Delft University of Technology

Monitoring hardware utilization when training on GPUs in distributed machine learning

Mark Provó Kluit

TU Delft

Monitoring hardware utilization when training on GPUs in distributed machine learning

by

Mark Provó Kluit

Master's Thesis in Computer Science

Distributed Systems group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

August 24, 2022

Student number:	1263099	
Graduation committee:	Dr. Lydia Chen,	TU Delft
	Prof Dr. Jan S. Rellermeyer,	TU Delft, supervisor
	Dr. Burcu Özkan,	TU Delft

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft Delft University of Technology

Abstract

Large-scale machine learning frameworks can accelerate training of a neural network by performing distributed training on a cluster using multiple GPUs per node and multiple nodes. Because distributed training on a cluster involves many nodes which need to communicate and load and exchange data, a machine learning framework may at certain times during training not fully utilize the available hardware of the system. Various techniques are assessed in their capability to measure the performance of specific parts of the hardware of a cluster. We present ML Board, a tool that measures and visualizes the utilization of the system while training a neural network model using some of the previously assessed techniques, and does so without requiring any changes to the used machine learning framework. ML Board can be used to identify straggling nodes, and by subsequently letting the user select different nodes using the Slurm job scheduler, can help to decrease the training time of a ResNet model by between 15 and 45% when using an ImageNet or CIFAR-10 dataset. Furthermore, the energy used by the GPUs can be measured and used to identify and replace GPUs to reduce the total used energy by between 5 to 16%.

Preface

Ever since being introduced to computers, I have had an interest in the performance aspect of software running on these systems. Naturally, during my study Computer Science, I was drawn to the field of parallel and distributed systems. With machine learning being a relevant field, having an opportunity to contribute to improving the performance of training machine learning models was a motivation for this thesis. The endeavor to help improving the performance has resulted in this thesis.

This thesis could not have been completed without the guidance and support of several people. First of all, I would like to thank my thesis advisor, Prof Dr. Jan S. Rellermeyer, for his guidance and support throughout the duration of this thesis. I want to thank Dr. Lydia Chen for chairing the thesis committee and Dr. Burcu Özkan for being a part of it. I would like to thank Shivansh Dhar for helping me to get started with GPUs on the DAS-5/VU cluster and Dan Graur for answering my questions about several machine learning frameworks. Most importantly, I want to thank my family and friends for inspiring me to follow this journey to the end, and in particular, my brother Robert for creating the front cover art and my parents for their continuous moral support.

Mark Provó Kluit
Delft, August 2022

Contents

Preface **v**

1 Introduction **1**
1.1 Problem Statement . . . 2
1.2 Research Approach . . . 2
1.3 Thesis Outline and Contributions . . . 3

2 Background **5**
2.1 Modern hardware . . . 5
2.1.1 Central Processing Units . . . 5
2.1.2 Graphics Processing Units . . . 7
2.1.3 Interconnects . . . 9
2.2 Execution models on GPUs . . . 9
2.2.1 CUDA . . . 10
2.3 Neural networks . . . 10
2.3.1 Layers . . . 11
2.3.2 Activation functions . . . 11
2.3.3 Convolution . . . 12
2.3.4 Downsampling and pooling . . . 13
2.3.5 Regularization and batch normalization . . . 13

3 Effect of cluster, datasets, and models on hardware utilization **15**
3.1 Research questions . . . 15
3.2 Set-up of the experiments . . . 16
3.2.1 Hardware used . . . 16
3.2.2 Software used . . . 16
3.2.3 Execution on the cluster . . . 17
3.3 Experiments . . . 17
3.3.1 Metrics . . . 17
3.3.2 Datasets . . . 17
3.3.3 Models and hyperparameters . . . 17
3.3.4 Experiments . . . 18
3.4 Results . . . 18
3.4.1 Collective communication backends . . . 18
3.4.2 Interconnects . . . 24
3.4.3 Batch sizes . . . 24
3.5 Performance profiling . . . 25
3.6 Discussion . . . 25

4 Monitoring techniques **27**
4.1 Research questions . . . 27
4.2 Tools and libraries . . . 27
4.2.1 nvidia-smi . . . 29
4.2.2 NVML . . . 29
4.2.3 Nsight Compute . . . 29
4.2.4 Nsight Systems . . . 29
4.2.5 perf . . . 30
4.2.6 perf_event_open . . . 30
4.2.7 eBPF . . . 30

4.2.8 psutil . . . 30
4.2.9 Perfetto . . . 30
4.2.10 CUPTI . . . 31
4.2.11 Netintercept . . . 31
4.2.12 sysfs . . . 31
4.3 Metrics . . . 32
4.3.1 nvidia-smi and NVML . . . 35
4.3.2 psutil . . . 35
4.4 Discussion . . . 37

5 ML Board **39**
5.1 Research questions . . . 39
5.2 Objectives . . . 39
5.3 Visualizations . . . 40
5.3.1 Idle time between kernels . . . 40
5.3.2 Statistics of launched kernels . . . 42
5.3.3 GPU utilization per node . . . 44
5.3.4 Common tasks of machine learning frameworks . . . 44
5.4 Architecture . . . 48
5.4.1 Context . . . 48
5.4.2 Functional . . . 49
5.4.3 Concurrency . . . 50
5.5 Discussion . . . 52

6 Evaluation **53**
6.1 Research questions . . . 53
6.2 Sets of nodes . . . 53
6.3 Datasets . . . 56
6.4 Models . . . 57
6.5 Batch sizes . . . 58
6.5.1 Energy . . . 58
6.5.2 Power . . . 59
6.5.3 Memory and GPU utilization . . . 60
6.6 Use cases . . . 61
6.6.1 Identifying straggling nodes . . . 61
6.6.2 Reducing total energy usage . . . 63
6.6.3 Selecting a batch size . . . 63
6.7 Discussion . . . 63

7 Conclusion **65**
7.1 Discussion of research questions . . . 66
7.2 Future Work . . . 67

Bibliography **69**

A Results **75**
A.1 Communication backends single-GPU multi-node . . . 75
A.2 Communication backends multi-GPU multi-node . . . 77
A.3 Interconnects single-GPU multi-node . . . 79
A.4 Interconnects multi-GPU multi-node . . . 81

1 Introduction

In the last decade or so, the increasing amount of data and computational power available, has caused *Machine Learning (ML)* to have seen a significant increase in usage to analyze and make predictions on data. Machine Learning can be applied when a problem is too complex to be solved by an algorithm. Examples of ML are image classification and semantic segmentation for medical [37] and automotive applications [66, 67], face recognition for surveillance and object detection for navigation and collision avoidance by autonomous vehicles [5, 22, 23, 39], speech recognition and synthesis in home assistant and IoT devices [6, 56], style transfer to generate images [15, 30], and reinforcement learning for AI in esports [46, 60].

As time passed by, the time to train models and the computation and memory requirements of these models for inference has been reduced with algorithmic improvements to these models and by reducing the memory bandwidth that is required. These include using different building blocks in layers or using a different architecture to reduce the number of hyperparameters [24, 54], and applying quantization during inference and using quantization-aware training [19]. However, sometimes the amount of data cannot fit into memory all at once or the distribution of the data does not allow one to train and infer on a single machine. Thus these conditions force the use of multiple machines to parallelize the training and evaluation of models.

To speed up training models with large-scale machine learning frameworks, one can scale up and scale out. Scaling up involves adding resources to a node, while scaling out means adding more nodes to the system.

Scaling up can occur by taking advantage of SIMD instruction sets in CPUs like AVX-512 in order to operate on multiple elements at once or to operate on special floating-point formats like bfloat16 [55] and int8 [48], where the reduced precision, compared to single-precision, can be used to speed up computations and reduce memory costs of deep learning applications [29]. Another way of scaling up is using hardware designed to process data with multiple hardware threads like *Graphics Processing Units (GPU)* or specialized hardware like Google's *Edge Tensor Processing Unit (Edge TPU)*, which can do inferencing of TensorFlow Lite models on a USB or PCIe accelerator [1].

