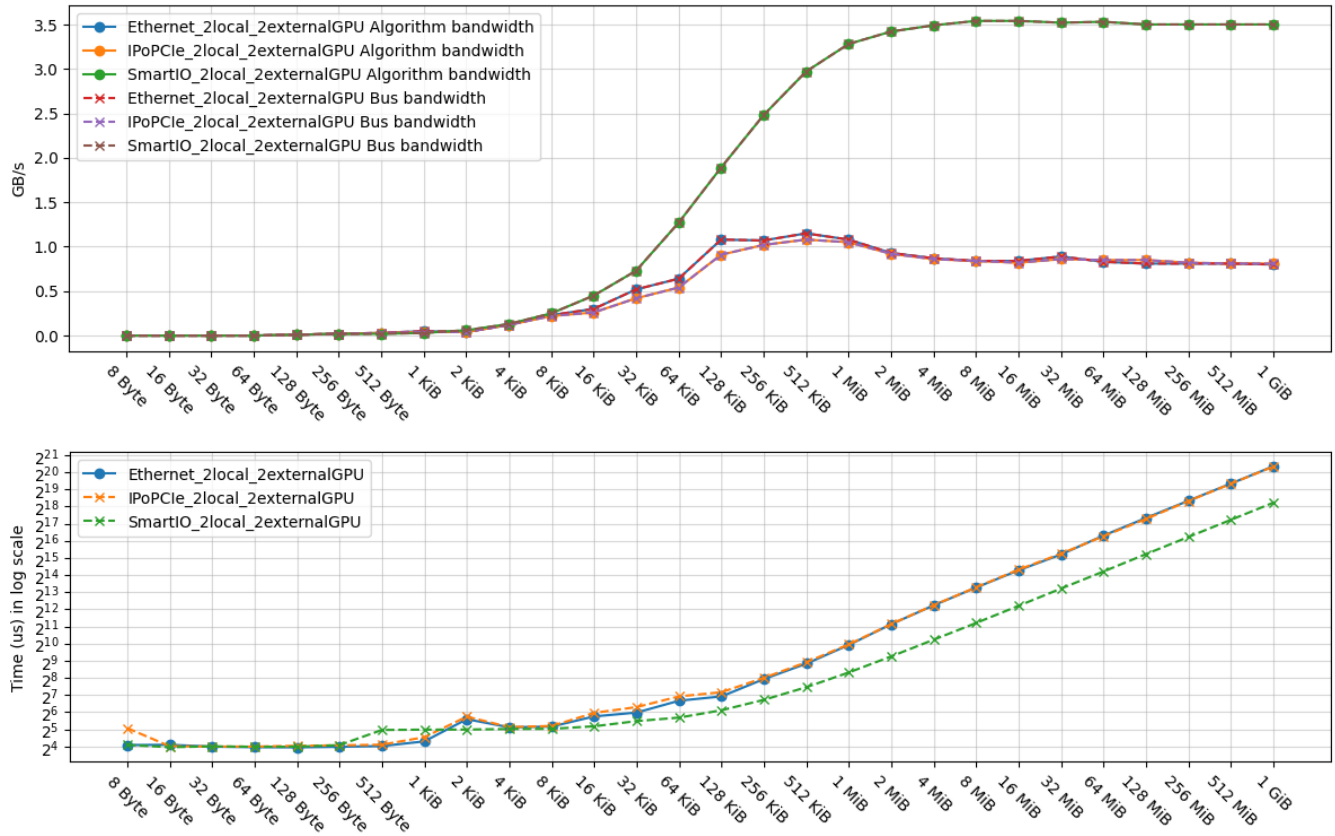


Figure 4.8: *Comparing multi-node interconnects*
4 GPU broadcast performance, 2 in each node



4.2.2 All-Reduce

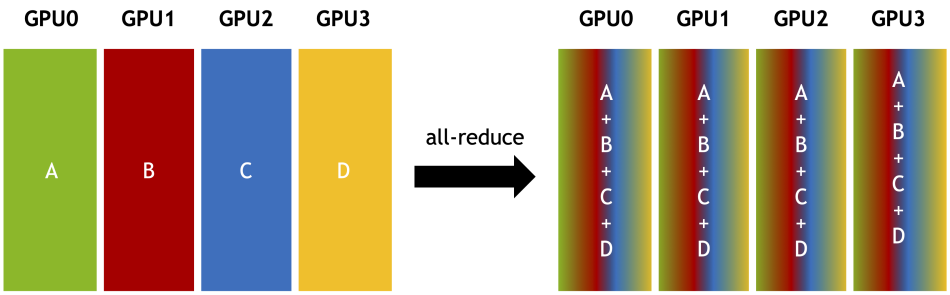
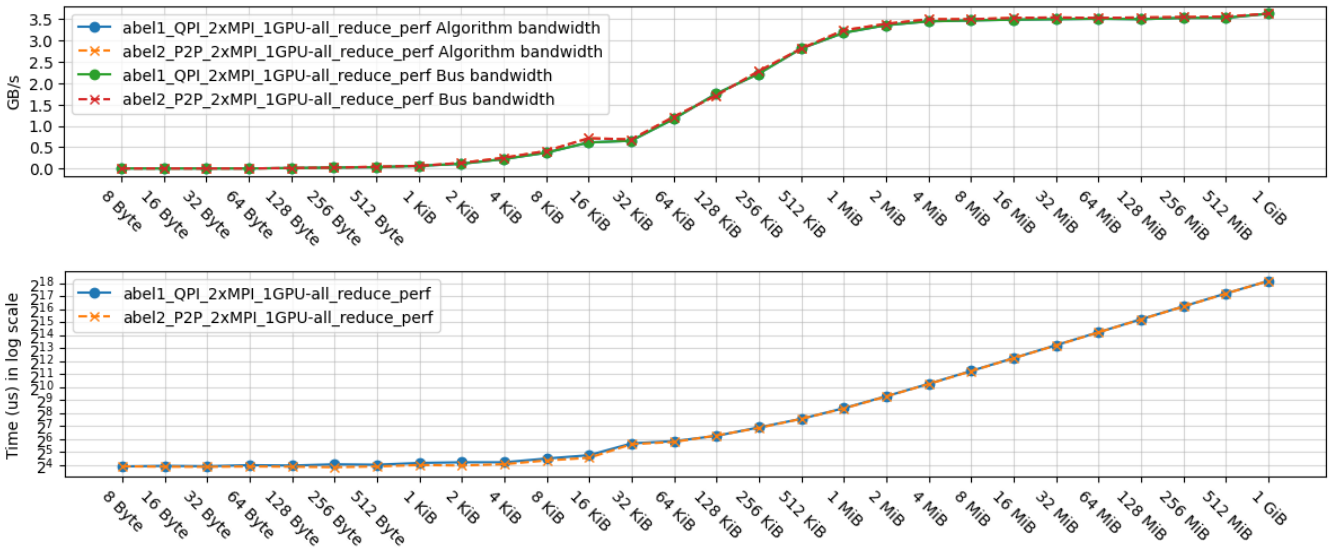


Figure 4.9: All-Reduce illustration [29]

All-Reduce is used in, e.g. Deep Learning and Molecular Dynamics [29] and is generally the most common in ML [22].

QPI vs P2P

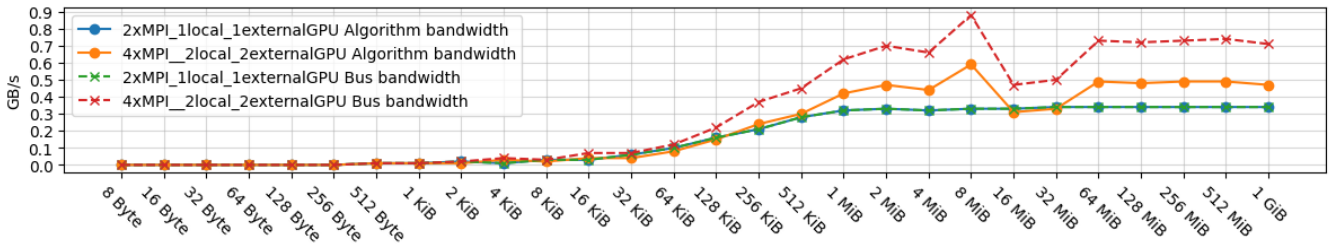
Figure 4.10: *All-reduce performance with GPUs over QPI (abel1), vs direct P2P (abel2)*



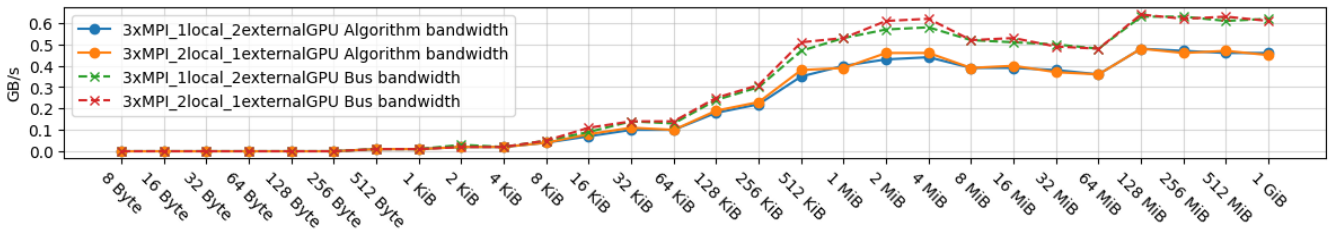
Gigabit Ethernet

Figure 4.11: *All-reduce performance over gigabit Ethernet*

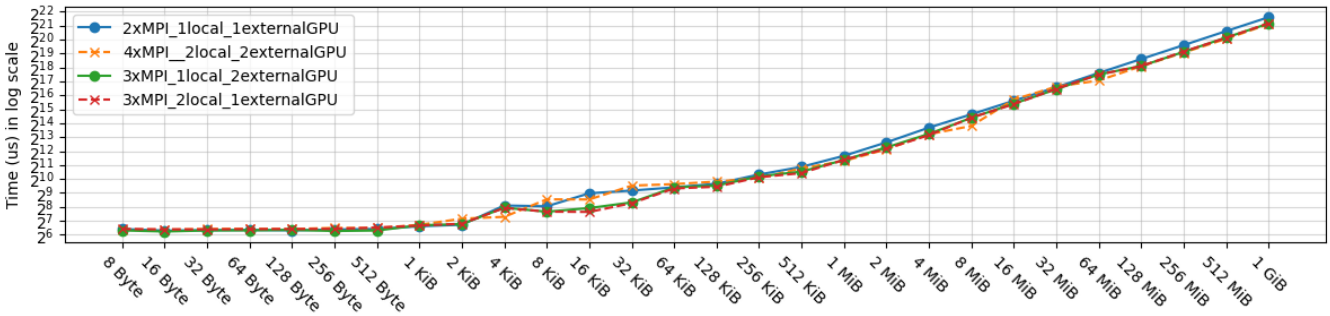
(a) *Difference running one vs two GPUs per node*



(b) *Difference in running two of three GPUs on local, vs remote node*



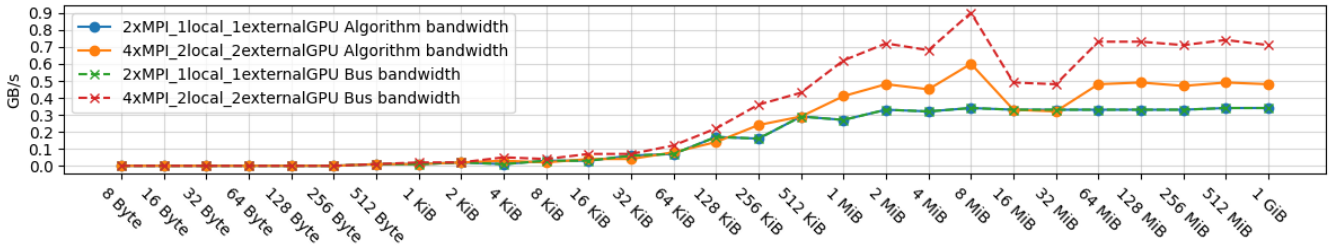
(c) *Time*



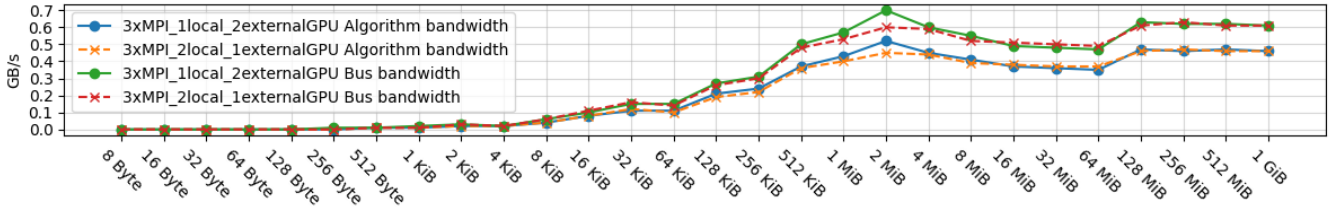
IPoPCIe

Figure 4.12: *All-reduce performance over IPoPCIe*

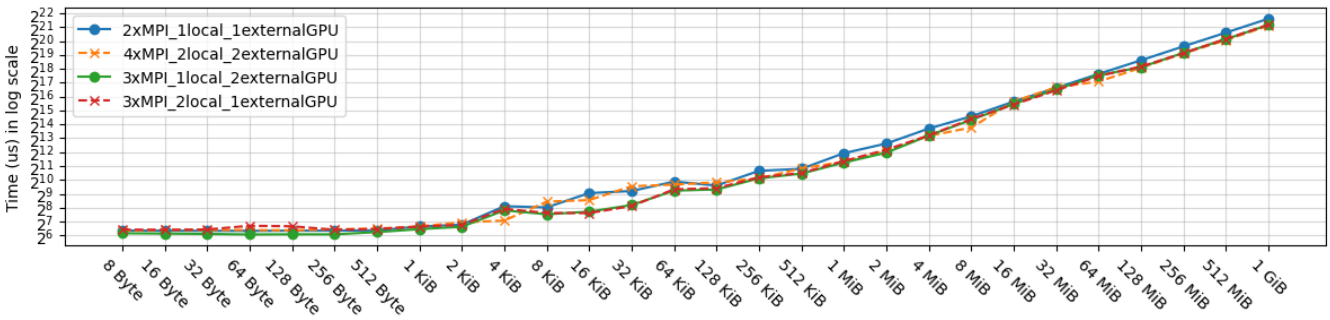
(a) *Difference running one vs two GPUs per node*



(b) *Difference in running two of three GPUs on local, vs remote node*

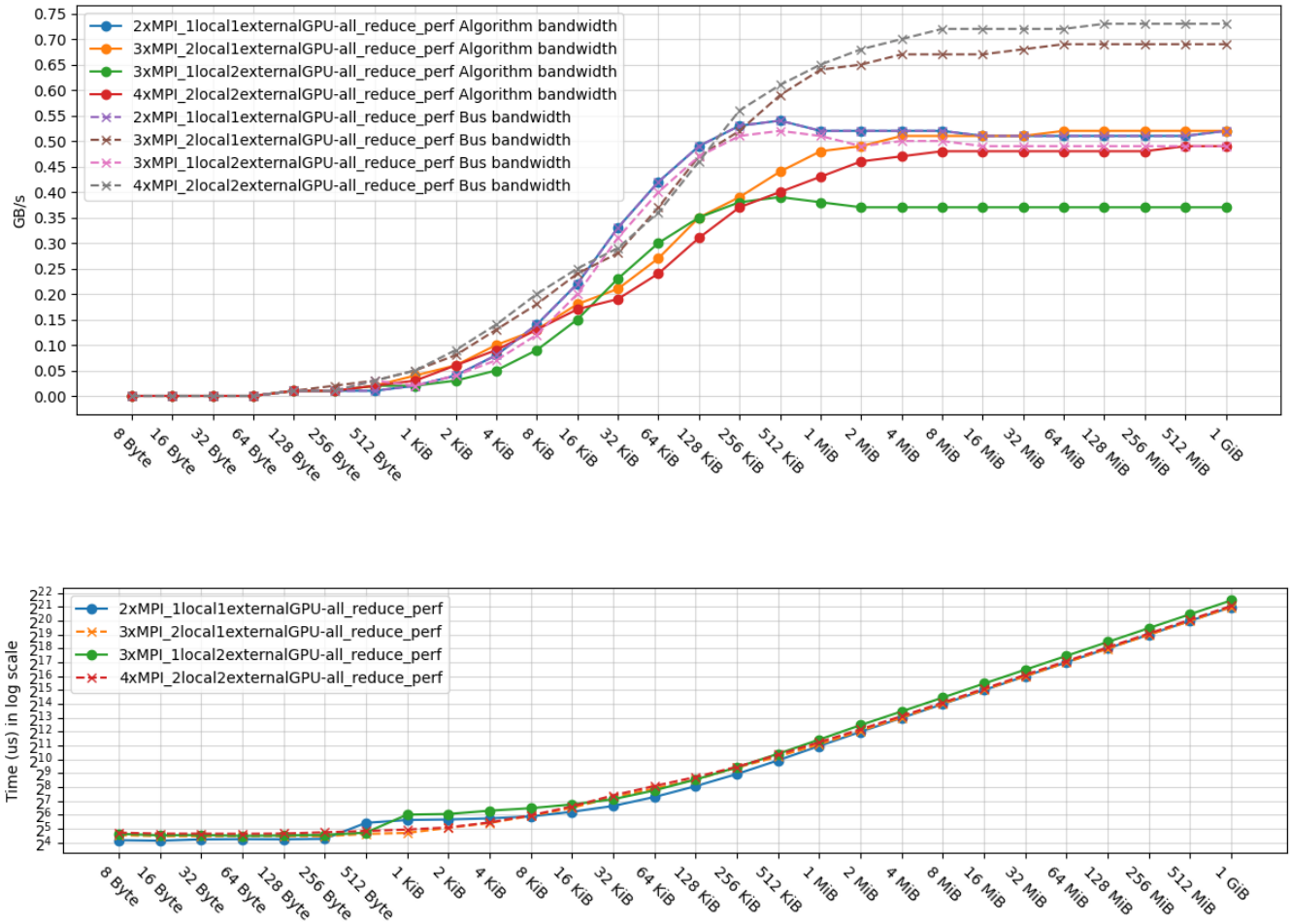


(c) *Time*



SmartIO

Figure 4.13: *All-reduce performance using SmartIO*



Stacked summary (multi-node)

Figure 4.14: *Comparing multi-node interconnects 2 GPU all-reduce performance, 1 in each node*

