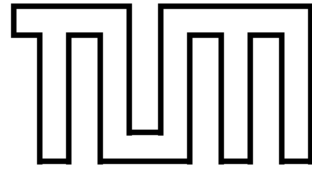


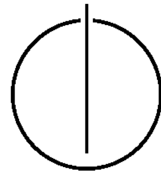
FAKULTÄT FÜR INFORMATIK

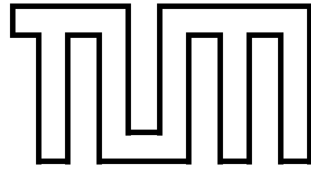
TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Evaluation of On-Node GPU Interconnects for Training Deep Neural Networks

Nane-Maiken Zarges





FAKULTÄT FÜR INFORMATIK

TECHNISCHEN UNIVERSITÄT MÜNCHEN

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Evaluation of On-Node GPU Interconnects for Training Deep Neural Networks

Evaluierung von On-Node GPU Interconnects für das Training von Deep Neural Networks

Author:	Nane-Maiken Zarges
Supervisor:	Prof. Dr. rer. nat. Martin Schulz
Advisor:	M. Sc. Amir Raoofy
Submission Date:	15.01.2019

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Garching bei München, 15.01.2019 Nane-Maiken Zarges

Abstract

Training deep neural networks is a computationally intensive task. The training time can be shortened by using hardware accelerators such as Graphics Processing Units (GPUs). Therefore, multi-GPU systems are used to speed up the training process. During the training process data needs to be copied from the CPU to the GPU and also between the GPUs. This thesis evaluates the performance of different on-node GPU interconnects: PCIe and NVLink.

Microbenchmarks are designed and conducted to measure the performance of these interconnects for basic operations involved in training deep neural networks. The practical impact of using different interconnects when training various deep neural networks on multi-GPU systems is measured for using PCIe interconnects and for using NVLink interconnects. These experiments are conducted on different multi-GPU systems that have distinct interconnect systems.

A classification approach is defined, which is used to classify the training workloads of deep neural networks into rather communication- or computation-intensive workloads. This classification is necessary to understand how much a certain training workload would benefit from a high-performance interconnect system.

Contents

Abstract **iii**

1. Introduction **1**
1.1. Motivation 1
1.2. Purpose 1
1.3. Thesis Structure 3

2. Deep Learning and its Hardware Requirements **4**
2.1. Introduction to Deep Learning 4
2.1.1. Learning Algorithms 4
2.1.2. Deep Neural Networks 5
2.1.3. Neural Network Architectures 7
2.1.4. Distributed Deep Learning 12
2.1.5. Classification of Deep Neural Networks 13
2.2. Hardware Accelerators for Deep Learning 15

3. Multi-GPU Computing Systems **19**
3.1. NVIDIA DGX-1 with 8 NVIDIA P100 GPUs 19
3.2. NVIDIA DGX-1 with 8 NVIDIA V100 GPUs 21
3.3. IBM Power System AC922 with 4 NVIDIA V100 GPUs 23

4. Experiment Design **25**
4.1. Microbenchmarks 25
4.2. Benchmarks for DNN training 26
4.3. Methods and Parameter Settings 27
4.3.1. Microbenchmarks 27
4.3.2. Benchmarks for DNN training 30

5. Experiment Results **33**
5.1. Microbenchmarks 33
5.1.1. DGX-1 P100 Results 34
5.1.2. DGX-1 V100 Results 37
5.1.3. AC922 Results 39

5.2. Benchmarks for DNN Training . . . 40
5.2.1. Workload Analysis using Nvprof . . . 40
5.2.2. Training Performance with PCIe only and with NVLink / PCIe . 41
5.3. Discussion of Results . . . 47
5.4. Limitations and Challenges . . . 50

6. Future Work **52**
6.1. Hardware Selection . . . 52
6.2. Deep Learning Workloads . . . 52
6.3. Comparison of NVLink and PCIe for DNN Training . . . 53

7. Conclusion **54**

A. Multi-GPU Computing Systems **56**

B. Experiment Setup **57**
B.1. Changes to mgbench . . . 57
B.1.1. Run only relevant tests . . . 57
B.1.2. Run the tests using PCIe . . . 58
B.1.3. Run the tests using different message sizes . . . 58
B.2. Changes to Tensorflow . . . 60
B.2.1. Changes in cuda_gpu_executor.cc . . . 60
B.2.2. Compiling changed Tensorflow . . . 60
B.2.3. Installing changed Tensorflow . . . 62

C. Experiment Results **63**
C.1. Microbenchmarks Results . . . 63
C.1.1. nvidia-smi Topology Query Results . . . 63
C.1.2. Mgbench Data Copies . . . 65
C.1.3. Latency . . . 73
C.2. DL Benchmarks Results . . . 77
C.2.1. Scaling Training using PCIe and NVLink . . . 77
C.2.2. Scaling Efficiency . . . 82

List of Figures **88**

List of Tables **91**

Bibliography **92**

1. Introduction

1.1. Motivation

Deep Learning as a part of Machine Learning is becoming increasingly important in Artificial Intelligence (AI) research and application. Current applications range from speech recognition to autonomous driving, medical diagnoses and many more. Although research on deep learning goes back several decades, during the last 5-10 years it gained a lot more attention. One reason for that is that learning weights of deep neural network models, known as training phase, is getting much faster thanks to exploiting hardware accelerators such as Graphics Processing Units (GPUs).[1] As GPUs are especially designed for highly parallel operations such as matrix multiplication, they perform a lot better than CPUs not only in computer graphics tasks such as real-time 3D graphics but also in training deep neural networks as this requires repetitive and highly parallel floating point operations.

Training a deep neural network can take hours to days, which can decelerate progress in Deep Learning.[2] [1] Therefore, finding solutions to speed up the training process has become crucial and the training is often parallelized on several GPUs. However, with more and ever faster GPUs, communication between the GPUs and also between GPU and CPU can become a bottleneck. Traditionally, PCIe interconnects are used to connect GPUs to the CPU. NVIDIA developed a high-performance, proprietary interconnect: NVLink. NVLink can be used to connect GPUs and in some systems also to connect the CPU with the GPUs. [3]

1.2. Purpose

The purpose of this thesis is to evaluate the performance of GPU interconnects for training deep neural networks (DNNs). Specifically, it is aimed at comparing the performance of NVLink and PCIe connections in the use case of training deep neural networks. In order to achieve a meaningful comparison, it is crucial to understand how the training process of DNNs works and what the differences of various neural network architectures are. Based on this knowledge, appropriate benchmarks need to be defined and a classification of deep neural networks based on their training workload has to be

elaborated.

The investigation is based on literature research and experiments. The experiments are performed on the following hardware setups:

- NVIDIA DGX-1, 8 Tesla P100 GPUs, NVLink 1.0 interconnect, Intel Xeon CPUs, PCIe Gen3
- NVIDIA DGX-1, 8 Tesla V100 GPUs, NVLink 2.0 interconnect, Intel Xeon CPUs, PCIe Gen3
- IBM AC922, 4 Tesla V100 GPUs, NVLink 2.0 interconnect, Power9 CPUs, PCIe Gen4

The hardware chosen for the benchmarks is manifold with regards to the interconnect systems: The two systems from NVIDIA use NVLink as GPU-to-GPU interconnects. For the connection between the CPU and the GPUs PCIe is used. The different NVLink versions do not only deliver different nominal bandwidth, but the different GPU versions also support a different number of NVLink connections, which enables further improvements in the interconnect system. The third system, IBM AC922 offers another specific characteristic: Not only the GPUs are connected via NVLink, but also the CPU-to-GPU connections are NVLink connections.

The main research question to be answered in this study is:

- How do different GPU interconnects compare in terms of performance for training Deep Neural Networks?

Consequentially, these subquestions need to be answered:

- Which metrics can be used to evaluate the performance of the interconnects for Deep Learning workloads?
- How bandwidth-sensitive are different neural network architectures?
- How do different architectures behave in the sense of scaling for multi-GPUs?
- How do the findings about interconnects explain this?

1.3. Thesis Structure

The remainder of this thesis is organized as follows:

In the second chapter, hardware requirements for efficiently performing deep learning workloads are elaborated. Therefore, a short introduction into deep learning is given including a description of different neural networks architectures. Also, a classification of deep neural networks based on the training workloads is elaborated. It is discussed why scaling to multi-GPUs is necessary and what kind of interconnects can be used to connect GPUs and CPUs.

In chapter 3, we take a deeper look into the hardware architecture of the used systems. All systems are presented focusing on the particular interconnect system.

Chapter 4 describes the experiment design in detail. At first, the general idea of two classes of benchmarks is presented. Then, the microbenchmarks and the designated deep neural network training benchmarks are explained in detail.

Chapter 5 presents the results of these benchmarks, discusses them and describes challenges in the benchmark design and run as well as limitations that apply to the results.

In chapter 6 prospects for future research are presented and chapter 7 concludes the findings of this work.

2. Deep Learning and its Hardware Requirements

2.1. Introduction to Deep Learning

Deep Learning (DL) "is part of the broad field of Artificial Intelligence (AI), which is the science and engineering of creating intelligent machines that have the ability to achieve goals like a human". [4, pp. 1-2] The term "Artificial Intelligence" was introduced by John McCarthy in his 1955 proposal on a summer research project, in which he explained some of the key aspects of AI, including "Neuron Nets" and how they can be arranged to form concepts. [5, p. 12]

Within AI there is a subfield called Machine Learning (ML). It enables the computer to learn from experience rather than to take take decisions based on hard-coded rules. [4, p. 1] Traditional ML approaches often require careful engineering and domain expertise to design a program that transforms raw data into representations, which can be used by a classifier to to detect or classify patterns in the input. [6, p. 436]

Deep Learning can be seen as a class of Machine Learning that uses multi layer (deep) neural networks to solve tasks without the need of predefined representations. By using Deep Learning algorithms the computer is not only able to solve problems, for which the process of solving can be described in a formal, mathematical way, but also problems that are solved intuitively by humans. The layers of features are learned from data. Deep Learning methods are so-called representation-learning methods, which means that the representations that are needed to classify or detect a certain pattern in the input data, are learned by the method itself. [6, p. 436] [7, pp. 1-4]

2.1.1. Learning Algorithms

A learning algorithm can be defined as "a mathematical framework or procedure that calculates the best output given a particular set of data". The calculation is adjusted based on the difference between the actual and the target output. [8] The learning procedure for DNNs is described in 2.1.2.

Mainly two different categories of learning are differentiated in Machine Learning: Supervised learning and unsupervised learning [7, p. 102].

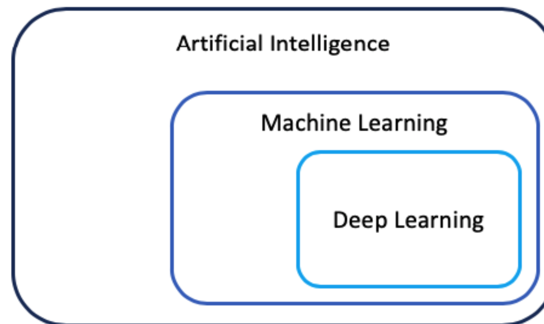


Figure 2.1.: Deep Learning in the context of Artificial Intelligence

In supervised learning algorithms the training data is labeled. For example, in image recognition we have a certain number of classes to which the images belong. In the training set each image is labeled with the class it belongs to. Unsupervised learning means that there is no target output given with the training set. This form of learning can for example be used for clustering or noise reduction tasks. The model learns the e.g. the classes during the training process itself. [7, p. 102]

Also some other categories exist, such as semi-supervised learning and reinforcement learning. In semi-supervised training part of the training data is labeled and the rest is not. [9, pp. 23] Reinforcement learning is based on a reward function. It is often used if the number of solutions is huge or infinite and if data sets are not independent but if there exist sequences of highly correlated states, a decision may influence all future decisions. A very famous example of Deep Reinforcement Learning is the paper by Mnih et. al. on "Playing Atari with Deep Reinforcement Learning".[10]

There are also hybrid approaches, especially the effect of unsupervised pretraining has been studied a lot.[11] [12]

In this thesis the focus is on supervised learning.

2.1.2. Deep Neural Networks

The design of neural networks was inspired by the human brain. [7, p. 13] The human brain consists of many neurons, which forward signals depending on their inputs. A neuron forwards information to a neighbor neuron if it receives enough impulses. At the point of entering the neighbor neuron, electric signals are converted to chemical signals and the synapse can strengthen or weaken the information forwarding depending on the chemical signal. [9, pp. 10]

The first algorithmically described neural network was the single layer perceptron described by Rosenblatt. The model consists of one node summing up weighted inputs

and an externally applied bias. The result of this operation is applied to a hard limiter to output a binary value: -1 if the hard limiter input was negative and +1 if it was positive. [13, pp. 78]

The Perceptron Convergence Algorithm was used to let the single layer perceptron learn. After some time steps, the weight vector, which is applied to the inputs, is updated based on the difference between the desired outcome and the actual outcome of the previous step. [13, pp. 80]

Deep Neural Networks are multi-layer systems, which were built based on the single layer perceptron. By using multiple non-linear layers, representations of the input data can be transformed into more abstract levels with every layer, which enables to learn more complex functions. [6, p. 436] As shown by Cybenko a so-called continuous feedforward neural network (FFN) with only one hidden layer can already approximate every continuous function. [14]

A feedforward deep neural network consists of at least three layers: The input and output layer and one or more hidden layers in between. [7, p. 165] The hidden layers compute a weighted sum on their inputs, perform a non-linear function on it, and then pass the result to the next layer. Very often the non-linear function used is the Rectified Linear Unit (ReLU) function $f(z) = max(z, 0)$. [6, p. 437]

Backpropagation of Error & Stochastic Gradient Descent

For deep neural networks the learning does not work as for a single layer perceptron since the error for hidden layers is not known. For a classifier problem the neural network would process the input through its layers and output a vector of scores for each class. An error function is defined, which measures the distance between the desired output vector (the label) and the actual output vector. The goal of the learning process is to minimize this error to a certain threshold. This is done by adjusting the internal parameters (also called weights) of the neural network by using a gradient vector. Minimizing the error function is done in several steps and each step involves two stages: The propagation of errors backwards and the weight adjustment using optimization schemes such as gradient descent. [15, pp. 241]

Stochastic Gradient Descent is an extension of the gradient descent algorithm. The gradient descent algorithm uses the derivatives of a function $f(x) = y$ to find a minimum. Because the derivative $f'(x)$ gives the slope of the function f at point x, it provides insights into how x needs to be changed to find the minimum: If $f'(x)$ is positive, x needs to be decreased, if $f'(x)$ is negative, x needs to be increased. For higher dimensional functions, partial derivatives are used. Partial derivatives measure

how f changes as only one of the inputs changes. The partial derivatives for all input variables are combined in the gradient vector.

$$\nabla_x f(x) = \begin{pmatrix} \frac{\partial}{\partial_{x_1} f(x)} \\ \frac{\partial}{\partial_{x_2} f(x)} \\ \dots \\ \frac{\partial}{\partial_{x_n} f(x)} \end{pmatrix}$$

Decreasing f by moving into the direction of negative gradient is called the gradient descent method. The size of steps is defined by the learning rate ϵ, so a new point is proposed by:

$$x_{new} = x - \epsilon \nabla_x f(x)$$

[7, pp. 80]

The **stochastic gradient descent algorithm** uses an expectation value to make the operation less cost-intensive. The outputs and errors are computed for a small set of inputs instead of for the whole training data. For the small set of data, an average gradient is computed. This process is repeated for several small test sets until the average error does not decrease anymore. [6, pp. 436] [7, pp. 149]

An efficient way to obtain the partial derivatives needed for the gradient descent algorithm is the backpropagation algorithm, which can be seen as a practical application of the chain rule for derivatives. [6, p. 438]

The learning process can be summarized as follows:

1. The weight matrix is initialized with random values and forwarded through the neural network

2. The labels are compared to the actual output and the difference is saved as error of the network. If this error is bigger than a certain threshold, the third step is applied. If not, the training is finished.

3. The error is propagated backwards through the network. [9, pp. 17]

2.1.3. Neural Network Architectures

There are different neural network architectures including Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs). The basic parts of these two architectures are described in the following sections.

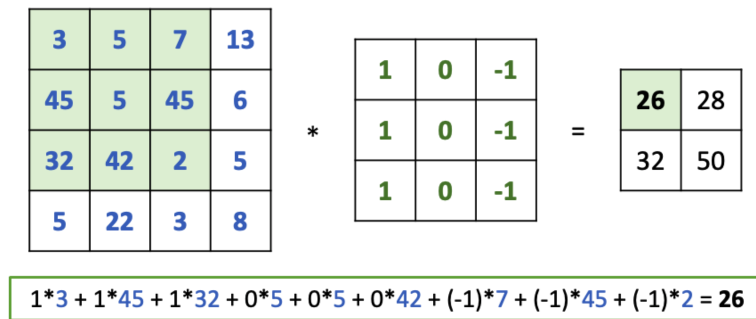


Figure 2.2.: Convolution operation, adapted from [9]

Convolutional Neural Networks

One of the earliest convolutional neural networks was presented by LeCun in 1998. He showed that character recognition can be done by a CNN without using hand-crafted heuristics, which was needed by fully-connected feed-forward networks. [16] The network he used is now called LeNet and described in this chapter.

CNNs are mainly used for computer vision tasks with a lot of training data. [17] [9, p. 26] Their main advantages over fully-connected neural networks are:
Sparse connectivity, which means that by applying filters that are smaller than the input the number of parameters is reduced. This leads to less computation time and less memory needed.
Weight sharing, which means that one weight is not only used once but applied to different input values. This also reduces memory requirements.
Equivariance leads to detecting a certain feature independent from where this feature is located in the input data. This makes the network resistant to local distortions. [7, pp. 329-335] [16, pp. 5-6]

Convolutional neural networks consist of three different types of weight layers: convolutional, pooling, fully-connected.

Each Convolutional Layer consists of several independent filters, which are used to extract different features within a picture. Each filter is moved over the whole input matrix of the respective layer in order to detect the respective feature in the whole picture. Figure 2.2 shows an example of the convolution operation. The filter applied to the input matrix would be a feature extractor for vertical edges. After that, a bias is added and a non-linear function such as ReLU is applied. [16]

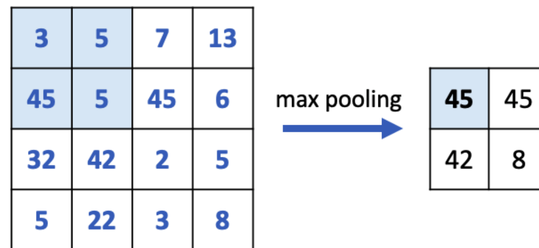


Figure 2.3.: Pooling operation, adapted from [9]

The Pooling Layer is used to drop unnecessary information. Examples for pooling operations are average-pooling, where the average value of a certain area is kept or max-pooling as shown in 2.3. [9, p. 25] In the original paper from LeCun [16] this layer is called sub-sampling layer. As by pooling layers the numbers of parameters is reduced, it leads to a reduced computation time and makes the network more robust to distortions.

Fully-connected Layers are often used at the end of a CNN in order to classify the resulting feature vectors.[6, p. 439]

LeNet5 is a five weight layer neural network, including three convolution layers and two fully-connected layers. After the first two convolution layers average pooling layers originally called subsampling layers) are used. One of the main advantages of LeNet as a CNN for image recognition was that no hand-designed feature extractors were needed anymore. Other ML algorithms used for recognizing handwritten digits still needed predefined feature extractors. [16]

AlexNet has won the ILSVCR[1] competition for object detection and image classification in 2012. It was the first neural network trained on multiple GPUs. It consists of eight weight layers, out of which the first five are convolutional layers and the remaining three are fully-connected ones. After each layer, ReLU is applied on the output. Additionally, after the first and second convolutional layer, a normalization and a max-pooling layer is added. The fifth convolutional layer is followed by another max-pooling layer. As the model is trained in a model parallel way (see 2.1.4), the model is split into two parts and at some points the GPUs need to exchange information: The kernels of the second convolutional layer are connected to the kernels of the third

[1] ImageNet Large Scale Visual Recognition Challenge, `http://image-net.org/challenges/LSVRC/2012/results.html`

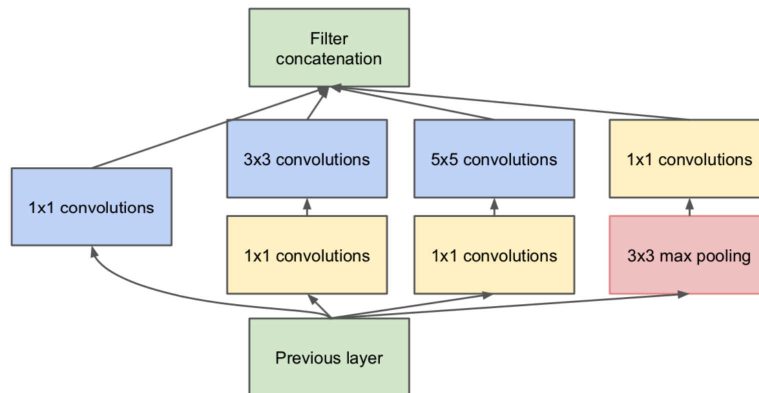


Figure 2.4.: Inception module with dimensionality reduction using 1x1 convolutions, [19]

convolutional layer on both GPUs and the fully-connected layers are also connected from one GPU to the other. [18]

GoogLeNet is also one of the winner networks of ILSVCR. The network code named Inception or Inception v1 won the image recognition challenge in 2014. It consists of 22 weight layers: Different to former networks it uses a so-called "inception module". The data going into the inception module has to go through several convolutions and a pooling operation in parallel and the resulting output is concatenated to form the input vector for the next layer. To reduce dimensions within the inception module, a 1x1 convolution is used before 3x3 and 5x5 convolutions as well as after 3x3 max pooling. Figure 2.4 shows one inception module. By using this method, GoogLeNet achieved higher accuracy than AlexNet two years earlier while having less parameters, which makes it less computation intensive. [19]

VGG is also one of the neural networks that performed very well at the ILSVRC 2014 competition. There exist different configurations with a different number of weight layers: eight to sixteen convolution layers and three fully-connected layers. Additionally, five maxpool layers follow some of the convolutional layers. VGG-11 consists of eight convolutional layers, VGG-19 consists of sixteen convolutional layers. The number of parameters range from 133 to 144 million. [20]

Inception networks (e.g. Inception v3 and Inception v4) are based on the architecture of GoogLeNet, which already used inception modules. The goal of the newer Inception

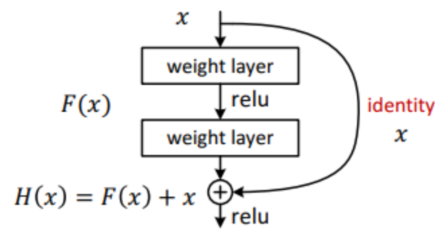


Figure 2.5.: A residual block, [23]

versions was to speed up computation by factorizing larger convolutions into several smaller ones (e.g. 5x5 to two 3x3 convolutions or nxn to a nx1 and a 1xn convolution). Another result of their research was that "the representation size should gently decrease from the inputs to the outputs" in order to "avoid representational bottlenecks". [21] In Inception v4 the inception modules were standardized to three different versions as they figured out that the former non-uniformity makes the model more complicated but does not add value in terms of performance or accuracy. [22]

ResNet has been developed by Microsoft Researchers and won several Computer Vision competitions in 2015, e.g. ILSVCR & COCO. There are different variants of ResNet with a different number of layers. The special thing about ResNet is that it can become quite deep without becoming too difficult to train. This difficulty was tackled by introducing "residual blocks". Each ResNet block has one identity mapping shortcut connection, which skips one or even more layers. In this way, no extra parameter or computational complexity is added. However, adding more depth to the network resulted in more accurate results. [23]

Recurrent Neural Networks

In this section the basic building blocks of Recurrent Neural Networks are described. As no RNNs are used in the experiments of this study, there is no detailed description of specific RNNs.

Recurrent Neural Networks are commonly used for sequential input data.[6, p. 441] For example, they are used in the fields of speech recognition, machine translation or sentiment classification to only name a few. [24, p. 1]

One of the advantages of using RNNs for e.g. speech recognition is that they can accept input vectors of variable length by using parameter sharing. As two sentences can have the same meaning but still be composed of different words and also have a different length, this is a very crucial characteristic. [7, pp. 367-368]

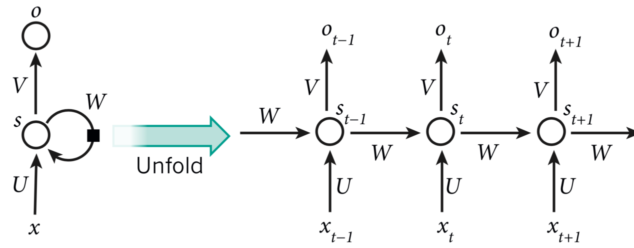


Figure 2.6.: RNN and the unfolding in time of the computation involved in its forward computation, [6, p. 442]

Figure 2.6 shows a simple RNN during forward propagation in the unfolded view. to illustrate how the different parts of the input sequence x influence the output computation of the following neurons. The artificial neuron s has time-depending values s_t. These depend on the value of the former neuron s_{t-1} multiplied with weight matrix W and the input x_t multiplied with weight matrix U. The output o_t of the neuron is its value multiplied by weight matrix V. [6, p. 442] There are also other more complex variants, but the basic principles remain the same.

One specific set of RNNs are Long Short Term Memory (LSTM) networks, which have been designed to enable the network to memorize past information. In order to do so, they use so-called memory or "LSTM" cells, which are special hidden units having an internal recurrence and acting like accumulators. [7, pp. 404-406]

One of the disadvantages of RNNs for language-related tasks is that the model of the network becomes very big when having a large vocabulary.[24, p. 1] Furthermore, RNNs are difficult to train and need a lot of computational power. Therefore, there is also research on using CNNs for sequence modeling tasks and therefore a discussion on the question if RNNs may be less important for these tasks in the future. [25][26]

2.1.4. Distributed Deep Learning

Distributed Deep Learning has become increasingly important as the size of DNNs and the training datasets has grown. Therefore, to train DNNs either the data (Data Parallelism) or the model (Model Parallelism) is distributed onto several GPUs. Those GPUs could either be on the same node or even on different nodes. [2, p. 1, 3] In this thesis, only multi-GPU training on one node is considered.

Data Parallelism

In the data parallelism strategy the neural network is copied onto every GPU and the data set is split up into pieces that are distributed to the GPU. Thus, each GPU is using

the same weights but different data. There are two parts of communication: First, the worker machines send the gradients resulting from backward propagation to the master. Then, out of the sub-gradients, the master computes the new weights and broadcasts them to the workers. [2, p. 3][27] This is done after each forward and backward pass. [28, p. 2] The synchronization can also be done via a multi-GPU communications collective. An allreduce operation would compute the mean of all the weights and then distribute the new weights. [29, p. 10] Data parallelism is an efficient method for networks with few parameters or a high computation effort per parameter. [28, p. 2]

Model Parallelism

In the model parallelism strategy the model is split up and each part is distributed to one GPU. Thus, each GPU is calculating part of each layer with all data. [2, p. 3] Following this strategy, less memory is needed at each GPU as only part of the network is stored there, and thus, very big models can be trained. For the convolution operation, model parallelism is rather inefficient as each GPU needs the results of the other GPUs to compute the final result of the layer. Therefore, communication is needed after every layer. [30, pp. 20-21] However, it is an efficient strategy for fully-connected layers as they have many parameters. [28, p. 2]

2.1.5. Classification of Deep Neural Networks

In order to select the right neural networks to benchmark on-node GPU interconnects, we need to understand two main points:

- Which network architectures benefit from using GPUs for training?
- Out of these architectures, which ones rely on heavy communication during the training process?

In the following different approaches found in literature and their results are presented.