To further accelerate training and inference, scaling up can be complemented by scaling out by adding more nodes to the system. This leads to a distributed system where nodes need to communicate, synchronize, and exchange data with each other. Input data is also often ingested from other machines in the data center, or even from remote locations in the cloud. With a system consisting of multiple machines, latencies exist between these machines, sometimes infinite if a machine or some other hardware struggles or fails completely. Due to the presence of latencies, nodes may fall idle, reducing the utilization of its hardware. It may also have an effect on the scalability and the ability to detect and recover from failures on fault-tolerance.

[1] https://coral.ai/products

Detecting when and in which components of the system hardware utilization is lower than expected might help in improving the scalability of the system. Thus the objective of this thesis is to answer the question: *which techniques can measure inefficiencies or bottlenecks when training models with machine learning frameworks on a distributed system?*

The remaining part of this chapter is as follows: Section 1.1 describes the problem and lists the research questions this thesis will answer, in Section 1.2 the methodology used to perform the experiments is explained, and in Section 1.3 an outline of the remaining chapters and the main contributions of this thesis are given.

1.1 Problem Statement

Large-scale ML frameworks have the ability to perform distributed training on multiple GPUs per node and on a cluster of multiple nodes. Because the system consists of multiple parts, these parts need to communicate and exchange data with each other. This distributed nature of the system means there are nodes which are idle at certain times, thus making inefficient use of the hardware of the cluster.

In order to measure inefficiencies, the duration of various parts of large-scale ML frameworks needs to be measured. There are currently a few popular ML frameworks used in research and production, like PyTorch and TensorFlow [1]. Previous work [17] evaluated the scalability of parameter update strategies using synthetic input data to avoid bias. In this thesis, instead experiments will be run with various well-known public datasets and models to learn whether the input data or model used has an effect on the utilization of the hardware.

To answer the question of what inefficiencies or bottlenecks exist in large-scale ML frameworks, the duration of the training and prediction phase and the parts of the system and the interaction between different nodes needs to be measured. For the training phase, the forward pass and backward pass needs to be measured. Another aspect is the loading of the dataset, where not only the size of the dataset is a factor, but also the method of transporting this data to the parts of the system that are responsible for training or testing the model.

This thesis will focus on the following research questions:

RQ1 Does the hardware utilization depend on the dataset, the model, hyperparameters, or configuration of the system like number of GPUs and used interconnect?

RQ2 Which parts of the system can be measured and visualized?

RQ3 How much time is spend on communication between nodes, loading data, updating parameters, and other aspects in popular large-scale machine learning frameworks?

RQ4 Which recommendations can be given to speed up training of neural network models?

1.2 Research Approach

The research questions mentioned above are answered by performing empirical research on a small cluster of nodes with GPUs and different network interconnects available. Experiments will use the PyTorch ML framework and use a small set of datasets like CIFAR-10 and ImageNet, and well-known models used in research papers like ResNet [21] and MobileNet V2 [54].

This thesis is inspired by [17], which focused on the scalability and resource usage of various parameter update strategies during training on CPUs. In this thesis, training is performed on GPUs instead, which are often used to reduce the time to do training and inference [61]. While [17] used synthetic data, this thesis does use various public datasets because the experiments happens on GPUs and because **RQ1** tries to answer whether using different datasets affect the hardware utilization.

To answer **RQ1** and **RQ3**, experiments are run on the DAS-5/VU [4] with up to 16 nodes with 2x E5-2630 v3 CPUs (16 cores and 32 threads per node) and 1x GeForce GTX TITAN X GPUs each. Experiments are repeated with 1, 2, 4, 8, and 16 nodes. A few nodes provide 2 or 3 GPUs and separate additional experiments are run with multiple GPUs on these nodes.

Some experiments will instead use a small number of nodes with 1x or 2x RTX 2080 Ti depending on the chipsets that are supported by the various tools of the NVIDIA CUDA toolkit that is installed on the system. These nodes also have a more modern CPU (Skylake generation). The cluster provides Ethernet and InfiniBand as network interconnect. Libraries like Gloo and NCCL are used for communication between GPUs and nodes by the ML frameworks.

Answering **RQ3** requires measuring the time spend during various phases of the training and testing workloads by different components of the ML frameworks. In order to measure the time, the code of the framework may need to be instrumented. Utilization of the hardware may be measured with the `nvidia-smi` tool. To limit the scope of the experiments, the thesis restricts itself to the ML framework PyTorch, which is one of the most popular frameworks used in research and production [20].

To answer **RQ2**, several tools and libraries are collected and assessed to which degree they are capable of measuring certain parts of the system. Useful tools and libraries are then used to build a new tool, ML Board, tailored to providing insight into the performance of machine learning workloads. ML Board can then be used to answer **RQ3** and **RQ4**.

1.3 Thesis Outline and Contributions

The remaining of this thesis is structured as follows: in Chapter 2 the background of the problem is described. Chapter 3 shows the hardware utilization during various experiments with a single or multiple GPUs, different number of nodes, and network interconnects when training and evaluating various datasets and models on the GPUs with PyTorch. In Chapter 4 a number of tools and libraries are investigated that can help to build a tool that visualizes the utilization of the hardware. Chapter 5 describes the framework of ML Board, the monitoring tool developed during this thesis. In Chapter 6 the tool is evaluated, and Chapter 7 provides the conclusions and future work. A schematic overview of the chapters is shown in Figure 1.1.

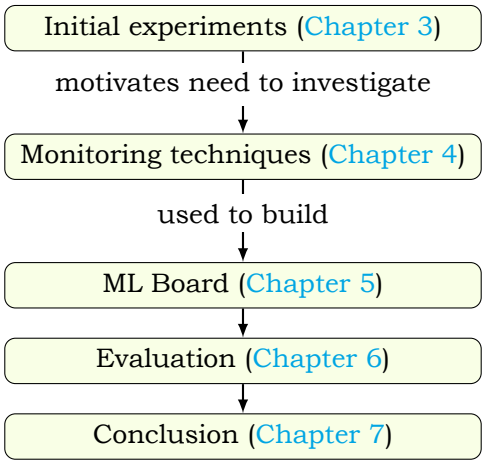


Figure 1.1: *Overview of the chapters of this thesis.*

The main contributions are:

- A prototype of a tool that visualizes the utilization and efficiency of the system, including hardware like GPUs.
- A survey of monitoring techniques and a list of visualizations that highlight the utilization of the system.
- Recommendations for the user to speed up training of a neural network model on the used hardware or to reduce the total used energy needed for training.

2 Background

In this chapter some background information is provided about the used hardware and software when training and testing neural network models on a distributed system. This chapter is structured as follows: Section 2.1 gives an overview of the general hardware used by the experiments in the following chapters. Section 2.2 describes how programs are run on video cards, and Section 2.3 explains the details of neural networks.

2.1 Modern hardware

2.1.1 Central Processing Units

An application or program is a set of algorithms and is made up of a sequence of instructions, executed by the *Central Processing Unit (CPU)*. The instructions are loaded by the *Operating System (OS)* into main memory and then fetched, decoded, and executed by the CPU. The CPU keeps track of which instruction to execute using a special register known as the *program counter*.

A small CPU like a microcontroller in an embedded system has usually one or two physical cores that can execute instructions from different threads concurrently. A modern desktop CPU has between four and 16 cores and server CPUs can have over a 100 cores. Because some of the instructions operate on different parts of the CPU, a CPU can expose a physical core as multiple logical cores. This allows multiple threads (from one or more programs) to run concurrently on one physical core. This technique is called *Simultaneous Multithreading (SMT)*. Common desktop CPUs support two logical cores per physical core, but some server CPUs are known to support up to eight logical cores. SMT may increase the efficiency of the hardware, but running two different applications on the same physical core can have an impact on security due to the shared use of hardware. A malicious application may exploit mistakes in the microarchitecture of the CPU to cause a core to inadvertently leak data from another application.