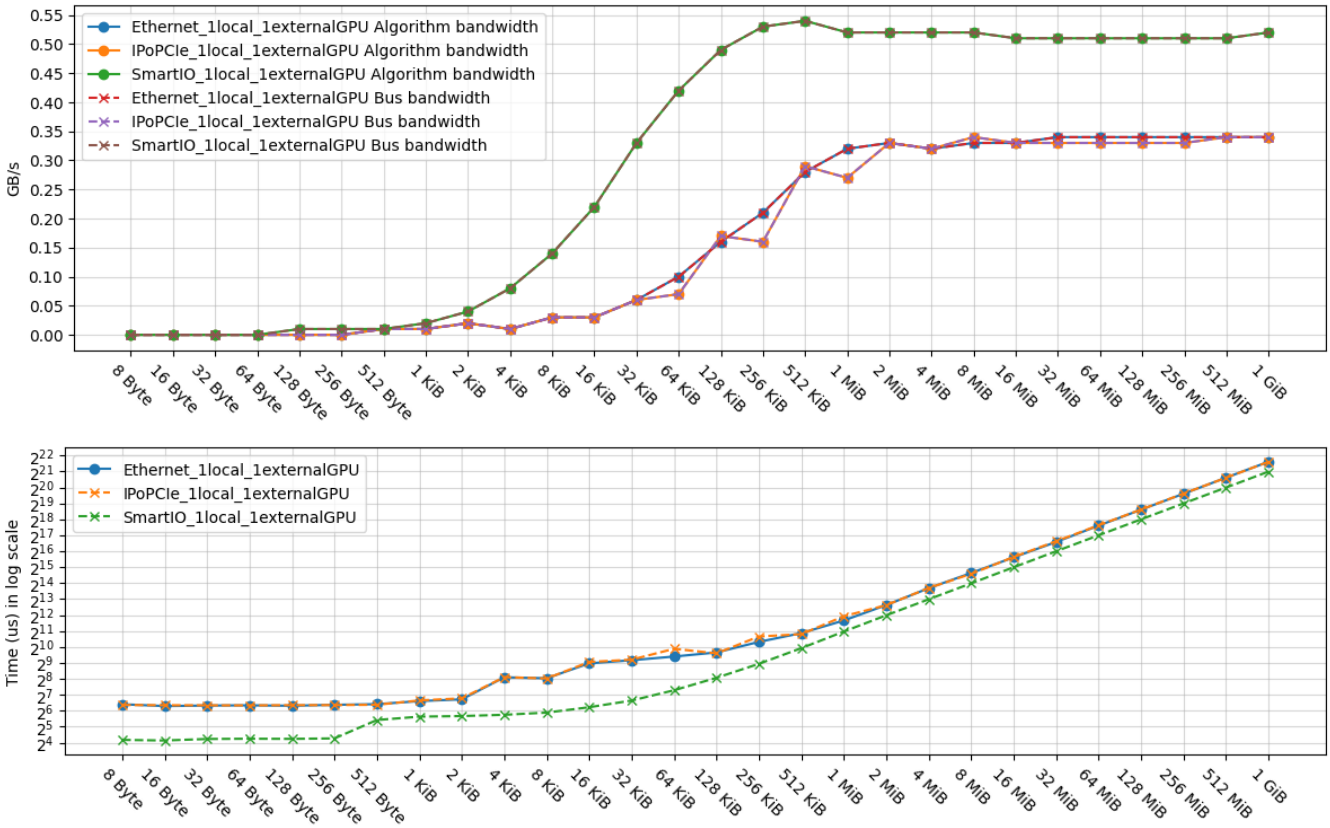
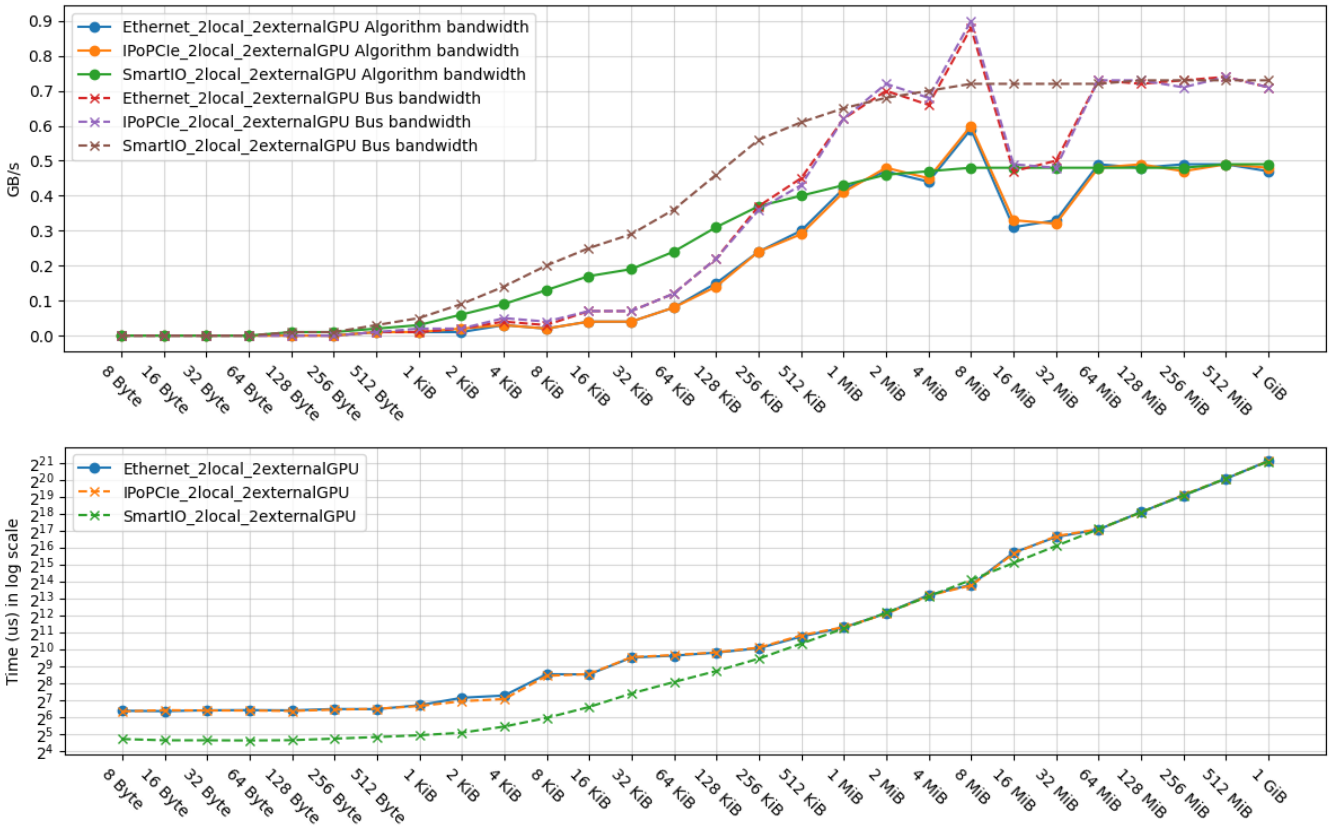


Figure 4.15: *Comparing multi-node interconnects*
4 GPU all-reduce performance, 2 in each node



4.2.3 All-to-All

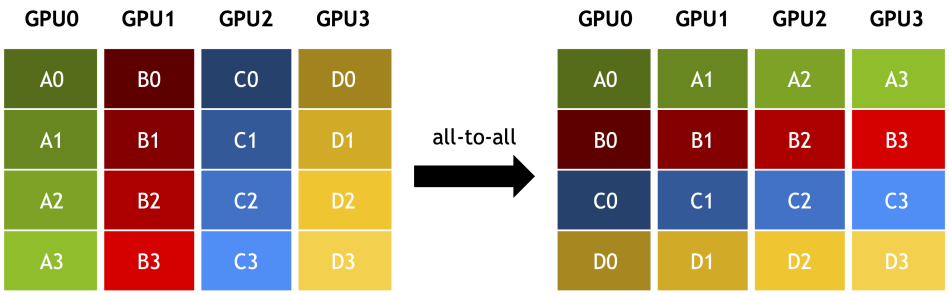
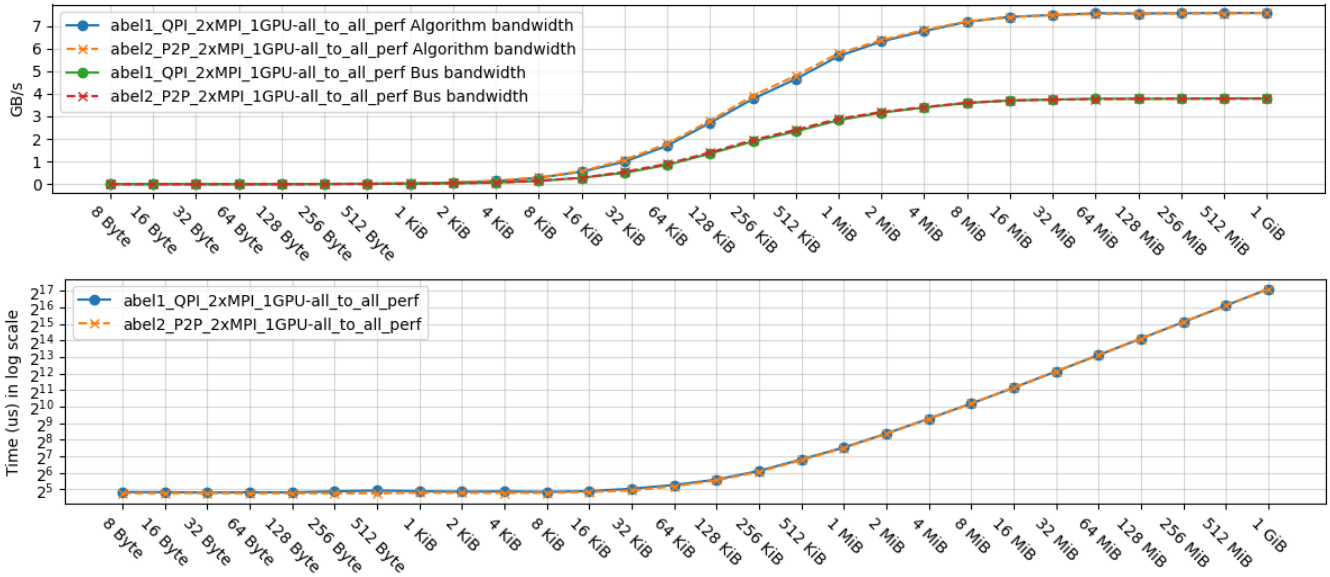


Figure 4.16: All-to-All illustration [29]

All-to-All is the most communication-intensive operation, with data transfers from every GPU to every other GPU. Use cases are, e.g. transposition of data, Parallel Fast Fourier Transform and Graph Analytics [29].

QPI vs P2P

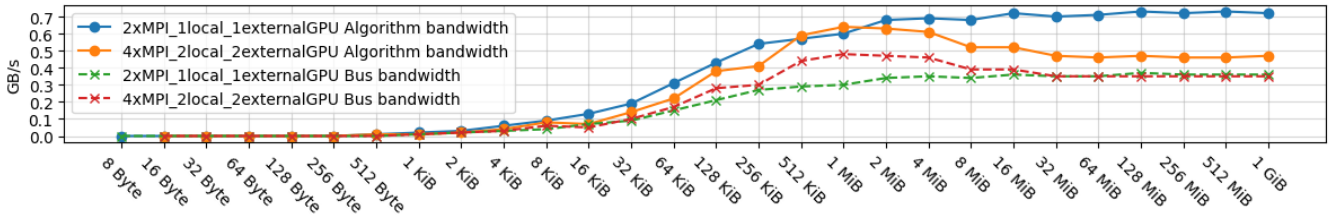
Figure 4.17: *All-to-All performance with GPUs over QPI (abel1), vs direct P2P (abel2)*



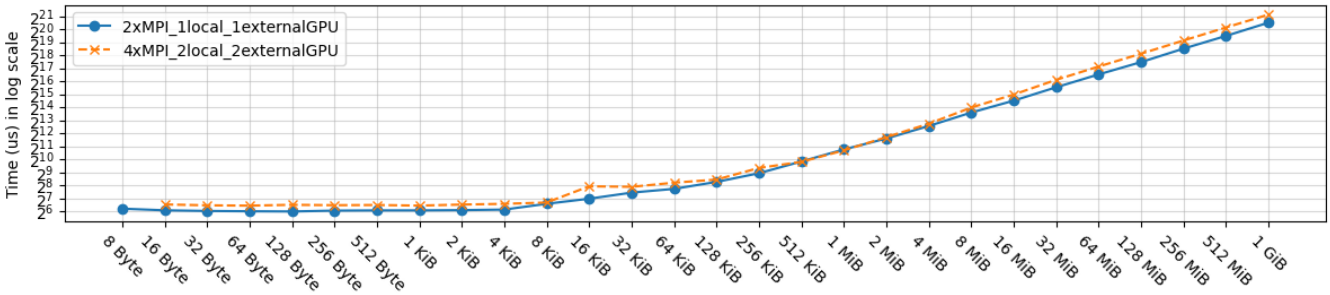
Gigabit Ethernet

Figure 4.18: *All-to-All performance over gigabit Ethernet*

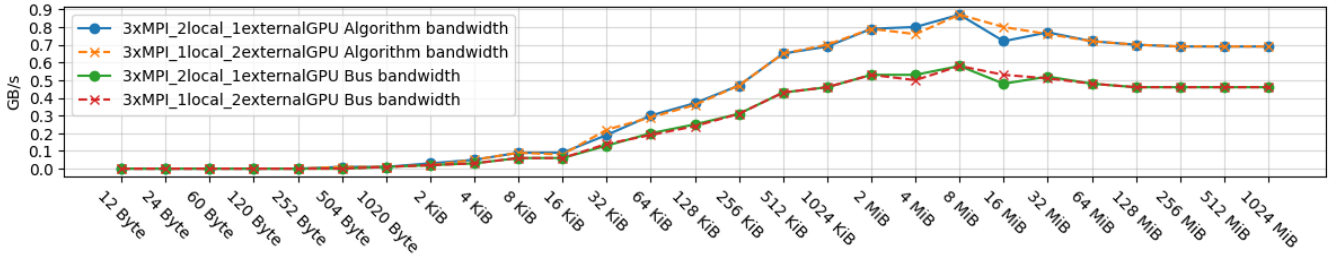
(a) *Difference running one vs two GPUs per node*



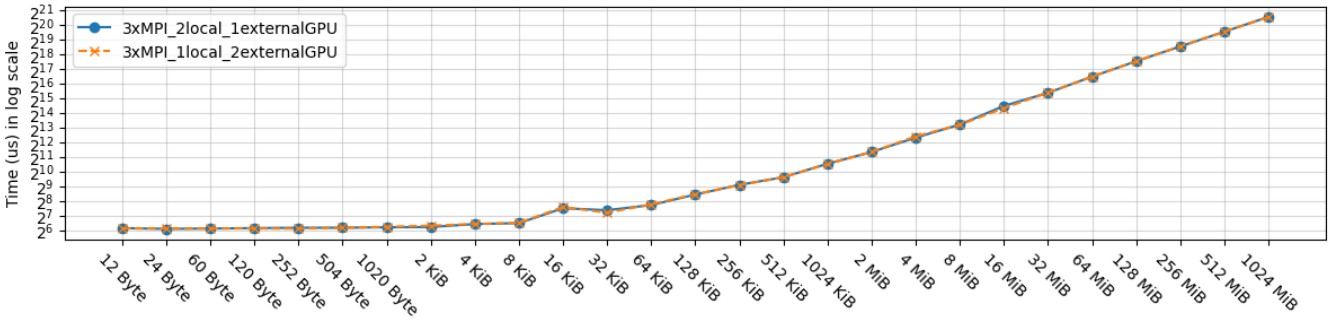
(b) *2 vs 4 GPU time*



(c) *Difference in running two of three GPUs on local, vs remote node*



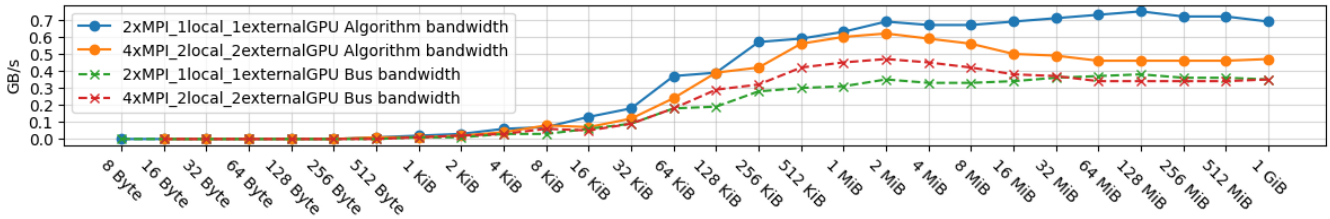
(d) *Internal vs external GPU time*



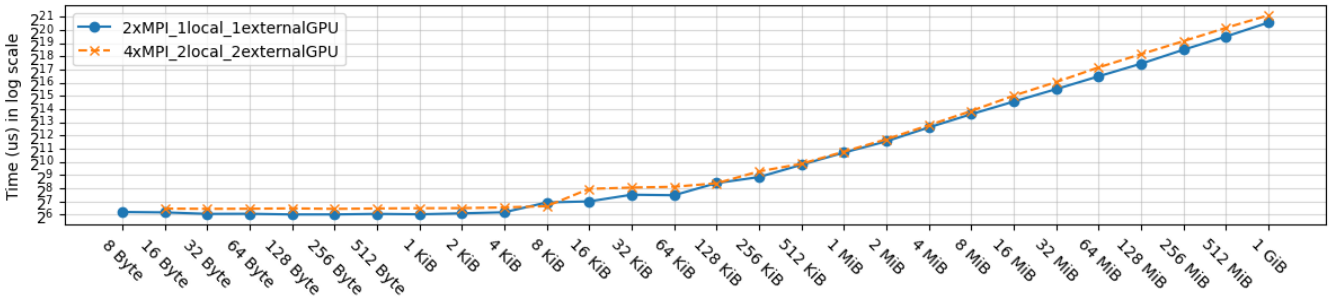
IPoPCIe

Figure 4.19: *All-to-All performance over IPoPCIe*

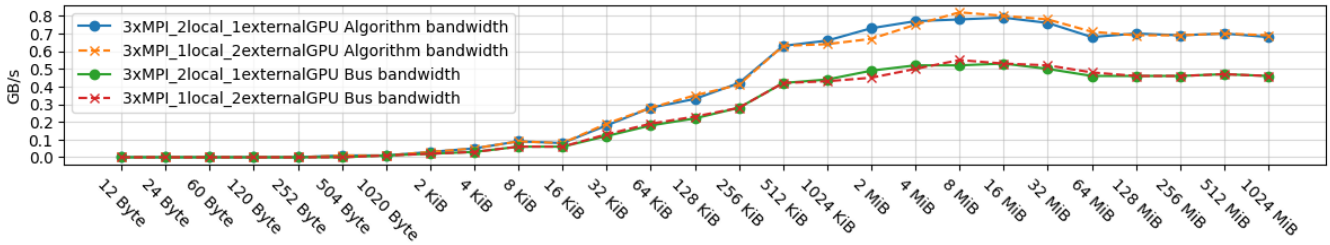
(a) *Difference running one vs two GPUs per node*