In order to create a collection of deep learning workloads that can be used to evaluate deep learning hardware performance, Adolf et al. have published Fathom. [31] They analyzed not only convolutional neural networks but also fully-connected ones as well as RNNs arguing that the latter architectures haven't been included in HW research as much as CNNs have. In figure 2.7 we can see how much the chosen eight networks benefit from using a GPU instead of a CPU for training. It can be observed that all CNNs (residual, vgg, alexnet, deepq) and the speech network, which consists of recurrent and fully-connected layers, have a considerably shorter training time on the GPU than on the CPU. Another interesting result from this work is the breakdown of

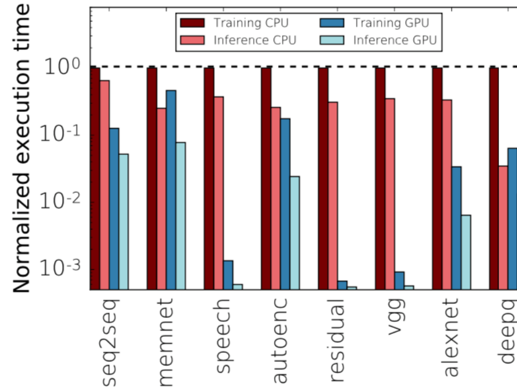


Figure 2.7.: FATHOM: normalized execution time, GPU vs. CPU [31]

execution time, which is presented in figure 2.8. The operation types that are highly parallelizable on GPUs are the groups B (matrix operations) and D (convolution). The networks, which benefit from using a GPU, spend 80-99% in these kinds of operations.

Another interesting group of operations from figure 2.8 is group G (data movement). As the objective of this thesis is to analyze the performance of interconnects, network architectures with a high amount of data movements are interesting. For the four CNNs at the bottom of the table, data movements are insignificant in comparison to the other operation groups. The RNN seq2seq as well as the memnet spend 24% of their execution time in moving data, which suggests a dependence on high bandwidth and low latency interconnects for training on multiple GPUs. The RNN speech spends 10% of its execution time with data movements. This implies that this network could be an interesting model for this research.

Tallent et al. have performed quite a similar research on on-node GPU interconnects as we do. They used GoogLeNet, AlexNet and a parameterized version of ResNet in their research. The parameterized version of ResNet - called ResNet/x - used x residual blocks as inner layers. In order to classify different DL workloads, a workload intensity metric was defined, which is a measure of communication/computation. For the training they used the data parallel approach. In their work they show that GoogLeNet is a quite computation-intensive workload whereas AlexNet is a quite communication-intensive workload. Depending on the depth of the ResNet variant, the workload is more or less computation-intensive. [32]

In a paper on the topic of parallelism in deep learning, Dettmers stresses another aspect, which affects the amount of communication while training on multiple GPUs: The parallelism strategy. As described in 2.1.4 the model parallelism strategy in general creates a lot more communication than the data parallelism strategy. [28]

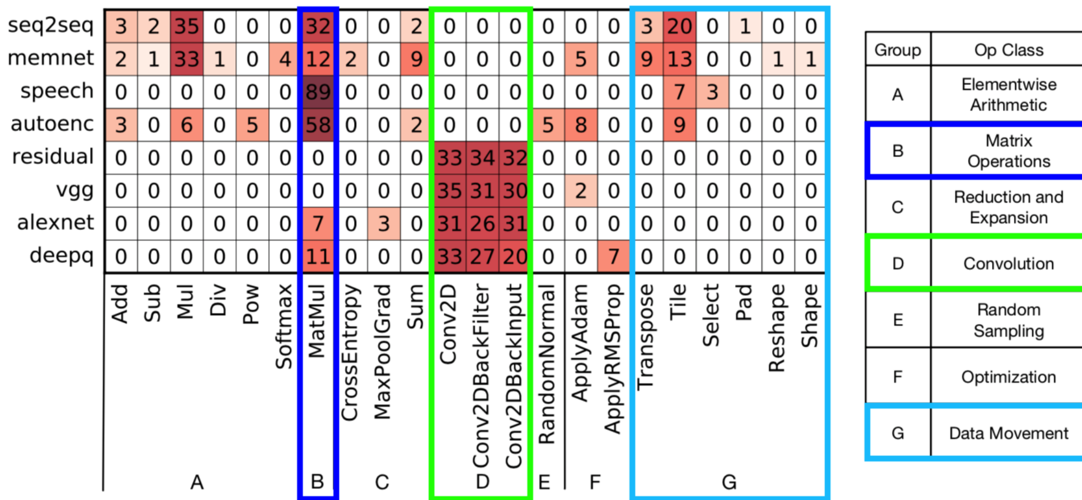


Figure 2.8.: Fathom: Breakdown of execution time by operation type, adapted from [31]

For this thesis, the used networks are classified based on their time spent in memory copy operations as these operations stress the interconnects.

2.2. Hardware Accelerators for Deep Learning

As the Central Processing Unit (CPU) is designed to not only compute but also control the system, it is helpful to transfer computationally intensive tasks such as the training of neural networks to Graphics Processing Units (GPUs) or other specialized hardware accelerators. The design of a GPU is useful for many parallel computations, especially for vector and matrix operations. [9, pp. 20-21][33, pp. 563-570] [2, p. 1]

As training neural networks involves a lot of matrix multiplications, researchers began to use GPUs to train their neural networks already in the early 2000s. [34, p. 89, p. 96] [35, pp. 1390-1391]

Having only one GPU to train a neural network limited the size of the networks. In 2012 Krizhevsky et al. won the ImageNet LSVRC-2012 contest with a convolutional neural network (see 2.1.3) that was trained on multiple NVIDIA GTX 580 GPUs, specifically two. Because their model was too big to fit into one GPU's memory (3 GB), they used the model parallel approach and split their model into two parts, half of the neurons being on each GPU. [18, p. 3]

Today, the most DL systems are a mix of CPU and GPU, where the GPU performs the computation-intensive tasks, and the CPU is responsible for loading the data into/from the memory of a graphics card and acting as a parameter server. [35] On top of having

fast GPUs, it is necessary to have performant interconnects between the CPU and the GPU to feed the GPU with training data as well as a fast connection between the GPUs in order to harness multiple GPU memories. [32]

On-Node Interconnects

In this part, the processor interconnect types used in the studied systems are described. All interconnects used for CPU-to-GPU as well as GPU-to-GPU connections are described.

PCI Express

Peripheral Component Interconnect (PCI) Express or short PCIe is a standard to connect expansion cards such as sound or graphics cards to PC motherboards.

There are two main factors, which affect the performance of PCIe slots: The number of lanes and the generation. A PCIe lane is a serial connection of differential signal pairs (transmission and reception) for data transfer. PCIe lanes can be bundled to increase the number of lanes, a bundle is denoted as xN where N stands for the lane width. By increasing the number of lanes, the bandwidth scales linearly. In the PCI Express Base Specification operations for x1, x2, x4, x8, x16 and x32 are described. [36, p. 38] The second factor is the generation: Over time, the bandwidth of one PCIe lane has increased as well. The first generation was launched in 2004 and had a bandwidth of 250 MB/s. At the time of writing, PCIe 4.0, introduced in 2017, is the most up-to-date release delivering a bandwidth of 1.97 GB/s for one lane.

The commonly used slot size for graphics cards is x16. Thus, a PCIe 3.0 connection to a graphics card delivers a bandwidth of 15.754 GB/s (985 MB/s per lane) and a PCIe 4.0 connection has a bandwidth of 31.508 GB/s. [37, p. 288-289]

The systems used in this study are equipped with either PCIe 3.0 or PCIe 4.0 slots. Table 2.1 shows the different PCIe generations and their bandwidths with respect to lane sizes. PCIe 5.0 is currently being reviewed. [38][39]

A PCIe Switch is used to connect a number of peripheral devices such as GPUs or Network Interface Cards (NIC) with the motherboard of a computer. It is defined as a logical assembly of multiple virtual PCIe-to-PCIe bridge devices. Several PCIe links are connected to one PCIe switch. [36, p. 45]

NVIDIA NVLink

NVLink is a proprietary system link architecture developed by NVIDIA, which was designed as an alternative to the PCIe connection, which can become a bottleneck for

Table 2.1.: PCIe bandwidths for different generations and lane widths
[40, p. 29] [41, p. 34][36, p. 40][42, p. 56][39, p. 234]

	x1	x4	x8	x16
PCIe 1.0	250 MB/s	1 GB/s	2 GB/s	4 GB/s
PCIe 2.0	500 MB/s	2 GB/s	4 GB/s	8 GB/s
PCIe 3.0	984.6 MB/s	3.94 GB/s	7.88 GB/s	15.75 GB/s
PCIe 4.0	1969 MB/s	7.88 GB/s	15.75 GB/s	31.5 GB/s
PCIe 5.0	3938 MB/s	15.75 GB/s	31.5 GB/s	63.0 GB/s

multi-GPU systems if data needs to be sent frequently. [43, p. 14]

In the first generation, NVLink delivers a unidirectional bandwidth of up to 20 GB/s. NVLink 2.0 delivers a unidirectional bandwidth of up to 25 GB/s. [3]

NVLink is compatible with NVIDIA's own GPU Instruction Set Architecture (ISA) for multi-GPU systems, which enables direct access to the memory of another GPU in the system. Thus, programs "can execute directly on data in the memory of another GPU" and also atomic memory operations can be performed on remote GPU memory addresses. [29, p. 6-7]

In systems with x86 CPUs, such as the DGX systems, NVLink is only used for GPU-to-GPU connections. At the time of writing, the only usage of NVLink for CPU-to-GPU connections is within IBM POWER systems. On NVIDIA's GTC 2016, the POWER8 CPU with NVLink was announced, which enabled the IBM "Minsky" Platform to use NVLink for CPU-to-GPU communication. The system is made of 2 POWER8 CPUs and 4 NVIDIA Tesla P100 GPUs. The official name is IBM S822LC. [44] [1]

For the study at hand, the IBM POWER System AC922 is used. This system uses NVLink 2.0 as CPU-to-GPU connection and is explained in detail in chapter 3.3.

NVIDIA NVSwitch is used in NVIDIA's DGX-2 systems to connect GPUs. One NVSwitch supports up to 18 NVLinks and each port supports 25 GB/s unidirectional bandwidth, which accounts for up to 900 GB/s bidirectional bandwidth for one NVSwitch. In total, there are six NVSwitches used to connect every GPU with all other GPUs within the system. [45]

NVSwitch is not part of the systems used in this study.

Intel Quick Path Interconnect

The Quick Path Interconnect (QPI) from Intel is used for point-to-point communication in multi-processor systems. Data is sent in parallel across multiple lanes and packets are

broken into multiple parallel transfers. The physical connectivity of each interconnect link consists of twenty differential signal. Each port supports a link pair consisting of two unidirectional links, which supports traffic in both directions simultaneously. [46, p. 8]

X Bus

The X Bus is a symmetric multi processing (SMP) interconnect developed by IBM. It is used in multi-CPU systems to connect several CPUs and enable communication as well as provide a shared memory space. The interconnect works at 16 GHz and 4 bytes, which results in a bandwidth of 64 GB/s per X Bus. [47, p. 12, 14]

3. Multi-GPU Computing Systems

This chapter presents the systems used in this study with an emphasis on the interconnect systems. The two NVIDIA DGX-1 systems are similar in their architecture but use different versions of NVLink. The IBM Power System AC922 is different from the DGX systems in the fact that it uses NVLink not only for GPU-to-GPU connections but also for CPU-to-GPU connections.

3.1. NVIDIA DGX-1 with 8 NVIDIA P100 GPUs

System Architecture

The core of this DGX-1 system are eight NVIDIA Tesla P100 GPUs, together with two 20-Core Intel Xeon E5-2698v4 CPUs running at 2.2 GHz.

The CPUs are mainly used for booting, storage management and deep learning framework coordination. [29, p. 2] The workload activities of training neural networks mainly take place on the GPUs. The host CPUs read the initial training data set into memory and transfer it to the GPUs. Therefore, the CPUs are not discussed in detail.

The DGX-1 system has 512 GB DDR4 system memory and four SSDs with 1.92 TB of storage. The whole system provides 170 TFLOPs computing power for single-precision workloads. [48]

NVIDIA Tesla P100 GPU

The NVIDIA Tesla P100 GPUs consists of 56 streaming multiprocessors (SMs), each with 64 single-precision (FP32) CUDA cores. Thus, one GPU comprises 3584 single-precision CUDA cores. Each GPU is equipped with 16 GB of High Bandwidth Memory 2 (HBM2) memory, which delivers a bandwidth of 732 GB/s, and a L2 Cache of 4096 KB. Tesla P100's peak computational throughput is 5.3 TFLOP/s for double-precision (FP64), 10.6 TFLOP/s for single precision (FP32), and 21.2 TFLOP/s for half-precision FP16. [49]

The Tesla P100 GPU is available as PCIe version [50] and as NVLink [51] version (form factor SXM2). The DGX-1 is equipped with NVLink connected GPUs. Each GPU supports up to four NVLink1.0 connections. [49, p. 20]

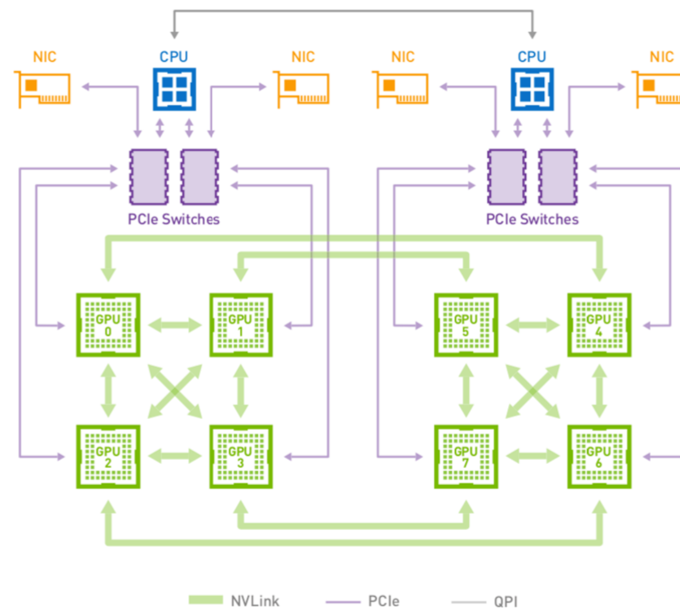


Figure 3.1.: NVIDIA DGX-1 with 8 NVIDIA P100 GPUs network topology [29, p. 9]

Interconnect System

The interconnects in this system are NVLink 1.0, PCIe 3.0 and QPI. Figure 3.1 shows the interconnect system as network topology.

Each GPU is connected via PCIe 3.0 x16 to one of the four PCIe switches. Each PCIe switch connects two GPUs with one of the two CPUs. Three types of connections using PCIe can be differentiated: Each GPU is connected to one other GPU through a PCIe switch. Two other GPUs can be reached by traversing the own PCIe switch, the CPU and the other PCIe switch connected to this CPU. The remaining four GPUs can only be reached via PCIe switches, the CPUs and the QPI connection between them. [29, p. 9]

Inside the DGX-1 system, the eight GPUs are connected in a hybrid cube-mesh NVLink network topology. Each GPU is connected point-to-point to four other GPUs: to all three in its own cluster and to one from the other cluster. Thus, the gap between the two clusters of four GPUs, which exists for the PCIe and QPI connections, is closed. NVLink supports the GPU Instruction Set Architecture (ISA), which means that programs cannot only execute on data on local memory but also on data, which is inside another GPU's memory. As each GPU supports up to four NVLink 1.0 connections, the aggregate bidirectional bandwidth accounts for 160 GB/s per GPU. [29, pp. 6]

System Configuration

On the DGX machines the NVIDIA application cotainer "Tensorflow 18.07 PY3" is preinstalled. Amongst other software packages it includes: [52]

- Ubuntu 16.04
- Python 3.5
- TensorFlow 1.8.0
- NVIDIA CUDA 9.0.176
- NVIDIA CUDA® Deep Neural Network library™ (cuDNN) 7.1.4
- NCCL 2.2.13

Changes to this configuration are described in detail in chapter 5.

3.2. NVIDIA DGX-1 with 8 NVIDIA V100 GPUs

System Architecture

Similiar to the DGX-1 system with P100 GPUs, this system also consists of eight GPUs and two CPUs. The GPUs are NVIDIA Tesla V100 GPUs, the CPUs are the same as in the system with P100 GPUs: two 20-Core Intel Xeon E5-2698v4 CPUs running at 2.2 GHz. This DGX-1 system also contains 512 GB DDR4 system memory and 4 SSDs with 1.92 TB of storage. [53, p. 4] The system provides 960 TFLOPS computing power for single-precision workloads. [48]

NVIDIA Tesla V100 GPU

The NVIDIA Tesla V100 GPU consists of 80 streaming multiprocessors, each with 64 single-precision (FP32) CUDA cores and in contrast to NVIDIA P100 GPUS each SM contains eight Tensor cores, which are especially important for the matrix-matrix multiplication operations when training neural networks. [54, p. 14] One V100 GPU comprises 5120 single-precision CUDA cores and 640 Tensor cores. Each GPU is equipped with 16 GB of High Bandwidth Memory 2 (HBM2), which delivers a bandwidth of 900 GB/s, and a L2 Cache of 6144 KB. [54, p. 8]

Tesla V100's peak computational throughput is 7.8 TFLOP/s for double-precision floating point (FP64), 17.7 TFLOP/s for single precision (FP32), and 125 Tensor TFLOP/s for mixed precision. [53, p. 5] [47, pp. 25-26]

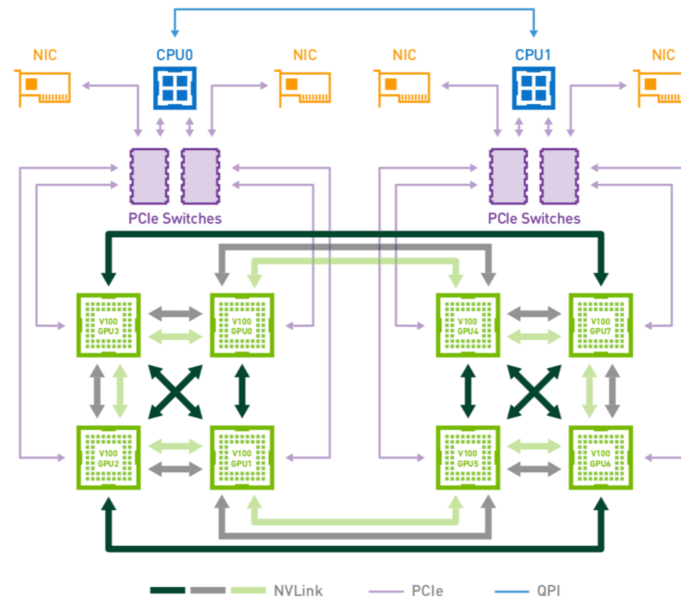


Figure 3.2.: NVIDIA DGX-1 with 8 NVIDIA V100 GPUs [53, p. 9]

The Tesla V100 GPU is available as PCIe version and as NVLink [55] version (form factor SXM2). The DGX-1 is equipped with NVLink connected GPUs. Each Tesla V100 GPU has six NVLink 2.0 connection points. [54, p. 19]

Interconnect System

The interconnects in this system are NVLink 2.0, PCIe 3.0 and Intel Quick Path Interconnect (QPI). Figure 3.2 shows the interconnect system as network topology.

The PCIe and QPI connections are the same as for the other DGX system, which is described in section 3.1.

The NVLink connections differ from the ones in the system with P100 GPUs since V100 GPUs support NVLink 2.0, thus a higher bandwidth, and have more NVLink connection points: Each GPU has six instead of four NVLink connection points. The GPUs are also ordered in a hybrid cube mesh topology, but some GPUs are connected to each other by using a bonded set of two NVLink connections. For these connections the theoretically achievable bandwidth accounts for 50 GB/s. The connections using one NVLink can deliver a bandwidth of 25 GB/s. Some GPUs are not directly connected to each other. This accounts for an aggregate bandwidth of 300 GB/s. [53, pp. 7-10]

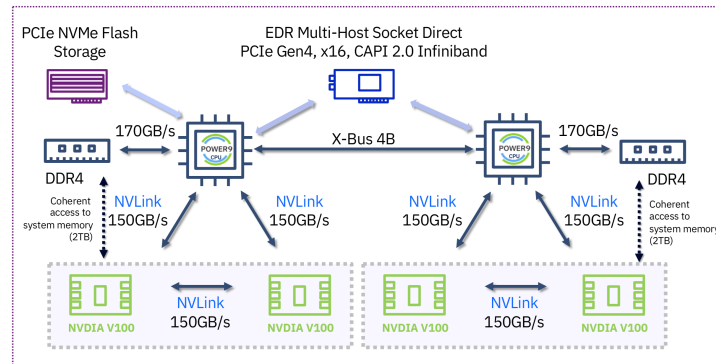


Figure 3.3.: IBM Power System AC922 with 4 V100 GPUs [56, p. 8]

System Configuration

The system configuration is consistent with the system configuration of the DGX-1 with P100 GPUs. The preinstalled NVIDIA application container is described in section 3.1.

3.3. IBM Power System AC922 with 4 NVIDIA V100 GPUs

System Architecture

The IBM AC922 system is available in different models: 8355-GTG and 8355-GTW. The main difference is the number of GPUs (four vs. six) and the cooling (air-cooled vs. water-cooled). For this study, we use the 8355-GTG version, which consists of two Power9 CPUs and four NVIDIA Tesla V100 GPUs. This system contains two Power9 processors, each of which has 20 cores that are based on a 64-bit architecture. The clockspeed constitutes 2.0 GHz, 2.87 GHz turbo. Per core, 512 KB L2 cache and per chip, 120 MB L3 cache is available. [47, pp. 4]

NVIDIA Tesla V100 GPU

The four Tesla V100 GPUs used in this system are the same graphics cards as described in chapter 3.2.

Interconnect System

This system uses NVLink 2.0 and XBus as interconnects between GPUs and CPUs. Figure 3.3 shows the interconnect system as network topology.

The main difference of this system compared to the other two are the NVLink connections between the Power9 CPUs and the GPUs in the corresponding cluster. As each V100 GPU supports up to six NVLink 2.0 connections, three of them are used to connect to the respective CPU and the other three are used to connect to the other GPU in the same cluster. This leads to a maximum unidirectional bandwidth of 150 GB/s for either of those connections. Another difference DGX systems is that the two clusters are not connected via NVLink GPU-to-GPU connections, but only by going over the XBus, which connects the two CPUs. XBus delivers a bandwidth of up to 64 GB/s. [47, pp. 12-14]

4. Experiment Design

The following chapter presents the experiments' setup. At first, the general experiment ideas are explained, then the implementation is presented in detail.

The experiments conducted in this study can be split into two categories: Microbenchmarks and designated DNN training benchmarks. The microbenchmarks are intended to achieve an understanding of how small operations are affected by using NVLink or PCIe without any specific use case. The benchmarks for DNN training are designed to understand the practical impact of using different GPU interconnects when training deep neural networks on multi-GPU systems.

4.1. Microbenchmarks

As explained in chapter 2, deep neural networks can be trained on several GPUs by splitting up the data to all nodes and then exchanging and adapting the resulting weights.

This requires a lot of data to be copied from CPU to GPU, between the GPUs and from GPU to CPU. Therefore, the microbenchmarks test the bandwidth and latency for memory copies of different data sizes. These tests are performed with and without enabling peer-to-peer access for the GPUs. If peer-to-peer access is enabled, one GPU can directly copy data to another GPU without using the CPU. Thus, the NVLink connection can be used. By using direct memory access (DMA), the GPUs are also able to read and write directly from / into another GPU's memory. DMA is only possible between NVLink connected GPUs. If peer-to-peer access is not enabled, the memory copies need to be performed over the CPU. This means, that the NVLink connections cannot be used in the DGX systems. Instead, PCIe and QPI are used to copy the data from one GPU to another.

By enabling or not enabling peer-to-peer access, bandwidths and latencies can be measured for NVLink or PCIe in the DGX-1 systems. As there are no PCIe connections between GPUs and CPUs in the IBM AC922 system, the benchmarks are not run without peer-to-peer access on this system.

4.2. Benchmarks for DNN training

In this section, we want to explore three aspects about the interconnect performance of PCIe and NVLink:

1. How communication-intensive is the training of different DNNs?
2. What is the difference in the performance (measured as number of images processed per second) for the same DNN, the same batch size and the same number of GPUs using either PCIe interconnects or NVLink interconnects?
3. What is the difference in scaling performance for scaling the workload over one to eight GPUs either using PCIe or NVLink?

To gain insights into these three aspects, different neural networks are trained on different numbers of GPUs. Starting with the training on only one GPU and scaling up to using eight GPUs. In order to measure PCIe performance, again peer-to-peer access needs to be disabled. Additionally, training on eight GPUs is profiled in order to understand how communication-intensive the training of different DNNs is.

Neural Networks

The neural networks used for the benchmark tests are: AlexNet, GoogLeNet, Inception v3, Inception v4, ResNet50, ResNet152, VGG11, VGG16, and LeNet5. The network architectures of these CNNs are described in chapter 2.1.3.

The selection of neural networks is based on the following criteria:

- Communication-intensity versus computation-intensity
- Comparability and relevance for industry and research
- Availability of implementations for training on GPUs

From the study of related work about DNN classification (see chapter 2.1.5), it is known that AlexNet is a communication-intensive workload while GoogLeNet is a computation-intensive workload. Choosing these two networks ensures that both workload types are covered and also enables a comparison to former work. In order to have a broader set of neural networks, other networks who have or had a major impact on research advances or performed well in competitions are chosen. The fact that not all implementations of neural networks are suitable to be trained on mutli-GPU systems without further adjustments also affects the selection.

The implementations used for the study are introduced in chapter 4.3.2.

4.3. Methods and Parameter Settings

In this section, the used code is described in detail.

4.3.1. Microbenchmarks

For the microbenchmarks, we used parts of the Multi-GPU Computing Benchmark Suite mgbench.[1]

The mgbench tests are divided into three categories: Level-0 tests, which are used to get information about the used system. The output of these tests is information about the CPUs, the GPUs and a DMA access matrix. It can be found in the log files `l0-info.log` and `l0-devices.log`. Level-1 tests comprise bandwidth tests for unidirectional data copies, bidirectional data exchanges, DMA between GPUs as well as between the host and GPUs, scatter and scaling tests. Level-2 tests perform multi-GPU matrix multiplications (sgemm) without using inter-GPU communications and game of life to test the correctness and inter-GPU communications. [57]

For this study, only the Level-1 tests are used since the information about the systems was already available and for the more practically-oriented tests, the NN benchmarks are run. In the following, the used Level-1 tests are described in more detail:

halfduplex.cpp

This test performs unidirectional data copies from GPU-to-GPU, CPU-to-GPU and GPU-to-CPU.

When running the test, some flags can be used to adjust the test setting including e.g. the number of repetitions or fixing the GPU index defining from where to where the data should be copied. In this study, only the size flag was changed to get bandwidth and latency figures for different message sizes: 1 KB, 10 KB, 100 KB, 1 MB, 10 MB, 100 MB and 1000 MB were tested.

In the main method (the number of available devices is gotten (`cudaGetDeviceCount`), and peer device memory access is enabled (`cudaDeviceEnablePeerAccess(j, 0)`). As calling this method grants access only unidirectionally, the call must be done for all pairs in both directions. In total, a device can have up to eight peer connections.[58]

After enabling the peer device memory access, the data copies are performed. There are three copy variants that need to be differentiated: GPU-to-GPU, CPU-to-GPU and GPU-to-CPU copies.

[1] mgbench repository on GitHub, master, commit ID: 6f12d3848020af8f718074a30c68e6f0b232bfb3: `https://github.com/tbennun/mgbench`

For GPU-to-GPU copies the method `CopySegment(int a, int b)` (lines 57-127) is called to copy data from GPU a to GPU b. First, memory is allocated on both devices and the devices are synchronized to make sure all other tasks have completed before the data copy is started. After that, a timer is started and the copy process begins. If the chunksize flag is not changed, the data will be copied in one chunk. The actual copy process is done by calling `cudaMemcpyPeerAsync (void * dst, int dstDevice, const void * src, int srcDevice, size_t count)`, where `dst` and `src` indicate the base device pointers of the destination respectively source memory, `dstDevice` and `srcDevice` specify the destination and source devices, and `count` defines the number of bytes to copy. The copy is done asynchronously as the CPU is not involved. The copy process is repeated as often as defined in the repetitions flag. At the end the timer is stopped and an average copy time is calculated. Based on the data size the bandwidth is calculated.