A CPU core consists of several execution units. These include units like the *Arithmetic Logic Unit (ALU)* and *Floating-Point Unit (FPU)*, which perform arithmetic on integers and floating-point numbers respectively, a unit that is responsible for loading and storing data from and to memory, and the branch predictor, which is responsible for predicting which instructions of a program are needed in the near future. Some microcontrollers do not have an FPU and in this case instructions on floating-point numbers need to be emulated.

Besides speeding up execution of a program by parallelizing parts of it so that these parts can be executed concurrently, known as *task parallelism*, a program may also use special instructions that operate on multiple numbers at once (*Single Instruction Multiple Data (SIMD)*), which is a form of *data parallelism*. SIMD extensions to CPU instruction sets were originally added for *Digital Signal Processing (DSP)* applications like video and audio decoding and encoding, but various new variants can be used for all kinds of algorithms, including matrix multiplication in linear algebra and parsing JSON files [32].

At a lower level, a CPU may transparently execute multiple different instructions without the application being aware of it if these instructions have no dependency on each other and use different parts (execution units) of the CPU, execute instructions in a different order than intended by the programmer or compiler, and use pipelining to fetch instructions, while decoding and executing others.

Parts of the application consists of instructions that must be run sequentially, while other parts may run concurrently in separate threads of control. This ratio between the sequential and parallel parts of an application gives an upper bound on the maximum speedup as a function of the number of CPU cores. For example, if 50% an application is run in parallel, the maximum speedup is 2, and if 90% is run in parallel, the maximum speedup is 10, even with an infinite number of CPU cores. This is known as Amdahl's law[1]. Fortunately, Amdahl's law assumes the application operates on a data set with a fixed size and more cores can be utilized by increasing the size of the data set on which the application operates.

Caches

After the instructions of a program have been loaded into main memory by the OS, the instructions are then fetched by the CPU. Because it would be quite slow to fetch each individual instruction, a window of instructions is fetched in one go. These instructions are stored in a small piece of memory on the CPU called the L1 instruction *cache*. Although the cache is quite small, it is much faster than accessing the main memory. As programs may contains loops (using an instruction that sets the program counter back to an earlier memory address) a *cache hit* will occur if the required instructions are found in the cache. Otherwise a *cache miss* will occur, which means the CPU has to fetch the instructions from main memory. Unfortunately, some parts of the application may not have been loaded into main memory by the OS yet. If this happens, an exception is raised by the *Memory Management Unit (MMU)* of the CPU. The OS is then asked to load the required instructions or data into main memory.

Besides the L1 instruction cache, various caches exist for data as well. Each CPU core has a small L1 data cache and a larger L2 cache which can contain instructions as well as data. Most processors also have a large L3 cache that is shared between multiple cores. In general, as the level of the cache increases, size and latency of it increases as well. Figure 2.1 shows the CPUs and its caches of a compute node in the DAS-5 cluster. Devices like GPUs and network interfaces are connected to the first NUMA node, and thus CPU cores in the second node must communicate with the devices through the first node, incurring extra latency.

Trends

One of the continuous trends is to move computations to specialized hardware to increase the performance [35] and move computations to software when flexibility is needed. Many mobile and desktop CPUs now have SIMD extensions suitable for many tasks, including DSP. Select server processors support the AVX-512 extensions, of which parts, like BF16 and VNNI, are designed to accelerate AI tasks. To increase flexibility, many parts of modern systems have microcontrollers which need firmware to operate.

Another trend is that systems have more and more hardware that can process data in parallel. With semiconductor companies trying to keep up with Moore's law, the prediction that the amount of transistors on a chip would double every two years, physical limits are being reached as the size of transistors decrease. Other ways to increase the performance of CPUs is to simply increase the clock frequency. A processor will in this case execute more instructions per second. Increasing the clock frequency results also in an increase in power usage, noticeable in the form of heat. There are, however, limits to how much heat typical cooling solutions can dissipate. Figure 2.2 shows that with the increase in single-core performance slowing down in the mid 2000s, CPUs started to get more and more cores.

With CPUs having many cores, applications now need to do parallel programming to take maximum use of the available computational power. Parallel programming, however, can be much more difficult than sequential programming and gives rise to problems like deadlock,

[1] Speedup $S = 1/(s + p/N)$ where s is the sequential part, p the parallel part, and N the number of CPU cores

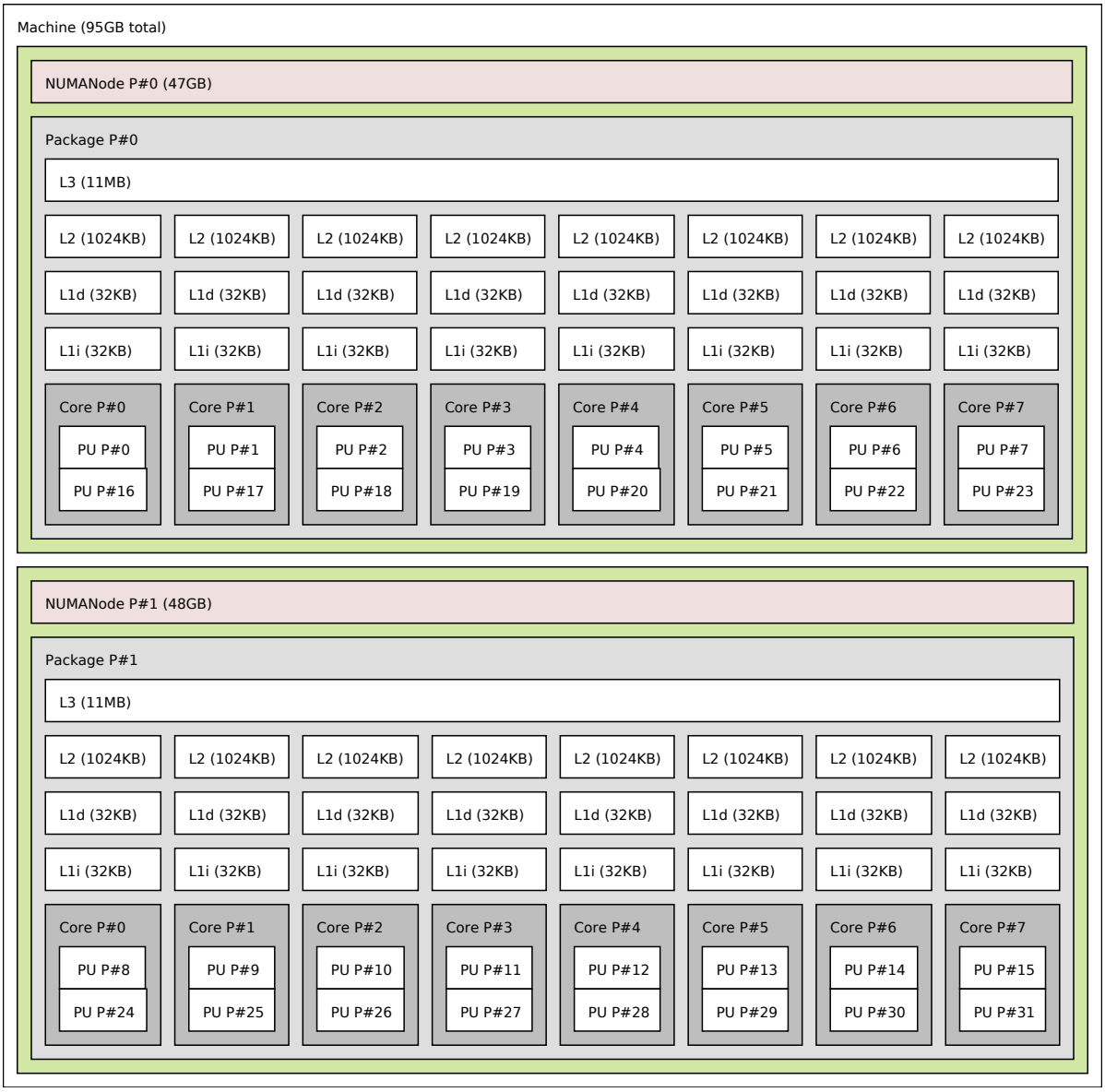


Figure 2.1: *Topology of two Skylake CPUs of a compute node in the DAS-5 cluster.*

where threads try to acquire resources that other threads are unable to release, halting a program (usually indefinitely). Unless a problem is embarrassingly parallel, communication and synchronization between threads is needed, which means the maximum speedup will be less than the increase in the number of cores due to the overhead of the communication.