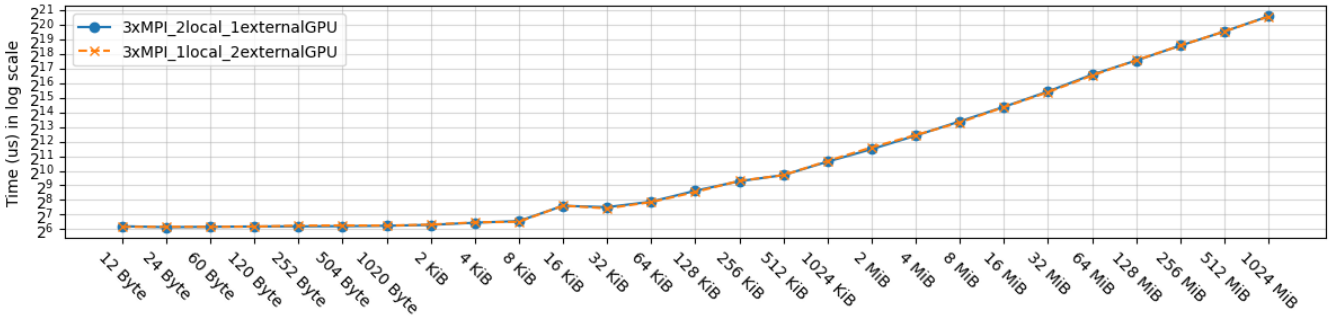
(b) *2 vs 4 GPU time*



(c) *Difference in running two of three GPUs on local, vs remote node*



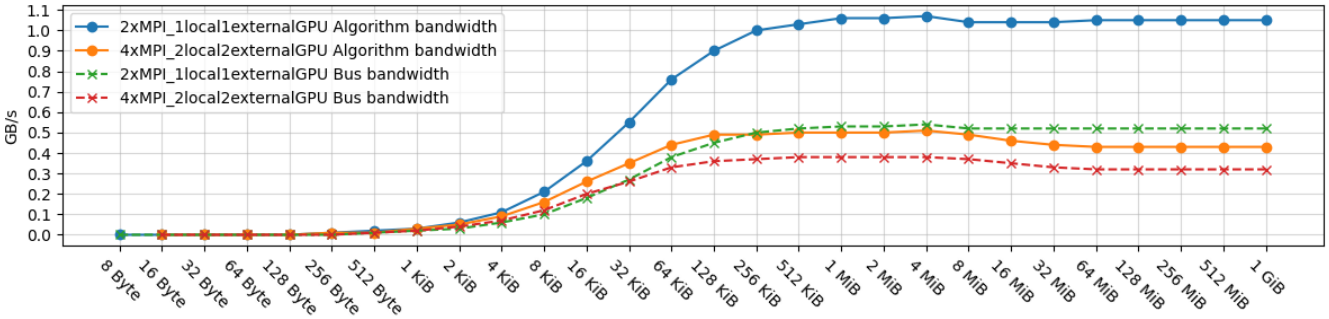
(d) *Internal vs external GPU time*



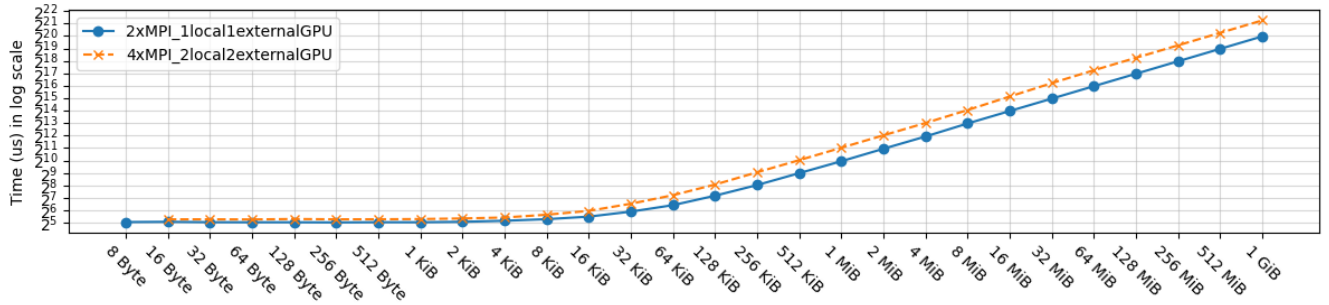
SmartIO

Figure 4.20: *All-to-All performance using SmartIO*

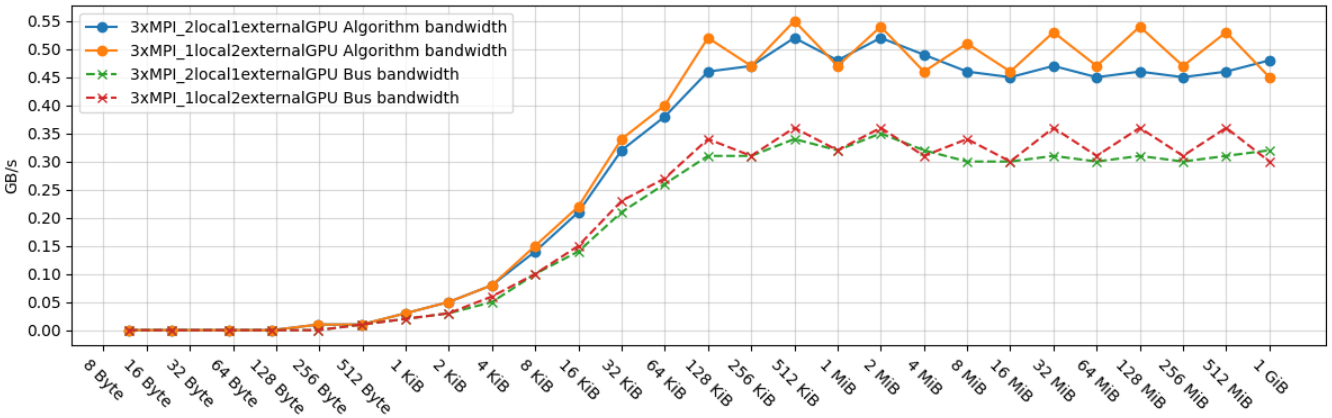
(a) *Difference running one vs two GPUs per node*



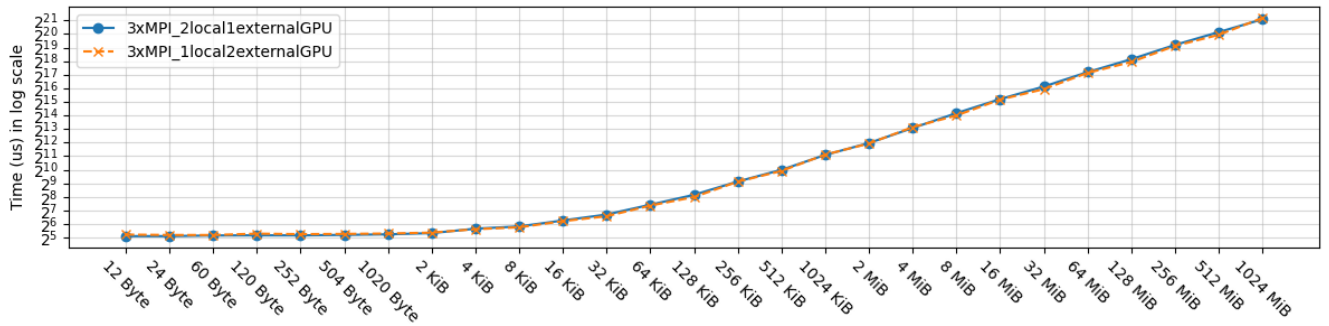
(b) *Time of one vs two GPUs per node*



(c) *Difference in running two of three GPUs locally, or externally*



(d) *Time when two of three GPUs are local or external*



Stacked summary (multi-node)

Figure 4.21: *Comparing multi-node interconnects*
2 GPU All-to-All performance, 1 in each node

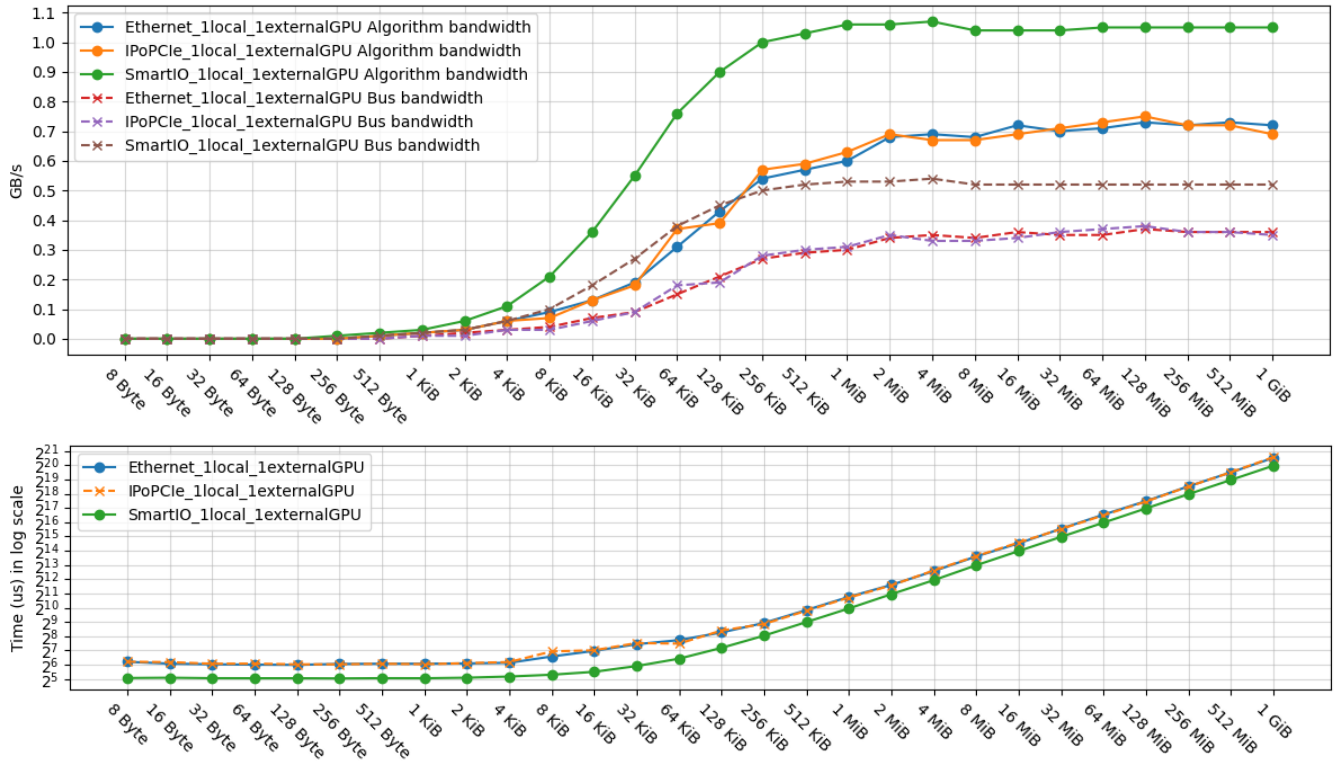
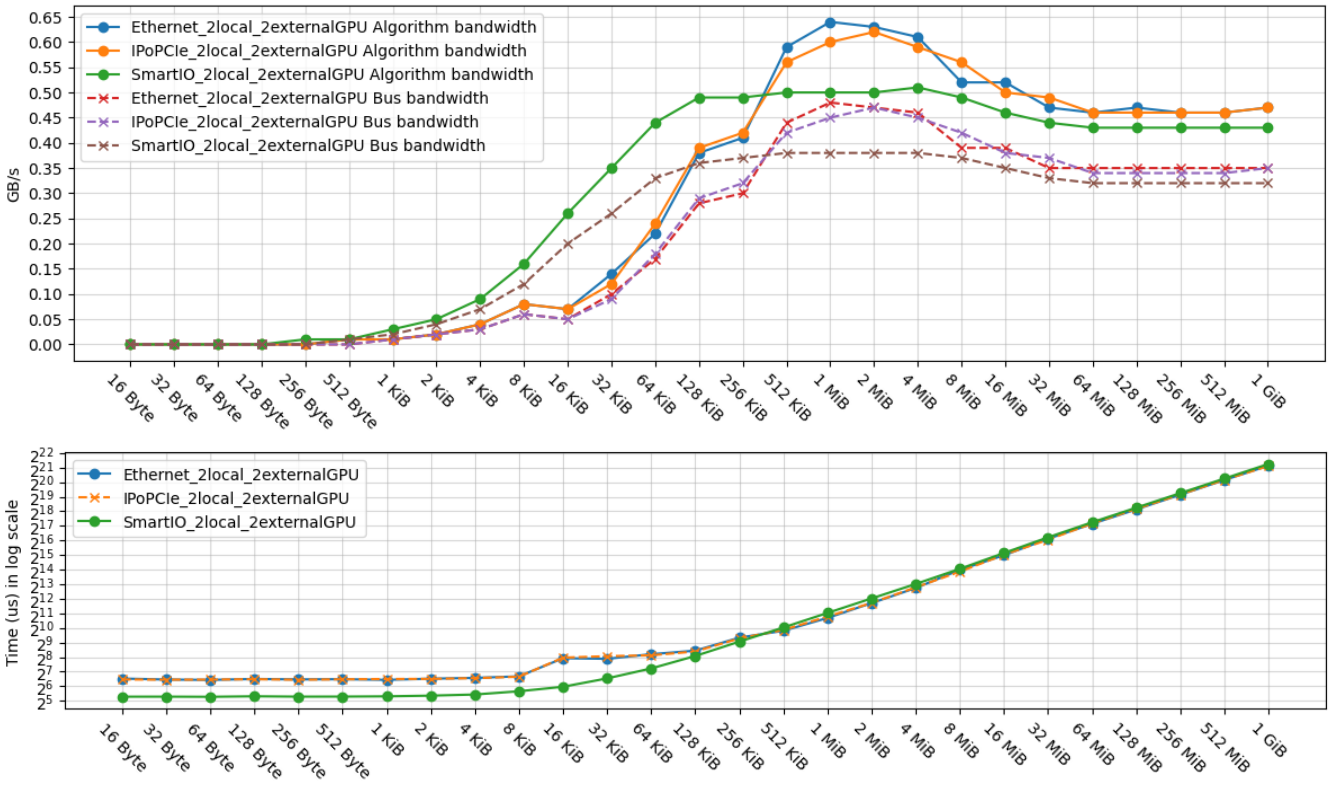


Figure 4.22: *Comparing multi-node interconnects*
4 GPU All-to-All performance, 2 in each node



4.2.4 All-Gather

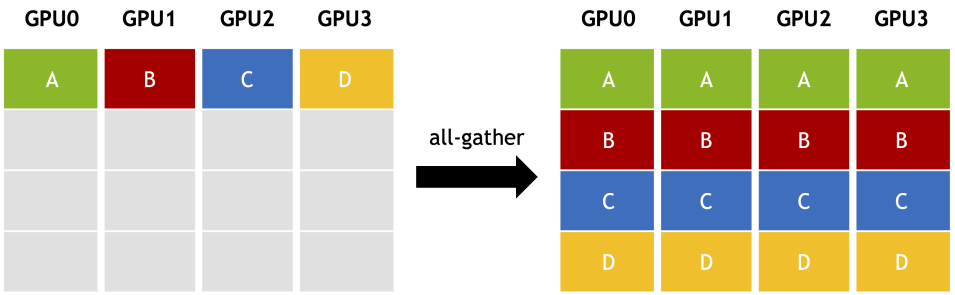
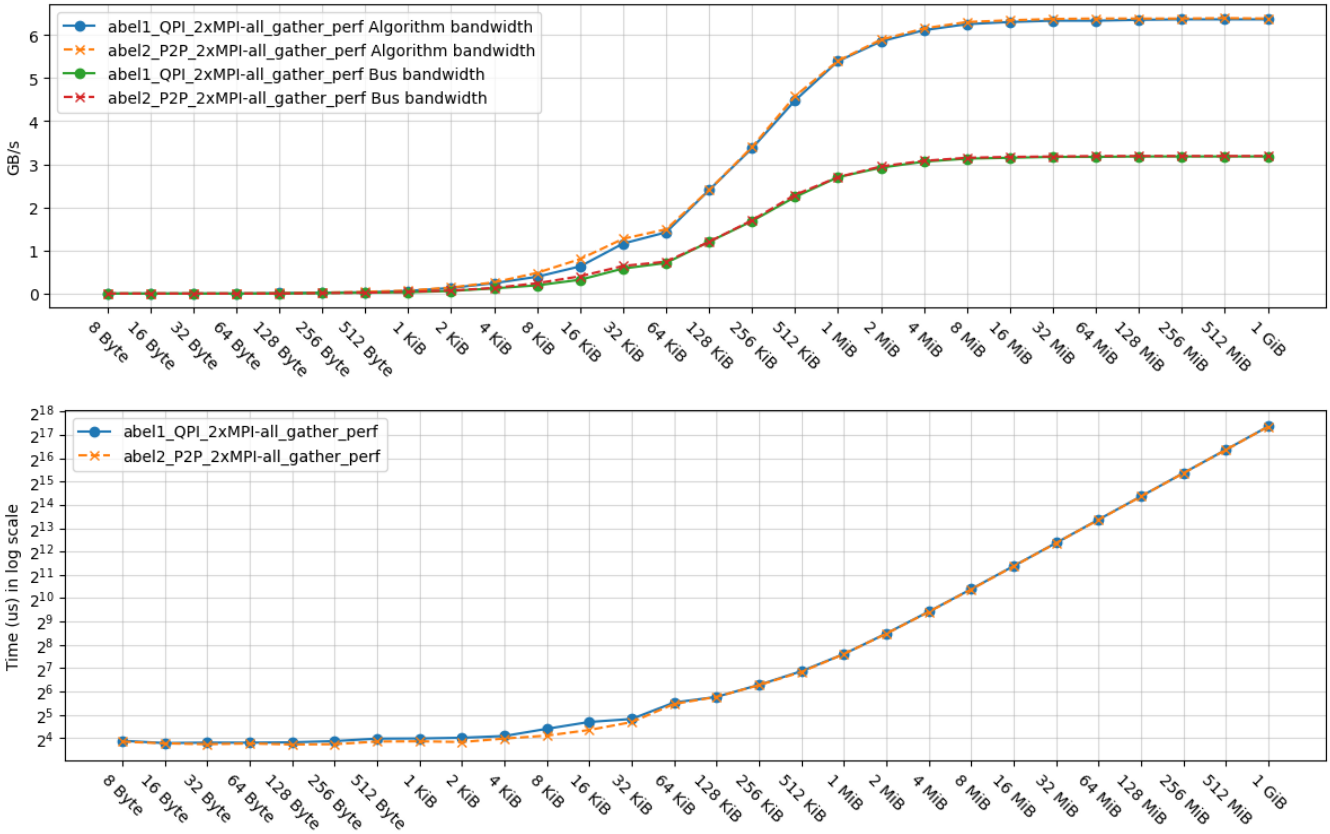


Figure 4.23: All-Gather illustration [29]

All-Gather is another communication-intensive operation, copying a sub-section of data from each GPU onto every other GPU.