For copies, in which the host is participating, the method `CopyHostDevice(int dev, bool d2h)` is called. This method is very similar to the `CopySegment` method described above. The main difference is that instead of calling `cudaMemcpyPeerAsync` the method `cudaMemcpyAsync(void * dst, const void * src, size_t count, enum cudaMemcpyKind kind)` is called. The last parameter specifies the used transfer type. In this case it is either `cudaMemcpyHostToDevice` for CPU-to-GPU or `cudaMemcpyDeviceToHost` GPU-to-CPU copies.

fullduplex.cpp

This test performs bidirectional data exchanges from GPU-to-GPU. In principle, the test works similar to the one in *halfduplex.cpp*. The main difference is that only GPUs are used and that copies are performed in two directions. The `CopySegment(int a, int b)` method in this file creates two non-blocking CUDA streams and calls the method `cudaMemcpyPeerAsync` twice: First, to copy data from device a to device b using stream b, then to copy data from b to a using stream a.

uva.cu

This test performs unidirectional and bidirectional read and write DMA operations. As in the other two files, there are some flags to adjust the test settings. The only parameter that was changed for this study is the size of the messages.

As in the other two tests, first the device count is retrieved, then peer-to-peer access is enabled. After that, four variants of the test are run:

For the unidirectional write test all possible pairs are tested in one direction. Using the `CopySegmentUVA(int a, int b)` method, memory is allocated and the devices are synchronized. Then the two device buffers are swapped and the method

`DispatchCopy(void *dst, const void *src, const size_t& sz, const size_t& type_size, const dim3& grid, const dim3& block, cudaStream_t stream)` is called to copy the kernel. The unidirectional read test is similar, but the device buffers are not swapped.

In contrast to the unidirectional tests, the bidirectional tests exchange data between GPUs. This means every pair is only tested once but swapping the buffers (for the write test) and copying the data is done twice, to cover both directions.

Changes to Mgbench

As for this study, the goal is to compare the performance of the different interconnects available in the systems, it was necessary to find a way to use either the NVLink or the PCIe connection. This paragraph is only relevant for the experiments run on the two DGX-1 machines since the IBM AC922 only has NVLink as GPU interconnects and a comparison of PCIe versus NVLink is therefore not possible.

In order to measure performance using NVLink connections, the source code of mgbench does not need to be changed. However, to measure PCIe performance, the CUDA peer-to-peer connection, which is described in chapter **??**, needs to be disabled. Therefore, the method `cudaDeviceEnablePeerAccess` must not be called. This can be done by changing a small part of the mgbench code.

In listings 4.1 and 4.2, an example of this change is illustrated for the file *fullduplex.cpp*. For *halfduplex.cpp*, the source code changes are listed in Appendix B. For the DMA read and write tests, peer-to-peer access cannot be disabled. Thus, there are no changes made to *uva.cu*.

Listing 4.1: original file - mgbench/src/L1/fullduplex.cpp

```
132 printf("Enabling␣peer-to-peer␣access\n");
133
134 // Enable peer-to-peer access
135 for(int i = 0; i < ndevs; ++i)
136 {
137     CUDA_CHECK(cudaSetDevice(i));
138     for(int j = 0; j < ndevs; ++j)
139         if (i != j)
140             cudaDeviceEnablePeerAccess(j, 0);
141 }
```

Listing 4.2: changed file - mgbench/src/L1/fullduplex.cpp

```
132 printf("NOT␣Enabling␣peer-to-peer␣access\n");
133 /*
134 // Enable peer-to-peer access
135 for(int i = 0; i < ndevs; ++i)
136 {
137     CUDA_CHECK(cudaSetDevice(i));
138     for(int j = 0; j < ndevs; ++j)
139         if (i != j)
140             cudaDeviceEnablePeerAccess(j, 0);
141 } */
```

4.3.2. Benchmarks for DNN training

For the benchmarks for Deep Neural Network Training parts of the TensorFlow benchmarks[2] were used.

The benchmarks repository consists of scripts for TensorFlow and keras benchmarks. For this study, only the tensorflow benchmarks are used. The implementations of the network models are designed to be as fast as possible, which is why they are interesting for this research. [59]

The distributed tests were run using different networks and batch sizes. All networks were trained on one to eight GPUs. The benchmark can be started using this python program:

```
python tf_cnn_benchmarks.py --num_gpus=N --batch_size=B --model=M
  --variable_update=parameter_server
```

The batch size that is specified by the batch size flag is the local batch size per GPU. When scaling to more GPUs the global batch size is increased by the factor of numbers of GPUs. The variable update method can be defined as well. For these tests "parameter_server" is used. In table 4.1 the used batch sizes for the respective models are listed.

Table 4.1.: Batch sizes for different neural network models

	alexnet	googlenet	inception3	inception4
batch size	512	128	128	64

	resnet50	resnet152	vgg11	vgg16	lenet5
batch size	128	64	128	64	512

[2]TensorFlow benchmarks repository on GitHub: https://github.com/tensorflow/benchmarks

Changes to Tensorflow

In order to not use the NVLink interconnects during the DNN training, it is necessary to conduct a small change inside the TensorFlow source code in order disable the peer-to-peer access similar to the approach with the microbenchmarks. As the NVIDIA application container that is used for the measurements of PCIe performance contains TensorFlow 1.8.0, the change is done in the source code of the branch r1.8 to get comparable results.[3] The part that was changed can be found under the path `/tensorflow/tensorflow/stream_executor/cuda/cuda_gpu_executor.cc` in lines 732 - 735. It corresponds to the same file in the current master branch (TF 1.12) lines 822 - 825. The changed code is presented in listing 4.3. Appendix B.2 shows the original and the changed code lines.

Using this small adjustment, the peer-to-peer connection is invisible, which means that all communication has to go over the host and therefore use the PCIe interconnects.

Listing 4.3: changed file: - branch: r1.8 - cuda_gpu_executor.cc

```
732  bool CUDAExecutor::CanEnablePeerAccessTo(StreamExecutorInterface *other) {
733    CUDAExecutor *cuda_other = static_cast<CUDAExecutor *>(other);
734    return false;
735  }
```

The compilation of TensorFlow r1.8 with the mentioned change was done on the DGX-1 P100 system using the NVIDIA application container Tensorflow 18.07 PY3. First, the existing TensorFlow and Bazel installations were uninstalled. After that, the compiler Bazel was installed in version 0.15.0.[4]

```
nccl2
├── include
│   └── nccl.h
├── lib
│   ├── libnccl.so
│   ├── libnccl.so.2
│   └── libnccl.so.2.2.13
└── NCCL-SLA.txt
```

[3] TensorFlow repository on GitHub, branch r1.8: https://github.com/tensorflow/tensorflow/tree/r1.8

[4] Bazel 0.15.0 installation files: https://github.com/bazelbuild/bazel/releases/tag/0.15.0

In order to use NCCL version 2.2.13, which is installed inside the used NVIDIA application container, some paths to the NCCL files need to be adjusted as they are split up into different locations by default. Therefore, a directory called nccl2 was created under /usr/lib. The content and structure of this directory is as shown above.

After setting the NCCL environment variable accordingly, the bazel build can be configured. The next step is to build the package and finally install the resulting pip package.

As a check that the new TensorFlow version is installed and used, one can run the python code from section 4.3.2. Before the actual benchmark is run, a GPU interconnect map is printed. Using the preinstalled TensorFlow, the matrix for the DGX-1 with P100 GPUs looks as shown in table 4.2.

Table 4.2.: GPU interconnect map - preinstalled TensorFlow version

GPU	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	N	Y	Y	Y	Y	N	N	N
1	Y	N	Y	Y	N	Y	N	N
2	Y	Y	N	Y	N	N	Y	N
3	Y	Y	Y	N	N	N	N	Y
4	Y	N	N	N	N	Y	Y	Y
5	N	Y	N	N	Y	N	Y	Y
6	N	N	Y	N	Y	Y	N	Y
7	N	N	N	Y	Y	Y	Y	N

Using the changed TensorFlow version, the matrix for the same system looks like the one in table 4.3

Table 4.3.: GPU interconnect map - changed TensorFlow version

GPU	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	N	N	N	N	N	N	N	N
1	N	N	N	N	N	N	N	N
2	N	N	N	N	N	N	N	N
3	N	N	N	N	N	N	N	N
4	N	N	N	N	N	N	N	N
5	N	N	N	N	N	N	N	N
6	N	N	N	N	N	N	N	N
7	N	N	N	N	N	N	N	N

The change in the TensorFlow source code and step-by-step instructions on how to compile and install the changed TensorFlow version are shown in appendix B.2.

5. Experiment Results

The following chapter presents the experiments' results. First, the results from the microbenchmarks are presented for each system. Then the results from the DNN benchmarks are presented including the profiling of the workloads and the performance. At the end of this chapter, the results are discussed and limitations and challenges regarding the experiment design and realization are explained. In order to maintain readability, not all figures are printed in this chapter. The full range of figures showing the experiments' results are to be found in appendix C.1 and C.2.

5.1. Microbenchmarks

Using the NVIDIA System Management Interface (nvidia-smi) one can query information about the used system. The query `nvidia-smi topo` prints topology information on the GPUs used in the system including the information how the GPUs are connected to each other. To understand the microbenchmark results, the queries are run on all systems. The results for DGX-1 P100 are shown in listing 5.1 and 5.2. To make it more readable, only GPU connections are displayed. The full output of the queries for all systems can be found in appendix C.1.1. The command `nvidia-smi topo -m` shows the NVLink connections, `nvidia-smi topo -mp` shows PCIe connections.

Listing 5.1: nvidia-smi topo -m DGX-1 P100

```
nvidia-smi topo -m

GPU0  X   NV1 NV1 NV1 NV1 SYS SYS SYS
GPU1  NV1 X   NV1 NV1 SYS NV1 SYS SYS
GPU2  NV1 NV1 X   NV1 SYS SYS NV1 SYS
GPU3  NV1 NV1 NV1 X   SYS SYS SYS NV1
GPU4  NV1 SYS SYS SYS X   NV1 NV1 NV1
GPU5  SYS NV1 SYS SYS NV1 X   NV1 NV1
GPU6  SYS SYS NV1 SYS NV1 NV1 X   NV1
GPU7  SYS SYS SYS NV1 NV1 NV1 NV1 X
```

The abbreviation NV# stands for a connection using a bonded set of # NVLinks, SYS shows a connection traversing PCIe as well as the SMP interconnect (e.g. QPI or XBus),

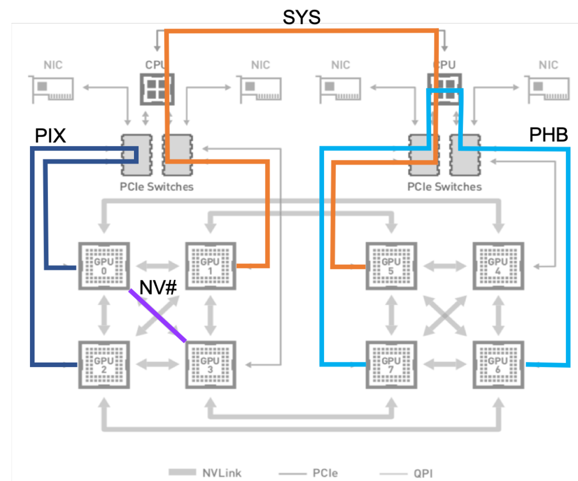


Figure 5.1.: Exemplary connection types used in nvidia-smi topo query result, adapted from [29]

PHB is used for a connection traversing PCIe as well as the CPU, and PIX displays a connection between GPUs connected to the same PCIe switch.

In figure 5.1 exemplary connections are shown.

Listing 5.2: nvidia-smi topo -mp DGX-1 P100

```
nvidia-smi topo -mp

GPU0   X   PIX PHB PHB SYS SYS SYS SYS
GPU1   PIX X   PHB PHB SYS SYS SYS SYS
GPU2   PHB PHB X   PIX SYS SYS SYS SYS
GPU3   PHB PHB PIX X   SYS SYS SYS SYS
GPU4   SYS SYS SYS SYS X   PIX PHB PHB
GPU5   SYS SYS SYS SYS PIX X   PHB PHB
GPU6   SYS SYS SYS SYS PHB PHB X   PIX
GPU7   SYS SYS SYS SYS PHB PHB PIX X
```

5.1.1. DGX-1 P100 Results

In figure 5.2 the bandwidths for unidirectional data copies of different sizes on the DGX-1 with P100 GPUs are shown. The theoretical bandwidths for PCIe 3.0 (15.754 GB/s) and NVLink 1.0 (20 GB/s) are also displayed. Even for small data copies (10-100 KB) the NVLink connection delivers more bandwidth than PCIe. For large data copies

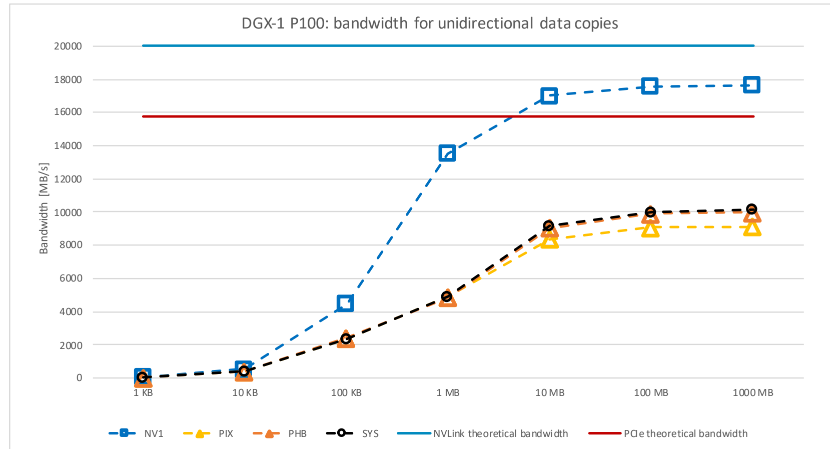


Figure 5.2.: DGX-1 P100 bandwidth for unidirectional data copies

(10 - 1000 MB) it achieves more bandwidth than the theoretical bandwidth of PCIe and at the largest data size tested, it achieves 17.626 GB/s, which is 88% of the theoretical bandwidth.

In contrast to that, even the fastest PCIe connection via a PCIe switch and the CPU (PHB) only achieves 9.987 GB/s, which is 62% of its theoretical bandwidth. It is also interesting to see that even the closest connection (PIX), which only goes through one PCIe switch and thus should be the fastest one of the PCIe connection types, is actually the slowest one.

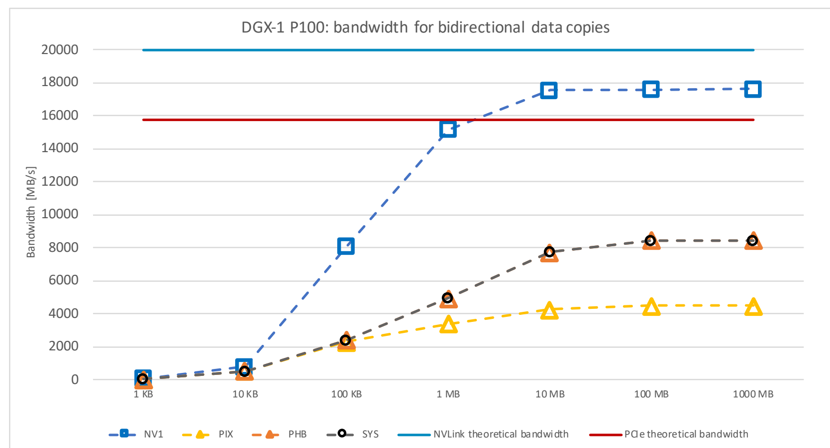


Figure 5.3.: DGX-1 P100 bandwidth for bidirectional data copies

In figure 5.3 the bandwidths for bidirectional data copies of different sizes on the

DGX-1 with P100 GPUs are shown. Similar to the unidirectional data copies, NVLink outperforms PCIe at all message sizes. For 1000 MB data copies, it achieves 88% of its theoretical bandwidth, while PCIe (PHB and SYS) only achieves 54% of its theoretical bandwidth. The PIX connection again is the slowest, only achieving 29% of the theoretical PCIe bandwidth.

In figure 5.4 latencies for the unidirectional data copies are displayed. The latency for a copy process from GPU 0 to all other GPUs is displayed. The figure does not change a lot for other starting GPUs. A cluster of close GPUs can clearly be identified for the NVLink figures. As shown in listing 5.1 GPU 0 is connected via one NVLink to GPUs 1-4. For all other GPUs, the data is sent over PCIe and QPI. For the tests, in which peer-to-peer access was disabled, there is almost no difference in the latency. Again we see that especially for the closest connection copying data needs the most time. This behaviour is also very noticeable in figure C.6, which shows the latencies for bidirectional data copies.

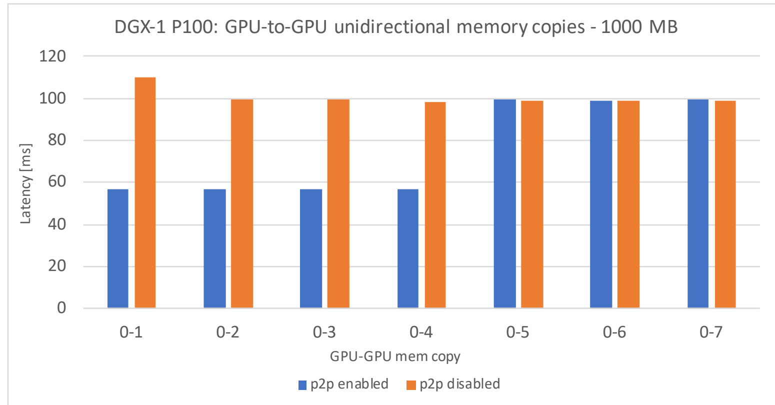


Figure 5.4.: Latency for GPU-to-GPU unidirectional memory copies on DGX-1 P100

In figure 5.5 the bandwidth for the DMA read operation is shown. As Direct Memory Access is not possible without enabling peer-to-peer access, the bandwidth of PCIe could only be measured for unidirectional operations including the CPU (host). For large data sizes the PCIe connection from the host to the GPUs achieves a bandwidth of 10.5 GB/s, which is 67% of its theoretical bandwidth. The NVLink connections reach 75% of the theoretical bandwidth for unidirectional DMA reads and 62% for bidirectional DMA reads. For the DMA write operation both connection types perform better: the PCIe connection reaches 76% of its theoretical bandwidth and the NVLink connections 83% for unidirectional as well as bidirectional write operations.

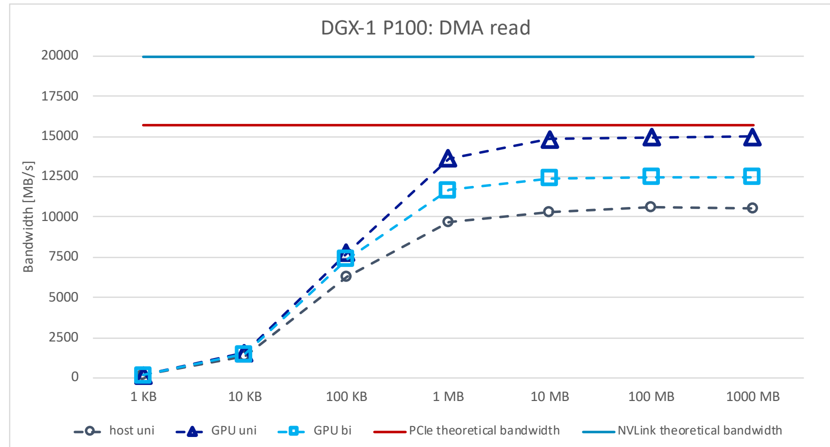


Figure 5.5.: DGX-1 P100 bandwidth for DMA read

5.1.2. DGX-1 V100 Results

In the DGX-1 system with V100 GPUs the connections between GPUs differ from the ones in the DGX-1 system with P100 GPUs in two aspects: NVLink 2.0 has higher bandwidth and there are more connections. In figure 5.6 the bandwidths for unidirectional data copies on this system are shown. The theoretical bandwidth of NV2 connections accounts for 50 GB/s, the one for NV1 connections for 25 GB/s and PCIe 3.0 remains at 15.754 GB/s. For small data sizes, NV1 and NV2 connections achieve similar bandwidths and already outperform the PCIe connections. For 1000 MB memory copies NV2 connections achieve a bandwidth of 46.23 GB/s, which is 92% of its theoretical bandwidth, NV1 connections achieve 23.128 GB/s, which is 93% of theoretical bandwidth. The PCIe connections PHB and SYS again are quite similar and achieve 63% of PCIe's theoretical bandwidth. PIX only achieves 56%.

Figure C.9 shows the bandwidths for bidirectional data copies on the DGX-1 V100 system. NV1 and NV2 connections achieve 93% of their theoretical bandwidth, while PHB and SYS achieve 52%. Again, the PIX connection is the slowest, reaching only 28% of the theoretical bandwidth. In figure C.12 the latencies for the unidirectional data copies are displayed. As for the DGX-1 P100 test results, only memory copies from GPU 0 to all other GPUs are illustrated as the figures for other sending GPUs look very similar. Especially for large data copies three clusters of GPUs can be identified when peer-to-peer access is enabled: Two are connected via two NVLink connections, another two are connected via a single NVLink connection and the remaining three are connected over PCIe and QPI. Figure C.13 shows the latencies for bidirectional memory copies. When peer-to-peer is disabled, there is almost no difference in latency when

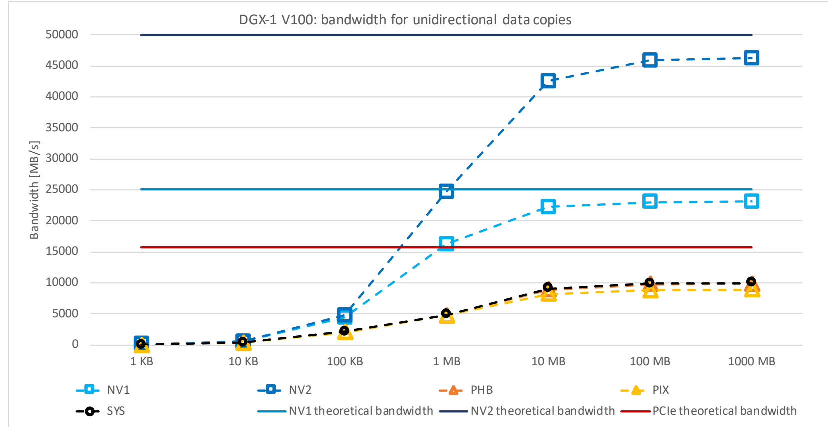


Figure 5.6.: DGX-1 V100 bandwidth for unidirectional data copies

copying large data sizes. In C.13(e) the anomaly of the PIX connection is seen again. It takes almost double the time to bidirectionally exchange data from GPU0 to GPU1 than from GPU0 to any other GPU.

Figure 5.7 presents the bandwidth for the DMA read operation. For large data sizes the PCIe connection from the host to the GPUs achieves a bandwidth of 10.6 GB/s, which is 67% of its theoretical bandwidth. The NVLink connections over a bonded set of 2 NVLinks reach 87% of the theoretical bandwidth (43.693 GB/s) for unidirectional DMA reads and 79% (39.326 GB/s) for bidirectional DMA reads. The single NVLink connections achieve 82% (20.485 GB/s) for unidirectional reads and 79% (19.666 GB/s) for bidirectional reads. For the DMA write operation all connection types perform better: the PCIe connection reaches 76% of its theoretical bandwidth, the single NVLink connections 82% for unidirectional and 87% for bidirectional write operations, and the paired NVLink connections achieves 87% of the theoretical bandwidth for both, unidirectional and bidirectional, DMA write operations.

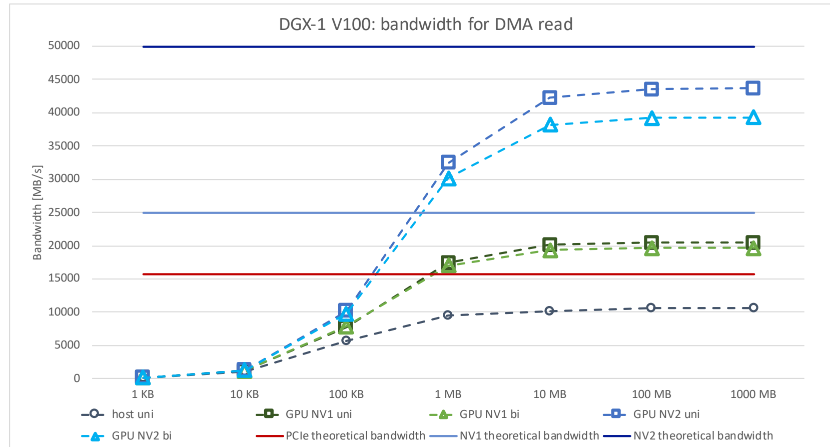


Figure 5.7.: DGX-1 V100 bandwidth for DMA read

5.1.3. AC922 Results

Figure 5.8 shows the bandwidths for unidirectional memory copies of different sizes as well as the theoretical bandwidth of the triple NVLink 2.0 connections (75 GB/s) and the theoretical bandwidth of the XBus between the two Power9 CPUs (64 GB/s). Memory copies to directly connected GPUs achieve 67.175 GB/s or 90% of theoretical bandwidth. Memory copies to GPUs connected to the other CPU only reach 27.847 GB/s or 44% of the theoretical XBus bandwidth. Memory copies from or to the closer host achieve a bandwidth of 89% of the theoretical bandwidth, for copies from to the other host, the bandwidth only accounts for 37.296 GB/s, which equals 58% of theoretical bandwidth.

For the bidirectional copies directly connected GPUs also achieve 90%. However, copies to other GPUs connected via SYS (XBus) only reach 16.484 GB/s of bandwidth, which is 26% of theoretical bandwidth. The complete bandwidths for bidirectional memory copies with different message sizes can be found in figure C.16.

The unidirectional DMA read operations achieve comparably high bandwidths: For NV3 connected GPUs a bandwidth of 63.555 GB/s was measured, which equals 85% of the theoretical bandwidth. The same is true for DMA read operations from a host to a directly connected GPU: 85%. DMA reads from one GPU to a GPU in the other cluster reaches a bandwidth of 33.434 GB/s, which is 52% of the theoretical bandwidth, and the DMA read from a host to a GPU from the other cluster achieves 39.507 GB/s or 62% of theoretical bandwidth. The bidirectional DMA reads show a little less performance: For NV3 connections 57.201 GB/s (76%) were measured and for SYS connections 17.64 GB/s (28%) were measured.

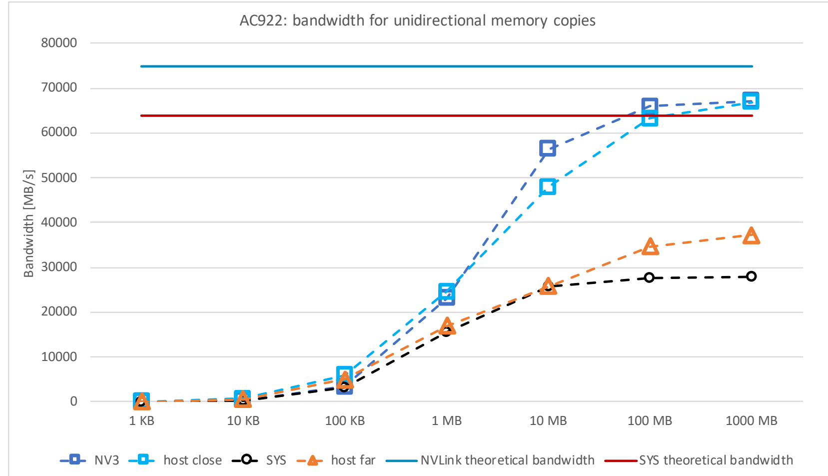


Figure 5.8.: AC922 bandwidth for unidirectional data copies

The bandwidths for DMA write operations account for: 85% of theoretical bandwidths for NV3 connections (GPU-to-GPU & CPU-to-GPU) and 56% respectively 44% for hosts respectively GPUs from the other cluster at unidirectional writes. For bidirectional DMA writes 63.555 GB/s (85%) was measured for NV3 connections and 16.402 GB/s (26%) was measured for SYS connections.