Just as CPUs contain now more cores to increase the task parallelism, the width of SIMD registers has doubled several times to increase the data parallelism. The MMX instruction set, introduced in 1997, uses registers of 64 bits, while SSE from 1999 operates on 128 bits, and AVX (from 2011) and AVX-512 (from 2016) use registers that are 256 and 512 bits wide, respectively.

2.1.2 Graphics Processing Units

In the mid and late 1990s, specialized tasks used in 3-D rendering like transformations and lighting had moved from the CPU to the *Graphics Processing Units (GPU)* and in the late 2010s GPUs gained the ability to hardware-accelerate ray tracing using AI acceleration hardware known as *tensor cores*. More specialized hardware (*Application Specific Integrated Circuit*

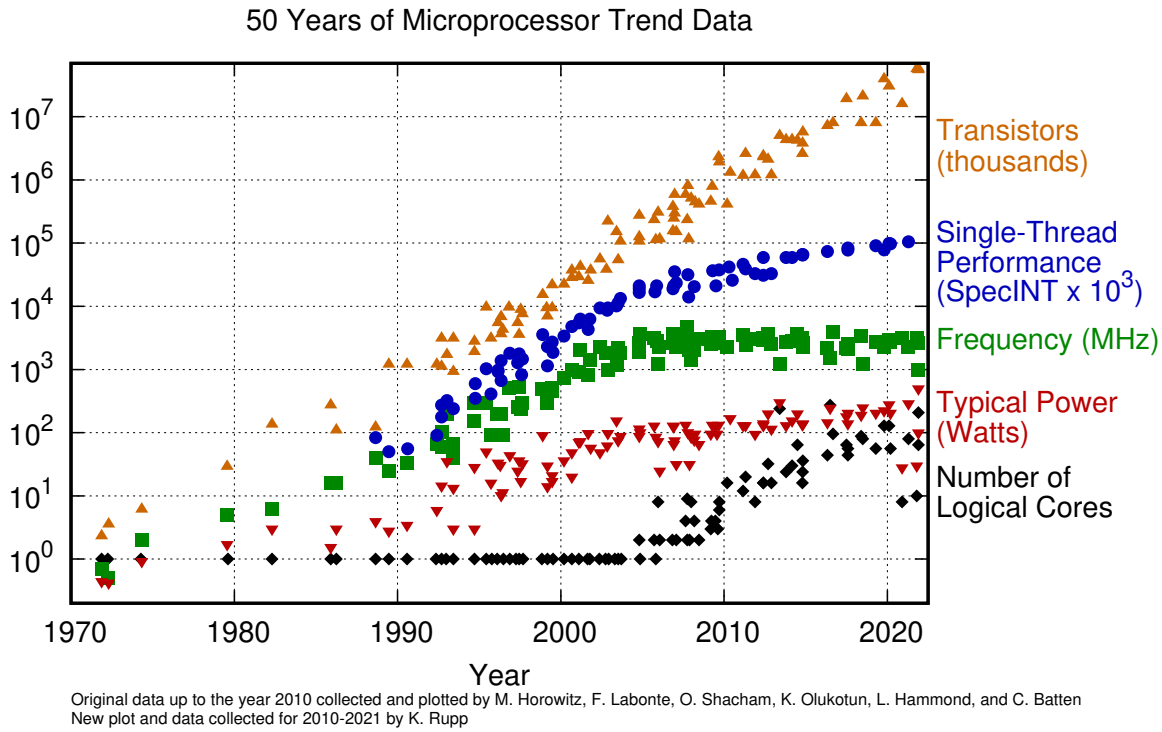


Figure 2.2: *Trends of CPUs [10, 35, 51] by K. Rupp [51] (CC-BY 4.0 International Public License).*

(ASIC)) exists like Google's *Edge Tensor Processing Unit (Edge TPU)* which can do inferencing (but not training) of some neural network models.

In the early and mid 2000s various parts of the *graphics pipeline*, a conceptual model that describes how 3-D models are transformed into 2-D pictures on the screen, became programmable with *shaders*, small programs that run on the GPU. Some steps of the pipeline, like the rasterization of triangles, are still performed using fixed-function hardware. APIs focused on general-purpose computing like CUDA and OpenCL, and APIs focused on rendering like OpenGL and Vulkan, provide access to the raw programmable hardware with compute shaders, or *kernels*, as they are called in APIs like CUDA. These kernels, which can read and write to large buffers, are part of a simpler *compute pipeline* and avoid hardware used only for rendering.

The hardware of GPUs will now be explained. While a CPU has a few cores, a GPU has in the order of tens of *Streaming Multiprocessors (SM)* (also known as Compute Units). Each SM can process multiple instructions in groups of 32 or 64 threads (depending on the vendor of the GPU). Such a group is also known as a *warp* and its threads are known as *lanes*. All lanes of a warp execute the same instruction and thus they operate in lockstep. This is known as *Single Instruction Multiple Threads (SIMT)*. In total a GPU has often a few thousand threads. An SM can execute a few warps in parallel and also have multiple warps in flight (since warps execute multiple instructions). Multiple warp schedulers determine which warps may execute instructions. If a warp needs to fetch data from memory, the scheduler may switch to another warp until the data has been fetched. Unlike a context switch on a CPU core from one program to another, switching between warps is very lightweight because an SM has a lot of registers for multiple warps.

Differences between the threads of an SM and the cores of a CPU are that CPUs devote a large amount of their die area to the branch predictor. GPUs instead are throughput-oriented and do not have out-of-order execution of instructions and predicting which branch of the program's control flow should be taken.

Because all lanes of a warp share a common program counter, they also must execute the same branch in the program's control flow. To handle *divergent control flow* when some threads must execute branch A while others must take branch B, the lanes of a warp will process both branches and use masking to enable or disable specific threads. Modern GPUs can execute just one branch if all threads decide to take that particular branch [3, 53].

The threads of a warp can communicate with each other using special instructions and each SM has a small amount of *shared memory* (around 64 or 96 KB) so that threads of different warps can communicate as well. Furthermore, a GPU has a large amount of L2 cache, and several *Texture Mapping Units (TMU)*, which fetch pixels from *textures* (n-dimensional images) and apply bilinear filtering in hardware. The automatic filtering can be taken advantage of to reduce the number of texture fetches when applying a convolutional filter to the pixels of an image [52]. Some other hardware that is used by the graphics pipeline but not by the compute pipeline is the *Raster Operations Pipeline (ROP)*, which does blending and antialiasing of framebuffers that are eventually displayed on the screen.

Some GPUs have special hardware optimized for processing matrix multiplications and additions, known by some vendors as tensor cores. These units can multiply values in a lower precision format (like FP16 or INT8), and then accumulate the result in the same or higher format like single-precision (FP32).

2.1.3 Interconnects

Ethernet and InfiniBand are standards that provide wired networking between different machines. Ethernet uses twisted pairs of copper cables or fiber optic cables. Most consumer hardware provides speeds of up to 1 Gbit/s, while hardware providing speeds of 10 or 100 Gbit/s is much more common in data centers. Data transmitted over Ethernet is split into separate packets called *frames*. Each frame has a source and destination address, called a MAC address. These addresses must be unique on the local network. Machines communicate with each other using IP packets, which is wrapped in a frame and transmitted over the wire. Machines use the ARP protocol to learn which MAC address belongs to an IP address. Many machines can be connected to one or more network switches, which often have between 8 and 48 ports. These switches use their *content-addressable memory (CAM)* to quickly lookup the network port on which a frame with some MAC address was last seen. Special addresses like `FF:FF:FF:FF:FF:FF` exist to broadcast a frame on all ports, which is needed for certain protocols.

InfiniBand is an interconnect that is typically used only by supercomputers. It supports *Remote Direct Memory Access (RDMA)*, which avoids involving the CPU and the OS. This results in a lower-latency than traffic that goes over the Ethernet interconnect and through the OS.