QPI vs P2P

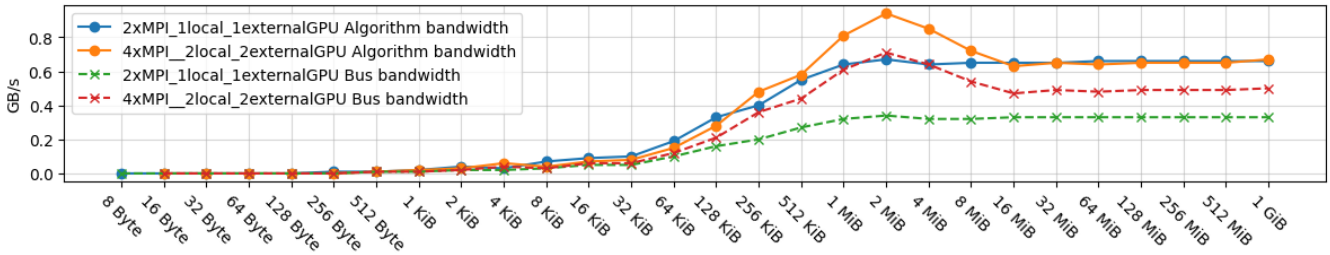
Figure 4.24: *All-Gather performance with GPUs over QPI (abel1), vs direct P2P (abel2)*



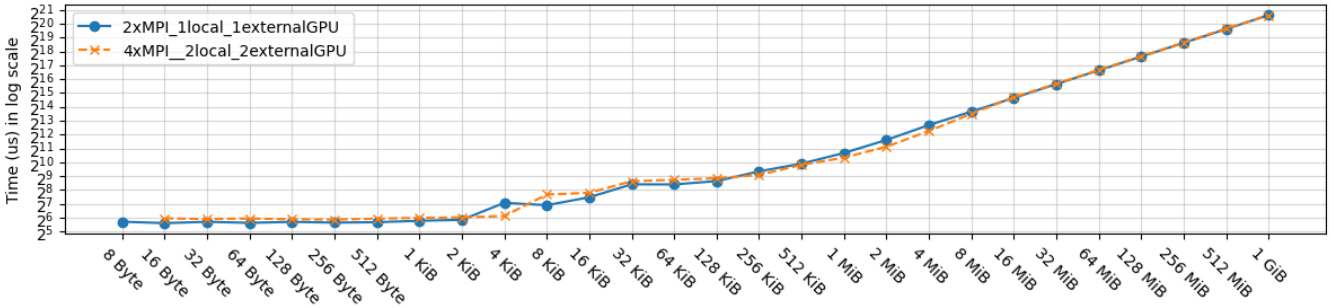
Gigabit Ethernet

Figure 4.25: *All-Gather performance over gigabit Ethernet*

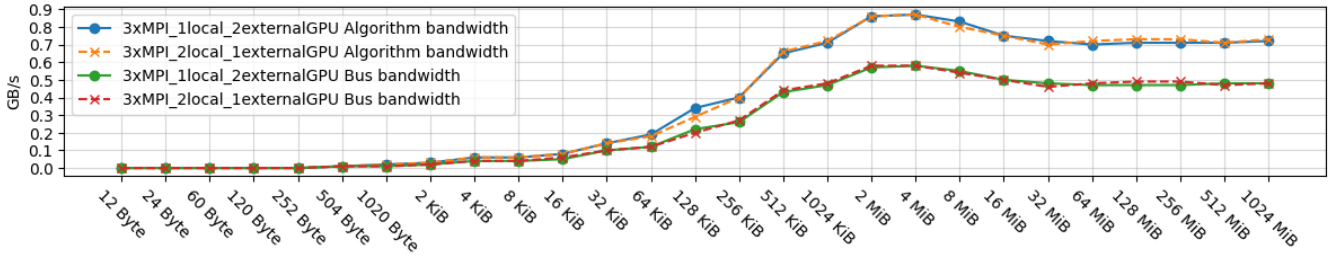
(a) *Difference running one vs two GPUs per node*



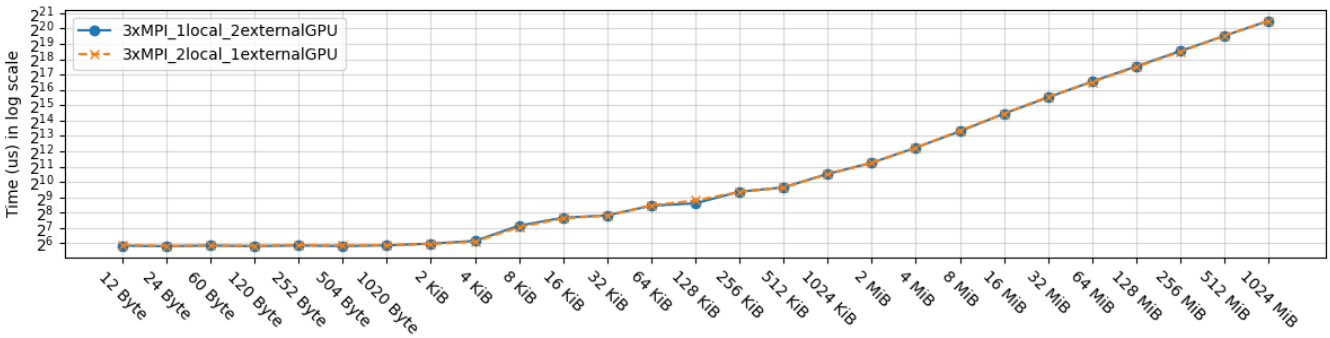
(b) *2 vs 4 GPU time*



(c) *Difference in running two of three GPUs on local, vs remote node*



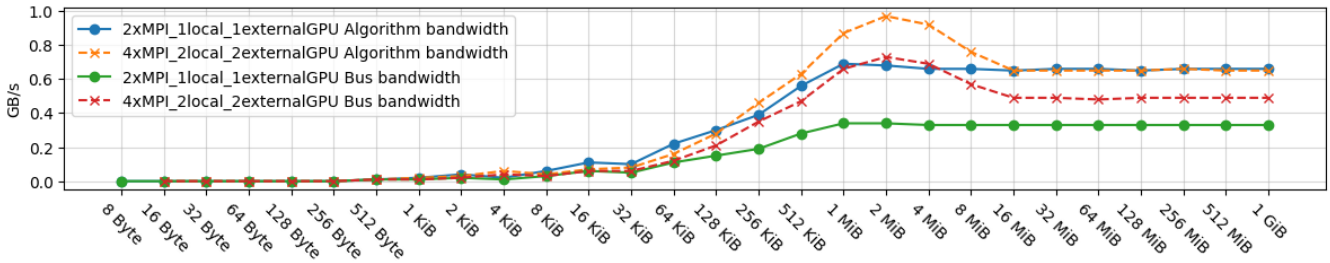
(d) *Internal vs external GPU time*



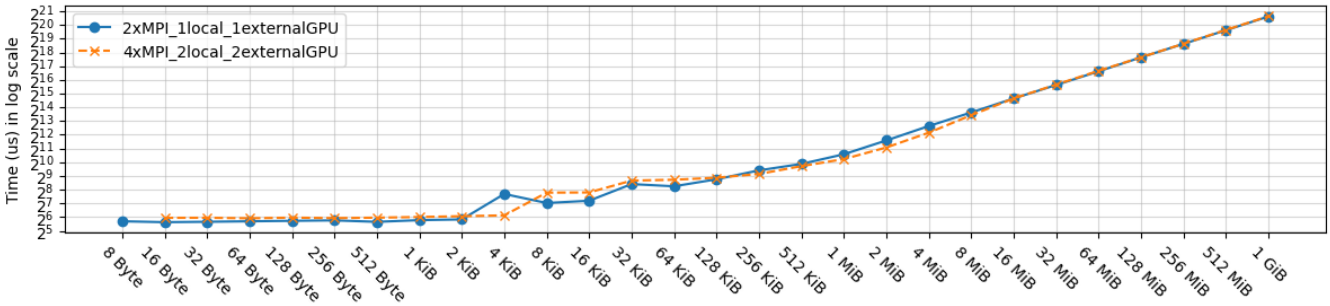
IPoPCIe

Figure 4.26: *All-Gather performance over IPoPCIe*

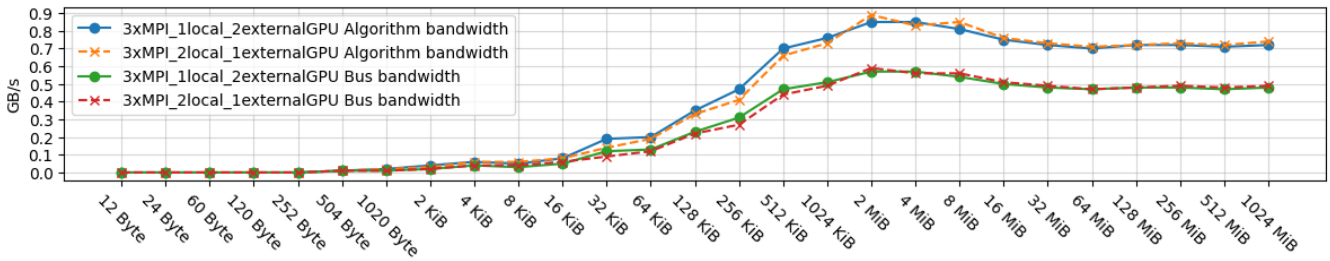
(a) *Difference running one vs two GPUs per node*



(b) *2 vs 4 GPU time*



(c) *Difference in running two of three GPUs on local, vs remote node*



(d) *Internal vs external GPU time*

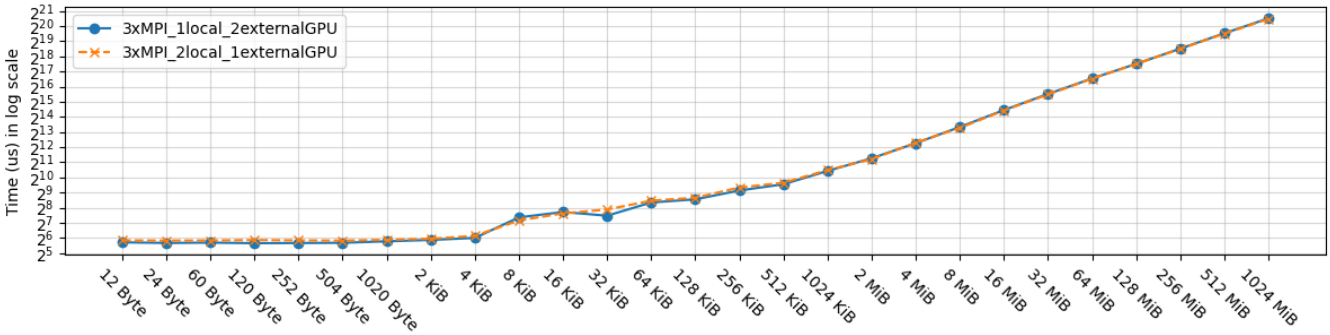
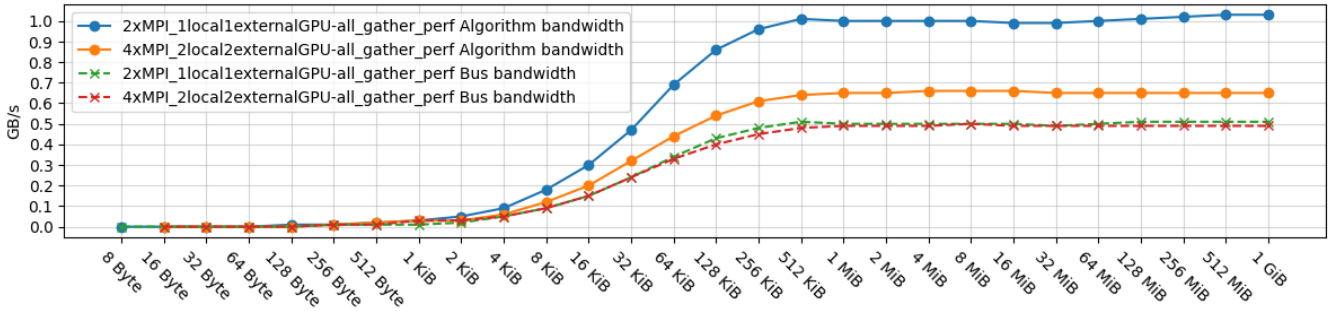
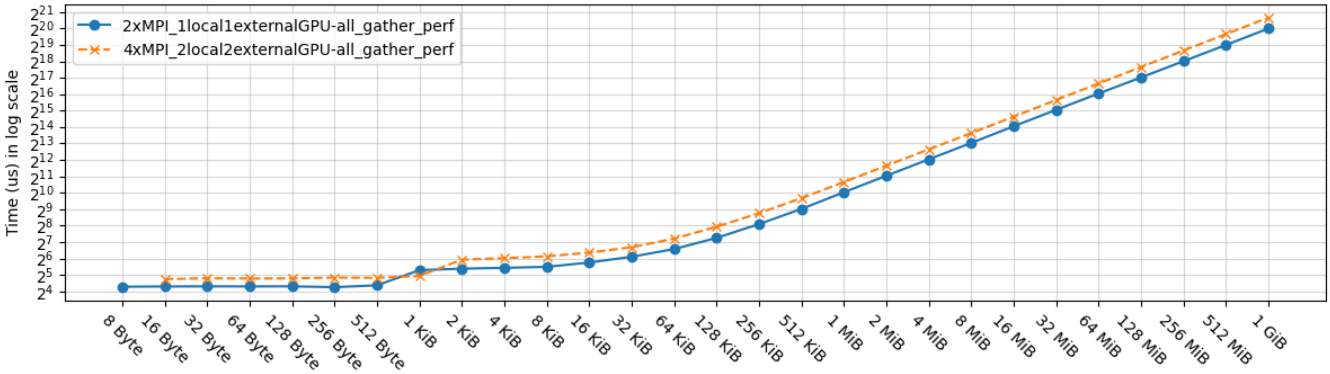


Figure 4.27: *All-Gather performance using SmartIO*

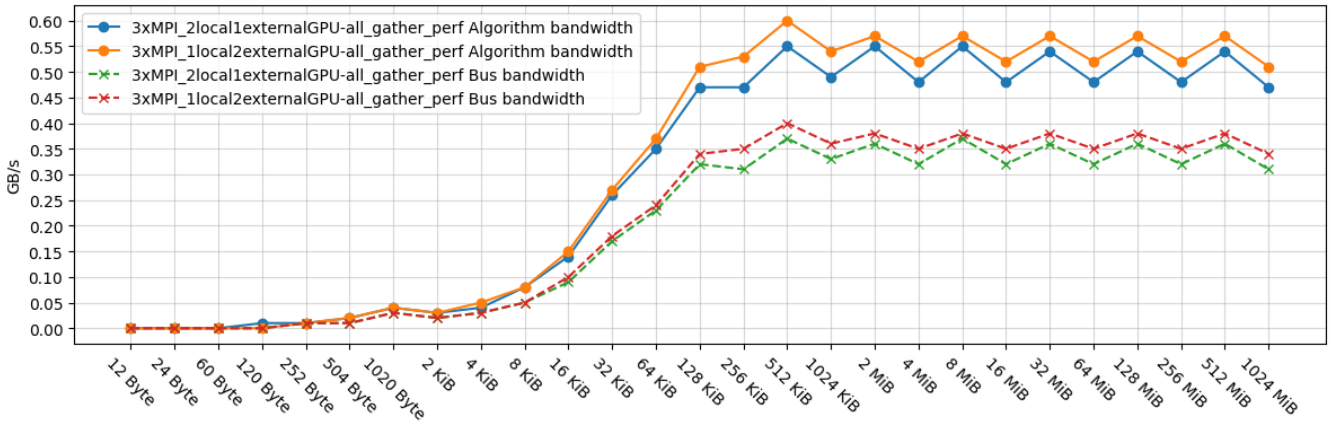
(a) *Difference running one vs two GPUs per node*



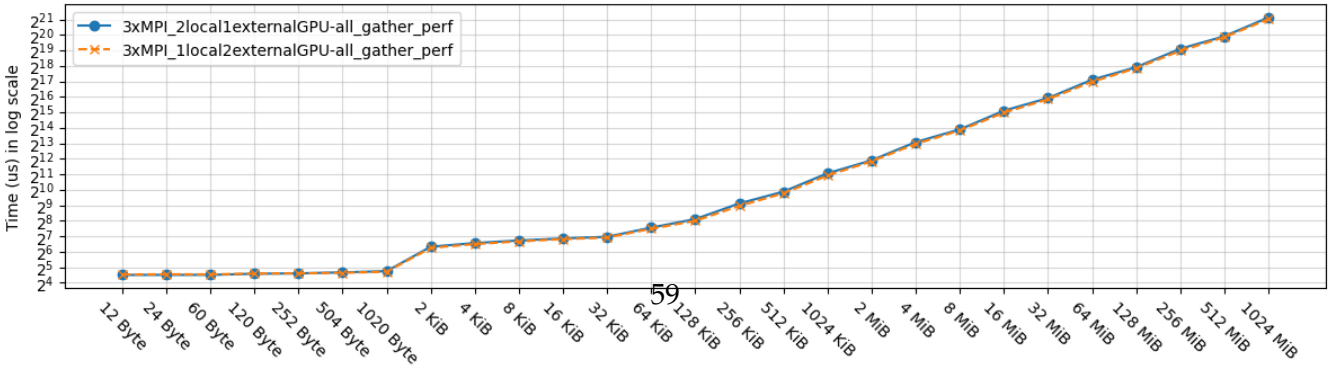
(b) *Time of one vs two GPUs per node*



(c) *Difference in running two of three GPUs locally, or externally*



(d) *Time when two of three GPUs are local or external*



Stacked summary (multi-node)

Figure 4.28: *Comparing multi-node interconnects*
2 GPU All-Gather performance, 1 in each node