5.2. Benchmarks for DNN Training

In this section, the results of the benchmarks for Deep Neural Network training are described. First, the results of the workload analysis using the NVIDIA profiling tool nvprof [1] are presented. These show, which workloads are more communication- or more computation-bound.

Then, all networks are trained on the DGX-1 P100 and on the DGX-1 V100 using one to eight GPUs and several batch sizes. The training is done using the preinstalled TensorFlow version to see the performance using NVLink and using the changed TensorFlow version to obtain performance measurements for only using PCIe interconnects. The performance is measured in processed images per second.

5.2.1. Workload Analysis using Nvprof

All networks are trained on eight GPUs using the profiler nvprof. Only the CUDA memcopy commands are listed. `CUDA memcpy DtoH` refers to memory copies from one of the

[1] see `https://docs.nvidia.com/cuda/profiler-users-guide/index.html`

hosts to a device, `CUDA memcpy HtoD` shows the percentage of total time used for copies from one of the CPUs to on of the GPUs, `CUDA memcpy PtoP` refers to communication between the GPUs.

	CUDA memcpy DtoH	CUDA memcpy HtoD	CUDA memcpy PtoP	sum
alexnet	11.25%	9.22%	5.44%	25,91%
googlenet	1.06%	1.19%	0.72%	2,97%
inception3	1.91%	1.86%	1.06%	4,83%
inception4	2.85%	2.77%	1.65%	7,27%
resnet50	3.39%	3.52%	1.94%	8,85%
resnet152	4.74%	5.03%	3.27%	13,04%
vgg11	9.14%	7.44%	6.96%	23,54%
vgg16	13.79%	12.10%	6.57%	32,46%
lenet	6.00%	6.17%	3.44%	15,61%

Table 5.1.: Profiling results nvprof, 8 GPUs

From the results in table 5.1 it can be assumed that disabling the faster NVLink connections should have more effect on AlexNet, VGG11, VGG16 and LeNet as on the remaining networks, because these seem to be more communication-intensive. It can also be assumed that the networks, which perform a lot communication from host to device would benefit from being trained on AC922.

5.2.2. Training Performance with PCIe only and with NVLink / PCIe

In the following sections the training performance using only PCIe connections compared to using NVLink is shown.

Because of clarity reasons not all figures are included in the text. The complete collection of figures can be found in appendix C.2.

AlexNet was trained using a local batch size of 512. In figure 5.9 the number of images per second is shown for training the network on one to eight GPUs. For the training on the DGX-1 P100 we can see a performance decrease when using only PCIe and not the NVLink connections (p2p disabled) for all training sessions independent from how many GPUs were used. On the DGX-1 V100 we can also see a performance increase when using NVLink (p2p enabled). However, especially the numbers for training on four and seven GPUs show some irregularities, which should be studied

further. Another measurement, which is unexpected, is that training on the DGX-1 V100 for one GPU shows different results for p2p enabled and p2p disabled.

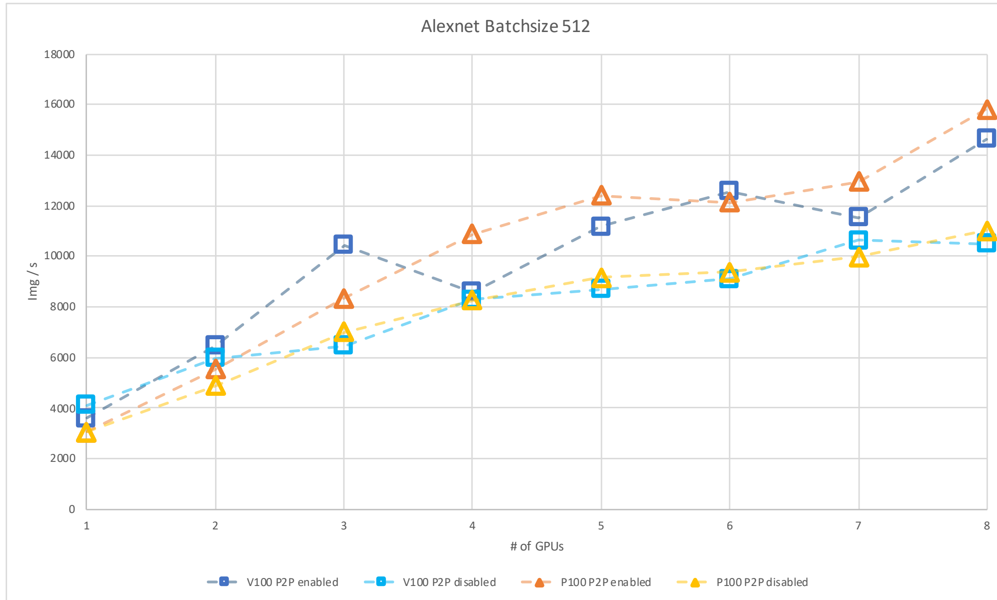


Figure 5.9.: Scaling training of alexnet with (p2p enabled) and without NVLink (p2p disabled)

GoogLeNet was trained using a local batch size of 128. The performance is shown in figure 5.10. On the DGX-1 P100 we can see almost no differences in performance for scaling the training on up to five GPUs. When scaling beyond this, we see that training the network with the changed TensorFlow version achieved even higher performance than training it with the preinstalled version using the NVLink connections. This result is surprising and should be studied further. On the DGX-1 V100 we can largely see the same behavior.

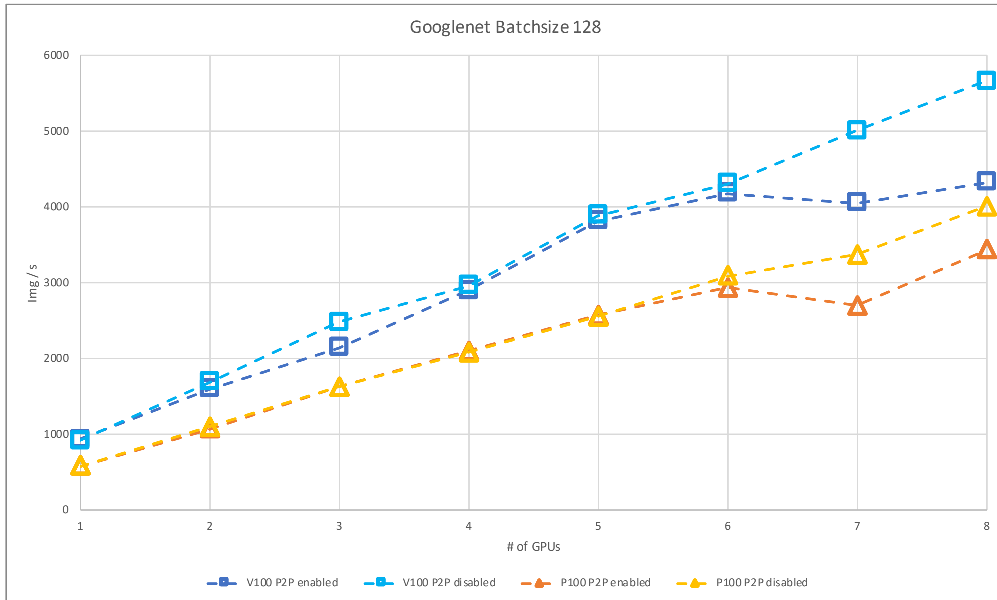
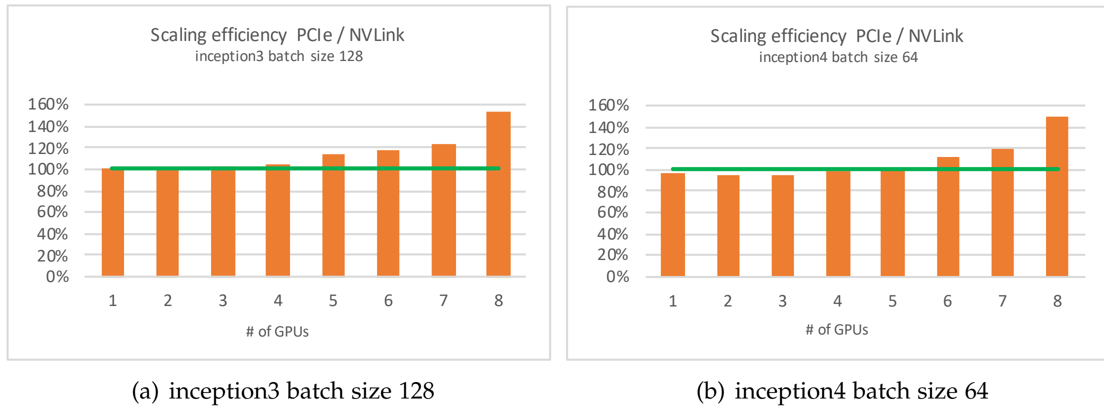


Figure 5.10.: Scaling training of GoogLeNet with (p2p enabled) and without NVLink (p2p disabled)

Inception v3 and Inception v4 Inception v3 was trained with a local batch size of 128 and Inception v4 was trained with a local batch size of 64. Figures 5.11(a) and 5.11(b) show the scaling efficiency of using PCIe interconnects versus NVLink interconnects on DGX-1 P100. Less than 100% means that training the networks only with PCIe is slower than with NVLink. More than 100% means that disabling peer-to-peer access makes training faster, which is a surprising result. Figures 5.12(a) and 5.12(b) show the same measurements for training Inception on the DGX-1 V100 machine.

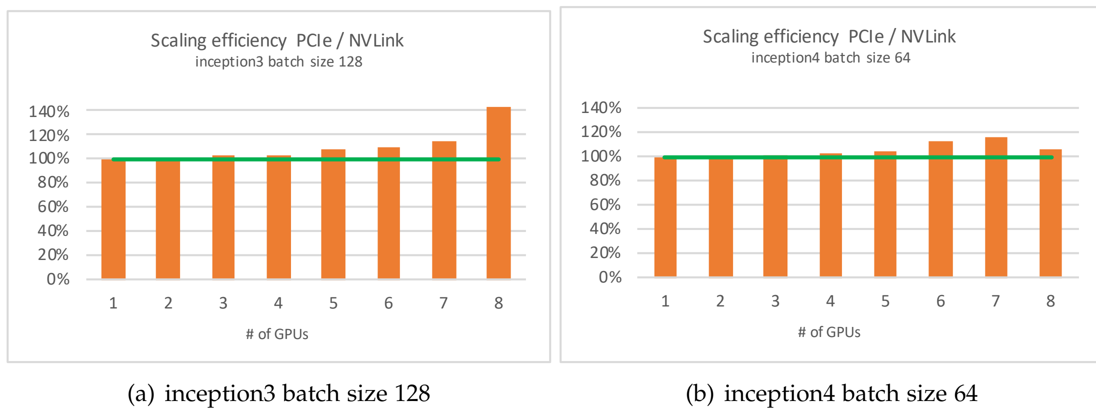
For both machines it can be observed that there is no effect for scaling to up to four GPUs. Training on more than four GPUs shows surprising results as well.



(a) inception3 batch size 128

(b) inception4 batch size 64

Figure 5.12.: Scaling efficiency using only PCIe compared to using NVLink on DGX-1 V100

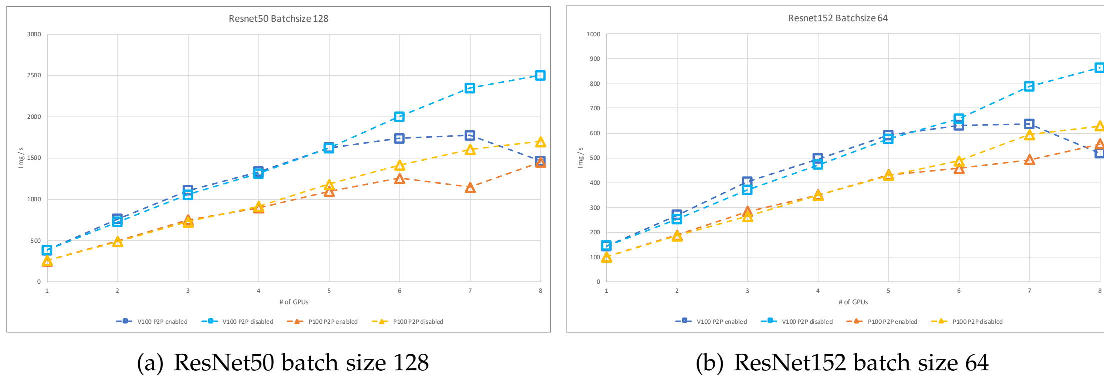


(a) inception3 batch size 128

(b) inception4 batch size 64

Figure 5.11.: Scaling efficiency using only PCIe compared to using NVLink on DGX-1 P100

ResNet50 and ResNet152 The performance results for ResNet50 and ResNet152 show similar behavior. ResNet50 was trained using a local batch size of 128, for ResNet152 the local batch size was 64. Of course, the absolute values of images processed per second differ as ResNet152 has significantly more layers than ResNet50 but the scaling behavior is similar: figure 5.13(a) shows scaling for ResNet50 and figure 5.13(b) shows scaling for ResNet152. For the training on up to five GPUs the difference between using NVLink and not using NVLink is marginal on both machines. For the training on six



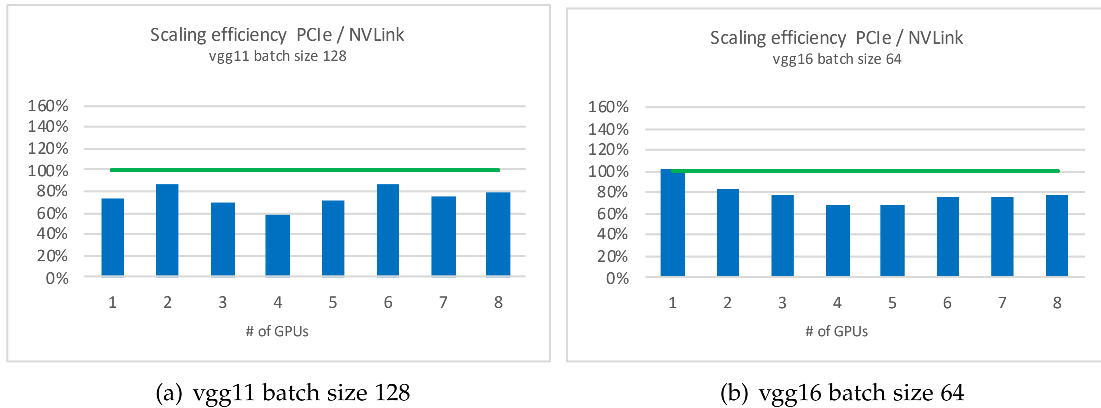
(a) ResNet50 batch size 128

(b) ResNet152 batch size 64

Figure 5.13.: Scaling efficiency with (p2p enabled) and without NVLink (p2p disabled)

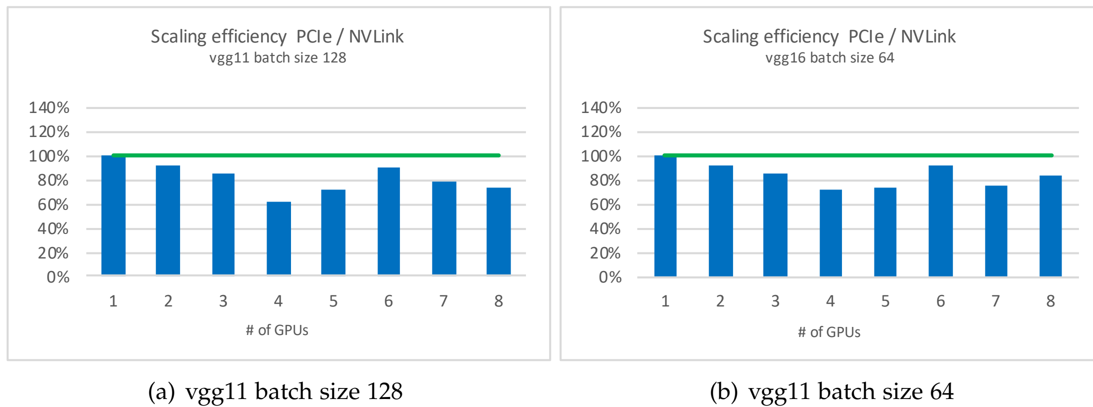
or seven GPUs the same inconsistency occurs as could be observed for GoogLeNet and Inception. Training on eight GPUs produced odd results for both networks on the DGX-1 V100: Using the NVIDIA TensorFlow version with p2p enabled resulted in lower performance than using the TensorFlow version where p2p was disabled. It even processed less images than on the DGX-1 P100 machine.

VGG11 and VGG16 VGG11 was trained with a batch size of 128 and VGG16 was trained using a batch size of 64. For both networks a decrease of performance is observed when disabling NVLink on both DGX-1 machines. Figures 5.14(a) and 5.14(b) show scaling of both networks on the DGX-1 P100. Figures 5.15(a) and 5.15(b) show the same for the DGX-1 V100 machine. For all scaling options training without NVLink is slower than with NVLink. In figure 5.15(a) it can be seen that training on one GPU also earned different performances.



(a) vgg11 batch size 128 (b) vgg16 batch size 64

Figure 5.15.: Scaling efficiency using only PCIe compared to using NVLink on DGX-1 V100



(a) vgg11 batch size 128 (b) vgg11 batch size 64

Figure 5.14.: Scaling efficiency using only PCIe compared to using NVLink on DGX-1 P100

LeNet The results for LeNet5 (batch size 512) are similar to the ones from AlexNet. For all scaling options a decrease of performance can be observed if only PCIe interconnects are used. However, also in these measurements, irregularities can be seen for the training on the DGX-1 V100 when using the preinstalled NVIDIA TensorFlow version. Figure 5.16 shows the results.

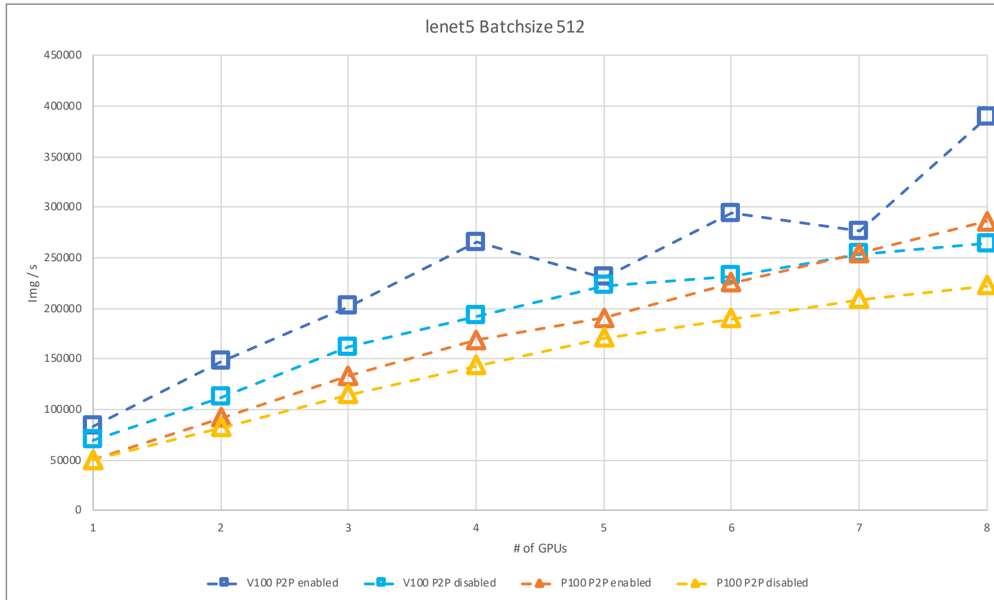


Figure 5.16.: Scaling training of lenet5 with (p2p enabled) and without NVLink (p2p disabled)

5.3. Discussion of Results

The conducted experiments generated some interesting results, which are summarized in the following.

Microbenchmarks

The microbenchmarks showed that NVLink achieves a higher bandwidth than PCIe for all tested operations. One aspect, which is interesting, is that NVLink's practical bandwidth for all operations is more than 61% of its nominal bandwidth (based on the results for the largest message sizes). In all operations except for bidirectional data copies on the DGX-1 P100, it is even higher than 74%. The highest percental bandwidth measured is 93%.

For the PCIe connections the highest percental bandwidth measured is 76% for DMA write operations from the CPU to the GPU on both DGX-1 systems. The lowest is 28% for the bidirectional data copies from one GPU to another GPU connected to the same PCIe switch. This is a surprising result, which can be observed on both DGX-1 systems and which suggests that PCIe switches might be a bottleneck for the data copies.

The XBus connection between the two CPUs in the AC922 system also by far does not

achieve its nominal bandwidth of 64 GB/s: The lowest percental bandwidth measured is 28%, the highest is 58%. However, this is still more than the nominal bandwidth of PCIe3.0 (15.754 GB/s).

For data copies or DMA read / write operations that include the host, an advantage of the AC922 system becomes clearly visible: The NVLink connections from the CPU to the GPU achieve 85% to 89% of their nominal bandwidth, which is 75 GB/s. In the DGX systems these operations are done via the PCIe connections and achieve 66% to 75% of their nominal bandwidth, which is only 15.754 GB/s.

These results suggest that for communication-intensive one should derive benefits from using NVLink instead of PCIe interconnects. Especially for workloads, in which a lot of communication occurs between GPU and CPU, the AC922 should show drastically higher performance.

Benchmarks for DNN Training

Using the results from the profiling, we can now classify AlexNet, VGG11, VGG16 and LeNet5 as communication-intensive neural networks. The remaining neural networks can be classified as computation-intensive.

For the communication-intensive workloads the training performance results fit to what one would expect: Using PCIe instead of NVLink leads to less performance in terms of fewer images processed per second.

However, for the computation-intensive workloads this behavior cannot be observed. In some cases disabling peer-to-peer and using only PCIe even lead to higher performance. This is a finding, which should be studied further in future research. Ideas for further study are presented in chapter 5.4.

However, these findings support the used approach using which neural networks can be classified. It also shows that in order to decide if high-performance interconnects such as NVLink are useful, the workloads should be well understood in terms of operation types. If a system with high-performance interconnects is available, users should put effort into optimizing their algorithms in a way that high bandwidth delivers high performance.

In terms of scaling, it can be observed that for the communication-intensive workloads, scaling up to eight GPUs with p2p disabled has a more negative effect in terms of reaching linear scaling than for the computation-intensive workloads. This can be explained by the rather low bandwidth achieved for SYS connections over QPI.

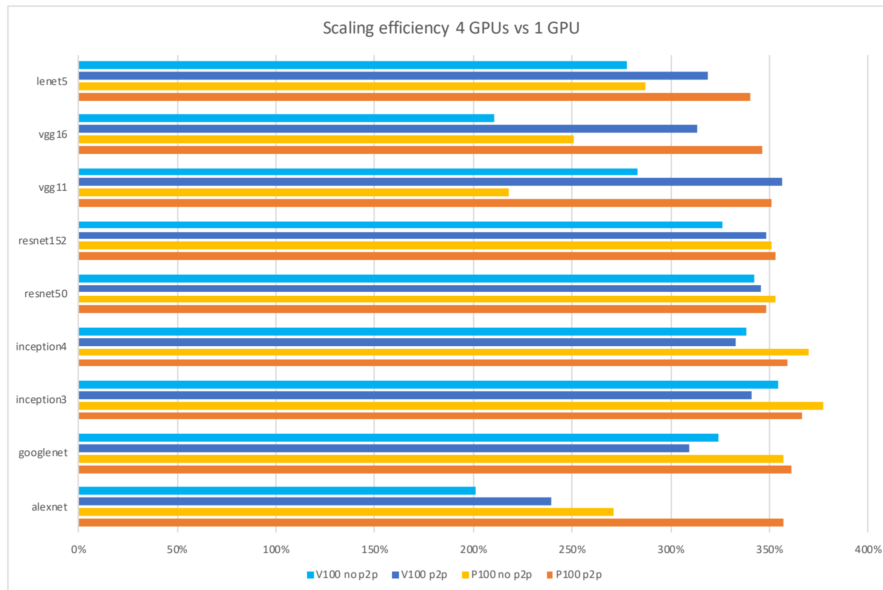


Figure 5.17.: Scaling efficiency: Training on 4 GPUs compared to 1 GPU)

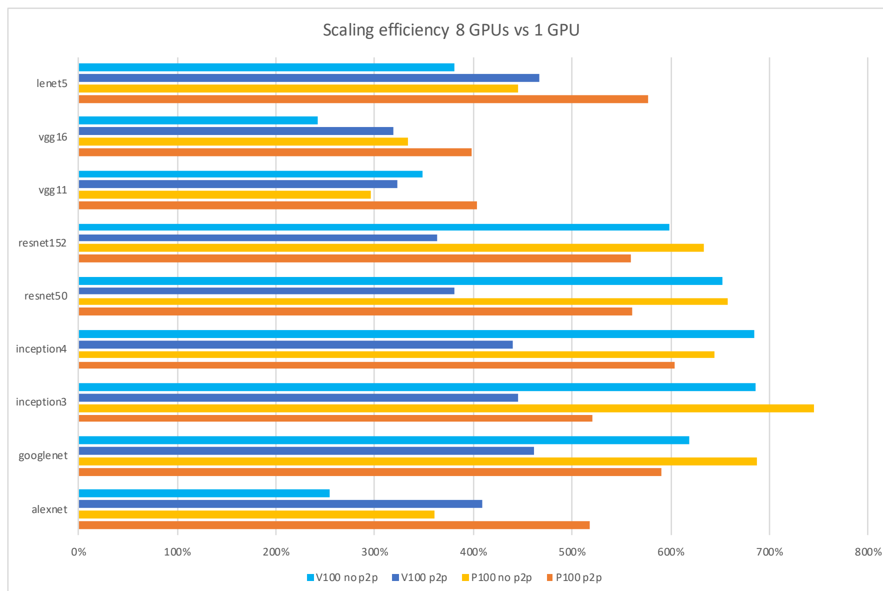


Figure 5.18.: Scaling efficiency: Training on 8 GPUs compared to 1 GPU)

5.4. Limitations and Challenges

During the design and run of the experiments, three major challenges needed to be overcome:

- Getting access to the systems used for the experiments took longer than expected. This was especially the case for the IBM Power system. Due to the system being available only to early users, documentation about login and scheduling was not yet available. For the two DGX-1 systems documentation and access was not an issue but sometimes reservation time became a bottleneck especially on the DGX-1 system with V100 GPUs.
- The workloads used in the Fathom paper described in chapter 2.1.5 are partly outdated and not available for current Tensorflow versions. Also, the license to use the TIMIT dataset, which was used to train the very interesting neural network DeepSpeech requires an LDC membership, which TUM doesn't hold.
- In order to compare the performance of training deep neural networks using different interconnects NVLink needed to be disabled in the systems. This was only possible by software changes in the DL library Tensorflow. Finding the right lines of code to make the change was one of the challenges. Compiling the changed Tensorflow version from source was another. First, it was tried to use the NVIDIA application container "CUDA 9.0 CUDNN7.1-DEVEL AND PGILINUX 2018-184", which has no Tensorflow installation, but CUDA and NCCL. Trying to compile the Tensorflow version on this image generated numerous errors. Finally, the NVIDIA application container "Tensorflow 18.07 PY3" was used and Tensorflow was uninstalled. After some minor changes described in 4.3.2 the compilation was straight-forward and the once compiled package could be easily installed on the other DGX-1 system.

Due to these challenges and experiment design decisions, some limitations apply to the results of this thesis:

- The benchmarks for DNN training could not be run on the IBM AC922 System due to availability and time constraints.
- As research on hardware for neural network architectures other than CNN is quite rare, benchmark implementations for other architectures were hard to find. Therefore, the benchmarks for DNN training only comprise convolutional neural networks.