2.2 Execution models on GPUs

Programs executed by the CPU have one or more distinct sequence of instructions, each executed by a different thread of control. On the GPU, a program, better known as a kernel, is just one sequence of instructions, but run multiple times on up to thousands of threads instead of just one. Thus the GPU runs a kernel in parallel on a massive number of threads, with some of the invocations done in concurrently and others done sequentially if the number of invocations is greater than the number of physical threads provided by the hardware. To make effective use of these threads, the threads must be logically and physically divided into a number of groups. Various different APIs for general-purpose computing, including CUDA, make use of a hierarchy of two levels:

- Grid of blocks
- Block of threads

Each level or layer has one, two, or three dimensions. The number of dimensions depends on what the application needs. For example, an application may want to apply element-wise

transformations to a large data set, and thus will use a 1-D grid and block. Another application may want to apply *Computational Fluid Dynamics (CFD)* to compute the aerodynamics of a vehicle and use a 3-D grid and block.

Each thread block is scheduled independently to one of the available SMs. No communication can occur between threads in different thread blocks since they may be scheduled sequentially to the same SM. It is, however, possible to use memory barriers to flush caches and use multiple kernel launches so that a thread can use data written by another thread in a previous launch.

All threads of a thread block are run on one particular SM. Because an SM can only hold a limited number of threads in its registers, there is a limit on the total number of threads in a block (1024 on NVIDIA GPUs). To achieve good utilization of the threads in an SM, the total number of threads must be a multiple of the warp size (32 on NVIDIA GPUs). While the warp size is an implementation detail, programs can take advantage of knowing the size of a warp to avoid using barriers when exchanging data between the threads in the same warp.

The threads of a thread block can communicate in various: using atomic instructions to read and write to global memory (this also allows threads from different blocks to modify data), using shared memory, and using special functions to broadcast data between active lanes in a warp.

2.2.1 CUDA

CUDA is an API and a set of libraries that applications can use to perform computations on the GPU. The API can be used to launch kernels and copy data between host and device memory. Some of the libraries provided by CUDA provide optimized kernels for linear algebra such as matrix-matrix multiplication by cuBLAS, *Fast Fourier Transforms (FFT)* by cuFFT, and kernels for deep learning by cuDNN.

2.3 Neural networks

Supervised machine learning tries to compute an unknown function f that maps a set of inputs $x_1, \dots, x_n$, to some output y:

$$y = f(x_1, \dots, x_n) \tag{2.1}$$

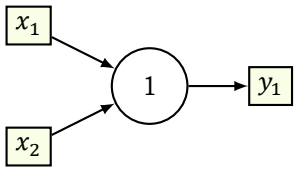
In regression the output can be a continuous value and in image classification the output is some discrete value such as a label. In object detection that are multiple labels, each with a corresponding bounding box that indicates the location of the object. While the function f is unknown, the goal of machine learning is to create a network that approximates f so that the error between an output y and the predicted $\hat{y}$ is minimized on average.

Such a function f is represented by a neural network, which consists of neurons, grouped into layers. Each neuron receives the outputs of some or all of the neurons of the previous layer as inputs, multiplies these with a set of *weights*, and adds a *bias*. For example, a 2-dimensional plane can be split into two halves by some straight line using a neural network that consists of a neuron with two inputs:

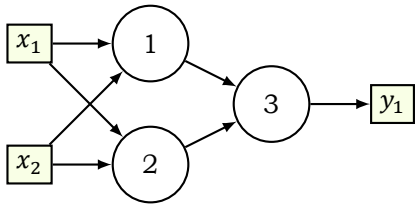
$$y = w_1 x_1 + w_2 x_2 + b \tag{2.2}$$

where w_1 and w_2 are the two weights and b the bias. A visual representation of the network in Equation (2.2) is shown in Figure 2.3a. If the output of a neuron in such a network is greater than zero ($y > 0$) then it is a binary classifier and called a perceptron. Single-layer perceptrons are used in the branch predictors of CPUs [28] because they are easy to train with no back-propagation needed.

The first layer of a neural network is called the *input layer*, the last layer as the *output layer*, and the layers in-between as the *hidden layers*. The earlier layers compute the most primitive features and deeper layers learn to recognize more abstract and higher-level features of the input composed of features learned by previous layers. A neural network with one input layer and one output layer is shown in Figure 2.3b. Each node has a bias and each link has



(a) *A network consisting of just one neuron.*



(b) *A network with one input layer with two neurons and an output layer with one neuron.*

Figure 2.3: *Examples of some small neural networks.*

a corresponding weight. More complex networks can have many layers with many neurons. The number of neurons in the layers indicate the *width* of the network and the number of layers indicate the *depth*.

Computing the values of the weights and biases is known as *training* a neural network. A trained neural network can then be used to make *predictions* by evaluating the function f given some inputs. To train a neural network, labels (the desired y values) and a lot of data (the inputs) is needed, hence the name *supervised machine learning*. Computing the weights and biases is done in an iterative algorithm called back-propagation [33] in which error gradients are propagated backwards through the network and which are then used to update the weights and biases.

There are two major types of neural networks: *Convolutional Neural Networks (CNN)* and *Recurrent Neural Networks (RNN)*. A CNN is a neural network with layers that apply convolution. This type is often used for classification of images and videos. An RNN is a network in which the neurons in a layer are fed back to the next neuron in the next iteration. This allows the network to maintain state and learn from previously seen data. RNNs are often used for tasks like speech recognition.

2.3.1 Layers

Most neural networks consists of a few type of layers that are used repeatedly. A CNN [34] uses the convolution operation to transform a set of images to another set of smaller images. The operation reuses a small set of weights for each pixel in an image. This sharing of weights reduces the number of parameters that need to be trained to a great extent compared to a fully connected layer like in Figure 2.3b. An interactive visualization of CNNs is available at [62]. Other important layers are layers that add non-linearity and layers that reduce the size of the images.

2.3.2 Activation functions

Activation functions add non-linearity to a neural network. Several functions exist, including:

1. **Step**: returns 1 or 0. Used by perceptrons.

$$f(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases} \tag{2.3}$$

2. **Sigmoid**: converts an output to a probability between 0 and 1.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.4}$$

3. **Softmax**: converts n outputs to n probabilities that add up to 1. It computes the exponential and then normalizes it using the sum of all the n exponentials.

$$f(x) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{2.5}$$

4. **Tanh**: convert an output to a value between -1 and 1.

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.6}$$

5. **ReLU**: clamps values to 0 for $x \leq 0$.

$$f(x) = \max(0, x) \tag{2.7}$$

6. **ELU**: like ReLU, uses the identity function for $x > 0$, but uses a hyperparameter α to control at which negative value the function saturates. Because it saturates at negative values, it can allow faster learning than ReLU [8].

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(\exp(x) - 1) & \text{if } x \leq 0 \end{cases} \tag{2.8}$$

2.3.3 Convolution

Convolution is the process of applying a small vector or matrix known as a *kernel* to each and all or a subset of all pixels (and the surrounding pixels of each pixel) of an image. This is done by multiplying the entries of the kernel with the pixel and its surrounding pixels, summing the values, and replacing the value of the pixel with the computed value. For example, Figure 2.4 shows a 3×3 matrix kernel being multiplied with each submatrix of pixel p_{ij} and its immediate surrounding pixels. This submatrix in the input image is called the *receptive field*.