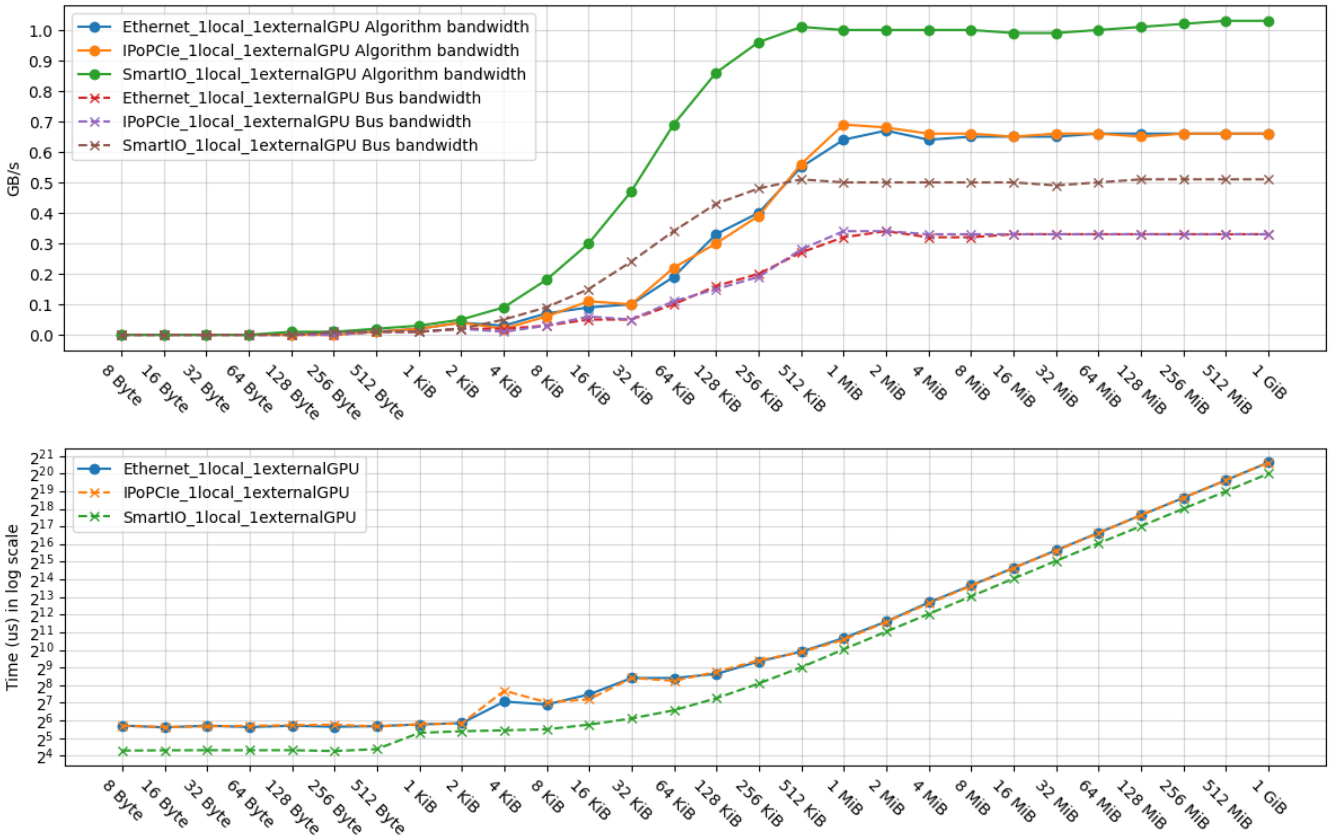
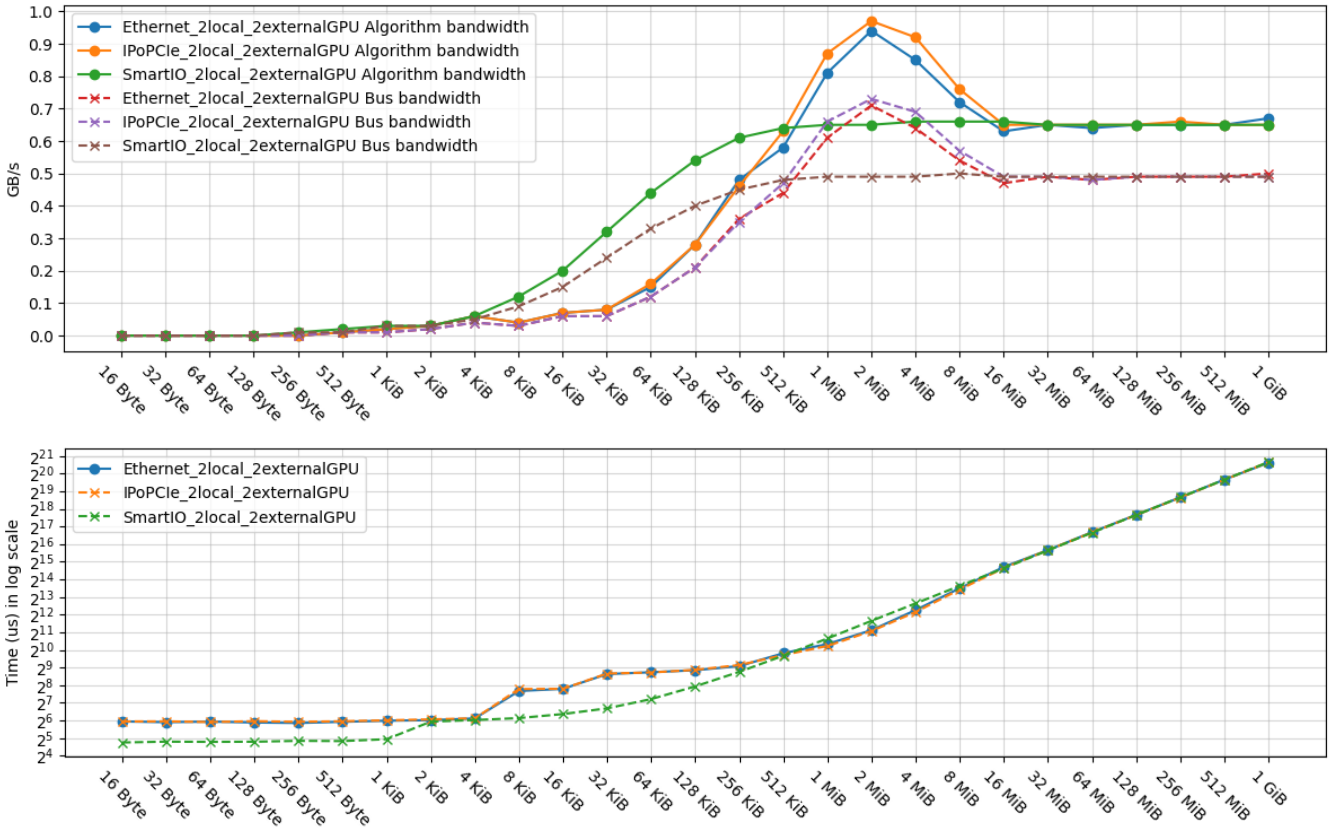


Figure 4.29: *Comparing multi-node interconnects*
4 GPU All-Gather performance, 2 in each node



4.3 Discussion

4.3.1 QPI vs P2P

With just a hair, P2P displays a slight overhead over QPI in All-to-All 4.17 and All-Gather 4.24 communication. Overall, however, they match in performance. This can be explained with a look at the system diagram of the server 3.1, QPI provide a point-to-point connection with a transfer speed of up to 8.0 GT/s. Quite enough for our gen2 GPUs. There is to be noted that our synthetic tests ran alone with little to no other software and services. QPI performance may go down if there is more cross-traffic from other processes. We would generally suggest connecting multiple GPUs to the same CPU if possible. If that is not an option, our results show no significant performance loss in splitting GPUs across NUMA nodes with QPI interconnect.

4.3.2 Gigabit Ethernet

We were surprised by how high the bandwidth of Gigabit Ethernet is during NCCL communication. Even sometimes beating IPoPCIe 4.22. While bandwidth between nodes is at most 112MB per second, bandwidth between GPUs inside each node is unhinged with high QPI and P2P bandwidth, thus increasing the total average.

In the Broadcast benchmark 4.4 there is a sudden drop at 256 KiB. Considering this drop is not visible in the other NCCL tests, we suspect it may be caused by some other traffic happening at the network switch during our testing.

In the All-Gather test 4.25, we suspect there is some optimization for segment sizes at around 2 megabytes, as bandwidth peaks there before slowing down and flattening. This is reflected in the IPoPCIe benchmark 4.26 as well. If we were to use NCCL in a production environment with only Gigabit Ethernet and heavy use of All-Gather, we would suggest choosing 2MB as the segment size.

4.3.3 IPoPCIe

To our big surprise, IPoPCIe with an 857 megabytes per second performance, lines up along the performance of gigabit Ethernet. Even going slower at intervals in All-Reduce 4.14.

In the Broadcast test with three GPUs 4.5, we see that the performance is lower if 2 of them are external rather than internal. The result seemed odd since the ring topology of NCCL should mean that there is one transfer across the link, regardless of the position of the GPUs. We expected equal performance. The underlying cause for this may be related to how Nvidia places GPU ranks based on PCIe bus addresses. The topology of our system makes Nvidia consistently rank the external GPUs before the internal ones.

The All-Gather test 4.26 shows the same peak at 2 MiB as with gigabit Ethernet. Considering that Ethernet and IPoPCIe use different drivers and

transfer across entirely separate interconnects, we suspect the explanation is an optimization in NCCL.

With some code changes to use SuperSockets instead of the default IP stack, we might have removed some overhead and made bandwidth increase. But overall, the results show that IPoPCIe, in a configuration like ours, is not a drop-in replacement to increase NCCL performance over built-in gigabit Ethernet.

4.3.4 SmartIO

In Broadcast 4.6 and All-reduce 4.13, we see that SmartIO drastically increase the NCCL bandwidth of our cluster. Moreover, the latency difference between SmartIO, Gigabit Ethernet and IPoPCIe, visualize how much the overhead of packaging and processing TCP packets is.

Same when our systems have a single GPU per node in All-to-All 4.20 and All-Gather 4.27 tests. But something is very off as soon as we add two extra GPUs to the cluster, SmartIO performance flattens far below Ethernet and IPoPCIe. The latter uses the same interconnect as SmartIO. We come back to our question if there is some optimization helping IP-based transfers around segment sizes of 2MiB.

In the Broadcast test 4.6 with one GPU in each node, we see the average bandwidth is around half compared to GPUs within a single server. This is because P2P is not supported, and RDMA is not used. Data travels via system memory in both directions, adding copy/write steps, and this adds an extra factor to consider when setting up a cluster. Without RDMA, the read and write speed of the main memory will impact GPU-to-GPU communication. When intercepting the traffic between the Dolphin interconnects during NCCL benchmark, we noticed the average payload is 64 bytes, despite our hardware supporting 256.

IOMMU can have an effect on the performance as well [14]:

> Since IOMMUs create a virtual address space, TLPs need to be routed through the root of the PCIe tree in order to resolve virtual IO addresses, effectively disabling peer-to-peer transfers. Processor designs are complex and often not well-documented, making it difficult to determine what exactly happens with the memory operations in progress once they leave the PCIe complex and enter the CPUs. Memory operations may be buffered, awaiting IOMMU translations, or the IOMMU may need to perform a multi-level table look up for resolving addresses.

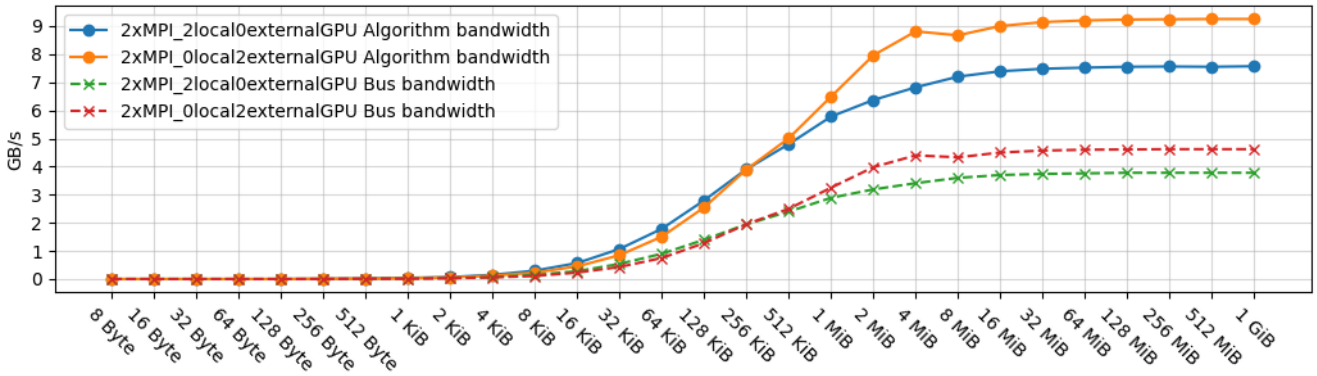
In the All-Reduce test 4.13, we see that the bandwidth curve flattens to meet along with the other interconnects in a dual GPU setup. The average bandwidth drops as we add GPUs, showing a bottleneck in the interconnect. Compared to QPI and P2P results, the drop is drastic, and we wonder if the communication consists of lots and lots of very small and fragmented data transfers. If so, upgrading the GPU to something modern with tensor cores optimized for All-Reduce operations might not improve

the communication performance across nodes. Another All-Reduce puzzle is the difference between having two GPUs local vs external in the three-GPU configurations. We could not find a good answer on why this happens.

The three-GPU tests in All-to-All 4.20 and All-Gather 4.27 tests display odd stepping pattern after 128 KiB. We are unsure of the cause, but think it might have something to do with how SmartIO schedule DMA transfers.

Against expectations, we registered that NCCL performance is higher when lending two external GPUs (from abel2) than using two internal GPUs. Meaning when we ran tests on only those two GPUs alone, enabling P2P for the whole test. To confirm the results, We ran the same benchmarks locally on abel2, where it also uses P2P. Still, the local run output results were worse than when abel1 borrowed the same GPUs. Something causes the GPUs lended out with SmartIO to perform better. We checked with p2pBandwidthLatencyTest from the CUDA-samples package and confirmed that P2P is on and that this should not be the bottleneck. The cause for the speedup for SmartIO is suspected in the difference in how P2P mappings are done by SmartIO compared to the NVIDIA driver.

Figure 4.30: *All-to-All performance running 2 GPUs locally on abel2, or borrowed to abel1 with SmartIO*



A deeper look into how exactly NCCL moves data between ranks could be of interest, considering DMA write operations to remote nodes yield significantly higher throughput than reading from remote nodes. If data transfer in the NCCL ring formation is done by pulling data rather than pushing, it could answer why we see lower bandwidth with SmartIO than, e.g. QPI.

4.3.5 Closing thoughts

We wonder if older GPUs with less VRAM could be useful in HPC-applications that does "streaming calculations", aka computation on data that doesn't need to stay long in VRAM. In workloads where a large portion of the computation time is moving data, lower GPU performance might not be as significant, however older cards might have slower

memory bandwidth and PCIe speeds, which negatively affects data transfer performance.

While our NVIDIA K20X hardly competes in half and single precision performance, the double precision performance are impressively high and competitive with modern models as seen in table 4.2. Meaning HPC and scientific computing that are dealing with double precision workloads can benefit from similar performance to modern hardware at a fraction of the cost.

A modern approach in scientific workloads is to exploit the high mixed precision performance to generate a rough result, then target specific areas of interest with double precision. In such scenarios, a hybrid solution might contain both new and old GPUs in order to exploit the mixed precision performance of newer GPUs, while saving cost by offloading double precision calculations to older hardware. Freeing time on the newer GPUs for more mixed-precision work. This is especially useful in clusters that try to reduce cost by using consumer grade hardware, such as Geforce RTX cards. The double precision performance of the most modern consumer grade cards today, such as Geforce RTX 4090, is still beaten by a cheap 10 year old data center GPU, as we can see in table 4.2.

Model	Release date	Half precision (GFLOPS)	Single precision (GFLOPS)	Double precision (GFLOPS)	TDP (watts)	Price
K20X	November 2012	N/A	3935	1312	235W	399 NOK
RTX 4090	*October 2022*	*82600*	*82600*	*1291*	*450W*	*24 799 NOK*
A40	October 2020	149680	37420	1168	300W	81 782 NOK
L40	October 2022	362066	90516	1414	300W	126 639 NOK
A100	May 14 2020	312000	19500	9700	250W	221 306 NOK
H100	March 2022	756449	51200	25600	350W	474 979 NOK

Table 4.2: Performance and price comparison[1] of some GPUs used in HPC environments [26]

An issue with an age-hybrid HPC environment is that software and code must be written with backwards compatibility. Backwards support costs extra, and it does not help that the hardware gets marked by manufacturers as legacy and driver support phased out. In some ways, this is a form of planned obsolesce, not in the way of hardware going defect, but software, caused by support-removal in drivers and compilers. An alternative is keeping the cluster in an offline environment, where running outdated hardware and software is safer.

[1] Prices are examples found on prisjakt.no and ebay.com at the time of writing

Chapter 5

Conclusions

5.1 Summary

We started with two ten-year-old servers, legacy GPUs, and a modern Dolphin PCIe interconnect. Then asked ourselves, do these GPUs still have value in a modern ML and HPC environment? We set the goal to get NCCL tests running and benchmark how they perform on legacy GPUs over various PCIe interconnects. And also using the newest OS and software available, like a modern HPC environment would expect to be able to run. We started by connecting the GPUs in two different configurations in each server, one configuration for testing QPI performance, and the other for P2P performance. Then rack-mounting the servers and updating BIOS and various firmware. Then configured IPMI console to gain remote access outside the server room. We installed Ubuntu 22.04 LTS, confirmed incompatibility troubles with `stdio.h` and downgraded to 20.04 LTS. We configured network, hostnames and SSH, and a backup and restore system for use during testing. Then we installed GPU drivers and confirmed they were on the Long Term Support Branch. Then installed CUDA, NVCC, NCCL and MPI through NVIDIA HPC SDK, discovered GPU incompatibility with the latest version, and had to downgrade to version CUDA 11.4. Then we downloaded and tried to compile CUDA samples and NCCL tests, learned how to force compilations against older architectures, and pointing NVCC to non-standard CUDA library locations. Then discovered NCCL tests were incompatible with the compiler for CUDA 11.4. We downgraded NCCL tests to an earlier version to circumvent the problem. Then we found methods to get MPI-run to work with NCCL tests over SSH, including sharing CUDA library paths over non-interactive SSH sessions. Then we installed drivers and software to use Dolphin PCIe NTB cards connecting the two servers with a high-speed interconnect and test frameworks such as IPoPCIe and SmartIO. Then we configured PCIe bar sizes and tested methods of unloading and reloading the NVIDIA kernel module to establish a stable method for SmartIO device lending, using our GPUs without the system freezing. Finally, we ran NCCL test over gigabit Ethernet, IPoPCIe and SmartIO to benchmark Broadcast, All-Reduce, All-to-All and All-Gather

communication.