- The Tensorflow version used to measure NVLink bandwidth, is the version from NVIDIA's application container. One might get slightly different results if the Github version used for PCIe would be compiled without the changes for PCIe and used as comparison to the one where peer-to-peer is disabled.
- For the training of the neural networks from the Tensorflow Github repository the variable update method parameter server was used, one could also use NCCL allreduce, which might lead to better performance for the NVLink connections.

6. Future Work

6.1. Hardware Selection

The workloads used in this thesis should definitely be run on the IBM AC922 system as well in order to have a holisitc comparison of the presented systems. Especially the results of workloads with a high amount of device-to-host and host-to-device communication could deliver insightful findings. Also, it would be interesting to see how the scaling performance behaves for the training on more than two (8355-GTG model) respectively three GPUS (8355-GTW model) since this would show the practical performance of the XBus connection, which was slower than expected in the microbenchmarks performed in this study.

Furthermore, the NVIDIA DGX-2 system could be included as it uses NVSwitch with NVLink 2.0 and 16 V100 GPUs with 32GB of GPU memory instead of 16GB. It would be interesting to see, which effects these changes in the system have. According to NVIDIA, for certain workloads, this system delivers 10-times the performance of the DGX-1 V100 system used here.[1]

6.2. Deep Learning Workloads

In this thesis only CNNs used for computer vision tasks are covered. In order to get a broader and deeper understanding of how the performance of training neural networks is affected by GPU interconnects, other domains, in which deep learning is currently used, should be explored. Other domains could be speech recognition or machine translation. The OpenNMT initiative could be used as a starting point.[2]

As it has been theoretically elaborated in chapter 2.1.5 that recurrent neural networks and fully-connected neural networks stress the interconnects more than convolutional neural networks, these network architectures should be covered by future research as well. On top of that, hybrid models should be taken into account. For example, DeepSpeech, which has been identified as an interesting model for the comparison of

[1] see https://www.nvidia.com/content/dam/en-zz/es_em/Solutions/Data-Center/dgx-2/nvidia-dgx-2-datasheet.pdf

[2] see http://opennmt.net/

performance using different types of interconnects, has been released in two newer versions, Deep Speech 2 and Deep Speech 3. It would be interesting to see how the model architecture evolved and what impacts it has on this reseach.

Additionally, model-parallel training of deep neural networks should be brought into focus.

Also, one could optimize a deep learning workload in order to stress the bandwidth of interconnects. This will lead to a stronger This might lead to less comparability, which was a key aspect of this thesis, but it

Having all these ideas in mind, the basis for further experiments should be a deeper analysis of DNN training workloads. This could be achieved by a more exhaustive profiling in order to determine which operations dominate the execution time, again focusing on the data movements as done in this thesis.

6.3. Comparison of NVLink and PCIe for DNN Training

As the test results for the training of Deep Neural Networks partly show inconsistency to what was expected, the Tensorflow version used for comparison should be examined again. It would be interesting to see if using the GitHub Tensorflow version of release 1.8.0[3] without changes delivers results different from the results that were gotten in the presented experiments using the NVIDIA preinstalled Tensorflow version.

[3]see `https://github.com/tensorflow/tensorflow/tree/r1.8`

7. Conclusion

In this thesis different GPU interconnects were compared based on their performance with regards to training deep neural networks. Therefore, different neural network architectures were studied and an idea how to classify neural networks was developed.

The bandwidth for data copies and DMA read and write operations were identified as metrics that can be used to evaluate the performance of interconnects for deep learning workloads. Microbenchmarks were performed in order to figure out the practically achievable bandwidth of interconnects. It was shown that NVLink can deliver a higher percentage of its nominal bandwidth than PCIe can and thus NVLink delivers considerably higher bandwidths.

In the DNN benchmarks it was shown that the identified metrics are mainly important for communication intensive workloads. A classification of deep neural networks based on their training workloads was defined. It could be demonstrated that using PCIe for communication intensive workloads leads to a lower performance in terms of images processed per second than using NVLink.

However, it was also shown that NVLink cannot outperform PCIe for all studied workloads. Only for four out of nine workloads a better performance in terms of images processed per second is achieved. This stresses the importance of a classification for deep learning workloads.

Regarding the comparison of multi-GPU systems, it could be evidenced for most workloads that the DGX-1 V100 systems outperforms the DGX-1 P100 system if the NVLink connections are used. When using the PCIe connections this behavior could also be seen for many workloads except for the scaling to eight GPUs. This might be caused by the slow PIX connection, which also showed lower performance in the microbenchmarks.

Based on the results of the microbenchmarks and the results from the profiling of the training workloads, one could assume that AC922 would outperform the DGX-1 systems because of its fast interconnects between CPU and GPU. However, it must be kept in mind that the AC922 system used in this study only has half of the GPUs that the DGX-1 V100 system has. The assumption could neither be proven nor disproved and

should be studied in future research considering the classification of neural networks developed in this thesis.

In summary, it can be said that NVLink outperforms the PCIe interconnects for communication-intensive workloads. However, for the majority of networks studied in this thesis, the advantage of having higher bandwidth did not lead to a performance increase in terms of speed of image processing during the training phase. This finding suggests that there cannot be a single answer to the question if training deep neural networks benefits from using high bandwidth interconnects but having an understanding of how the algorithm works and especially how often data needs to be transferred is still necessary.

A. Multi-GPU Computing Systems

B. Experiment Setup

B.1. Changes to mgbench

B.1.1. Run only relevant tests

Listing B.1: original file - mgbench/run.sh

```
83  # Run L1 tests
84  echo ""
85  echo "L1␣Tests"
86  echo "--------"
87
88  echo "1/8␣Half-duplex␣(unidirectional)␣memory␣copy"
89  ./build/halfduplex > l1-halfduplex.log
90
91  echo "2/8␣Full-duplex␣(bidirectional)␣memory␣copy"
92  ./build/fullduplex > l1-fullduplex.log
93
94  echo "3/8␣Half-duplex␣DMA␣Read"
95  ./build/uva > l1-uvahalf.log
96
97  echo "4/8␣Full-duplex␣DMA␣Read"
98  ./build/uva --fullduplex > l1-uvafull.log
99
100 echo "5/8␣Half-duplex␣DMA␣Write"
101 ./build/uva --write > l1-uvawhalf.log
102
103 echo "6/8␣Full-duplex␣DMA␣Write"
104 ./build/uva --write --fullduplex > l1-uvawfull.log
105
106 #echo "7/8 Scatter-Gather"
107 #./build/scatter > l1-scatter.log
108
109 #echo "8/8 Scaling"
110 #./build/sgemm -n 4096 -k 4096 -m 4096 --repetitions=100 --regression=false
        --scaling > l1-scaling.log
```

B.1.2. Run the tests using PCIe

mgbench/src/L1/fullduplex.cpp

Listing B.2: changed file - mgbench/src/L1/fullduplex.cpp

```
132 printf("NOT␣Enabling␣peer-to-peer␣access\n");
133 /*
134 // Enable peer-to-peer access
135 for(int i = 0; i < ndevs; ++i)
136 {
137     CUDA_CHECK(cudaSetDevice(i));
138     for(int j = 0; j < ndevs; ++j)
139         if (i != j)
140             cudaDeviceEnablePeerAccess(j, 0);
141 } */
```

mgbench/src/L1/halfduplex.cpp

Listing B.3: changed file - mgbench/src/L1/halfduplex.cpp

```
224 printf("NOT␣Enabling␣peer-to-peer␣access\n");
225 /*
226 // Enable peer-to-peer access
227 for(int i = 0; i < ndevs; ++i)
228 {
229     CUDA_CHECK(cudaSetDevice(i));
230     for(int j = 0; j < ndevs; ++j)
231         if (i != j)
232             cudaDeviceEnablePeerAccess(j, 0);
233 } */
```

B.1.3. Run the tests using different message sizes

halfduplex.cpp 10 MB

Listing B.4: changed file - mgbench/src/L1/halfduplex.cpp

```
37 DEFINE_uint64(size, 10*1024*1024, "The␣amount␣of␣data␣to␣transfer");
38 DEFINE_uint64(chunksize, 0, "If␣not␣zero,␣fragments␣the␣data␣into␣chunksize-
       byte␣chunks");
39 DEFINE_uint64(repetitions, 100, "Number␣of␣repetitions␣to␣average");
40 DEFINE_bool(sync_chunks, false, "If␣true,␣synchronizes␣at␣the␣end␣of␣each␣
       fragment␣transfer");
```

fullduplex.cpp 1 MB

Listing B.5: changed file - mgbench/src/L1/halfduplex.cpp

```
37 DEFINE_uint64(size, 1*1024*1024, "The␣amount␣of␣data␣to␣transfer");
38 DEFINE_uint64(repetitions, 100, "Number␣of␣repetitions␣to␣average");
39
40 DEFINE_int32(from, -1, "Only␣copy␣from␣a␣single␣GPU␣index,␣or␣-1␣for␣all");
41 DEFINE_int32(to, -1, "Only␣copy␣to␣a␣single␣GPU␣index,␣or␣-1␣for␣all");
```

uva.cu 100 KB

Listing B.6: changed file - mgbench/src/L1/uva.cu

```
39 DEFINE_uint64(size, 0.1*1024*1024, "The␣amount␣of␣data␣to␣transfer");
40 DEFINE_uint64(type_size, sizeof(float), "The␣size␣of␣the␣data␣chunk␣to␣"
41              "transfer,␣e.g.␣4␣for␣a␣4-byte␣float");
42 DEFINE_uint64(repetitions, 100, "Number␣of␣repetitions␣to␣average");
43 DEFINE_uint64(block_size, 32, "Copy␣kernel␣block␣size");
44 DEFINE_bool(fullduplex, false, "True␣for␣bi-directional␣copy");
45 DEFINE_bool(write, false, "Perform␣DMA␣write␣instead␣of␣read");
46 DEFINE_bool(random, false, "Use␣random␣access␣instead␣of␣coalesced");
```

B.2. Changes to Tensorflow

B.2.1. Changes in cuda_gpu_executor.cc

The following has been changed in the file `/tensorflow/tensorflow/stream_executor/cuda/cuda_gpu_executor.cc` from branch r1.8 [60]

Listing B.7: original file - branch: r1.8 - cuda_gpu_executor.cc

```
732 bool CUDAExecutor::CanEnablePeerAccessTo(StreamExecutorInterface *other) {
733   CUDAExecutor *cuda_other = static_cast<CUDAExecutor *>(other);
734   return CUDADriver::CanEnablePeerAccess(context_, cuda_other->context_);
735 }
```

Listing B.8: changed file: - branch: r1.8 - cuda_gpu_executor.cc

```
732 bool CUDAExecutor::CanEnablePeerAccessTo(StreamExecutorInterface *other) {
733   CUDAExecutor *cuda_other = static_cast<CUDAExecutor *>(other);
734   return false;
735 }
```

B.2.2. Compiling changed Tensorflow

The compilation of Tensorflow r1.8 with the

Listing B.9: Step-by-step: Compile TensorFlow on DGX-1 P100

```
# delete existing tf version
sudo rm -r /opt/tensorflow

# delete existing bazel version
sudo rm -r /usr/local/lib/bazel
sudo rm -r /usr/local/bin/bazel

# update apt-get
sudo apt-get update

# install bazel version 0.15.0 with
./bazel-0.15.0-installer-linux-x86_64.sh --user
export PATH="$PATH:$HOME/bin"
TEST_TMPDIR=/tmp/bazel/ bazel version
export LC_ALL=C
```

```
# update / install several packages that are needed
sudo apt install python-pip
sudo apt install python-numpy python-scipy python-wheel python-mock python-six
sudo pip install --upgrade setuptools
sudo pip install keras
sudo pip install keras-preprocessing

# in order to use the preinstalled NCCL version (2.2.13), some files need to be
     reordered, because they are by default split into different locations
cd /usr/lib
sudo mkdir nccl2
cd nccl2
sudo mkdir lib
sudo mkdir include
sudo ln -s /usr/lib/x86_64-linux-gnu/libnccl.so /usr/lib/nccl2/lib/libnccl.so
sudo ln -s /usr/include/nccl.h /usr/lib/nccl2/include/nccl.h
sudo ln -s /usr/lib/x86_64-linux-gnu/libnccl.so.2.2.13 /usr/lib/nccl2/lib/
    libnccl.so.2.2.13
sudo ln -s /usr/lib/x86_64-linux-gnu/libnccl.so.2 /usr/lib/nccl2/lib/libnccl.so
    .2
sudo ln -s /usr/ /usr/lib/nccl2/NCCL-SLA.txt
sudo chmod -R 777 nccl2
export TF_NCCL_VERSION='2.2.13'
export NCCL_INSTALL_PATH=/usr/lib/nccl2

# inside the tensorflow directory, configure the build
cd tensorflow
TEST_TMPDIR=/tmp/bazel/ ./configure

# make tensorflow package builder
TEST_TMPDIR=/tmp/bazel/ bazel build --config=opt --config=cuda //tensorflow/
    tools/pip_package:build_pip_package

# build the package
./bazel-bin/tensorflow/tools/pip_package/build_pip_package /tmp/tensorflow_pkg

# install the changed tensorflow via pip (the package can be found in /tmp/
    tensorflow_pkg)
sudo pip install tensorflow-1.8.0-cp27-cp27mu-linux_x86_64.whl
```

B.2.3. Installing changed Tensorflow

Listing B.10: Step-by-step: Install changed TensorFlow

```
# delete existing TensorFlow version
sudo rm -r /opt/tensorflow

# update / install several packages that are needed
sudo apt-get update
sudo apt install python-pip
sudo apt install python-numpy python-scipy python-wheel python-mock python-six
sudo pip install --upgrade setuptools
sudo pip install keras
sudo pip install keras-preprocessing

# install the changed TensorFlow version
sudo pip install tensorflow-1.8.0-cp27-cp27mu-linux_x86_64.whl
```

C. Experiment Results

C.1. Microbenchmarks Results

C.1.1. nvidia-smi Topology Query Results

	GPU0	GPU1	GPU2	GPU3	GPU4	GPU5	GPU6	GPU7	mlx5_0	mlx5_2	mlx5_1	mlx5_3	CPU Affinity
GPU0	X	NV1	NV1	NV1	NV1	SYS	SYS	SYS	PIX	SYS	PHB	SYS	0-19,40-59
GPU1	NV1	X	NV1	NV1	SYS	NV1	SYS	SYS	PIX	SYS	PHB	SYS	0-19,40-59
GPU2	NV1	NV1	X	NV1	SYS	SYS	NV1	SYS	PHB	SYS	PIX	SYS	0-19,40-59
GPU3	NV1	NV1	NV1	X	SYS	SYS	SYS	NV1	PHB	SYS	PIX	SYS	0-19,40-59
GPU4	NV1	SYS	SYS	SYS	X	NV1	NV1	NV1	SYS	PIX	SYS	PHB	20-39,60-79
GPU5	SYS	NV1	SYS	SYS	NV1	X	NV1	NV1	SYS	PIX	SYS	PHB	20-39,60-79
GPU6	SYS	SYS	NV1	SYS	NV1	NV1	X	NV1	SYS	PHB	SYS	PIX	20-39,60-79
GPU7	SYS	SYS	SYS	NV1	NV1	NV1	NV1	X	SYS	PHB	SYS	PIX	20-39,60-79
mlx5_0	PIX	PIX	PHB	PHB	SYS	SYS	SYS	SYS	X	SYS	PHB	SYS	
mlx5_2	SYS	SYS	SYS	SYS	PIX	PIX	PHB	PHB	SYS	X	SYS	PHB	
mlx5_1	PHB	PHB	PIX	PIX	SYS	SYS	SYS	SYS	PHB	SYS	X	SYS	
mlx5_3	SYS	SYS	SYS	SYS	PHB	PHB	PIX	PIX	SYS	PHB	SYS	X	

Table C.1.: nvidia-smi topo DGX-1 P100; nvidia-smi topo -m

	GPU0	GPU1	GPU2	GPU3	GPU4	GPU5	GPU6	GPU7	mlx5_0	mlx5_2	mlx5_1	mlx5_3	CPU Affinity
GPU0	X	PIX	PHB	PHB	SYS	SYS	SYS	SYS	PIX	SYS	PHB	SYS	0-19,40-59
GPU1	PIX	X	PHB	PHB	SYS	SYS	SYS	SYS	PIX	SYS	PHB	SYS	0-19,40-59
GPU2	PHB	PHB	X	PIX	SYS	SYS	SYS	SYS	PHB	SYS	PIX	SYS	0-19,40-59
GPU3	PHB	PHB	PIX	X	SYS	SYS	SYS	SYS	PHB	SYS	PIX	SYS	0-19,40-59
GPU4	SYS	SYS	SYS	SYS	X	PIX	PHB	PHB	SYS	PIX	SYS	PHB	20-39,60-79
GPU5	SYS	SYS	SYS	SYS	PIX	X	PHB	PHB	SYS	PIX	SYS	PHB	20-39,60-79
GPU6	SYS	SYS	SYS	SYS	PHB	PHB	X	PIX	SYS	PHB	SYS	PIX	20-39,60-79
GPU7	SYS	SYS	SYS	SYS	PHB	PHB	PIX	X	SYS	PHB	SYS	PIX	20-39,60-79
mlx5_0	PIX	PIX	PHB	PHB	SYS	SYS	SYS	SYS	X	SYS	PHB	SYS	
mlx5_2	SYS	SYS	SYS	SYS	PIX	PIX	PHB	PHB	SYS	X	SYS	PHB	
mlx5_1	PHB	PHB	PIX	PIX	SYS	SYS	SYS	SYS	PHB	SYS	X	SYS	
mlx5_3	SYS	SYS	SYS	SYS	PHB	PHB	PIX	PIX	SYS	PHB	SYS	X	

Table C.2.: nvidia-smi topo DGX-1 P100; nvidia-smi topo -mp

	GPU0	GPU1	GPU2	GPU3	GPU4	GPU5	GPU6	GPU7	mlx5_0	mlx5_2	mlx5_1	mlx5_3	CPU Affinity
GPU0	X	NV1	NV1	NV2	NV2	SYS	SYS	SYS	PIX	SYS	PHB	SYS	0-19,40-59
GPU1	NV1	X	NV2	NV1	SYS	NV2	SYS	SYS	PIX	SYS	PHB	SYS	0-19,40-59
GPU2	NV1	NV2	X	NV2	SYS	SYS	NV1	SYS	PHB	SYS	PIX	SYS	0-19,40-59
GPU3	NV2	NV1	NV2	X	SYS	SYS	SYS	NV1	PHB	SYS	PIX	SYS	0-19,40-59
GPU4	NV2	SYS	SYS	SYS	X	NV1	NV1	NV2	SYS	PIX	SYS	PHB	20-39,60-79
GPU5	SYS	NV2	SYS	SYS	NV1	X	NV2	NV1	SYS	PIX	SYS	PHB	20-39,60-79
GPU6	SYS	SYS	NV1	SYS	NV1	NV2	X	NV2	SYS	PHB	SYS	PIX	20-39,60-79
GPU7	SYS	SYS	SYS	NV1	NV2	NV1	NV2	X	SYS	PHB	SYS	PIX	20-39,60-79
mlx5_0	PIX	PIX	PHB	PHB	SYS	SYS	SYS	SYS	X	SYS	PHB	SYS	
mlx5_2	SYS	SYS	SYS	SYS	PIX	PIX	PHB	PHB	SYS	X	SYS	PHB	
mlx5_1	PHB	PHB	PIX	PIX	SYS	SYS	SYS	SYS	PHB	SYS	X	SYS	
mlx5_3	SYS	SYS	SYS	SYS	PHB	PHB	PIX	PIX	SYS	PHB	SYS	X	

Table C.3.: nvidia-smi topo DGX-1 V100; nvidia-smi topo -m

	GPU0	GPU1	GPU2	GPU3	GPU4	GPU5	GPU6	GPU7	mlx5_0	mlx5_2	mlx5_1	mlx5_3	CPU Affinity
GPU0	X	PIX	PHB	PHB	SYS	SYS	SYS	SYS	PIX	SYS	PHB	SYS	0-19,40-59
GPU1	PIX	X	PHB	PHB	SYS	SYS	SYS	SYS	PIX	SYS	PHB	SYS	0-19,40-59
GPU2	PHB	PHB	X	PIX	SYS	SYS	SYS	SYS	PHB	SYS	PIX	SYS	0-19,40-59
GPU3	PHB	PHB	PIX	X	SYS	SYS	SYS	SYS	PHB	SYS	PIX	SYS	0-19,40-59
GPU4	SYS	SYS	SYS	SYS	X	PIX	PHB	PHB	SYS	PIX	SYS	PHB	20-39,60-79
GPU5	SYS	SYS	SYS	SYS	PIX	X	PHB	PHB	SYS	PIX	SYS	PHB	20-39,60-79
GPU6	SYS	SYS	SYS	SYS	PHB	PHB	X	PIX	SYS	PHB	SYS	PIX	20-39,60-79
GPU7	SYS	SYS	SYS	SYS	PHB	PHB	PIX	X	SYS	PHB	SYS	PIX	20-39,60-79
mlx5_0	PIX	PIX	PHB	PHB	SYS	SYS	SYS	SYS	X	SYS	PHB	SYS	
mlx5_2	SYS	SYS	SYS	SYS	PIX	PIX	PHB	PHB	SYS	X	SYS	PHB	
mlx5_1	PHB	PHB	PIX	PIX	SYS	SYS	SYS	SYS	PHB	SYS	X	SYS	
mlx5_3	SYS	SYS	SYS	SYS	PHB	PHB	PIX	PIX	SYS	PHB	SYS	X	

Table C.4.: nvidia-smi topo DGX-1 V100; nvidia-smi topo -mp

	GPU0	GPU1	GPU2	GPU3	mlx5_1	mlx5_0	CPU Affinity
GPU0	X	NV3	SYS	SYS	SYS	NODE	0-0,4-4,8-8,12-12,16-16,20-20,24-24,28-28, 32-32,36-36,40-40,44-44,48-48,52-52,56-56, 60-60,64-64,68-68,72-72,76-76
GPU1	NV3	X	SYS	SYS	SYS	NODE	0-0,4-4,8-8,12-12,16-16,20-20,24-24,28-28, 32-32,36-36, 40-40,44-44,48-48,52-52,56-56, 60-60,64-64,68-68,72-72,76-76
GPU2	SYS	SYS	X	NV3	NODE	SYS	80-80,84-84,88-88,92-92,96-96,100-100,104-104, 108-108,112-112,116-116,120-120,124-124, 128-128,132-132,136-136,140-140,144-144
GPU3	SYS	SYS	NV3	X	NODE	SYS	80-80,84-84,88-88,92-92,96-96,100-100,104-104, 108-108,112-112,116-116,120-120,124-124, 128-128,132-132,136-136,140-140,144-144
mlx5_1	SYS	SYS	NODE	NODE	X	SYS	
mlx5_0	NODE	NODE	SYS	SYS	SYS	X	