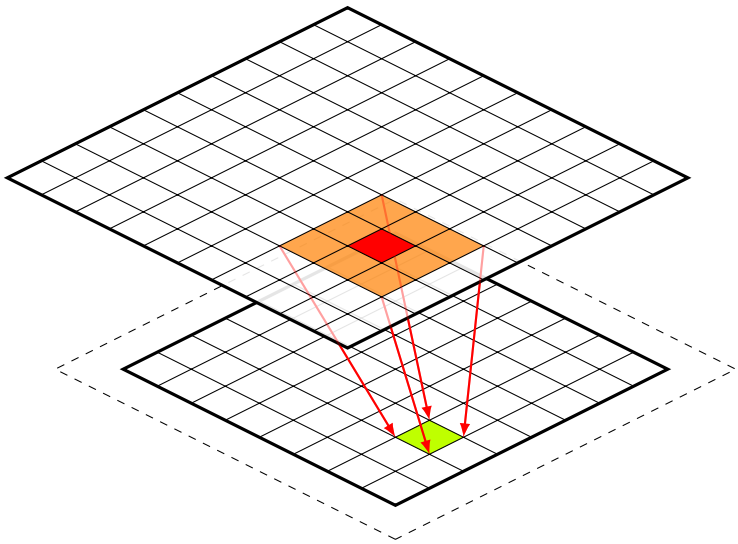


Figure 2.4: *A convolution operation of a* 3×3 *kernel to an image of* 10×10*. No padding was added, thus the output image has a smaller size of* 8×8*. The same kernel (consisting of nine weights) is used for each and all pixels.*

The result of the convolution depends on the entries of the kernel. If each of the nine entries of the 3×3 matrix kernel is 1 / 9, then applying this kernel to the pixels will compute the average of the pixel and the pixels around it. This is known as a *box blur*. Repeating this process will approximate a blur whose kernel has a Gaussian distribution because of the central limit theorem.

Besides the size of a kernel, a stride will specify the distance between pixels that are used when applying the convolution to each pixel. For a kernel of 3×3, a stride of one will select a submatrix of 3×3 pixels, while a stride of two will select a submatrix of 5×5 pixels.

A padding adds a border of pixels to the input image. This can be used to keep the width and height of the output image equal to the input image. The bias adds a learnable bias to the computed pixels.

Another way to perform convolution is to use a 1×1 kernel over the depth of the image (channels or feature maps) instead of the width and height. This is called *depthwise separable convolution* [7, 59].

2.3.4 Downsampling and pooling

With several convolution layers allowing the network to recognize more abstract features, these higher-level features make less stringent demands on the precise location of recognized features than the more primitive learned features. Pooling is a form of subsampling to reduce the spatial resolution of images by reducing the size of the images. This makes the network more robust to translation and distortion of features. Because of the lower resolution of the outputs, the number of neurons in the next layer is sometimes increased by the same amount to keep the computational cost constant. Without it, each successive layer has less and less parameters (the weights and biases).

There are two often used forms of pooling: max pooling and average pooling. Max pooling takes the maximum value of a 2×2 block of pixels. This halves the width and height of each image. Applying convolution with a stride of 2 effectively halves the size as well and can be used to avoid separate pooling layers.

2.3.5 Regularization and batch normalization

Regularization helps to prevent overfitting (optimizing the network for the training data and decreasing the generalization to test data) by driving weights to near or exactly zero. L2 regularization reduces the weights by taking the sum of the square of the weights (Euclidean distance) into account, while L1 uses the sum of the absolute values of the weights (Manhattan distance). Batch normalization [25] (used by network models like ResNet [21]) is another technique to reduce overfitting.

3

Effect of cluster, datasets, and models on hardware utilization

In this chapter the effect of the cluster configuration, datasets, and models on the hardware utilization is examined. A number of experiments will be performed to assess the hardware utilization and determine to which amount various aspects of the configuration have an effect on the utilization. Various datasets and models are used to determine whether the hardware is utilized to different amounts.

This chapter will focus on determining whether and how the performance of distributed training and inference depends on various aspects or that a new tool could give better insight in how effective the hardware is used to perform training and inference.

This chapter is structured as follows: Section 3.1 will state the research questions this chapter will focus on. Section 3.2 explains the set-up of the initial experiments, Section 3.3 lists the experiments and their datasets, models, and variables. Section 3.4 shows the results of the experiments.

3.1 Research questions

This chapter will focus on the following research questions of Section 1.1:

RQ1 Does the hardware utilization depend on the dataset, the model, hyperparameters, or configuration of the system like number of GPUs and used interconnect?

RQ3 How much time is spend on communication between nodes, loading data, updating parameters, and other aspects in popular large-scale machine learning frameworks?

To answer research question **RQ1**, several models will be trained using data parallelism [61] with several datasets, collective communication backends, interconnects, number of nodes, and number of GPUs per node. The wallclock time to train and test, as well as the time to reach a certain accuracy is measured. Due to the very few number of machines with multiple GPUs, it may be difficult to say anything meaningful about the effect of the number of GPUs on the hardware utilization. A significant difference between a pair of experiments in which one variable is changed would indicate a lower utilization of the hardware. However, the absence of any difference does not suggest the hardware is optimally utilized in both experiments.

Performance profiling of the code of the used machine learning framework should show possible areas in the code where the frameworks do not make optimal use of the hardware and help to answer research question **RQ3**. By instrumenting the code and collecting the traces in a file, the time spend in and between functions and be analyzed. Although trace files are collected per process (one process per node), the system time can be recorded in some event as early as possible in the trace file, so that events in different trace files can be synchronized with each other.

3.2 Set-up of the experiments

3.2.1 Hardware used

The experiments are run on machines at the VU site of the DAS-5 [4] cluster. Each machine has two sockets containing an Intel Xeon E5-2630 v3 CPU (Haswell generation) [1] with a total of 16 cores and 32 threads per node. 15 machines have a single NVIDIA GeForce GTX Titan X GPU (Maxwell generation) with 24 SMs (3072 threads) and 12 GiB of VRAM, providing single-precision processing power of about 6 TFLOPS per GPU [2] or 98 TFLOPS in total. Two machines have 3 GPUs each, of which 2 per machine are NVIDIA Titan X GPUs (Pascal generation) with 28 SMs (3584 threads) and 12 GiB of VRAM, providing about 10 TFLOPS of single-precision operations due to the 4 extra SMs and the much higher clock speed. This gives a total of about 40 TFLOPS. A path exists between the two Titan X GPUs via a PCIe host bridge, and a path with the third GPU (which has a different NUMA affinity) via an SMP interconnect. The two Titan X GPUs are not connected through NVLink according to the `nvidia-smi` application.

Each node in the cluster at the VU site is connected to the other nodes through multiple interconnects. The network interface *ib0* on a node provides access through the InfiniBand (at a speed of 56 Gbit/s) interconnect and *eth0* through the Ethernet (at a speed of 1 Gbit/s) interconnect. The collective communication backend Gloo of PyTorch supports only Ethernet, while NCCL supports Ethernet and InfiniBand. Thus some of the experiments can only be performed using the Ethernet interconnect.

Storage for the datasets is provided on SATA or SAS drives via an LSI RAID storage controller and accessible on the compute nodes over the network through NFS via InfiniBand.

3.2.2 Software used

The experiments will be performed with a popular machine learning framework called PyTorch. Since the experiments are focused on distributed training with GPUs, all of which are from NVIDIA, the machine learning framework will make use of several software technologies for GPUs of this vendor. These include CUDA, a platform and API that allows one to write algorithms that will run on the GPU, and NCCL [26], a collective communication library that supports multiple interconnect to exchange data between multiple GPUs and multiple nodes.

PyTorch makes use of CUDA through `.cu` files, which are compiled by CUDA's `nvcc` compiler, in which C++ code that will run on the CPU and kernel code that will run on the parallel threads of the GPU can be mixed in a single file. The `nvcc` compiler will generate instructions or CUDA library calls for copying data between host and device memory and launching kernels on the GPU.

Several backends for collective communications are available in PyTorch[3]. These include Gloo, MPI, and NCCL. The MPI backend provides functions for CPU tensors only, and thus will not be used in the experiments. The documentation of PyTorch mentions that Gloo should run slower than NCCL when performing distributed training on GPUs, but does not mention why. The experiments in Section 3.3 should show how much slower Gloo is relative to NCCL and whether there are any limits to the scalability of training when using the Gloo backend. NCCL does not support any functions for CPU tensors, while Gloo supports almost all functions for tensors on the CPU and a few for tensors on the GPU.