We confirmed that OS and GPU drivers still had support, but not with the newest versions. We found that the GPUs age hindered the use of the latest compiler, breaking compatibility with the newest NCCL tests version. We documented strong NCCL-communication performance from SmartIO and that there was little difference between gigabit Ethernet and IPoPCIe in our NCCL test case. We concluded that legacy GPUs value in a modern ML and HPC environment is limited, mainly because of software support. However beneficial for those seeking cost-effective double precision performance.

5.2 Main Contributions

We found challenges concerning software support, packages and drivers when using legacy GPUs. Ubuntu 20.04 LTS and GPU drivers branch R470 is still supported a few years more, but any new software created today with Ubuntu 22.04 LTS and newer GPU drivers and CUDA versions in mind will not work. The GPUs age and status as legacy within NVIDIA cause support to be removed in today's compilers and cause compatibility issues with the newer CUDA code, such as the one we found in NCCL tests.

Regarding PCIe, We found it could be used efficiently in ML communication frameworks such as NCCL, and we benchmarked internal (QPI, P2P) and external (IPoPCIe, SmartIO) PCIe interconnects. We found that with our legacy GPUs, in NUMA setups where GPU-to-GPU communication crosses QPI, it had little to no bottleneck in performance compared to a P2P configuration.

We confirmed that SmartIO increased bandwidth and lowered latency in multi-node communication, reaching bandwidth at more than 3.5 gigabytes per second in the Broadcast test. But the more GPUs added, the lower the average bandwidth went. We see a potential for improvements with RDMA support, with some prototypes such as a sisci-nccl plugin [21] already existing. We also found that IPoPCIe did not improve NCCL communication performance over built-in gigabit Ethernet in our usecase.

We confirmed using PCIe device lending with SmartIO on legacy GPUs is possible. It did help ML communication performance, with some odd results, such as borrowed GPUs in a P2P setup performing better than internal GPUs in a P2P setup. The software is, however, bleeding edge, with a few odd behaviours and a setup process that can easily cause the server to freeze.

And with that we answer our research question with the following:
The value of legacy GPUs in a modern ML and HPC environment is limited as they age, mainly because of software support. However, for HPC environments that do not mind supporting older software packages, they offer cost-effective double-precision performance.

5.3 Future Work

We would like to test the SISCI NCCL plugin to see GPU Direct RDMA performance of the Dolphin cards. Eliminating copies into system memory could elevate a significant bottleneck in the current implementation. The same goes for P2P support between nodes in SmartIO. Furthermore, we would like to see how that compares to the current RDMA implementation of Infiniband.

As mentioned in the scope and limitations section, 1.3, we have used Dolphin interconnect at gen3 x8, and with GPUs at gen2 x16, they somewhat match bandwidth at just below 8GB/s. We would like to test how performance differs from using interconnects and GPUs at gen4 and x16 width, considering they offer around 400% higher theoretical bandwidth. Then compare against our existing benchmarks to confirm the location of bottlenecks in the NCCL communication.

We would also like to test non-synthetic benchmarks. The question remains on how much (or how little) the bandwidth limit between nodes affects performance in real ML workloads using TensorFlow and PyTorch.

Appendix A

Captains log - The adventure of setting up our system

Terminology:
Commands starting with $ are run with normal user privileges.
Commands starting with # are run with root privileges (sudo).

A.1 Initial system

Hardware:
Two desktop computers
CPU: Intel i5-4590 on MSI 297-G55 SLI
RAM: 8GB DDR3
GPU: 2xNVIDIA Quadro K2200, 4GB GDDR5.
Interconnect: Dolphin PXH812

Software:
OS: Ubuntu 22.04.1 LTS server
GPU: CUDA Version 11.7 from NVIDIA HPC SDK 22.9
NTB: Dolphin eXpressWare pipeline 20777
System snapshots: Timeshift

A.1.1 Old setup

Installed Desktop version of Ubuntu 22.04 with proprietary drivers. NVIDIA-drivers with CUDA are included and activated in a graphical environment as such:
Open Software & updates -> Additional Drivers -> "Using NVIDIA driver metapackage from nvidia-driver-515 (proprietary, tested")
Installed NVIDIA HPC SDK (includes NCCL).

A.1.2 Defective IOMMU

Simple trainingjob with tensorflow ran. Latest version of NVIDIA NCCL-tests compiled and ran successfully. A hardware or firmware fault cause

the nodes to crash during boot if IOMMU were enabled in the boot options as such:

```
# vi /etc/default/grub
GRUB_CMDLINE_LINUX_DEFAULT="intel_iommu=on"
# update-grub
```

Updating BIOS to the latest version did not change behaviour.

Without IOMMU, data transfers from GPUs need to be copied into RAM before transfer over the dolphin interconnect. For this reason we changed nodes to ones with working IOMMU. The ones available at the time were a set of legacy headless GPU-servers.

A.2 Final system

Hardware:
Two Supermicro x9 SuperServer 1027GR-TRF, Chassis Model: 118-18, Motherboard X9DRG-HF with a NUMA configuration, illustrated in 3.1
CPU: 2xIntel Xeon E5-2609 @ 2.40GHz
RAM: 64GB DDR3 split between the NUMA nodes
GPU: 2xNVIDIA Tesla K20X, 6GB GDDR5.
Interconnect: Dolphin PXH810

Software
OS: Ubuntu 20.04 LTS server
GPU: NVIDIA-driver-470-server
NVIDIA driver version 470.182.03
CUDA Version 11.4
NVIDIA HPC SDK 21.9
NTB: Dolphin eXpressWare pipeline 22529
System snapshots: Timeshift

A.2.1 New setup

On the first server, named "abel1", each NUMA node (CPU socket) has one GPU. In such a configuration, data transfer between GPUs is trough the Intel QuickPath Interconnect (QPI) between the NUMA nodes.
On the second server, named "abel2", both GPUs are placed on the first NUMA node. Data transfer between GPUs in such configuration are direct (P2P).
In both servers, a Dolphin PXH810 interconnect is connected to the available x8 gen 3 slot on the fist NUMA node, referenced as slot 5 in 3.1.

A.3 Server configuration

A.3.1 Firmware & BMC/IPMI

The BIOS and firmware on the BMC (Baseboard Management Controller, also known as IPMI) and Dolphin PXH810 were outdated. We downloaded newest BIOS and BMC version from Supermicro websites. Installed by booting a USB-pen with freeDOS.
Dolphin PXH810 were updated by running

```
# upgrade_eeprom.sh
```

included in Dolphin eXpressWare after Ubuntu was installed.

To gain access to remote server management (BMC/IPMI), network settings were configured in the BIOS. Our setup used static IP instead of DHCP.

Complications

BMC firmware upgrade caused IP settings to be reset to default. After reapplying the static network configuration through BIOS, the BMC stayed unreachable over the network. From the host OS we downloaded and ran the `IPMICFG` tool from supermicro. When running

```
# ./IPMICFG-Linux.x86_64 -linkstatus
```

and

```
# ./IPMICFG-Linux.x86_64 -vlan
```

the culprit was found. VLAN was preset to 100. This was not obvious as VLAN were not one of the configuration options for the BMC in BIOS. Since we´re not using host-configured VLAN in our network, the problem was solved by disabling this option:

```
# ./IPMICFG-Linux.x86_64 -vlan off
```

A.3.2 Console access

The BMC (Baseboard Management Controller) offers sensor data and remote console through it's web interface. The console requires Java Web Start, a deprecated framework since Java 9, and removed since Java 11. Trying to open the console returns you a JNLP file that modern browsers does not support. Supermicro however provide an app named `IPMIView`, for Windows, Linux, iPadOS and iOS. And we were working from an ARM based Mac.
First try was the iPad app, it can technically run on the Mac as they are binary compatible. The console returned image, but the app did not accept

any keyboard inputs.
Second we tried running the JNLP-file using OpenWebStart, an open source re-implementation of the Java Web Start. It failed with the error message

```
no iKVM64 in java.library. path
```

Third try was using IPMIView App wrapper for MacOS. The installer script failed due to a hashing fault. We corrected the fault and made the installer run to completion. But the App failed to open, with no error message. At this point we'd spent multiple days troubleshooting. So we gave up and accepted that we'd have to walk to the server room if needed. Later we dug out an outdated iPad mini and ran the official `IPMIView` app on it, as shown in figure A.1. This would turn out very useful to quickly reset the servers, as we frequently experienced total system-freezes while setting up SmartIO later on.

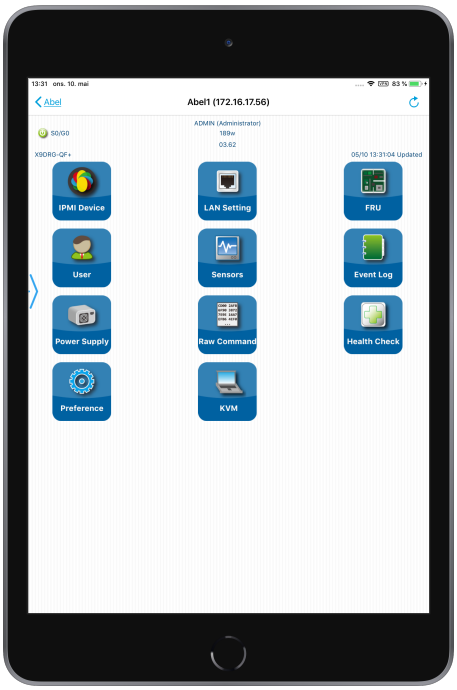


Figure A.1: IPMIView on an iPad

A.3.3 Operating system

Initial trial was with Ubuntu server 22.04 LTS. It proved incompatible with the older NVIDIA HPC SDK that we needed to use for our legacy GPUs (more on this in A.3.8). Compiling compatible version of NVIDIAs NCCL-test code using NVCC for CUDA 11.4 would throw the following error

```
"/usr/include/stdio.h(189): error: attribute __malloc__" does
↪  not take arguments
```

Since stdio.h is distributed as part of the OS, we installed Ubuntu 20.04 LTS on another server to compare against 22.04 LTS. The error went away on the older version, so we downgraded both servers to 20.04 LTS.

Then we check IOMMU support

```
$ sudo dmesg | grep IOMMU
```

If "IOMMU disabled" is listed, we add "`intel_iommu=on`" in /etc/default/grub like this:

```
GRUB_CMDLINE_LINUX_DEFAULT="intel_iommu=on"
```

then activate the change for the next reboot with

```
$ sudo update-grub
```

A.3.4 Snapshot tool

To keep reference points of the system as we tested various drivers and tools, we use `timeshift` to generate snapshots of the filesystem. Restoring from snapshots saved us a tremendous amount of time from having to reinstall the OS whenever things broke, and let us easily switch between known configurations during testing. It's installed as such:

```
$ sudo apt install timeshift
```

Some commands:

```
# timeshift --snapshot-device /dev/sda           #Configures
↪  sda as the mount point where backups are stored. Relevant
↪  if systemdisk (ssd) and storage disk (hdd) are different.
# timeshift --create --comments "A new backup"   #Create a
↪  snapshot
# timeshift --list                               #List
↪  snapshots
# timeshift --restore                            #Restore to a
↪  snapshot.
```

Restores does not include anything inside /home and /root folder. If restore is needed, it won't restore or change configurations in .profile or ssh-keys.

A.3.5 Network

Network was configured by modifying a yaml config file for use with netplan.

```
# vi /etc/netplan/*.yaml
```

Static IP were configured for both ethernet (eth0, eth1) and the Dolphin PXH810 (dis0) There were no redundancy, so only one of the two available ethernet ports were in use. A fault in our Supermicro x9 servers would

cause the renaming of network interface to not stick to one port. During boot, at random either eth0 or eth1 were renamed to eno1. Not practical when using just one port. This was solved by disabling renaming in grub by adding

```
GRUB_CMDLINE_LINUX="net.ifnames=0".
```

into

```
/etc/default/grub
```

If iommu is also on, it is

```
GRUB_CMDLINE_LINUX="net.ifnames=0 intel_iommu=on"
```

Then run

```
$ sudo update-grub
```

And for good measure, eth0 and eth1 were bonded in active-backup mode. Below is an example of /etc/netplan/*.yaml with with bonding for ethernet:

```
network:
  bonds:
    bond0:
      dhcp4: false
      addresses:
      - 10.174.0.57/22
      routes:
      - to: default
        via: 10.174.0.1
      interfaces:
      - eth0
      - eth1
      nameservers:
        addresses:
        - 172.16.16.57
        - 1.1.1.1
        - 8.8.8.8
        search:
        - simula.no
      parameters:
        mode: active-backup
        mii-monitor-interval: 60s
  ethernets:
    eth0:
     dhcp4: false
```

```
    optional: true
   eth1:
    dhcp4: false
    optional: true
 version: 2
```

To activate a new netplan configuration, run

```
$ sudo netplan apply.
```

A.3.6 Hosts

We made hostnames for a more human readable experience instead of using IPs. These were not configured in DNS, so we manually add the hostnames and their IPs on both servers by modifying

```
/etc/hosts
```

For example, we added the following:

```
10.174.0.55 abel1.simula.no     abel1           #Ethernet port
10.174.0.56 abel1-bmc.simula.no abel1-bmc       #Server
↪  management
10.174.0.57 abel2.simula.no     abel2           #Ethernet port
10.174.0.58 abel2-bmc.simula.no abel2-bmc       #Server
↪  management
10.0.0.4 abel1-dis.cluster      abel1-dis0      #Dolphin
↪  interconnect port
10.0.0.8 abel2-dis.cluster      abel2-dis0      #Dolphin
↪  interconnect port
```

A.3.7 SSH

For easy login between the nodes, .ssh/config was configured as such:

```
Host abel1
        User audunjoh
        Hostname 10.174.0.55

Host abel2
        User audunjoh
        Hostname 10.174.0.57
```

MPI and the Dolphin driver installer need password-less SSH to be set up between all nodes in order to work. So we generate and distribute a public SSH key for both the user and root onto the authorized_keys file across all nodes, including it self:

```
# ssh-copy-id user@host1
# ssh-copy-id user@host2
...
# ssh-copy-id root@host1
# ssh-copy-id root@host2
...
```

A.3.8 NVIDIA drivers & tools

Driver branch 470 is the latest one to still support Tesla K20X GPUs. We install the drivers, diagnostic tools and other tools needed as such

```
# apt install nvidia-headless-470-server
# apt install nvidia-utils-470-server
# apt install environment-modules
```

For visual monitoring of the GPU load, we used the AppImage version of nvtop We avoided the apt version since it included a load of older NVIDIA libraries as dependencies, cluttering any debugging later on.
To make it easy to run every time we login, we made an alias in the .profile file pointing to the executable

```
# Alias for nvtop AppImage
alias nvtop='~/./nvtop-2.0.4-x86_64.AppImage'
```

For our project we need CUDA, NVCC, NCCL and MPI. NVIDIAs HPC SDK includes all this as well as environment modules that makes setting it up across nodes consistent and stable. For our Tesla K20X GPUs, the latest working SDK version is 21.9, with CUDA 11.4. Instructions for adding their repository and downloading it can be found on the NVIDIA HPC SDK 21.9 site. Installation was done as following:

```
$ curl https://developer.download.nvidia.com/hpc-sdk/ubuntu/⌋
↪  DEB-GPG-KEY-NVIDIA-HPC-SDK | sudo gpg --dearmor -o
↪  /usr/share/keyrings/nvidia-hpcsdk-archive-keyring.gpg
$ echo 'deb [signed-by=/usr/share/keyrings/⌋
↪  nvidia-hpcsdk-archive-keyring.gpg]
↪  https://developer.download.nvidia.com/hpc-sdk/ubuntu/amd64
↪  /' | sudo tee /etc/apt/sources.list.d/nvhpc.list
$ sudo apt update -y
$ sudo apt install -y nvhpc-21-9
```

After the install, we load in the environment module as such

```
$ module use /opt/nvidia/hpc_sdk/modulefiles
$ module load nvhpc/21.9
```

This way, the location of the the various SDK libraries such as CUDA, NCCL and MPI are added to the `LD_LIBRARY_PATH` for all software to find. As well as relevant programs such as NVCC and mpirun are added to PATH, so we can call and run them without prompting the path of the executable. To list available environment modules, we can run

```
$ module avail
```

To load the modules automatically when interactively SSHing into the nodes, we inserted the following into the users .profile file

```
# set module so it includes NVIDIA HPC libraries if it exists
if [ -d "/opt/nvidia/hpc_sdk/" ] ; then
        module use /opt/nvidia/hpc_sdk/modulefiles
        module load nvhpc/21.9 -v
fi
```

For testing various CUDA capabilities, CUDA samples provided as part of the CUDA toolkit are good resources. The various code-samples include makefiles for easy building and running. On our setup, we must pass trough arguments for our non-standard compiler and library locations as well as compute architecture. Example:

```
$ make NVCC=/opt/nvidia/hpc_sdk/Linux_x86_64/21.9/compilers/
↪  bin/nvcc
↪  CUDA_PATH=/opt/nvidia/hpc_sdk/Linux_x86_64/21.9/cuda/11.4/
↪  SMS="35" -j8
```

With SmartIO, the borrowed GPUs appear as if local. So our method to ensure that benchmarks run on borrowed GPUs only was to deactivate the local GPUs as such:
To deactivate a GPU on PCIe address `0000:85:00.0`:

```
$ sudo nvidia-smi drain -p 0000:85:00.0 -m 1
```

To reactivate:

```
$ sudo nvidia-smi drain -p 0000:85:00.0 -m 0
```

cuDNN

To accelerate deep learning with TensorFlow on our GPUs, we install NVIDIA cuDNN. While we didn't get the time to properly run any training on the final system, we installed it so it was ready.
Add repository and install:

```
$ sudo apt-key adv --fetch-keys
 ↪  https://developer.download.nvidia.com/compute/cuda/repos/
 ↪  ubuntu2004/x86_64/3bf863cc.pub
$ sudo add-apt-repository "deb https://developer.download.
 ↪  nvidia.com/compute/cuda/repos/ubuntu2004/x86_64/
 ↪  /"
$ sudo apt install libcudnn8
$ sudo apt install libcudnn8-dev
```

For tests, download deb-package. Extract it, then extract `cudnn_samples_v8` within. To compile the tests on our setup:

```
$ make NVCC=/opt/nvidia/hpc_sdk/Linux_x86_64/21.9/compilers/
 ↪  bin/nvcc
 ↪  CUDA_PATH=/opt/nvidia/hpc_sdk/Linux_x86_64/21.9/cuda/11.4
```

Complications

The K20X GPUs support at most CUDA capability 3.5 and CUDA 11.4. Initially we tried to use the newest SDK. First 22.7 and later 22.9 as it was released. Specifically the multipack version including CUDA 10.2, 11.0 and 11.7. However they are missing environment module files for anything but CUDA 11.7. So when we do

```
$ module load nvhpc/22.9
```

We'd get nvcc for CUDA 11.7

```
$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2022 NVIDIA Corporation
Built on Wed_Jun__8_16:49:14_PDT_2022
Cuda compilation tools, release 11.7, V11.7.99
Build cuda_11.7.r11.7/compiler.31442593_0
```