Table C.5.: nvidia-smi topo AC922; nvidia-smi topo -m

C.1.2. Mgbench Data Copies

DGX-1 P100

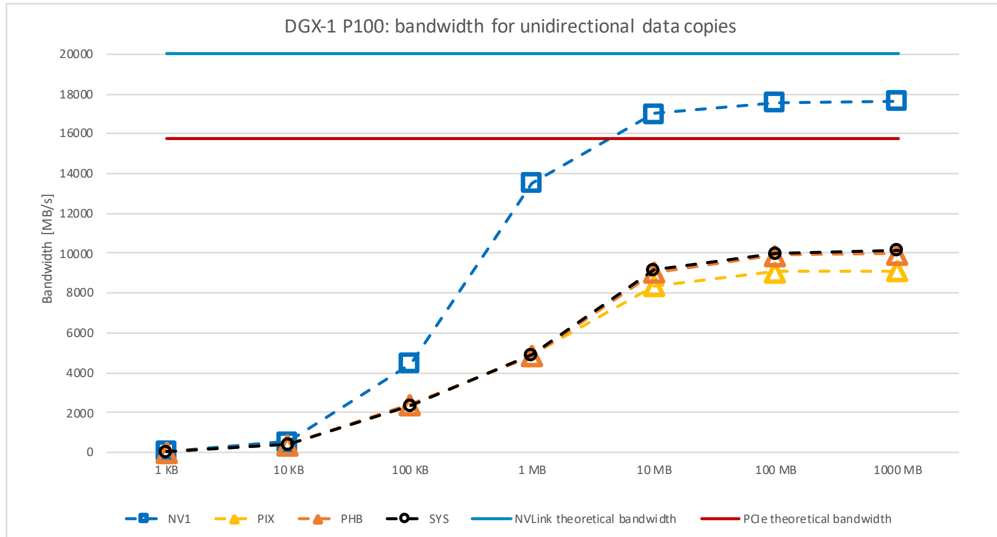


Figure C.1.: DGX-1 P100 bandwidth for unidirectional data copies

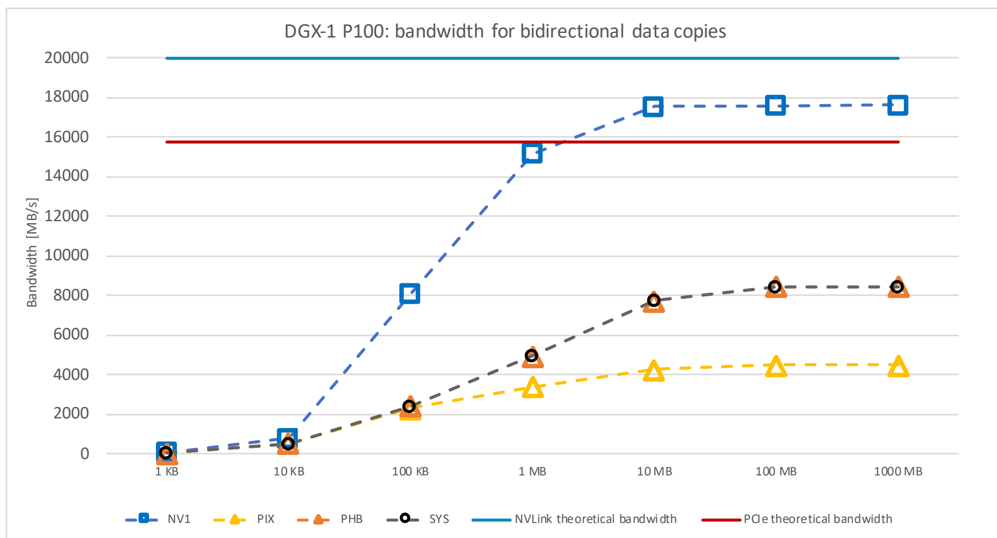


Figure C.2.: DGX-1 P100 bandwidth for bidirectional data copies

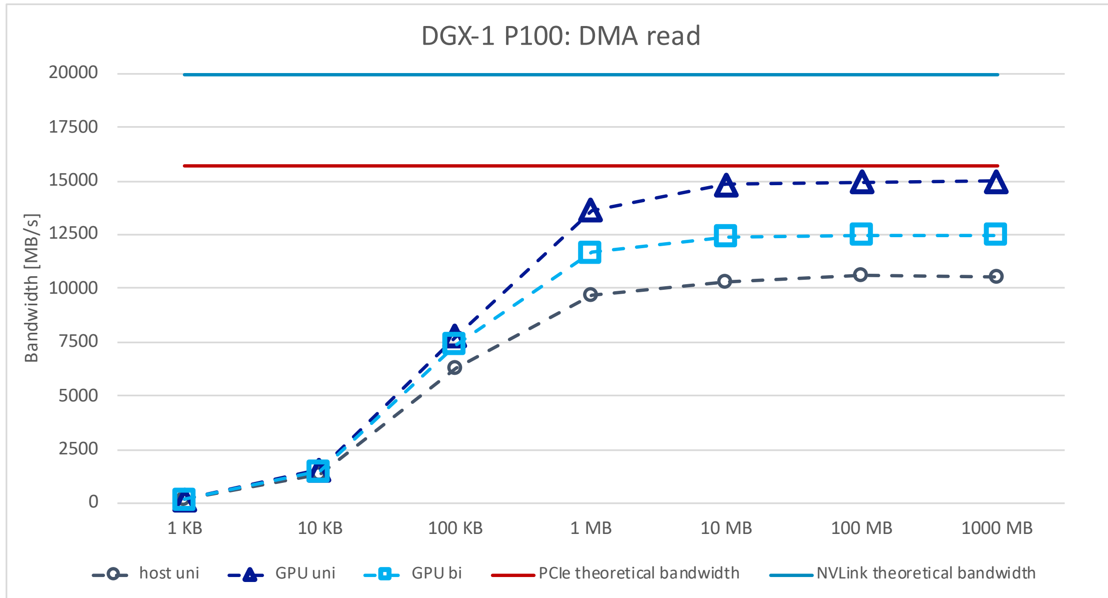


Figure C.3.: DGX-1 P100 bandwidth for DMA read

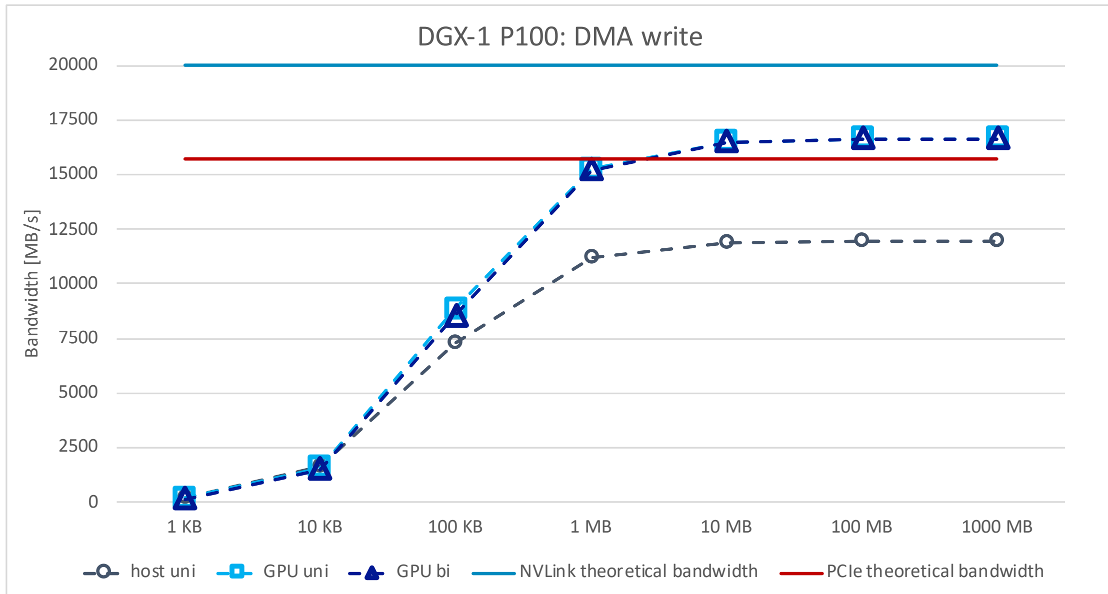


Figure C.4.: DGX-1 P100 bandwidth for DMA write

(a) data size: 1 KB

(b) data size: 10 KB

(c) data size: 100 KB

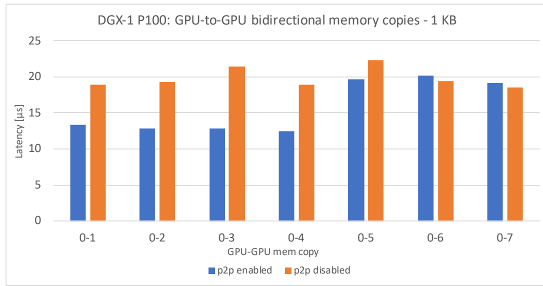
(d) data size: 1 MB

(e) data size: 10 MB

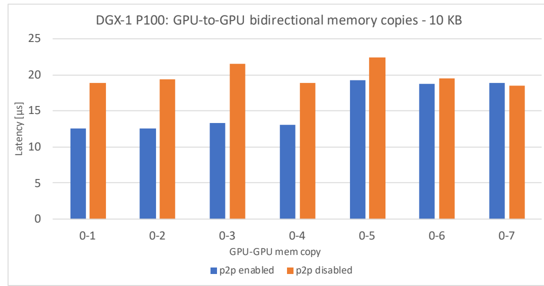
(f) data size: 100 MB

(g) data size: 1000 MB

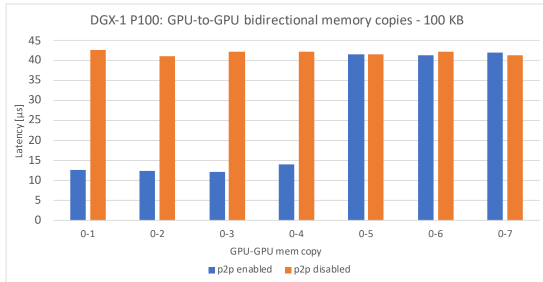
Figure C.5.: Latency for GPU-to-GPU unidirectional memory copies on DGX-1 P100



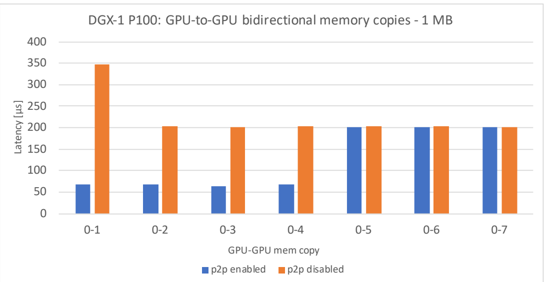
(a) data size: 1 KB



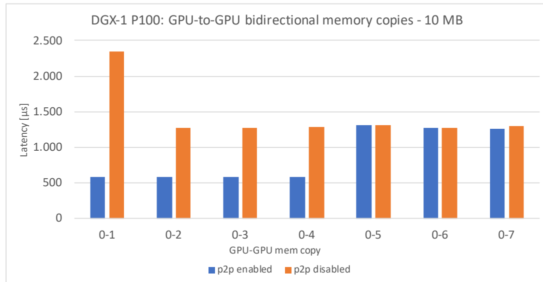
(b) data size: 10 KB



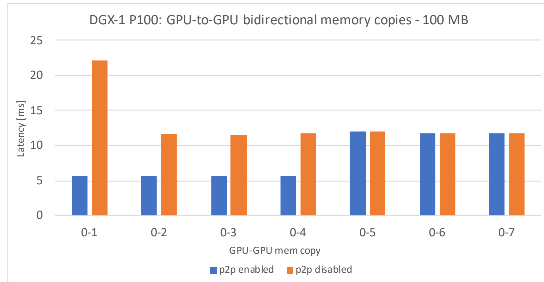
(c) data size: 100 KB



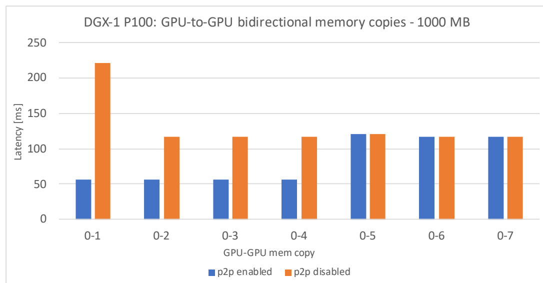
(d) data size: 1 MB



(e) data size: 10 MB

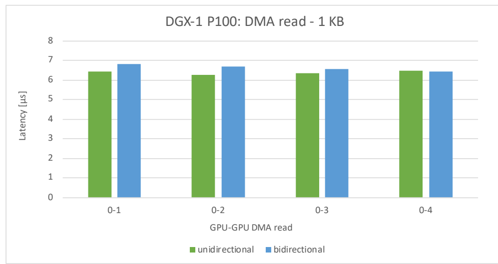


(f) data size: 100 MB

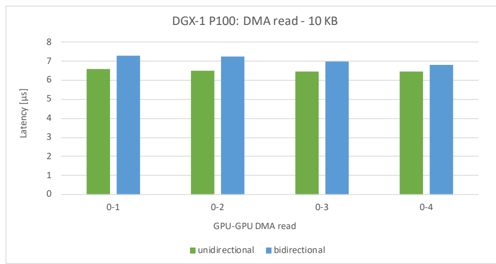


(g) data size: 1000 MB

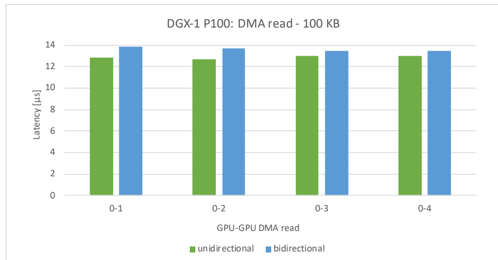
Figure C.6.: Latency for GPU-to-GPU bidirectional memory copies on DGX-1 P100



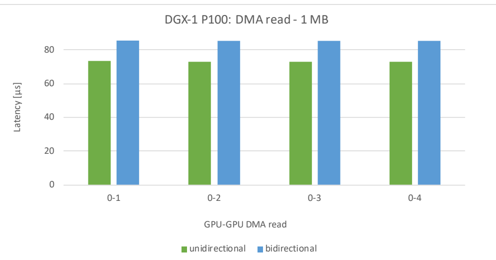
(a) data size: 1 KB



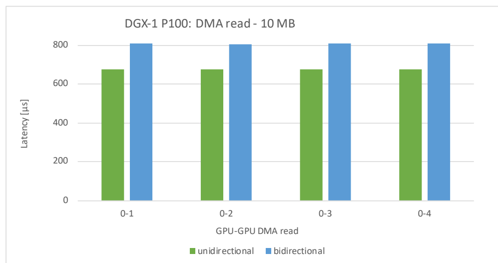
(b) data size: 10 KB



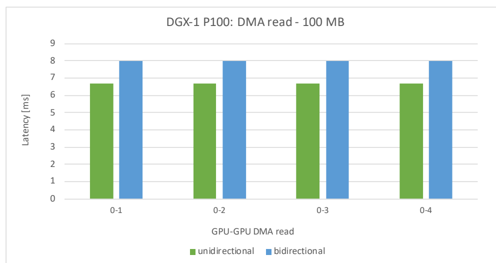
(c) data size: 100 KB



(d) data size: 1 MB



(e) data size: 10 MB



(f) data size: 100 MB

(g) data size: 1000 MB

Figure C.7.: Latency for GPU-to-GPU DMA read on DGX-1 P100

DGX-1 V100

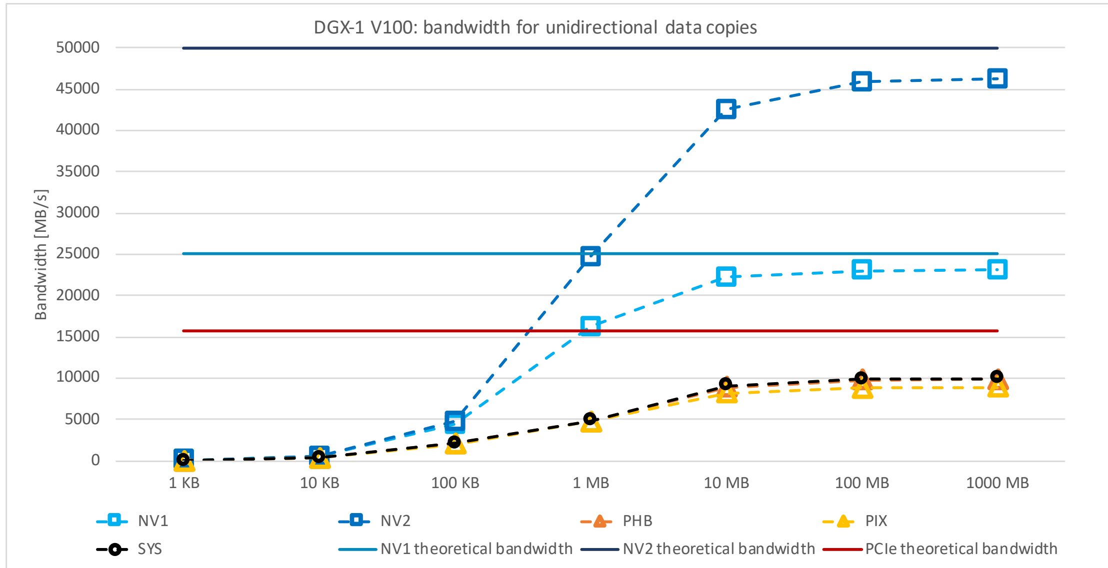


Figure C.8.: DGX-1 V100 bandwidth for unidirectional data copies

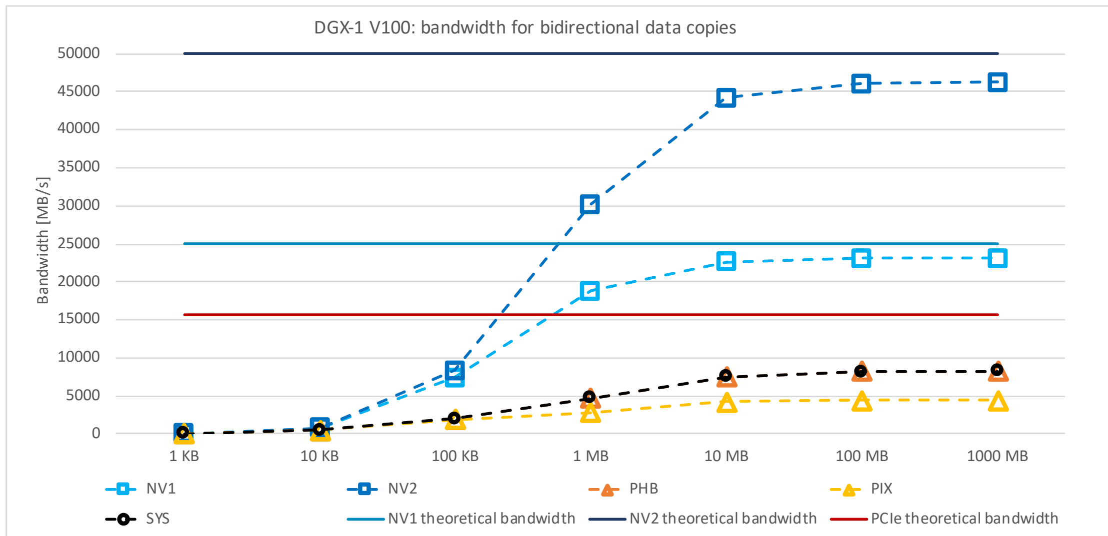


Figure C.9.: DGX-1 V100 bandwidth for bidirectional data copies

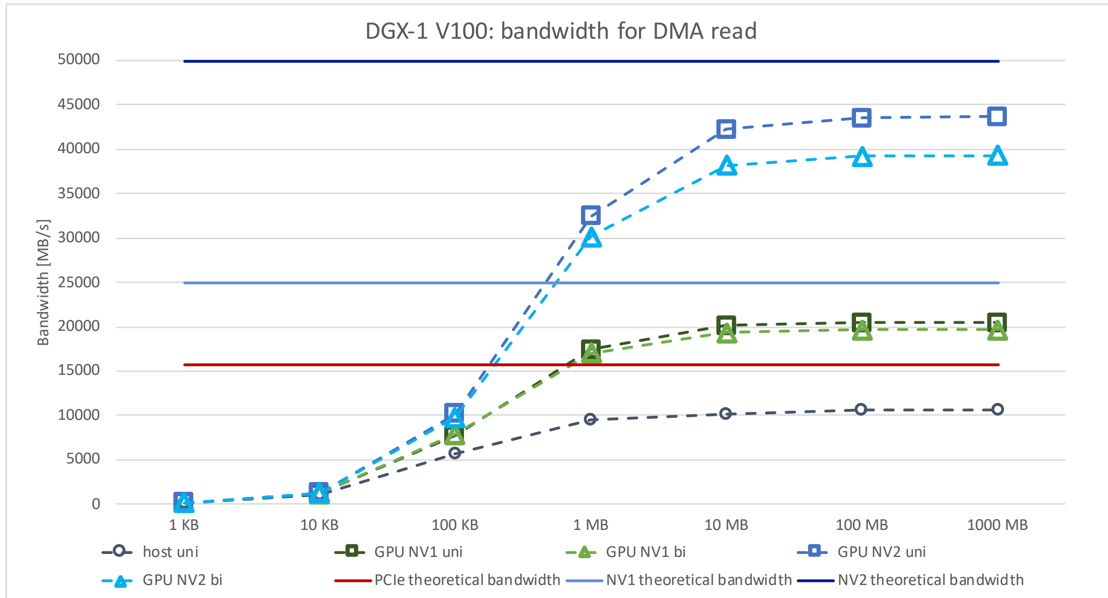


Figure C.10.: DGX-1 V100 bandwidth for DMA read

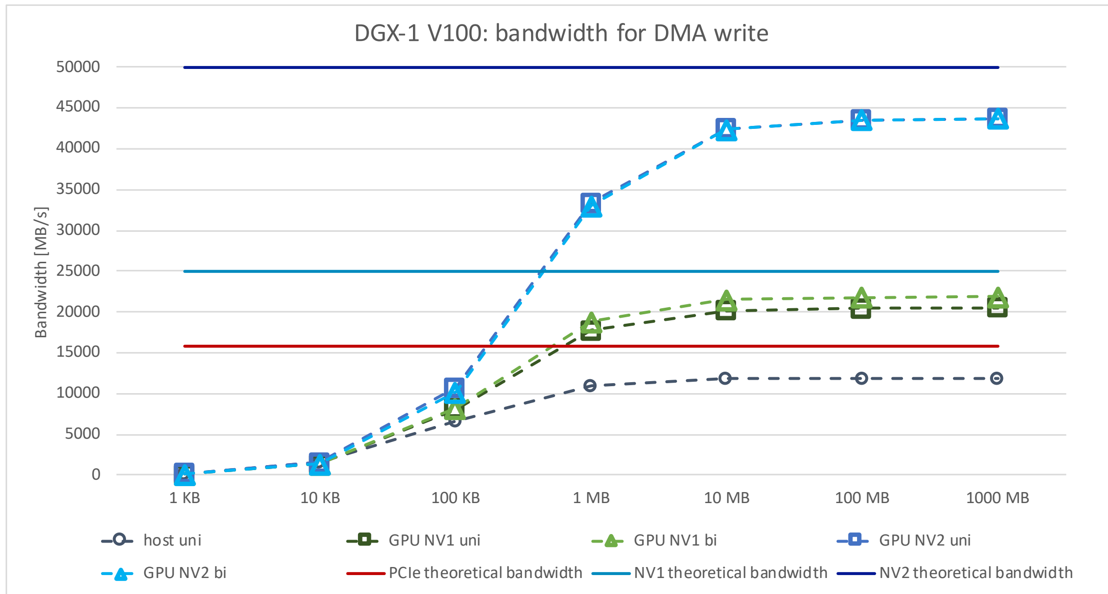
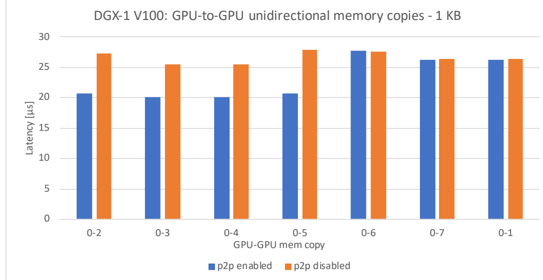
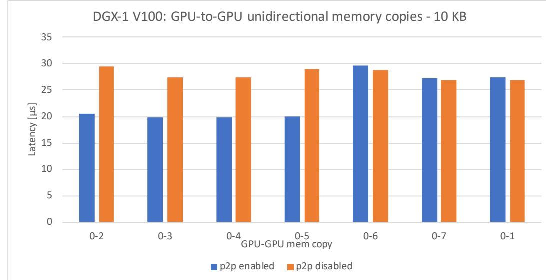


Figure C.11.: DGX-1 V100 bandwidth for DMA write

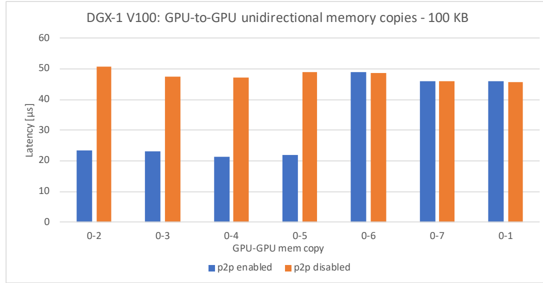
C.1.3. Latency



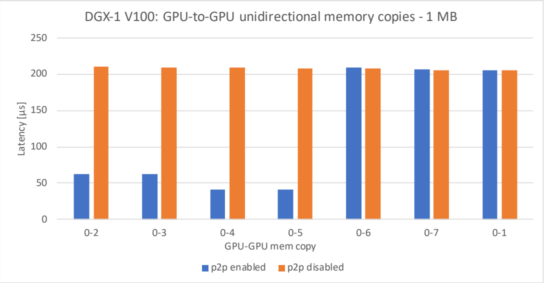
(a) data size: 1 KB



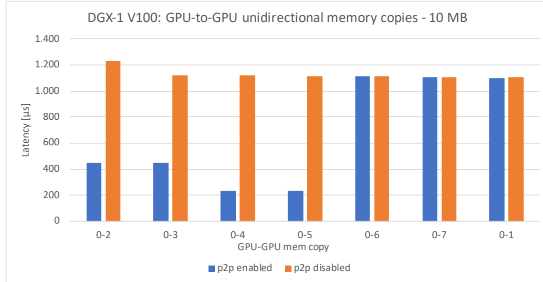
(b) data size: 10 KB



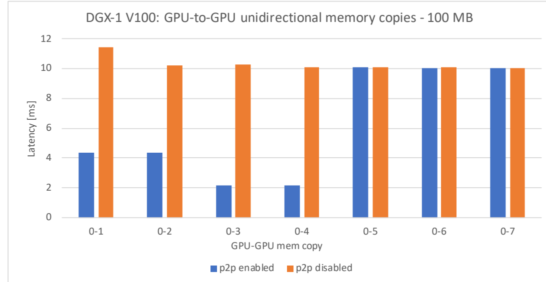
(c) data size: 100 KB



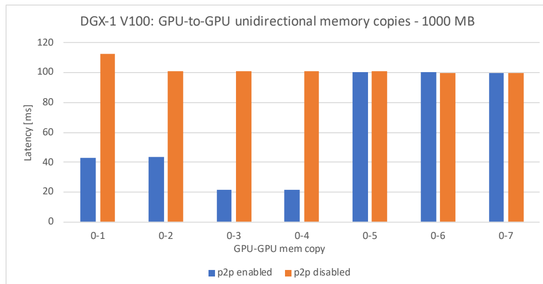
(d) data size: 1 MB



(e) data size: 10 MB

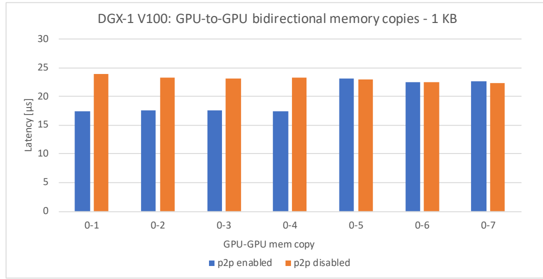


(f) data size: 100 MB

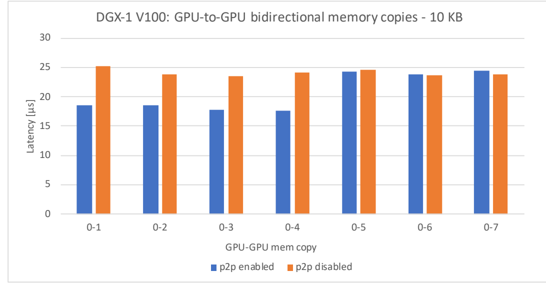


(g) data size: 1000 MB

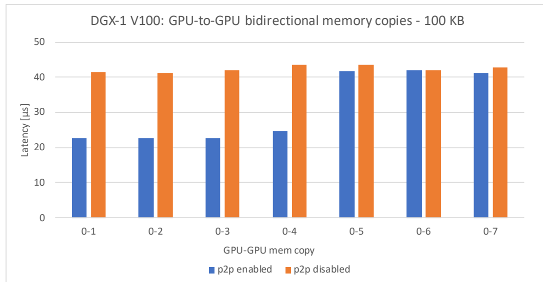
Figure C.12.: Latency for GPU-to-GPU unidirectional memory copies on DGX-1 V100