Data parallel training on multiple nodes and GPUs is enabled through the `DistributedDataParallel` class (DDP) in the `torch.nn.parallel` module of PyTorch. DDP uses the *allreduce* function, which is supported by NCCL and is also one of the few functions supported by Gloo.

Performance profiling is done with Perfetto [16], which provides an SDK for instrumenting C++ code. After the trace has been written to a `.pftrace` file, the file can be loaded into the

[1] https://ark.intel.com/content/www/us/en/ark/products/83356/intel-xeon-processor-e5-2630-v3-20m-cache-2-40-g
html
[2] https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units
[3] https://pytorch.org/docs/stable/distributed.html

Perfetto UI[4] web application for analysis.

3.2.3 Execution on the cluster

Experiments are scheduled using the SLURM job scheduler [57]. Each `.job` file contains several `#SBATCH` instructions to allocate the required resources, set environment variables, and then run a shell script on the compute node to launch the machine learning framework which then performs training and testing of a model. For experiments that use NCCL, various `NCCL_*` environment variables are set. The shell script is executed by `srun` and, in the case of PyTorch, the actual Python script is then executed via the Python module `torch.distributed.launch` with one process per node. PyTorch will use Python's `multiprocessing` module to load the dataset using a configurable number of child processes (the Python script that does the actual training and testing uses a default of 8).

The power draw, temperature, and memory and SM utilization of the GPUs is measured with the `nvidia-smi` utility.

3.3 Experiments

3.3.1 Metrics

For all experiments, the following metrics are collected on each node and per epoch:

- Wallclock time time to train the DDP model and then run the forward and backward pass.
- Wallclock time to evaluate the trained model.
- Accuracy obtained after evaluating the model.

The metrics are printed to standard output on each node and collected to a single `.out` file by the Slurm system. Afterwards the `.out` files are processed and grouped into folders based on the communication backend, interconnect, dataset, and model. The first 5 epochs are used to warm-up the nodes and are not used in the figures that show the boxplots.

Memory and SM utilization of the GPUs is measured once per second using the `nvidia-smi` tool.

3.3.2 Datasets

All experiments are run with a small set of well-known of datasets. These include: CIFAR10 [31], ImageNet [12], and QMNIST [63]. All datasets are stored in a folder in `/var/scratch/` that is accessible over the network. The datasets are loaded by PyTorch's `CIFAR10`, `ImageFolder`, and `QMNIST` classes in the `torchvision.datasets` module. The ImageNet images were downloaded using the ImageNet-Datasets-Downloader[5] repository on GitHub.

3.3.3 Models and hyperparameters

For the experiments in this chapter, a ResNet [21] model is used. Due to time constraints imposed by the DAS-5 cluster, a ResNet-18 model, an instance of ResNet with just 18 layers, is used for most of the experiments. All models are represented by the corresponding classes in PyTorch's `torchvision.models` module.

Each dataset used during various experiments uses specific values for certain hyperparameters. These hyperparameters include the number of epochs, which controls for how many iterations the model is trained and evaluated, the batch size, which is used by the `DataLoader` from the `torch.utils.data` module, and the momentum, learning rate, and weight decay, which are used by the `SGD` optimizer from the `torch.optim` module. The specific values that have been used are shown in Table 3.1.

[4] https://ui.perfetto.dev/
[5] https://github.com/mf1024/ImageNet-Datasets-Downloader

Table 3.1: *Hyperparameters used by data loader and SGD optimizer during the experiments with various datasets.*

	Dataset		
Hyperparameter	CIFAR-10	ImageNet	QMNIST
Epochs	80	80	20
Batch size (train)	1024	1024	64
Batch size (test)	32	128	64
Learning rate	0.1	0.1	0.1
Momentum	0.9	0.9	0.9
Weight decay	1e-5	1e-5	1e-5

Some of the experiments use different batch sizes for the training phase. These include the sizes 256, 512, 1024, 2048, and 4096.

3.3.4 Experiments

The following experiments have been performed:

- Comparing the collective communication backends Gloo and NCCL
  - On between 1 and 15 nodes with 1 GPU on each node. This experiment should show how well each communication backend scales as a function of the number of nodes and the difference in utilization between the backends.
  - On 1 and 2 nodes with 1 or 2 GPUs on each node. This experiment should show if there is any difference between the backends in a set-up with n nodes with k GPUs each. This experiment is of little value because the maximum n is very small.
- Comparing InfiniBand and Ethernet interconnects
  - On between 1 and 15 nodes with 1 GPU each.
  - On nodes with a single GPU and nodes with multiple GPUs.
- Comparing different batch sizes for the training phase

Because the Gloo backend does not support InfiniBand, the experiments that involve this backend are all run over an Ethernet interconnect. Experiments that compare InfiniBand with Ethernet are run using the NCCL backend.

3.4 Results

3.4.1 Collective communication backends

Single-GPU multi-node set-up

Figures 3.1 and 3.2 shows the average wallclock time needed to test and train a ResNet-18 model on a CIFAR-10 dataset with PyTorch on between 1 and 15 nodes with 1 GPU each using `DistributedDataParallel` from the `torch.nn.parallel` module.

The top row of each figure shows the time needed to evaluate the trained model while the bottom row shows the time needed to train the model. A column represents an experiment with a given number of nodes (between 1 and 15). Each cell shows one or multiple boxplots, one for each node. A boxplot displays the median, quantiles, and outliers of the measured wallclock times of the 75 epochs. The boxplots are sorted by the mean of each plot. Above each cell the average mean and the average standard deviation of the boxplots is displayed, as well as the number nodes, total number of GPUs, and epochs. The number of nodes and GPUs above each column is equal since 1 GPU per node was used.

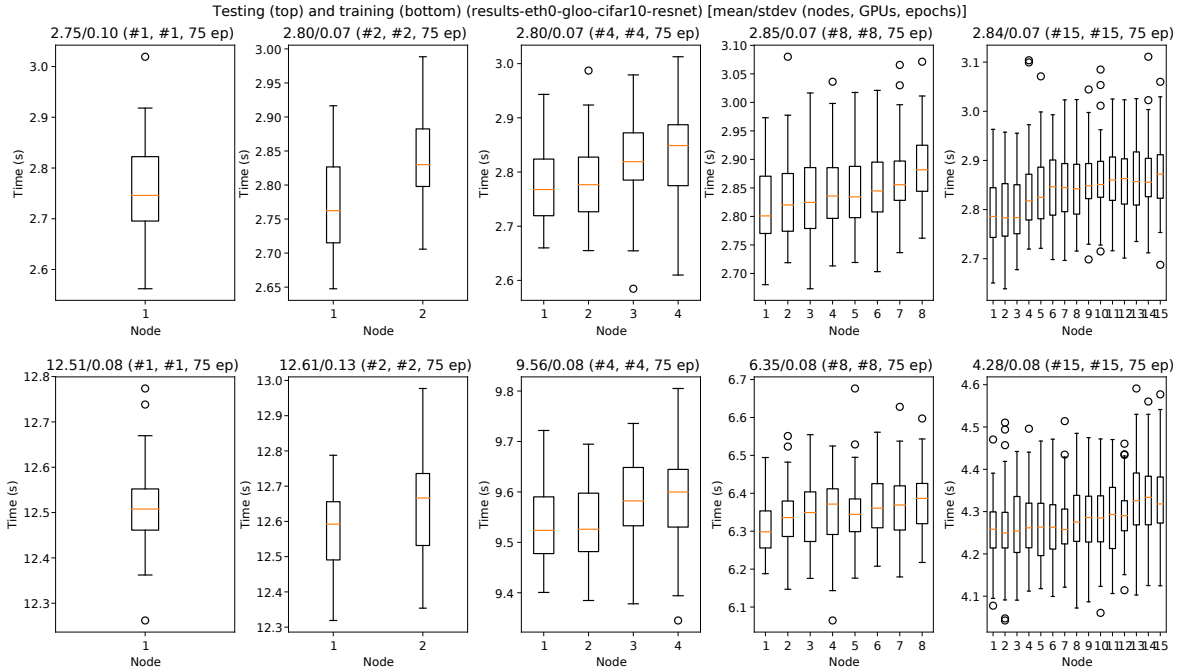


Figure 3.1: *Wallclock time for testing and training when using the Gloo communication backend, on nodes with 1 GPU per node, Ethernet interconnect, ResNet-18 model, and CIFAR-10 dataset.*