If we load module nvhpc/22.9, and try to compile NCCL-test while pointing it to CUDA 11.0 included in 22.9, we'd get pgc++ errors

```
$ make
 ↪  CUDA_HOME=/opt/nvidia/hpc_sdk/Linux_x86_64/22.9/cuda/11.0/
 ↪  NCCL_HOME=/opt/nvidia/hpc_sdk/Linux_x86_64/22.9/
 ↪  comm_libs/nccl
 ↪  NVCC_GENCODE="-gencode=arch=compute_35,code=sm_35"
make -C src build
make[1]: Entering directory
 ↪  '/home/audunjoh/cudastuff/nccl-tests/src'
Compiling  all_reduce.cu                        >
 ↪  ../build/all_reduce.o
```

```
nvcc warning : The 'compute_35', 'compute_37', 'compute_50',
 ↪  'sm_35', 'sm_37' and 'sm_50' architectures are deprecated,
 ↪  and may be removed in a future release (Use
 ↪  -Wno-deprecated-gpu-targets to suppress warning).
nvc++-Warning-CUDA_HOME has been deprecated. Please, use
 ↪  NVHPC_CUDA_HOME instead.
nvc++-Warning-CUDA_HOME has been deprecated. Please, use
 ↪  NVHPC_CUDA_HOME instead.
"/opt/nvidia/hpc_sdk/Linux_x86_64/22.9/cuda/11.0//bin/../
 ↪  targets/x86_64-linux/include/crt/host_config.h", line 118:
 ↪  catastrophic error: #error directive: -- unsupported pgc++
 ↪  configuration! Only pgc++ 18, 19 and 20 are supported!
  #error -- unsupported pgc++ configuration! Only pgc++ 18, 19
 ↪  and 20 are supported!
   ^

1 catastrophic error detected in the compilation of
 ↪  "all_reduce.cu".
Compilation terminated.
```

Even if we try pointing to the known working CUDA 11.4 from 21.9 while using nvcc loaded from 22.9, We'd get the following

```
$ make
 ↪  CUDA_HOME=/opt/nvidia/hpc_sdk/Linux_x86_64/21.9/cuda/11.4/
 ↪  NCCL_HOME=/opt/nvidia/hpc_sdk/Linux_x86_64/21.9/
 ↪  comm_libs/nccl
 ↪  NVCC_GENCODE="-gencode=arch=compute_35,code=sm_35"
make -C src build
make[1]: Entering directory
 ↪  '/home/audunjoh/cudastuff/nccl-tests/src'
Compiling  all_reduce.cu                        >
 ↪  ../build/all_reduce.o
nvcc warning : The 'compute_35', 'compute_37', 'compute_50',
 ↪  'sm_35', 'sm_37' and 'sm_50' architectures are deprecated,
 ↪  and may be removed in a future release (Use
 ↪  -Wno-deprecated-gpu-targets to suppress warning).
nvc++-Warning-CUDA_HOME has been deprecated. Please, use
 ↪  NVHPC_CUDA_HOME instead.
nvc++-Warning-CUDA_HOME has been deprecated. Please, use
 ↪  NVHPC_CUDA_HOME instead.
"/opt/nvidia/hpc_sdk/Linux_x86_64/21.9/cuda/11.4//bin/../
 ↪  targets/x86_64-linux/include/crt/host_config.h", line 118:
 ↪  catastrophic error: #error directive: -- unsupported pgc++
 ↪  configuration! Only pgc++ 18, 19, 20 and 21 are supported!
 ↪  The nvcc flag '-allow-unsupported-compiler' can be used to
 ↪  override this version check; however, using an unsupported
 ↪  host compiler may cause compilation failure or incorrect
 ↪  run time execution. Use at your own risk.
```

```
  #error -- unsupported pgc++ configuration! Only pgc++ 18,
 ↪  19, 20 and 21 are supported! The nvcc flag
 ↪  '-allow-unsupported-compiler' can be used to override this
 ↪  version check; however, using an unsupported host compiler
 ↪  may cause compilation failure or incorrect run time
 ↪  execution. Use at your own risk.
    ^

1 catastrophic error detected in the compilation of
 ↪  "all_reduce.cu".
Compilation terminated.
```

If we use SDK 21.9, we don't get pgc++ errors.

Originally the .deb instructions for 21.9 release had an outdated gpg-signature. We solved it by adding the newest gpg-signature from the latest SDK release, then installing the old version. After an email exchange with NVIDIA, they updated the instructions and signatures on their sites.

At one point we discovered that the benchmark results was off compared to earlier runs. We found that the GPU on Address: 0000:05:00.0 in abel2 reported x4 width, instead of x16. We extracted this info by reading the ports PCIe generation and width by running

```
$ sudo dmidecode --type 9"
```

One of the ports with GPU plugged in reported "PCI-E 3.0 X4 (IN X8 SLOT)", despite being an x16 port. Similar info from `nvidia-smi` with

```
$ nvidia-smi
 ↪  --query-gpu=pcie.link.width.max,pcie.link.width.current
 ↪  --format=csv
```

Solution to get it back to x16 width was to physically reseat the GPU.

A.3.9 NCCL Tests

The code to benchmark NCCL performance and correctness is distributed by NVIDIA and called nccl-tests. In order to compile to our legacy GPUs using NVCC for CUDA 11.4, the newest release was incompatible. It includes newer commands that would not compile on the slightly older NVIDIA C compiler. The newest verified git-release that compiled and worked on our system was git-commit 8274cb4 (27 May 2022), so we went ahead and used that as such:

```
$ git checkout 8274cb4
```

In the makefile `nccl-tests/src/Makefile` NVIDIA has added a check for CUDA capability to define NVCC_GENCODE in order to reduce compile time. After the release of CUDA 11, we see that the makefile was modified to check the CUDA version of the NVCC-compiler. If it's CUDA 11, it will only compile to GPUs with CUDA capability 6.0 and newer. Our GPUs were at CUDA capability 3.5, so in order to compile the code using CUDA 11 we could either modify the makefile, or manually override by adding

```
NVCC_GENCODE="-gencode=arch=compute_35,code=sm_35"
```

as a compiler argument. We chose to manually override.

When using NVIDIAs HPC SDK, the locations for CUDA, NCCL and MPI libraries are not in the default location. As an example, NCCL test only look for the CUDA library in `/usr/local/cuda`, while the HPC SDK puts it at `/opt/nvidia/hpc_sdk/Linux_x86_64/21.9/cuda/`. This means all non-standard locations must be explicitly defined as an argument when compiling. Example compiling nccl-tests without MPI:

```
$ make
↪  CUDA_HOME=/opt/nvidia/hpc_sdk/Linux_x86_64/21.9/cuda/11.4/
↪  NCCL_HOME=/opt/nvidia/hpc_sdk/Linux_x86_64/21.9/
↪  comm_libs/nccl
↪  NVCC_GENCODE="-gencode=arch=compute_35,code=sm_35 -j8"
```

Example run of `all_reduce_perf` without MPI (local only):

```
$ ~/cudastuff/nccl-tests/build/all_reduce_perf -b 8 -e 128M -f
↪  2 -g 2
```

-g are number of GPUs.

Thus to compile NCCL test with MPI for multi node testing using the NVIDIA HPC SDK, we ran the following

```
$ make MPI=1 MPI_HOME=/opt/nvidia/hpc_sdk/Linux_x86_64/21.9/
↪  comm_libs/mpi
↪  CUDA_HOME=/opt/nvidia/hpc_sdk/Linux_x86_64/21.9/cuda/11.4/
↪  NCCL_HOME=/opt/nvidia/hpc_sdk/Linux_x86_64/21.9/
↪  comm_libs/nccl
↪  NVCC_GENCODE="-gencode=arch=compute_35,code=sm_35"
```

To speed up the compilation time, `-j8` can be added to compile using all 8 threads in our system.

Compiling with MPI also means we have to run the program with mpirun. To run two MPI processes on a single node:

```
$ mpirun -np 2 -host abel1:2
↪  ~/cudastuff/nccl-tests/build/all_reduce_perf -b 8 -e 128M
↪  -f 2 -g 1
```

To run four MPI processes across our two nodes

```
$ mpirun -np 4 -host abel1:2,abel2:2 -x LD_LIBRARY_PATH
↪  ~/cudastuff/nccl-tests/build/all_reduce_perf -b 8 -e 128M
↪  -f 2 -g 1
```

MPI args:
-np are number of MPI processes.
-x export our specified environment variables to the remote nodes before executing the program. In this case: `"LD_LIBRARY_PATH"`. This is critical since MPI SSH onto each node in an non-interactive way. This means the .profile won't run, thus the environment modules won't be loaded, thus the program wont find NCCL or CUDA in `"LD_LIBRARY_PATH"` on the remote nodes.
"PATH" may also be added if we want to access executable programs that are loaded from the environment modules. To test this, here is an example that returns the location of nvcc:

```
$ mpirun -np 4 -host abel1:2,abel2:2 -x PATH -x
↪  LD_LIBRARY_PATH which nvcc
```

Note that this method only works if the libraries and programs are installed at identical locations on all nodes. I learned this on hpc.uni.lu/old/users/howtos/UsingMPIstacksWithModules.html.
Each of our nodes has only two GPUs. Therefore, each node can either run two processes with one GPU: `"-np 2 -g 1"`, or one process with two GPUs: `"np 1 -g 2"`.

Complications

It was time-consuming to find the source of faults causing NCCL test to not compile. After some time researching and trying to debug the source of the various compiler errors, we just began exploiting the repository version history in git. We tested an earlier commit from around the same time as the release of the SDK we used. They compiled without a hitch and confirmed that the compilation errors were caused by changes in the code rather than a fault in our setup. After that, we started from the head and went backwards until we found the newest verified commit that would compile. However, as we got compiler errors out of the way, the executable returned a runtime error

```
Test CUDA failure common.cu:381 'no kernel image is available
↪  for execution on the device'
```

Turns out the makefile is set up to not compile to our GPUs gencode 35 if compiler happens to be for CUDA 11. After searching trough the issue-pages of the NCCL test github repository , we found a single post with with the same error message. The solution given was to explicitly define our old NVCC_GENCODE during compilation. Thus, by using an older version of the code while overriding the NVCC_GENCODE, we're finally able to use the official NCCL synthetic benchmarking tool.

The following is an example of what errors arrive when trying to compile the latest NCCL test version (commit 365b92a as of writing) with `nvcc release 11.4, V11.4.100, Build cuda_11.4.r11.4/compiler.30188945_0`.

```
$ make
 ↪  CUDA_HOME=/opt/nvidia/hpc_sdk/Linux_x86_64/21.9/cuda/11.4/
 ↪  NCCL_HOME=/opt/nvidia/hpc_sdk/Linux_x86_64/21.9/
 ↪  comm_libs/nccl
 ↪  NVCC_GENCODE="-gencode=arch=compute_35,code=sm_35"
make -C src build
 ↪  BUILDDIR=/home/audunjoh/cudastuff/nccl-tests/build
make[1]: Entering directory
 ↪  '/home/audunjoh/cudastuff/nccl-tests/src'
Compiling  timer.cc                              >
 ↪  /home/audunjoh/cudastuff/nccl-tests/build/timer.o
Compiling /home/audunjoh/cudastuff/nccl-tests/build/
 ↪  verifiable/verifiable.o
nvcc warning : The 'compute_35', 'compute_37', 'compute_50',
 ↪  'sm_35', 'sm_37' and 'sm_50' architectures are deprecated,
 ↪  and may be removed in a future release (Use
 ↪  -Wno-deprecated-gpu-targets to suppress warning).
../verifiable/verifiable.cu(124): warning: function
 ↪  "<unnamed>::castTo<Y>(float) [with Y=__nv_bfloat16]" was
 ↪  declared but never referenced

../verifiable/verifiable.cu(119): warning: function
 ↪  "<unnamed>::castTo<Y>(float) [with Y=half]" was declared
 ↪  but never referenced

../verifiable/verifiable.cu(147): warning: function
 ↪  "<unnamed>::ReduceSum::operator()(half, half) const" was
 ↪  declared but never referenced

../verifiable/verifiable.cu(155): warning: function
 ↪  "<unnamed>::ReduceSum::operator()(__nv_bfloat16,
 ↪  __nv_bfloat16) const" was declared but never referenced

"../verifiable/verifiable.cu", line 353: error: expected a ")"
  return (uint64_t)((((unsigned __int128)a) * ((unsigned
 ↪  __int128)b)) >> 64);
```

```
                                  ^

"../verifiable/verifiable.cu", line 353: error: expected a ")"
  return (uint64_t)((((unsigned __int128)a) * ((unsigned
 ↪  __int128)b)) >> 64);
                                                        ^

"../verifiable/verifiable.cu", line 353: warning: shift count
 ↪  is too large
  return (uint64_t)((((unsigned __int128)a) * ((unsigned
 ↪  __int128)b)) >> 64);

 ↪  ^

2 errors detected in the compilation of
 ↪  "/tmp/tmpxft_00001276_00000000-6_verifiable.cudafe1.cpp".
make[1]: *** [../verifiable/verifiable.mk:11: /home/audunjoh/
 ↪  cudastuff/nccl-tests/build/verifiable/verifiable.o] Error
 ↪  2
make[1]: Leaving directory
 ↪  '/home/audunjoh/cudastuff/nccl-tests/src'
make: *** [Makefile:20: src.build] Error 2
```

A.3.10 Dolphin eXpressWare

Dolphin install notes:

```
$ sudo bash
 ↪  Dolphin_eXpressWare-Linux-x86_64-PX-66aa356545_c0e0d090cc.
 ↪  ubuntu20.04.sh --disable-gui --enable-smartio
 ↪  --enable-supersockets

Explanation:
--disable-gui           #Graphical interface. Disabled since
 ↪  the server is headless.
--enable-smartio        #Enables smartIO functionality.
--enable-supersockets   #Enables SuperSockets
```

Beware: if the kernel is updated after the Dolphin driver is installed, it will break. Solution is to run the install script again to build against the new kernel.

To automatically load the various Dolphin tools into PATHs when interactively logging in with SSH, we inserted the following into the users .profile file

```
# set PATH so it includes DIS sbin if it exists
if [ -d "/opt/DIS/sbin" ] ; then
    PATH="/opt/DIS/sbin:$PATH"
fi
```

For status:

```
$ dis_services status -l
```

SmartIO

Prefetch: Default prefetch size on the PX card is 512MB. This can be seen by running `dis_config`. The prefetch size for the GPUs can be found running `lspci -vs [pci device ID]`. In our setup, we observer that one Tesla K20X need 256M prefetched memory space. In theory, with 512MB prefetch for the PX card, we have just enough for two GPUs. However if we try to borrow two GPUs using smartIO, the second will fail. A look into `dmesg` reveal

```
BAR X: no space for [mem size 0x10000000 64bit pref]
BAR X: failed to assign [mem size 0x10000000 64bit pref]
```

A look into the SmartIO manual informs that "The sum of PCIe BAR sizes + natural alignment for all added devices must be smaller than the prefetch space allocated by the Dolphin NTB board."

To give us some ample headroom, we increase the the PX cards prefetch allocation to 4096MB. This would for older systems be the limit, however if we wanted our systems could also handle more, as it supports "Above 4G Decoding". It enables 64bit capable Devices to be Decoded in Above 4G Address Space. This is needed for newer GPUs that support Resizable BAR (Base Address Register).

For debugging SmartIO, we run

```
$ dmesg
```

This is how we used SmartIO to borrow GPUs between the nodes. Based on documentation from dolphinics.no.

We can check with nvidia-smi that each of our two nodes contain two GPUs.
abel1:

```
abel1$ nvidia-smi
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.182.03   Driver Version: 470.182.03   CUDA Version: 11.4     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
```

```
|===============================+======================+======================|
|   0  Tesla K20Xm         Off  | 00000000:05:00.0 Off |                    0 |
| N/A   32C    P0    55W / 235W |      0MiB /  5700MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   1  Tesla K20Xm         Off  | 00000000:85:00.0 Off |                    0 |
| N/A   29C    P0    58W / 235W |      0MiB /  5700MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
```

abel2:

```
abel2$ nvidia-smi
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.182.03   Driver Version: 470.182.03   CUDA Version: 11.4     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla K20Xm         Off  | 00000000:04:00.0 Off |                    0 |
| N/A   26C    P0    55W / 235W |      0MiB /  5700MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   1  Tesla K20Xm         Off  | 00000000:05:00.0 Off |                    0 |
| N/A   31C    P0    57W / 235W |      0MiB /  5700MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
```

Then we do the following, slowly one by one, to borrow and lend two GPUs from one of the nodes.

```
// Lender side, if it's abel2:
// Connect to abel1
$ sudo /opt/DIS/sbin/smartio_tool connect 4
// Get PCI-addresses to the GPUs
$ lspci | grep NVIDIA
// Add them to the lender list
$ sudo /opt/DIS/sbin/smartio_tool add 04:00.0
$ sudo /opt/DIS/sbin/smartio_tool add 05:00.0
// Make them available for borrowers
$ sudo /opt/DIS/sbin/smartio_tool available 04:00.0
$ sudo /opt/DIS/sbin/smartio_tool available 05:00.0

// Borrower side:
// We begin by stopping the sisci service
$ sudo systemctl stop dis_sisci
// Then we list available GPUs to borrow
$ sudo /opt/DIS/sbin/smartio_tool list
// We borrow the two GPUs with their ID and the DMA window
 ↪  size
$ sudo /opt/DIS/sbin/smartio_tool borrow 80400 1024
$ sudo /opt/DIS/sbin/smartio_tool borrow 80500 1024
// Then we enable p2p between the remote GPUs so they locally
 ↪  can talk directly with each other
$ sudo /opt/DIS/sbin/smartio_tool enable-p2p 80400 80500
```

```
$ sudo /opt/DIS/sbin/smartio_tool enable-p2p 80500 80400
// Now for us to use the remote GPUs, we must reload the
 ↪  NVIDIA kernel module. We must unload in the order of its
 ↪  dependencies as seen by running "$ lsmod | grep nvidia"
$ sudo modprobe --remove nvidia_uvm nvidia_drm nvidia_modeset
 ↪  nvidia
// Then reload it back in again. Dependencies will follow
 ↪  along
$ sudo modprobe nvidia
// And finally, start the sisci service again
$ sudo systemctl start dis_sisci
// To confirm that the lending was sucessfull, we check that
 ↪  the remote GPUs are listed
$ nvidia-smi
```

The borrower node now display four GPUs:

```
abel1:~$ nvidia-smi
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.182.03   Driver Version: 470.182.03   CUDA Version: 11.4     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla K20Xm         Off  | 00000000:03:04.0 Off |                    0 |
| N/A   28C    P0    54W / 235W |      0MiB /  5700MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   1  Tesla K20Xm         Off  | 00000000:03:05.0 Off |                    0 |
| N/A   33C    P0    56W / 235W |      0MiB /  5700MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   2  Tesla K20Xm         Off  | 00000000:05:00.0 Off |                    0 |
| N/A   32C    P0    56W / 235W |      0MiB /  5700MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   3  Tesla K20Xm         Off  | 00000000:85:00.0 Off |                    0 |
| N/A   29C    P0    57W / 235W |      0MiB /  5700MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
```