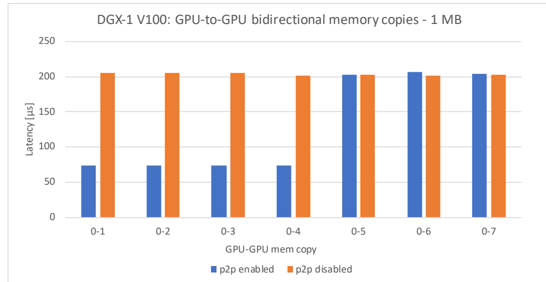
(a) data size: 1 KB



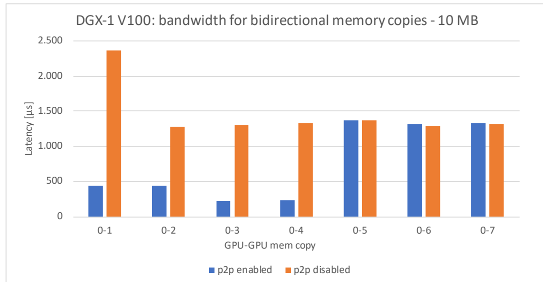
(b) data size: 10 KB



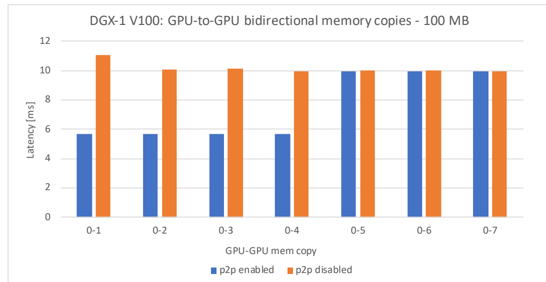
(c) data size: 100 KB



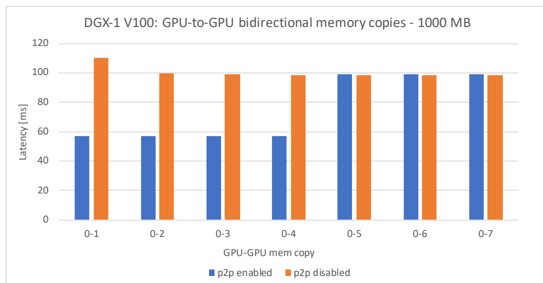
(d) data size: 1 MB



(e) data size: 10 MB

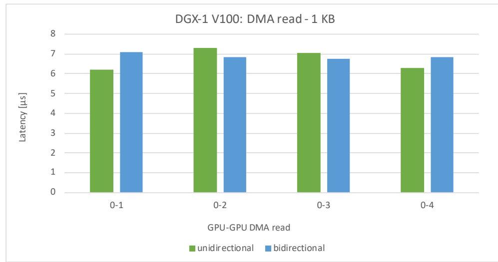


(f) data size: 100 MB

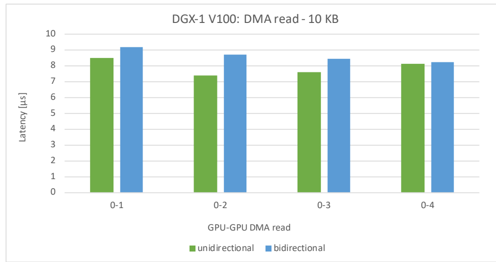


(g) data size: 1000 MB

Figure C.13.: Latency for GPU-to-GPU bidirectional memory copies on DGX-1 V100

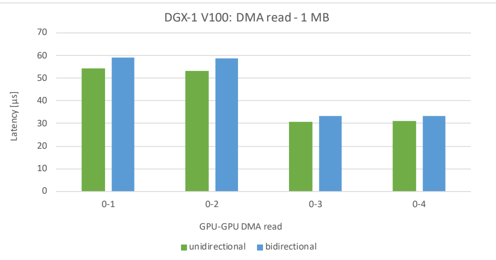


(a) data size: 1 KB

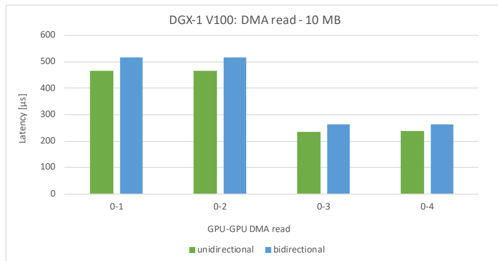


(b) data size: 10 KB

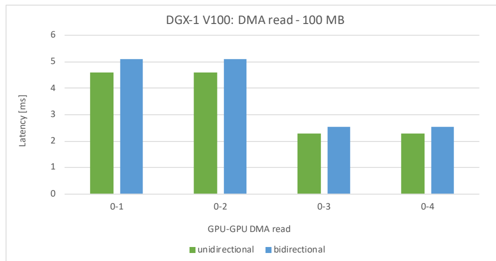
(c) data size: 100 KB



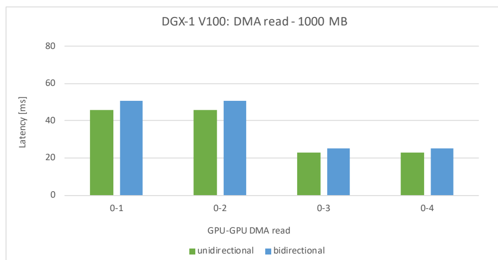
(d) data size: 1 MB



(e) data size: 10 MB



(f) data size: 100 MB



(g) data size: 1000 MB

Figure C.14.: Latency for GPU-to-GPU DMA read on DGX-1 V100

AC922

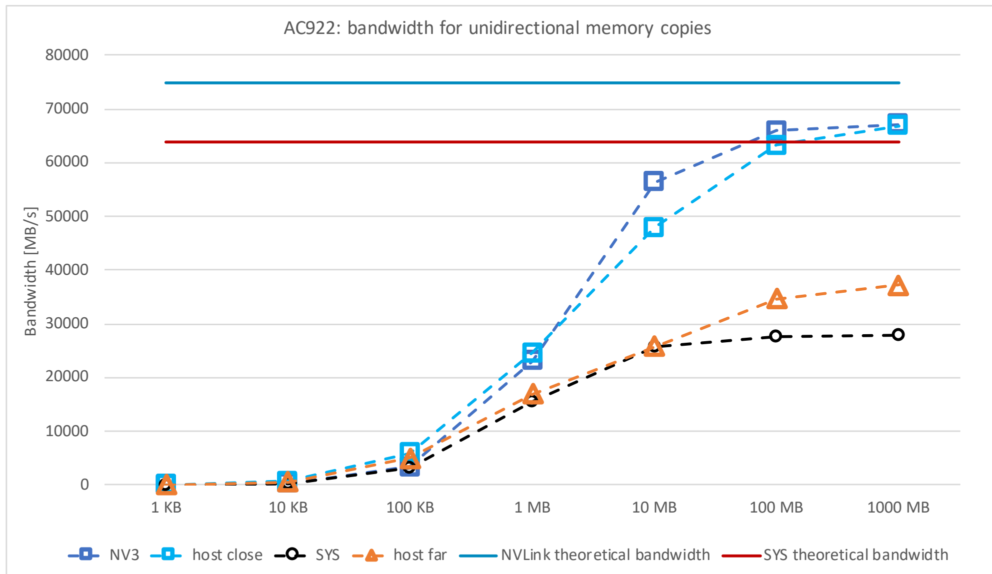


Figure C.15.: AC922 bandwidth for unidirectional data copies

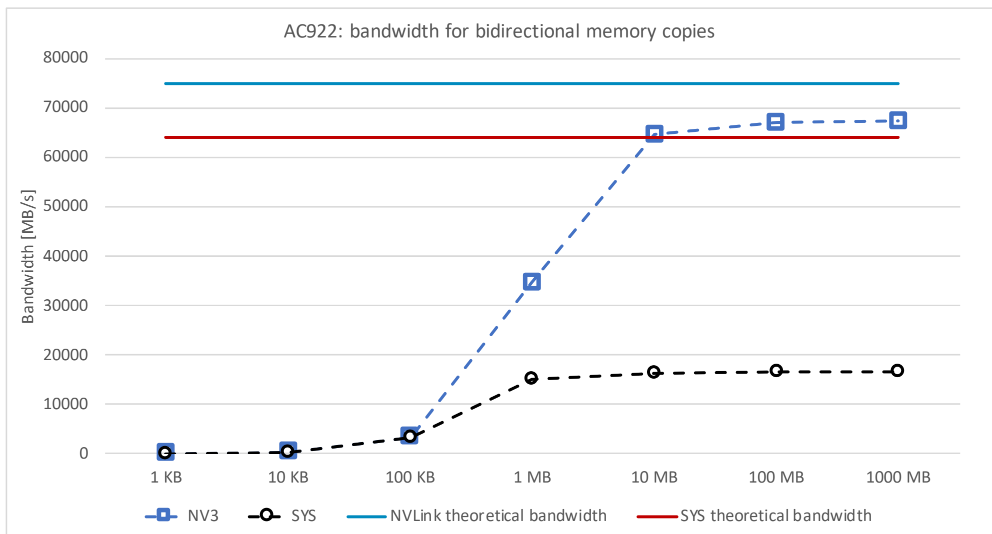


Figure C.16.: AC922 bandwidth for bidirectional data copies

C.2. DL Benchmarks Results

C.2.1. Scaling Training using PCIe and NVLink

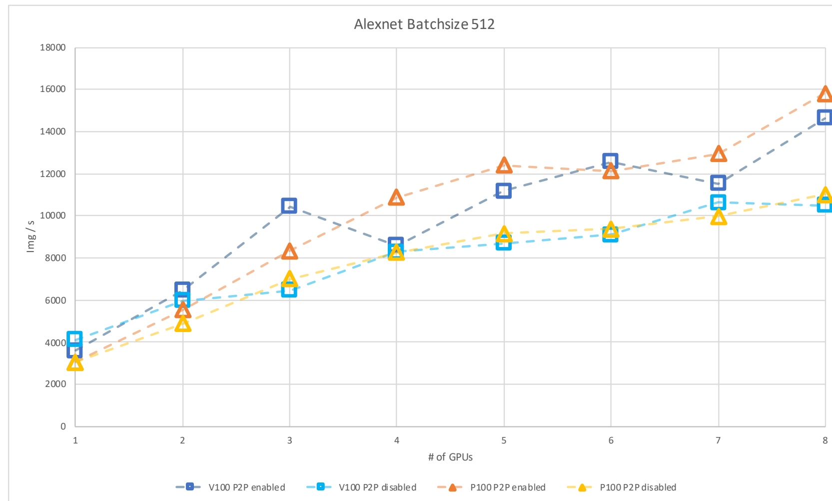


Figure C.17.: Scaling training of alexnet with (p2p enabled) and without NVLink (p2p disabled)

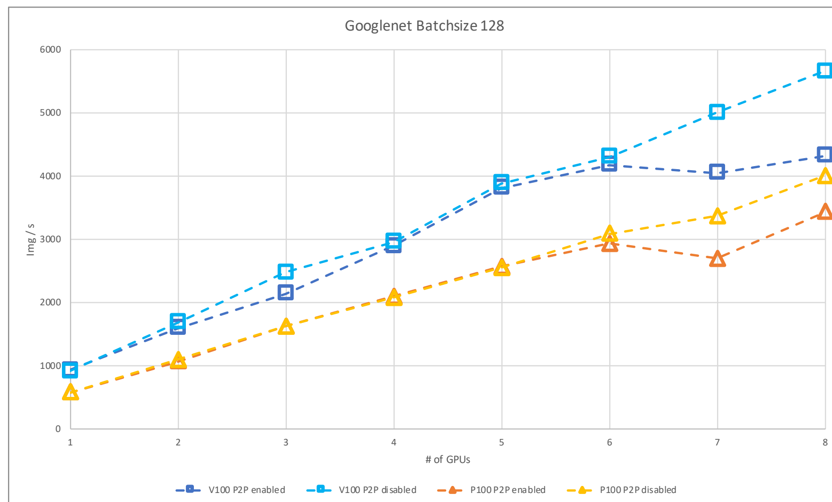


Figure C.18.: Scaling training of googlenet with (p2p enabled) and without NVLink (p2p disabled)

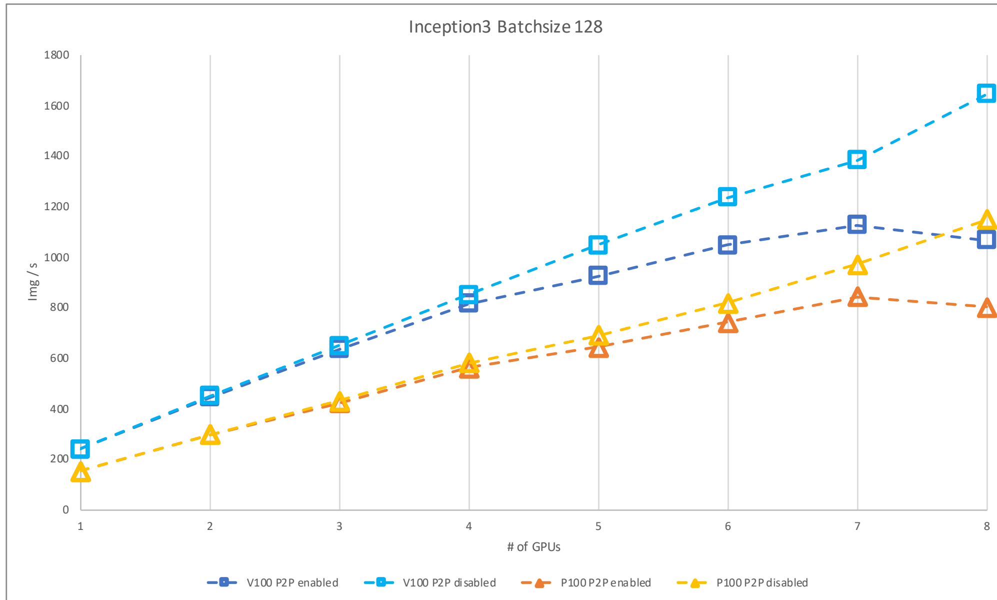


Figure C.19.: Scaling training of inception3 with (p2p enabled) and without NVLink (p2p disabled)

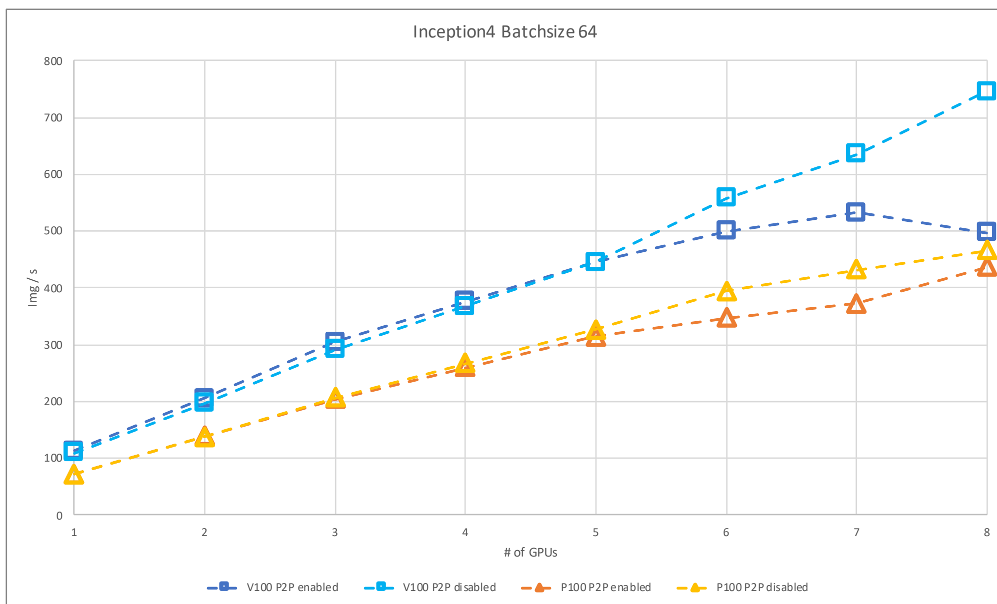


Figure C.20.: Scaling training of inception4 with (p2p enabled) and without NVLink (p2p disabled)

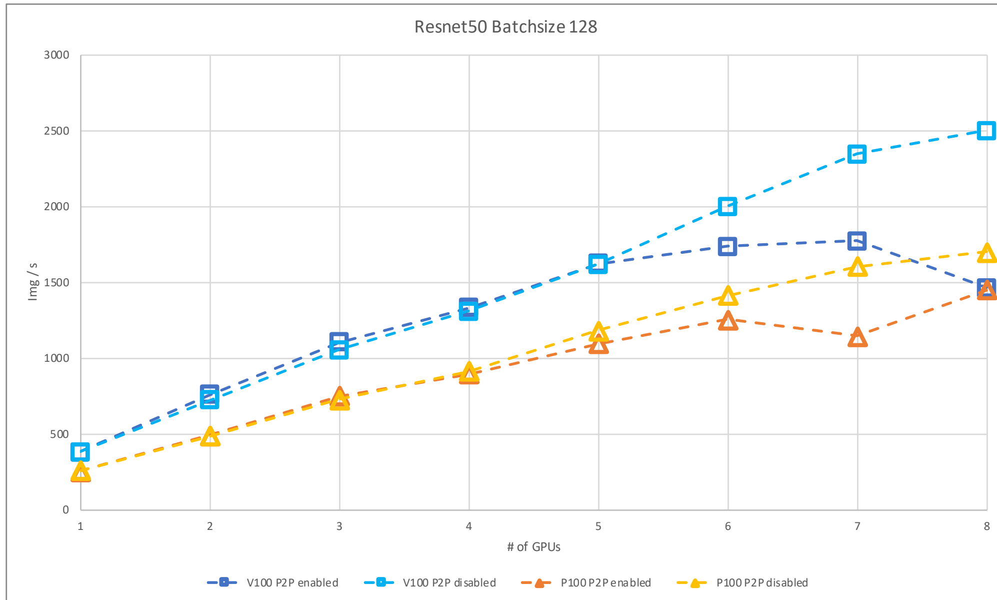


Figure C.21.: Scaling training of resnet50 with (p2p enabled) and without NVLink (p2p disabled)

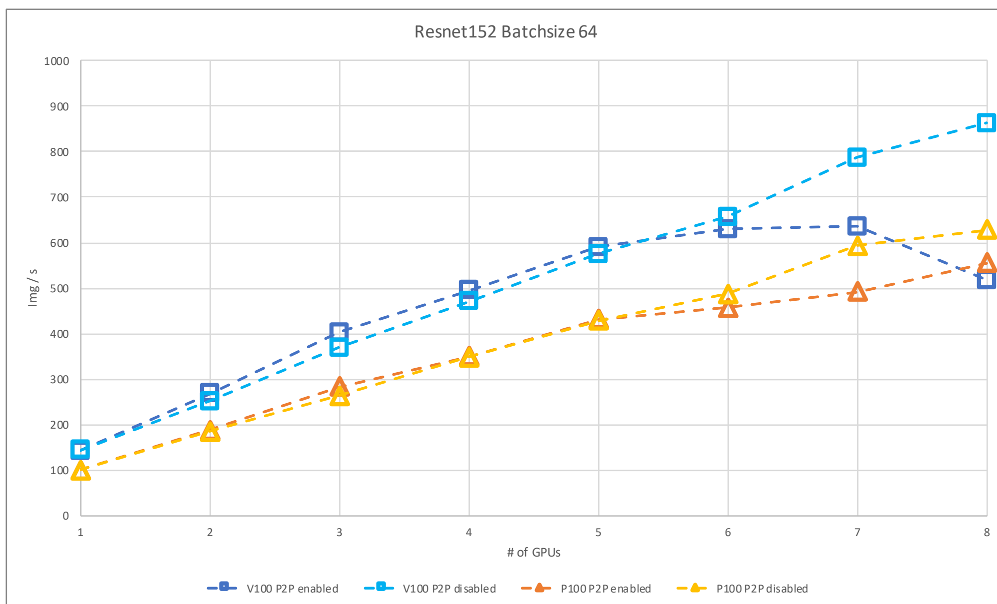


Figure C.22.: Scaling training of resnet152 with (p2p enabled) and without NVLink (p2p disabled)

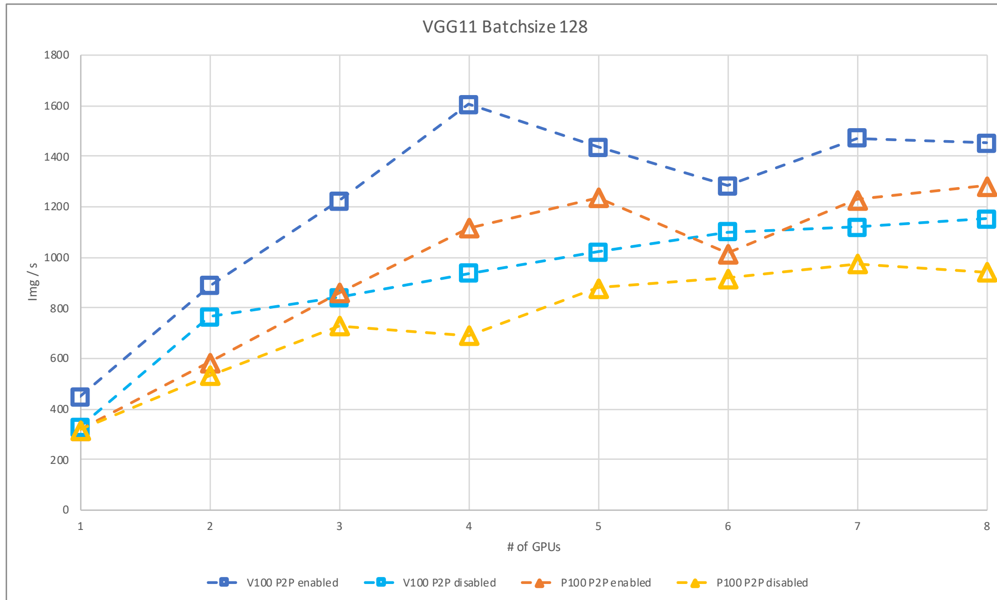


Figure C.23.: Scaling training of vgg11 with (p2p enabled) and without NVLink (p2p disabled)

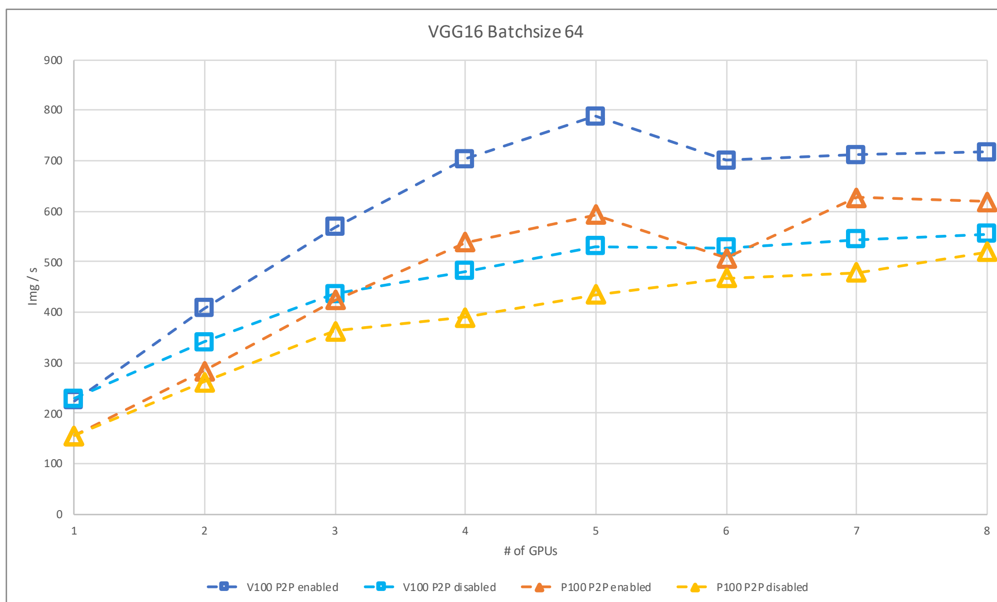


Figure C.24.: Scaling training of vgg16 with (p2p enabled) and without NVLink (p2p disabled)

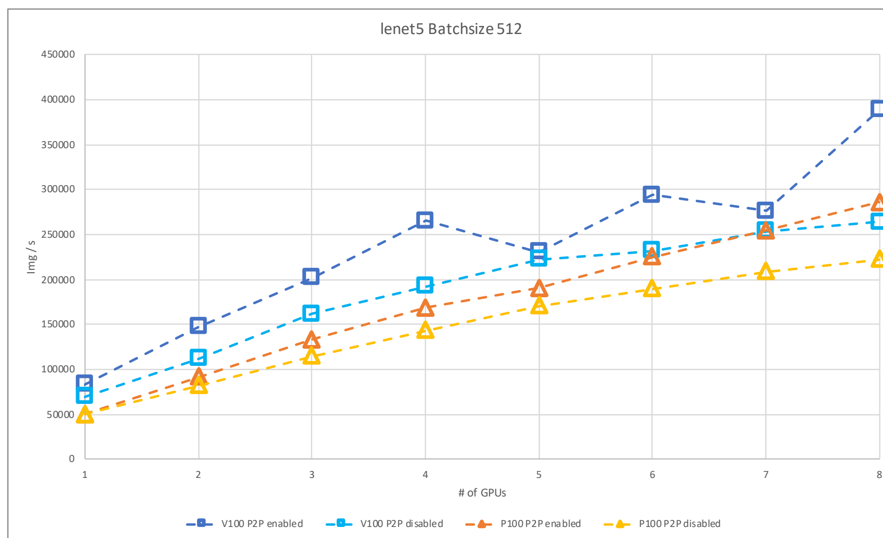


Figure C.25.: Scaling training of lenet5 with (p2p enabled) and without NVLink (p2p disabled)

C.2.2. Scaling Efficiency

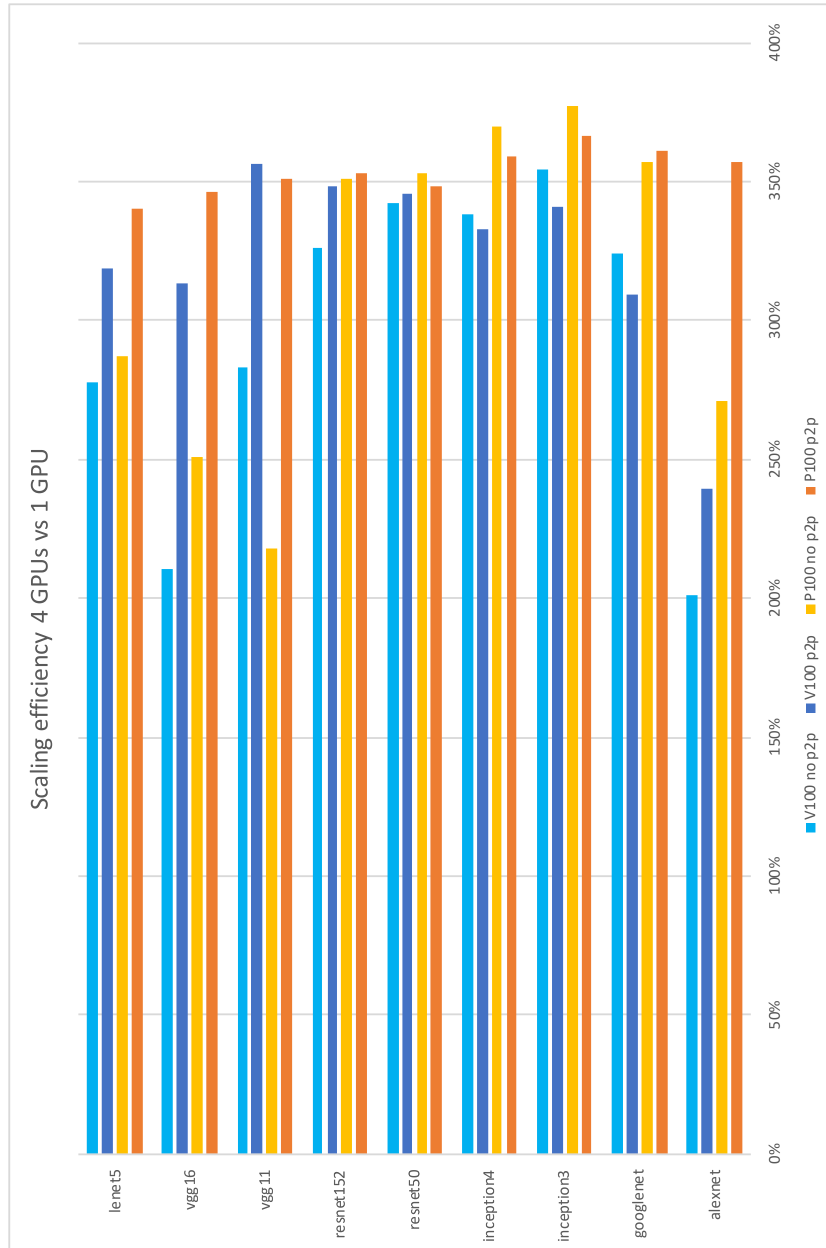


Figure C.26.: Scaling efficiency 4 GPUs versus 1 GPU with (p2p enabled) and without NVLink (p2p disabled)

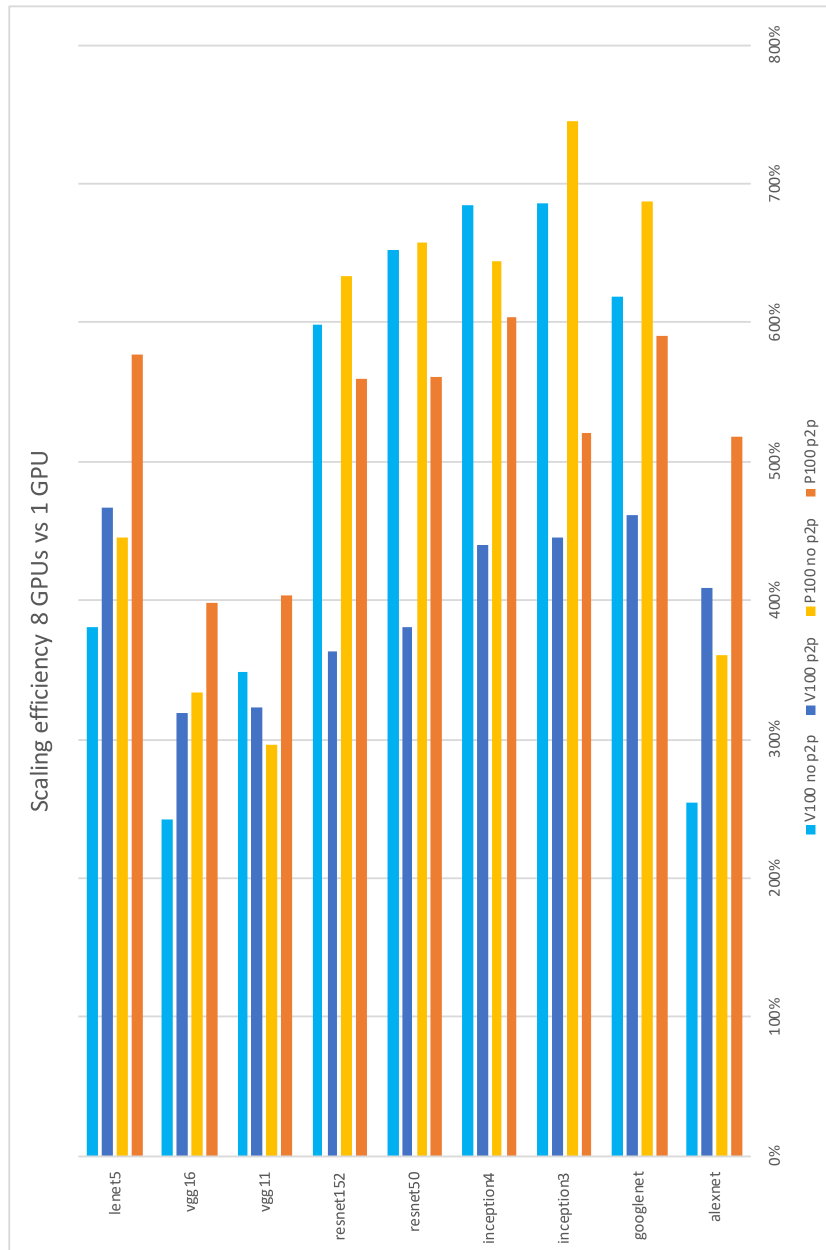


Figure C.27.: Scaling efficiency 8 GPUs versus 1 GPU with (p2p enabled) and without NVLink (p2p disabled)

(a) alexnet batch size 512

(b) googlenet batch size 128

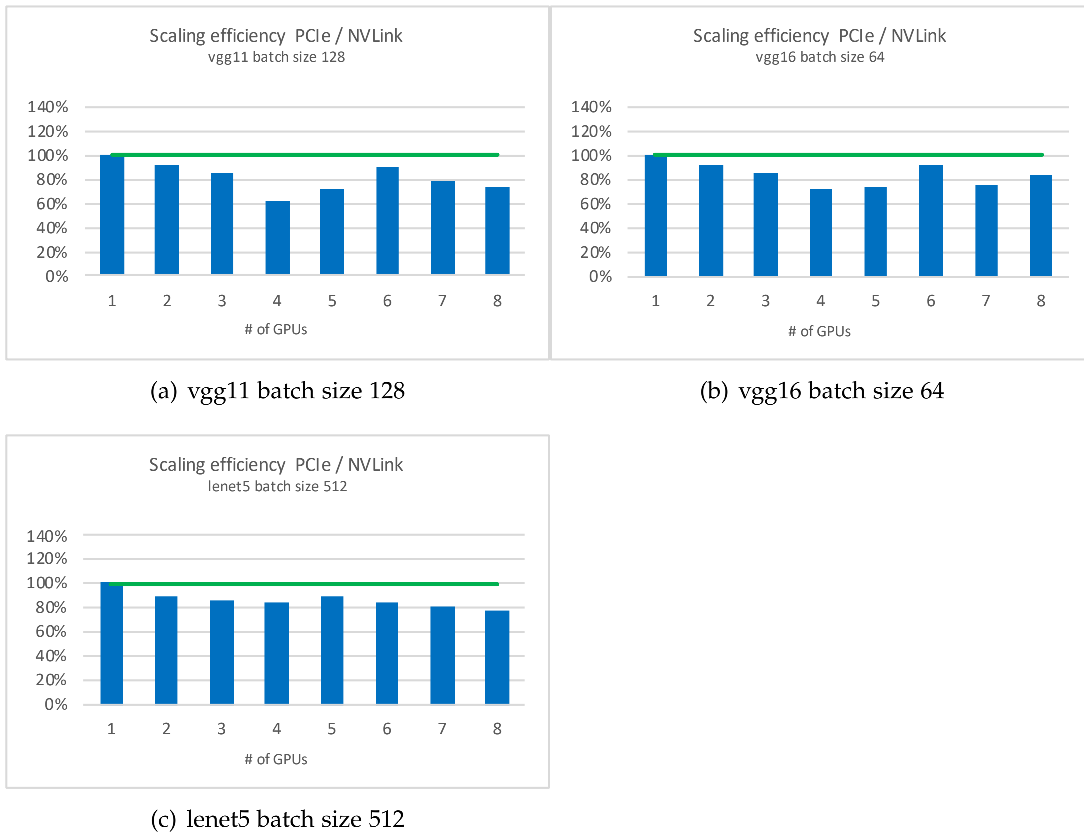
(c) inception3 batch size 128

(d) inception4 batch size 64

(e) resnet50 batch size 128

(f) resnet152 batch size 64

Figure C.28.: Scaling efficiency using only PCIe compared to using NVLink on DGX-1 P100

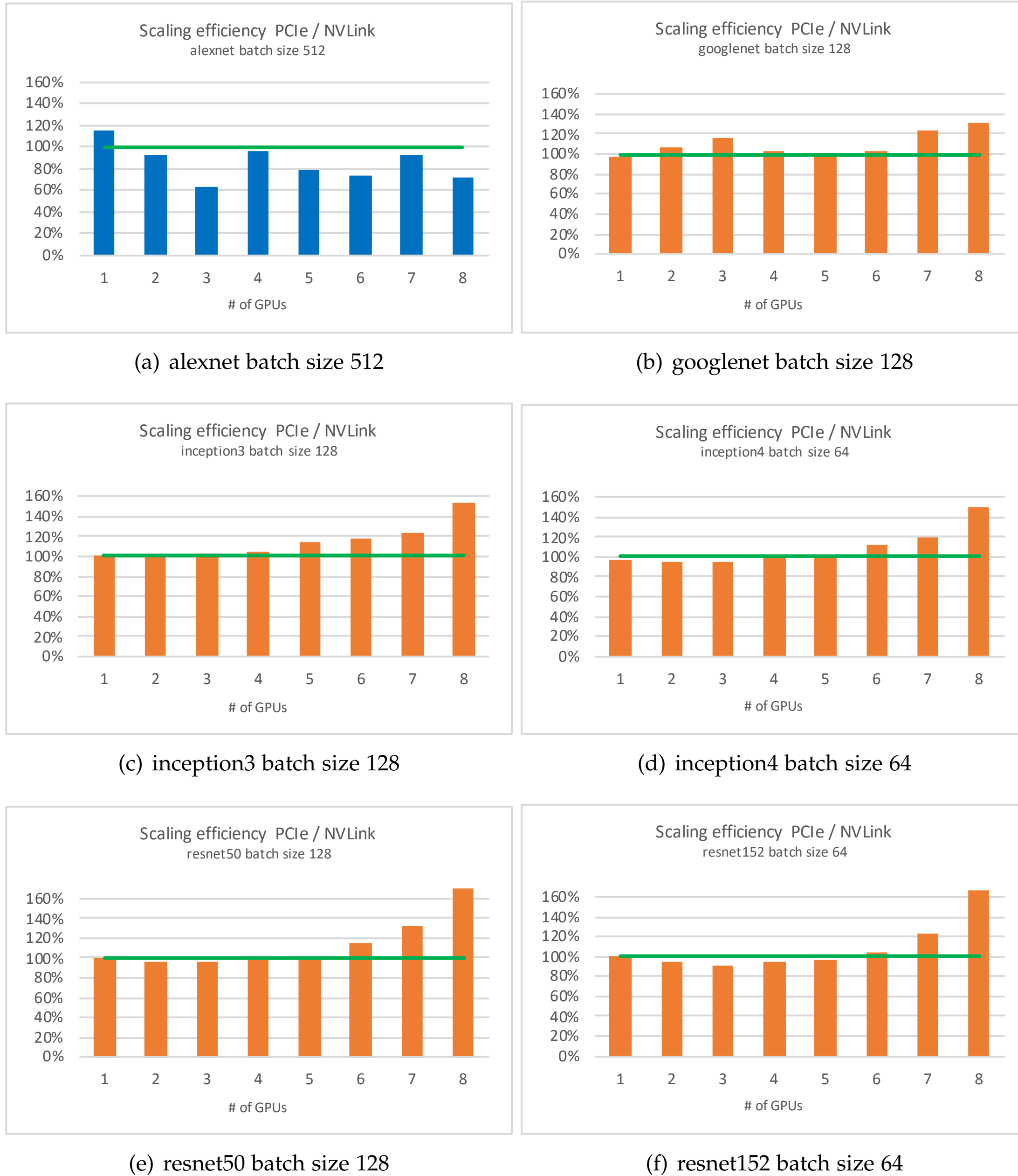


(a) vgg11 batch size 128

(b) vgg16 batch size 64

(c) lenet5 batch size 512

Figure C.29.: Scaling efficiency using only PCIe compared to using NVLink on DGX-1 P100



(a) alexnet batch size 512

(b) googlenet batch size 128

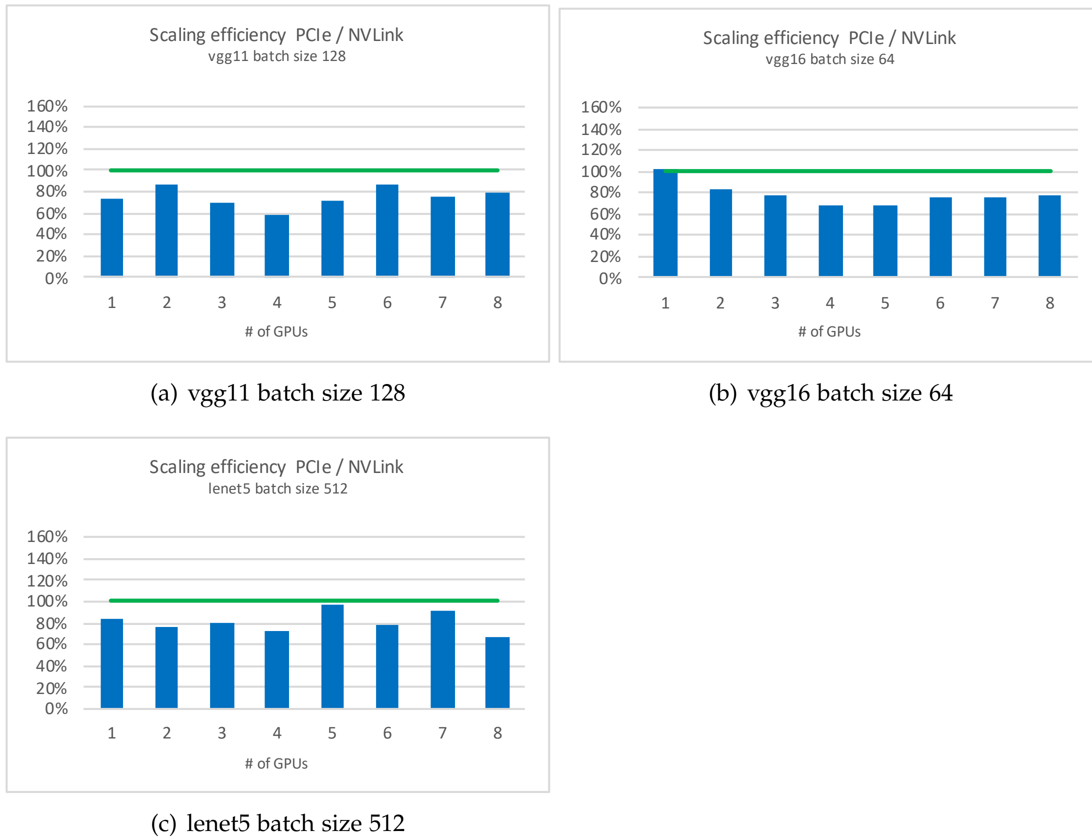
(c) inception3 batch size 128

(d) inception4 batch size 64

(e) resnet50 batch size 128

(f) resnet152 batch size 64

Figure C.30.: Scaling efficiency using only PCIe compared to using NVLink on DGX-1 V100



(a) vgg11 batch size 128

(b) vgg16 batch size 64

(c) lenet5 batch size 512

Figure C.31.: Scaling efficiency using only PCIe compared to using NVLink on DGX-1 V100

List of Figures

2.1. Deep Learning in the context of Artificial Intelligence 5
2.2. Convolution operation, adapted from [9] 8
2.3. Pooling operation, adapted from [9] 9
2.4. Inception module with dimensionality reduction using 1x1 convolutions, [19] 10
2.5. A residual block, [23] 11
2.6. RNN and the unfolding in time of the computation involved in its forward computation, [6, p. 442] 12
2.7. FATHOM: normalized execution time, GPU vs. CPU [31] 14
2.8. Fathom: Breakdown of execution time by operation type, adapted from [31] 15

3.1. NVIDIA DGX-1 with 8 NVIDIA P100 GPUs network topology [29, p. 9] 20
3.2. NVIDIA DGX-1 with 8 NVIDIA V100 GPUs [53, p. 9] 22
3.3. IBM Power System AC922 with 4 V100 GPUs [56, p. 8] 23

5.1. Exemplary connection types used in nvidia-smi topo query result, adapted from [29] 34
5.2. DGX-1 P100 bandwidth for unidirectional data copies 35
5.3. DGX-1 P100 bandwidth for bidirectional data copies 35
5.4. Latency for GPU-to-GPU unidirectional memory copies on DGX-1 P100 36
5.5. DGX-1 P100 bandwidth for DMA read 37
5.6. DGX-1 V100 bandwidth for unidirectional data copies 38
5.7. DGX-1 V100 bandwidth for DMA read 39
5.8. AC922 bandwidth for unidirectional data copies 40
5.9. Scaling training of alexnet with (p2p enabled) and without NVLink (p2p disabled) 42
5.10. Scaling training of GoogLeNet with (p2p enabled) and without NVLink (p2p disabled) 43
5.12. Scaling efficiency using only PCIe compared to using NVLink on DGX-1 V100 44
5.11. Scaling efficiency using only PCIe compared to using NVLink on DGX-1 P100 44

5.13. Scaling efficiency with (p2p enabled) and without NVLink (p2p disabled) 45
5.15. Scaling efficiency using only PCIe compared to using NVLink on DGX-1 V100 . . . 46
5.14. Scaling efficiency using only PCIe compared to using NVLink on DGX-1 P100 . . . 46
5.16. Scaling training of lenet5 with (p2p enabled) and without NVLink (p2p disabled) . . . 47
5.17. Scaling efficiency: Training on 4 GPUs compared to 1 GPU) . . . 49
5.18. Scaling efficiency: Training on 8 GPUs compared to 1 GPU) . . . 49

C.1. DGX-1 P100 bandwidth for unidirectional data copies . . . 65
C.2. DGX-1 P100 bandwidth for bidirectional data copies . . . 65
C.3. DGX-1 P100 bandwidth for DMA read . . . 66
C.4. DGX-1 P100 bandwidth for DMA write . . . 66
C.5. Latency for GPU-to-GPU unidirectional memory copies on DGX-1 P100 67
C.6. Latency for GPU-to-GPU bidirectional memory copies on DGX-1 P100 . 68
C.7. Latency for GPU-to-GPU DMA read on DGX-1 P100 . . . 69
C.8. DGX-1 V100 bandwidth for unidirectional data copies . . . 70
C.9. DGX-1 V100 bandwidth for bidirectional data copies . . . 70
C.10. DGX-1 V100 bandwidth for DMA read . . . 71
C.11. DGX-1 V100 bandwidth for DMA write . . . 71
C.12. Latency for GPU-to-GPU unidirectional memory copies on DGX-1 V100 73
C.13. Latency for GPU-to-GPU bidirectional memory copies on DGX-1 V100 74
C.14. Latency for GPU-to-GPU DMA read on DGX-1 V100 . . . 75
C.15. AC922 bandwidth for unidirectional data copies . . . 76
C.16. AC922 bandwidth for bidirectional data copies . . . 76
C.17. Scaling training of alexnet with (p2p enabled) and without NVLink (p2p disabled) . . . 77
C.18. Scaling training of googlenet with (p2p enabled) and without NVLink (p2p disabled) . . . 77
C.19. Scaling training of inception3 with (p2p enabled) and without NVLink (p2p disabled) . . . 78
C.20. Scaling training of inception4 with (p2p enabled) and without NVLink (p2p disabled) . . . 78
C.21. Scaling training of resnet50 with (p2p enabled) and without NVLink (p2p disabled) . . . 79
C.22. Scaling training of resnet152 with (p2p enabled) and without NVLink (p2p disabled) . . . 79

C.23. Scaling training of vgg11 with (p2p enabled) and without NVLink (p2p disabled) 80
C.24. Scaling training of vgg16 with (p2p enabled) and without NVLink (p2p disabled) 80
C.25. Scaling training of lenet5 with (p2p enabled) and without NVLink (p2p disabled) 81
C.26. Scaling efficiency 4 GPUs versus 1 GPU with (p2p enabled) and without NVLink (p2p disabled) 82
C.27. Scaling efficiency 8 GPUs versus 1 GPU with (p2p enabled) and without NVLink (p2p disabled) 83
C.28. Scaling efficiency using only PCIe compared to using NVLink on DGX-1 P100 84
C.29. Scaling efficiency using only PCIe compared to using NVLink on DGX-1 P100 85
C.30. Scaling efficiency using only PCIe compared to using NVLink on DGX-1 V100 86
C.31. Scaling efficiency using only PCIe compared to using NVLink on DGX-1 V100 87

List of Tables

2.1. PCIe bandwidths for different generations and lane widths 17

4.1. Batch sizes for different neural network models 30
4.2. GPU interconnect map - preinstalled TensorFlow version 32
4.3. GPU interconnect map - changed TensorFlow version 32

5.1. Profiling results nvprof, 8 GPUs 41

C.1. nvidia-smi topo DGX-1 P100; nvidia-smi topo -m 63
C.2. nvidia-smi topo DGX-1 P100; nvidia-smi topo -mp 63
C.3. nvidia-smi topo DGX-1 V100; nvidia-smi topo -m 64
C.4. nvidia-smi topo DGX-1 V100; nvidia-smi topo -mp 64
C.5. nvidia-smi topo AC922; nvidia-smi topo -m 64

Bibliography

[1] M. Guignard, M. Schild, C. S. Bederián, N. Wolovick, and A. J. Vega, "Performance characterization of state-of-the-art deep learning workloads on an ibm "minsky" platform," *Frontiers in AI and Software Engineering*, Jan. 2018.

[2] Y. You, A. Buluç, and J. Demmel, "Scaling deep learning on gpu and knights landing clusters," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17, Denver, Colorado: ACM, 2017, 9:1–9:12, ISBN: 978-1-4503-5114-0. DOI: 10.1145/3126908.3126912. [Online]. Available: http://doi.acm.org/10.1145/3126908.3126912.

[3] N. Corporation, *Nvlink fabric*. [Online]. Available: https://www.nvidia.com/en-us/data-center/nvlink/.

[4] V. Sze, Y. Chen, T. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017, ISSN: 0018-9219. DOI: 10.1109/JPROC.2017.2761740.

[5] J. McCarthy, M. Minsky, N. Rochester, and C. E. Shannon, "A proposal for the dartmouth summer research project on artificial intelligence, august 31, 1955," *AI Magazine*, vol. 27, pp. 12–14, 2006.

[6] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, p. 436, May 2015. [Online]. Available: https://doi.org/10.1038/nature14539.

[7] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[8] 2019. [Online]. Available: https://www.nature.com/subjects/learning-algorithms.

[9] R. Wartala, *Praxiseinstieg Deep Learning*. O'Reilly Media, Inc., 2018.

[10] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013. arXiv: 1312.5602. [Online]. Available: http://arxiv.org/abs/1312.5602.

[11] Y. Bengio, “Deep learning of representations for unsupervised and transfer learning,” in *Proceedings of the 2011 International Conference on Unsupervised and Transfer Learning Workshop - Volume 27*, ser. UTLW’11, Washington, USA: JMLR.org, 2011, pp. 17–37. [Online]. Available: http://dl.acm.org/citation.cfm?id=3045796.3045800.

[12] D. Erhan, Y. Bengio, A. Courville, P.-A. Manzagol, P. Vincent, and S. Bengio, “Why does unsupervised pre-training help deep learning?” *J. Mach. Learn. Res.*, vol. 11, pp. 625–660, Mar. 2010, ISSN: 1532-4435. [Online]. Available: http://dl.acm.org/citation.cfm?id=1756006.1756025.

[13] S. S. Haykin, *Neural networks and learning machines*, Third. Upper Saddle River, NJ: Pearson Education, 2009.

[14] G. Cybenko, “Approximation by superpositions of a sigmoidal function,” *Mathematics of Control, Signals and Systems*, vol. 2, no. 4, pp. 303–314, 1989, ISSN: 1435-568X. DOI: 10.1007/BF02551274. [Online]. Available: https://doi.org/10.1007/BF02551274.

[15] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006, ISBN: 0387310738.

[16] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998, ISSN: 0018-9219. DOI: 10.1109/5.726791.

[17] D. Mishkin, N. Sergievskiy, and J. Matas, “Systematic evaluation of CNN advances on the imagenet,” *CoRR*, vol. abs/1606.02228, 2016. arXiv: 1606.02228. [Online]. Available: http://arxiv.org/abs/1606.02228.

[18] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS’12, Lake Tahoe, Nevada: Curran Associates Inc., 2012, pp. 1097–1105. [Online]. Available: http://dl.acm.org/citation.cfm?id=2999134.2999257.

[19] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going deeper with convolutions,” in *Computer Vision and Pattern Recognition (CVPR)*, 2015. [Online]. Available: http://arxiv.org/abs/1409.4842.

[20] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *CoRR*, vol. abs/1409.1556, 2014. arXiv: 1409.1556. [Online]. Available: http://arxiv.org/abs/1409.1556.

[21] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," *CoRR*, vol. abs/1512.00567, 2015. arXiv: `1512.00567`. [Online]. Available: `http://arxiv.org/abs/1512.00567`.

[22] C. Szegedy, S. Ioffe, and V. Vanhoucke, "Inception-v4, inception-resnet and the impact of residual connections on learning," *CoRR*, vol. abs/1602.07261, 2016. arXiv: `1602.07261`. [Online]. Available: `http://arxiv.org/abs/1602.07261`.

[23] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. arXiv: `1512.03385`. [Online]. Available: `http://arxiv.org/abs/1512.03385`.

[24] X. Li, T. Qin, J. Yang, and T. Liu, "Lightrnn: Memory and computation-efficient recurrent neural networks," *CoRR*, vol. abs/1610.09893, 2016. arXiv: `1610.09893`. [Online]. Available: `http://arxiv.org/abs/1610.09893`.

[25] M. Elbayad, L. Besacier, and J. Verbeek, "Pervasive attention: 2d convolutional neural networks for sequence-to-sequence prediction," *CoRR*, vol. abs/1808.03867, 2018. arXiv: `1808.03867`. [Online]. Available: `http://arxiv.org/abs/1808.03867`.

[26] S. Bai, J. Z. Kolter, and V. Koltun, "An empirical evaluation of generic convolutional and recurrent networks for sequence modeling," *CoRR*, vol. abs/1803.01271, 2018. arXiv: `1803.01271`. [Online]. Available: `http://arxiv.org/abs/1803.01271`.

[27] O. Yadan, K. Adams, Y. Taigman, and M. Ranzato, "Multi-gpu training of convnets," *CoRR*, vol. abs/1312.5853, 2013. arXiv: `1312.5853`. [Online]. Available: `http://arxiv.org/abs/1312.5853`.

[28] T. Dettmers, "8-bit approximations for parallelism in deep learning," *CoRR*, vol. abs/1511.04561, 2015. arXiv: `1511.04561`. [Online]. Available: `http://arxiv.org/abs/1511.04561`.

[29] "Nvidia dgx-1 system architecture," NVIDIA Corporation, White Paper, 2017. [Online]. Available: `https://www.azken.com/images/dgx1_images/dgx1-system-architecture-whitepaper1.pdf`.

[30] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *CoRR*, vol. abs/1802.09941, 2018. arXiv: `1802.09941`. [Online]. Available: `http://arxiv.org/abs/1802.09941`.

[31] R. Adolf, S. Rama, B. Reagen, G. Wei, and D. M. Brooks, "Fathom: Reference workloads for modern deep learning methods," *CoRR*, vol. abs/1608.06581, 2016. arXiv: `1608.06581`. [Online]. Available: `http://arxiv.org/abs/1608.06581`.

[32] N. R. Tallent, N. A. Gawande, C. Siegel, A. Vishnu, and A. Hoisie, "Evaluating on-node gpu interconnects for deep learning workloads," in *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, S. Jarvis, S. Wright, and S. Hammond, Eds., Cham: Springer International Publishing, 2018, pp. 3–21, ISBN: 978-3-319-72971-8.

[33] D. A. P. J. L. Hennesy, *Rechnerorganisation und Rechnerentwurf*. DeGruyter, 2016.

[34] J. Schmidhuber, "Deep learning in neural networks: An overview," *CoRR*, vol. abs/1404.7828, 2014. arXiv: 1404.7828. [Online]. Available: http://arxiv.org/abs/1404.7828.

[35] S. Shams, R. Platania, K. Lee, and S. Park, "Evaluation of deep learning frameworks over different hpc architectures," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017, pp. 1389–1396. DOI: 10.1109/ICDCS.2017.259.

[36] "Pci express base specification revision 3.0," PCI-SIG, Specification, 2010.

[37] H. Roland, "Rechnerarchitektur, einführung in den aufbau moderner computer," 2016. DOI: 10.1515/9783110496642. [Online]. Available: https://www.degruyter.com/view/product/476754.

[38] NI, "Pci express – an overview of the pci express standard," National Instruments, White Paper, 2014. [Online]. Available: http://www.ni.com/white-paper/3767/en/.

[39] "Pci express base specification revision 5.0," PCI-SIG, Specification, 2018.

[40] "Pci express base specification revision 1.1," PCI-SIG, Specification, 2005.

[41] "Pci express base specification revision 2.0," PCI-SIG, Specification, 2006.

[42] "Pci express base specification revision 4.0," PCI-SIG, Specification, 2017.

[43] P. Czarnul, *Parallel Programming for Modern High Performance Computing Systems*. Feb. 2018, ISBN: 9781138305953.

[44] D. Vandeth, *Accelerating applications with cpu-gpu nvlink*, 2016. [Online]. Available: http://on-demand.gputechconf.com/gtcdc/2016/video/dcs16172-drew-vandeth-accelerating-applications.mp4.

[45] *Nvswitch technical overview*, 2018. [Online]. Available: http://images.nvidia.com/content/pdf/nvswitch-technical-overview.pdf.

[46] Intel, "An introduction to the intel quickpath interconnect," Intel Corporation, Tech. Rep., 2009. [Online]. Available: https://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html.

[47] IBM, "Ibm power system ac922 - introduction and technical overview," IBM Corporation, Tech. Rep., 2016. [Online]. Available: http://www.redbooks.ibm.com/redpapers/pdfs/redp5472.pdf.

[48] *Nvidia dgx-1 datasheet*, 2017. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/dgx-1/dgx-1-ai-supercomputer-datasheet-v4.pdf.

[49] *Nvidia tesla p100*, White Paper, 2017. [Online]. Available: http://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper-v1.2.pdf.

[50] *Nvidia tesla p100 gpu accelerator - pcie version*, Data Sheet, 2016. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-p100/pdf/nvidia-tesla-p100-PCIe-datasheet.pdf.

[51] *Nvidia tesla p100 gpu accelerator - nvlink version*, Data Sheet, 2016. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-p100/pdf/nvidia-tesla-p100-datasheet.pdf.

[52] *Tensorflow release 18.07*, 2018. [Online]. Available: https://docs.nvidia.com/deeplearning/dgx/tensorflow-release-notes/.

[53] *Nvidia dgx-1 with tesla v100 system architecture*, 2017. [Online]. Available: http://images.nvidia.com/content/pdf/dgx1-v100-system-architecture-whitepaper.pdf.

[54] *Nvidia tesla v100 gpu architecture*, 2017. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[55] *Nvidia tesla v100 gpu accelerator*, 2018. [Online]. Available: https://images.nvidia.com/content/technologies/volta/pdf/tesla-volta-v100-datasheet-letter-fnl-web.pdf8.

[56] F. Manila, *Ibm powerai deep learning platform*. [Online]. Available: https://www.slideshare.net/ganesannarayanasamy/powerai-deep-dive.

[57] T. Ben-Nun, *Mgbench: Multi-gpu computing benchmark suite*. [Online]. Available: https://github.com/tbennun/mgbench (visited on).

[58] NVIDIA, *Peer device memory access*, 2018. [Online]. Available: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__PEER.html#group__CUDART__PEER_1g2b0adabf90db37e5cfddc92cbb2589f3.

[59] Google, *Tensorflow benchmarks*. [Online]. Available: https://github.com/tensorflow/benchmarks.

[60] *Tensorflow branch r1.8*, 2018. [Online]. Available: https://github.com/tensorflow/tensorflow/tree/r1.8.