SmartIO complications

If the series of smartio_tool commands for lender and borrower are input and executed too quickly after each other, for example by pasting a block of them into the terminal, it may appear to have worked. `nvidia-smi` may show external GPUs as expected. However CUDA-programs that try to run on them will fail quickly. `p2pBandwidthLatencyTest` from CUDA-samples will return:

```
Cuda failure p2pBandwidthLatencyTest.cu:610: 'unknown error'
```

NCCL-benchmarks will also report 'unknown error', state it exited with non-zero status and segmentation fault:

```
$ bash ~/sisci-nccl-benchmark/nccl-tests-benchmark/
↪  Benchmarking-scripts/
↪  SmartIO_2external0internalGPU_NCCLtests2CSV.sh
Running abel1_2xMPI_0local2externalGPU-broadcast_perf
# nThread 1 nGpus 1 minBytes 8 maxBytes 1073741824 step:
↪  2(factor) warmup iters: 5 iters: 20 validation: 1
# Parallel Init Enabled: threads call into NcclInitRank
↪  concurrently
#
# Using devices
abel1: Test CUDA failure common.cu:1045 'unknown error'
 .. abel1 pid 3326: Test failure common.cu:1007
abel1: Test CUDA failure common.cu:1045 'unknown error'
 .. abel1 pid 3325: Test failure common.cu:1007
--------------------------------------------------------------------------
Primary job  terminated normally, but 1 process returned
a non-zero exit code. Per user-direction, the job has been
↪  aborted.
--------------------------------------------------------------------------
--------------------------------------------------------------------------
mpirun detected that one or more processes exited with
↪  non-zero status, thus causing
the job to be terminated. The first process to do so was:

  Process name: [[39348,1],0]
  Exit code:    2
--------------------------------------------------------------------------
Running abel1_2xMPI_0local2externalGPU-all_reduce_perf
# nThread 1 nGpus 1 minBytes 8 maxBytes 1073741824 step:
↪  2(factor) warmup iters: 5 iters: 20 validation: 1
# Parallel Init Enabled: threads call into NcclInitRank
↪  concurrently
#
# Using devices
abel1: Test CUDA failure common.cu:1045 'unknown error'
 .. abel1 pid 3352: Test failure common.cu:1007
abel1: Test CUDA failure common.cu:1045 'unknown error'
 .. abel1 pid 3351: Test failure common.cu:1007
--------------------------------------------------------------------------
Primary job  terminated normally, but 1 process returned
a non-zero exit code. Per user-direction, the job has been
↪  aborted.
--------------------------------------------------------------------------
--------------------------------------------------------------------------
mpirun detected that one or more processes exited with
↪  non-zero status, thus causing
the job to be terminated. The first process to do so was:
```

```
  Process name: [[39007,1],0]
  Exit code:    2
--------------------------------------------------------------------------
Running abel1_2xMPI_0local2externalGPU-all_gather_perf
# nThread 1 nGpus 1 minBytes 8 maxBytes 1073741824 step:
 ↪  2(factor) warmup iters: 5 iters: 20 validation: 1
# Parallel Init Enabled: threads call into NcclInitRank
 ↪  concurrently
#
# Using devices
abel1: Test CUDA failure common.cu:1045 'unknown error'
 .. abel1 pid 3379: Test failure common.cu:1007
abel1: Test CUDA failure common.cu:1045 'unknown error'
 .. abel1 pid 3378: Test failure common.cu:1007
--------------------------------------------------------------------------
Primary job  terminated normally, but 1 process returned
a non-zero exit code. Per user-direction, the job has been
 ↪  aborted.
--------------------------------------------------------------------------
^Cmpirun: abort is already in progress...hit ctrl-c again to
 ↪  forcibly terminate

[abel1:03374] *** Process received signal ***
[abel1:03374] Signal: Segmentation fault (11)
[abel1:03374] Signal code: Address not mapped (1)
[abel1:03374] Failing at address: 0x30
[abel1:03374] *** End of error message ***
/opt/nvidia/hpc_sdk/Linux_x86_64/21.9/comm_libs/mpi/bin/
 ↪  mpirun: line 15:  3374 Segmentation fault      (core
 ↪  dumped) $MY_DIR/.bin/$EXE "$@"
```

The "solution" is to wait a second or two between executing each `smartio_tool` command. We suspect `smartio_tool` is exposed to race conditions.

Unloading the NVIDIA kernel module on the lender side will cause the server to freeze as soon as a GPU is added to the SmartIO list:

```
// Freeze reproduction #1:
$ sudo modprobe --remove nvidia_uvm nvidia_drm nvidia_modeset
 ↪  nvidia
$ sudo /opt/DIS/sbin/smartio_tool add 05:00.0
(server freezes here)

// Freeze reproduction #2:
$ sudo /opt/DIS/sbin/smartio_tool add 05:00.0
```

```
$ sudo modprobe --remove nvidia_uvm nvidia_drm nvidia_modeset
↪  nvidia
$ sudo /opt/DIS/sbin/smartio_tool remove 05:00.0
$ sudo /opt/DIS/sbin/smartio_tool add 05:00.0
(server freezes here)
```

Solution was to not bother with unloading the NVIDIA kernel module on the lender side.

CUDA and Dolphin complications

CUDA is an option in the eXpressWare installation. We initially thought this was necessary to set up for our benchmarks to run over the PXH810, but it turned out not to be relevant unless we modified NCCL test code to add SISCI API calls. GPUDirect over Dolphin ICS SISCI might have increased the bandwidth and lowered latency; alas we did not have time to explore this. The following is the tale of us trying to get CUDA setup during the eXpressWare installation:

The install script for Dolphin eXpressWare expects very specific paths and preconditions. This caused some issues when trying to enable CUDA on our setup.

#1. The script appear not to be made with server(headless)-versions of the GPU driver in mind:

```
ERROR: Couldn't find
↪  '.*/((nvidia-([0-9]+\.?)+(/nvidia)?)|nvgpu/.*)/nv-p2p.h$'
↪  in '/usr/src/'
```

Cause:
Script is unable to locate `/usr/src/nvidia-srv-470.141.03`, because of "`-srv`" in the folder name.
Solution:
Copy and rename `/usr/src/nvidia-srv-470.141.03` to `/usr/src/nvidia-470.141.10`.
Or specify correct path to the configuration as shown in solution for error #3.

#2. The script doesn't compile any 3.party tools it needs from the GPU driver. The user must do this manually.

```
ERROR: Couldn't find '.*/Module.symvers$' in
↪  '/usr/src/nvidia-470.141.03'
```

Cause:
Source code in `/usr/src/nvidia-470.141.10` are not compiled.
Solution:
Go in /usr/src/nvidia-srv-470.141.10, run `make` to compile headers and

modules so 3.party drivers, such as eXpressWare, can access GPU functionality. `peermem` and `uvm` are of interest for our dolphin setup.

#3. The script check for CUDA in a specific path, without an option to pass trough alternative locations as an argument.

```
ERROR: Couldn't find '.*/cuda-[0-9.]+$' in '/usr/local'
```

Cause:
CUDA is located elsewhere when using NVIDIAs HPC SDK. Location on our setup is `/opt/nvidia/hpc_sdk/Linux_x86_64/21.9/cuda/11.4/`
Solution:
The script needs to be dismantled and content within modified and manually installed. It starts by extracting the source of the eXpressWare install script by running

```
$ bash
 ↪  Dolphin_eXpressWare-Linux-x86_64-PX-66aa356545_c0e0d090cc.
 ↪  ubuntu20.04.sh
 ↪  --get-source
```

Then in the DIS folder

```
$ sudo ./configure --with-adapter=PX --with-cuda-api=/opt/
 ↪  nvidia/hpc_sdk/Linux_x86_64/21.9/cuda/11.4/
 ↪  --with-cuda-drv=/usr/src/nvidia-srv-470.161.03
 ↪  --disable-gui --enable-sisci-development --enable-smartio
 ↪  --enable-supersockets --enable-cuda-support

Explanation:
--with-adapter=PX       #The card in use is a PXH810
--with-cuda-api         #CUDA location
--with-cuda-drv         #3.party GPU tools location
--disable-gui           #Graphical interface. Disabled since
 ↪  the server is headless.
--enable-smartio        #Enables smartIO functionality.
--enable-supersockets   #Enables SuperSockets and IPoPCIe.
```

Then

```
$ sudo make -j8 && sudo make install -j8

Explanation:
make -j8                #Run make using 8 CPU threads.
```

Then

```
$ cd /opt/DIS/sbin && sudo su
```

On the head-node run all the *-setup files with

```
$ for i in *-setup; do ./$i -i; done
```

On the other nodes, run all the *-setup, except `networkmgr-setup`. This is because only one node in the cluster may run `dis_networkmgr`. Then run

```
$ dis_config
```

to change prefetch size. Then

```
$ sudo /opt/DIS/sbin/dis_mkconf -fabrics 2 -stt 2 -nodes abel1
↪  abel2
```

to generate cluster configuration. Then try

```
$ sudo /opt/DIS/sbin/dis_services restart
```

Run

```
$ dis_services status -l
```

for status. If there are errors in the status, try a reboot.

A.3.11 Tensorflow

We explored how to get TensorFlow up and running on the original desktop setup. These are the notes on how we did that:
To install and use Tensorflow in virtual python environment:
Setup virtual environment

```
$ sudo apt install python3.8-venv
$ mkdir tf-demo
$ cd tf-demo
$ python3 -m venv tensorflow-dev
```

To enter virtual environment:

```
$ source tensorflow-dev/bin/activate
```

To exit virtual environment:

```
$ deactivate
```

Now install tensorflow

```
$ pip install tensorflow
```

Test that GPUs are seen by tensorflow by running the following python code:

```
    import tensorflow as tf
    print("TensorFlow version: ", tf.__version__)
    tf.test.gpu_device_name()
```

A.3.12 Misc notes

For hardware localisation, such as PCI-e devices per NUMA node:

```
$ sudo apt install hwloc
$ lstopo-no-graphics -.ascii
```

Driver support compared to CUDA version can be found at docs.nvidia.com/cuda/cuda-toolkit-release-notes.
Good info on compiling for correct gencode and architecture can be found at: arnon.dk/matching-sm-architectures-arch-and-gencode-for-various-nvidia-cards.

IPoPCIe setup: TCP/IP over PCIe is included when installing SuperSockets. Network interface dis0 should then be discoverable. Setup IP on the iterface with i.e netplan or network-manager.

To get a live feed of what PCIe-gen is running on the GPUs, run:

```
$ nvidia-smi --query-gpu=timestamp,name,pci.bus_id,pstate,
↪  pcie.link.gen.max,pcie.link.gen.current,pcie.link.width.
↪  max,pcie.link.width.current  --format=csv -l
↪  5
```

Example to extract illustrations from PDF to transparent PNG:

```
$ pdftocairo X9DRG-HF-diagram.pdf X9DRG-HF-diagram -png -r 600
↪  -transp

Explanation:
-png to get png file
-r to set PPI (default is 150)
-transp to get transparent background instead of white
```

To confirm P2P is enabled, p2pBandwidthLatencyTest from NVIDIA CUDA-samples can be used. Example from abel2:

```
./p2pBandwidthLatencyTest
[P2P (Peer-to-Peer) GPU Bandwidth Latency Test]
Device: 0, Tesla K20Xm, pciBusID: 4, pciDeviceID: 0,
 ↪  pciDomainID:0
Device: 1, Tesla K20Xm, pciBusID: 5, pciDeviceID: 0,
 ↪  pciDomainID:0
Device=0 CAN Access Peer Device=1
Device=1 CAN Access Peer Device=0

***NOTE: In case a device doesn't have P2P access to other
 ↪  one, it falls back to normal memcopy procedure.
So you can see lesser Bandwidth (GB/s) and unstable Latency
 ↪  (us) in those cases.

P2P Connectivity Matrix
     D\D     0     1
     0       1     1
     1       1     1
Unidirectional P2P=Disabled Bandwidth Matrix (GB/s)
   D\D     0      1
     0 179.09   5.97
     1   6.08 179.21
Unidirectional P2P=Enabled Bandwidth (P2P Writes) Matrix
 ↪  (GB/s)
   D\D     0      1
     0 178.88   5.31
     1   5.31 178.56
Bidirectional P2P=Disabled Bandwidth Matrix (GB/s)
   D\D     0      1
     0 179.27   7.56
     1   7.59 179.40
Bidirectional P2P=Enabled Bandwidth Matrix (GB/s)
   D\D     0      1
     0 179.33  10.05
     1  10.02 179.90
P2P=Disabled Latency Matrix (us)
   GPU     0      1
     0   4.75  19.67
     1  18.61   4.72

   CPU     0      1
     0   4.46   9.85
     1   9.81   4.44
P2P=Enabled Latency (P2P Writes) Matrix (us)
   GPU     0      1
     0   4.77   1.60
     1   1.66   4.79
```

```
   CPU     0      1
     0   4.56   3.01
     1   2.99   4.50

NOTE: The CUDA Samples are not meant for performance
 ↪  measurements. Results may vary when GPU Boost is enabled.
```

Bibliography

[1] Intel Corporation. *An Introduction to the Intel® QuickPath Interconnect*. Tech. rep. 320412-001US. 2009. URL: https://www.intel.com/content/dam/doc/white-paper/quick-path-interconnect-introduction-paper.pdf.

[2] NVIDIA Corporation. *NCCL and MPI*. 2020. URL: https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/mpi.html (visited on 10/12/2021).

[3] NVIDIA Corporation. *NCCL Tests*. URL: https://github.com/NVIDIA/nccl-tests (visited on 09/05/2023).

[4] NVIDIA Corporation. *Non-Transparent Bridging and PCIe Interface Communication*. 2019. URL: https://docs.nvidia.com/drive/drive_os_5.1.6.1L/nvvib_docs/index.html#page/DRIVE_OS_Linux_SDK_Development_Guide/System%20Programming/sys_components_non_transparent_bridging.html (visited on 06/12/2021).

[5] NVIDIA Corporation. *NVIDIA Deep Learning NCCL Documentation*. 2021. URL: https://docs.nvidia.com/deeplearning/nccl/ (visited on 05/12/2021).

[6] NVIDIA Corporation. *NVIDIA H100 Tensor Core GPU*. 2023. URL: https://www.nvidia.com/en-us/data-center/h100/ (visited on 03/05/2023).

[7] NVIDIA Corporation. *What's a legacy driver?* URL: https://www.nvidia.com/en-us/drivers/unix/legacy-gpu/ (visited on 23/04/2023).

[8] P.J. Denning et al. 'Computing as a discipline'. eng. In: *Computer (Long Beach, Calif.)* 22.2 (1989), pp. 63–70. ISSN: 0018-9162.

[9] Luke Durant et al. *Inside Volta: The World's Most Advanced Data Center GPU*. URL: https://developer.nvidia.com/blog/inside-volta/ (visited on 10/05/2023).

[10] John Gulbrandsen. *PCI Express Physical Layer*. 2016. URL: https://youtu.be/EHkuzkNWXFk (visited on 21/11/2021).

[11] Super Micro Computer Inc. *X9DRG-HF X9DRG-HTF user's manual*. 1.0c. 15th Nov. 2013.

[12] Sylvain Jeaugey. *NCCL 2.0*. 2017. URL: https://on-demand.gputechconf.com/gtc/2017/presentation/s7155-jeaugey-nccl.pdf (visited on 10/12/2021).

[13] Canonical Ltd. *The Ubuntu lifecycle and release cadence*. URL: https://ubuntu.com/about/release-cycle (visited on 13/05/2023).

[14] Jonas Sæther Markussen. 'SmartIO : device sharing and memory disaggregation in PCIe clusters using non-transparent bridging'. PhD thesis. 2022.

[15] Alec Radford et al. *Robust Speech Recognition via Large-Scale Weak Supervision*. 2023. URL: https://github.com/openai/whisper (visited on 23/04/2023).

[16] Clara Santato and Pierre-Jean Alarco. 'The Global Challenge of Electronics: Managing the Present and Preparing the Future'. eng. In: *Advanced materials technologies* 7.2 (2022), 2101265–n/a. ISSN: 2365-709X.

[17] Richard Solomon. *PCI Express® Basics & Background*. 2021. URL: https://pcisig.com/sites/default/files/files/PCI_Express_Basics_Background.pdf#page=26 (visited on 09/12/2021).

[18] Dolphin Interconnect Solutions. *Accelerated PCI Express Network Performance. Optimized TCP IP Network driver*. URL: https://www.dolphinics.no/products/fast_TCP_UDP_IP_network_driver.html (visited on 09/05/2023).

[19] Dolphin Interconnect Solutions. *Dolphin eXpressWare Installation and Reference Guide*. 2022. URL: http://ww.dolphinics.no/download/PX_5_X_X_LIN_DOC/ (visited on 07/05/2023).

[20] Dolphin Interconnect Solutions. *Dolphin PCIe SmartIO technology*. URL: https://www.dolphinics.no/solutions/pcie_smart_io.html (visited on 09/05/2023).

[21] Dolphin Interconnect Solutions. *SISCI NCCL plugin*. URL: https://github.com/Dolphinics/sisci-nccl/ (visited on 08/05/2023).

[22] Joost Verbraeken et al. 'A Survey on Distributed Machine Learning'. In: *ACM Comput. Surv.* 53.2 (Mar. 2020). ISSN: 0360-0300. DOI: 10.1145/3377454. URL: https://doi.org/10.1145/3377454.

[23] Jeffrey Voas, Nir Kshetri and Joanna F. DeFranco. 'Scarcity and Global Insecurity: The Semiconductor Shortage'. eng. In: *IT professional* 23.5 (2021), pp. 78–82. ISSN: 1520-9202.

[24] Wikipedia. *Input–output memory management unit*. URL: https://en.wikipedia.org/wiki/Input-output_memory_management_unit (visited on 13/05/2023).

[25] Wikipedia. *Intel Ultra Path Interconnect*. URL: https://en.wikipedia.org/wiki/Intel_Ultra_Path_Interconnect (visited on 13/05/2023).

[26] Wikipedia. *List of Nvidia graphics processing units*. URL: https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units#Data_Center_GPUs (visited on 07/05/2023).

[27] Wikipedia. *PCI Express*. URL: https://en.wikipedia.org/wiki/PCI_Express (visited on 05/12/2021).

[28] Wikipedia. *The Portland Group Compilers*. URL: https://en.wikipedia.org/wiki/The_Portland_Group#Compilers (visited on 07/05/2023).

[29] Cliff Woolley. *NCCL: Accelerated multi-gpu collective communications*. 2015. URL: https://images.nvidia.com/events/sc15/pdfs/NCCL-Woolley.pdf (visited on 10/12/2021).