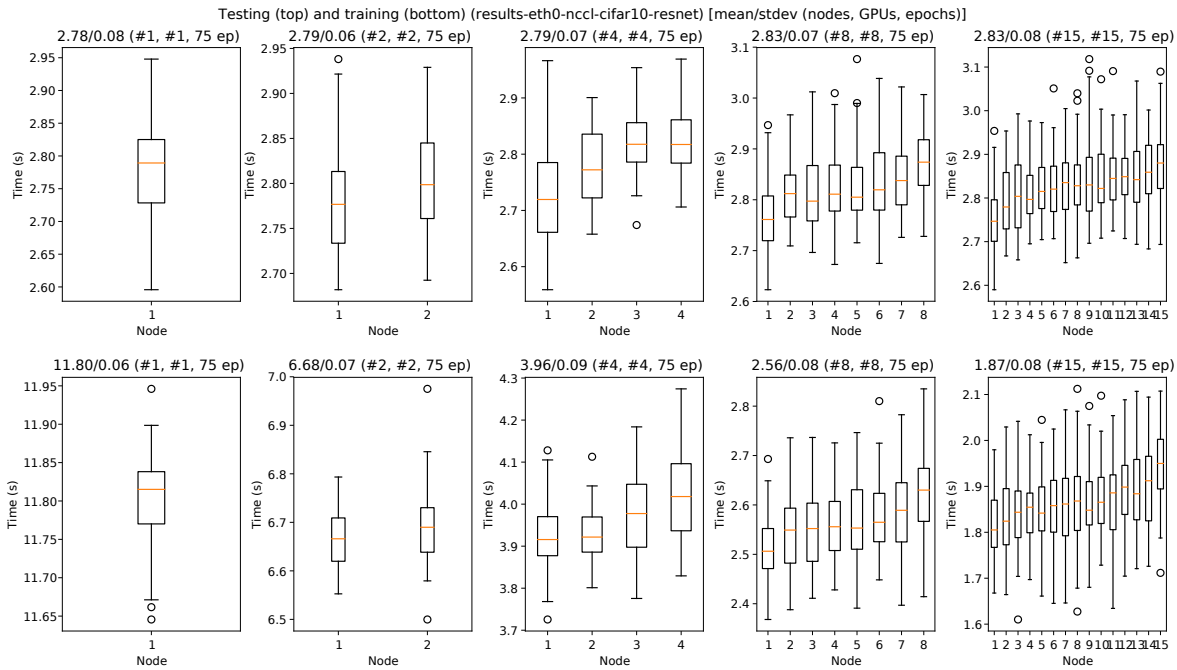


Figure 3.2: *Wallclock time for testing and training when using the NCCL communication backend, on nodes with 1 GPU per node, Ethernet interconnect, ResNet-18 model, and CIFAR-10 dataset.*

Comparing the average mean of the top rows of these two figures shows that there is little difference between the Gloo and NCCL backends during testing. Both figures show that roughly 2.8 seconds is needed irrespective of the number of nodes or the communication backend used. The bottom rows show the time needed for training. A large difference between the two communication backends can be seen in the figures. While there is negligible

Table 3.2: *Speed-up of training time in Figures 3.1 and 3.2 compared to training on a single node. Each node has 1 GPU and communication happens over the Ethernet interconnect. The percentages indicate the percentage of the maximum speed-up (based on the total number of nodes). Closer to 100% is better.*

Nodes	NCCL		Gloo	
1	1.00	(100%)	1.00	(100%)
2	1.77	(89%)	0.99	(50%)
4	2.98	(75%)	1.31	(33%)
8	4.61	(58%)	1.96	(25%)
15	6.30	(42%)	2.91	(19%)

difference when training occurs on a single node (training takes about 12 seconds on a single node), with multiple nodes the wallclock time decreases much more as the number of nodes used increases when the NCCL library is used than when the Gloo backend is used. Table 3.2 shows the speed-up, compared to the case with one node, of the average training time over all the nodes used in an experiment when using NCCL and Gloo as a communication backend. The table clearly shows that the NCCL backend scales much better with the number of nodes than Gloo, giving speedups closer to the theoretical maximum. On 8 nodes, the average wallclock time needed to train the model during one epoch using NCCL is ≈ 21.7% of the time needed on a single node (ideally ≈ 12.5%), while Gloo requires ≈ 51.0% (1.0/1.96) of the time of a single node for the case with 8 nodes. For the case with 2 nodes, Gloo is actually no faster than training on a single node. These experiments were run with a fixed batch size, and a greater speed-up for synchronous SGD might be possible when increasing the batch size as a function of the number of nodes due to less collective communication operations, according to [27, 65].

Figure 3.3 shows the accuracy as a function of the time when using Gloo and NCCL with the CIFAR-10 dataset. Interestingly, the accuracy converges slower when training with 15 nodes than with 8 nodes when using the NCCL backend while this behavior does not occur with the Gloo backend. For the Gloo backend, the accuracy had a few sharp drops when training on 8 nodes (these drops are smoothed in the figure because of the median filtering that has been applied). This behavior does not seems to occur for the ImageNet dataset, as shown in Figure A.2. Comparing Figures 3.3 and 3.4 shows that the accuracy decreases as the number of nodes increases for the CIFAR-10 dataset, while for the ImageNet dataset this effect is much weaker and the accuracy is generally greater than for the case of a single node. Also, for the CIFAR-10 dataset, the accuracy obtained with 8 or 15 nodes is no better than for the case of a single node.

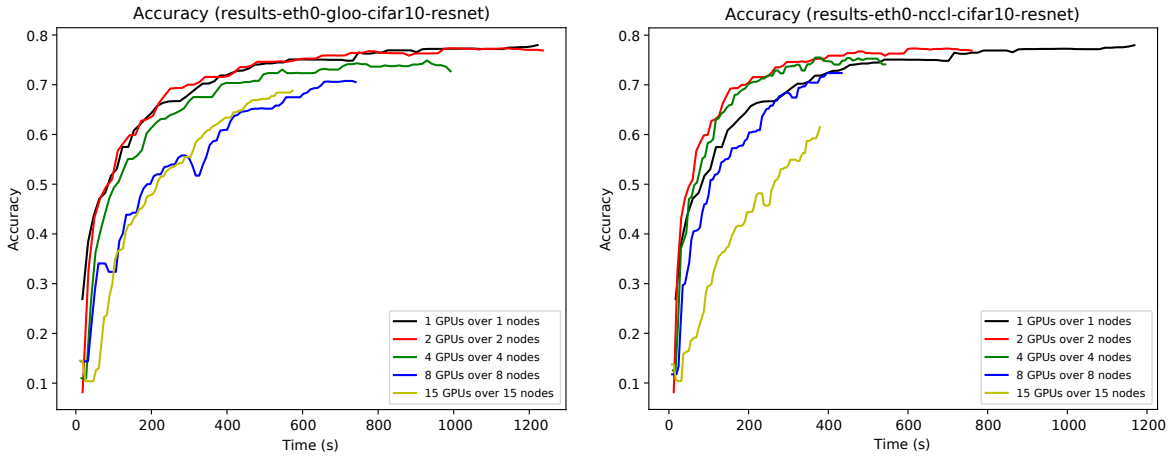
Figures showing the accuracy as a function of the epoch for the CIFAR-10 and ImageNet datasets can be found in Appendix A.1.

Multi-GPU multi-node set-up

Figures 3.5 and 3.6 shows the training time when using Gloo and NCCL on nodes with a single or two GPUs with the CIFAR-10 dataset. (Results for ImageNet can be found in Figure A.5) The first column in both figures shows one node with one GPU. The second and third columns show the experiments with two GPUs in total, divided over one (second column) or two nodes (third column). The fourth and last column shows four GPUs in total, divided over two nodes.

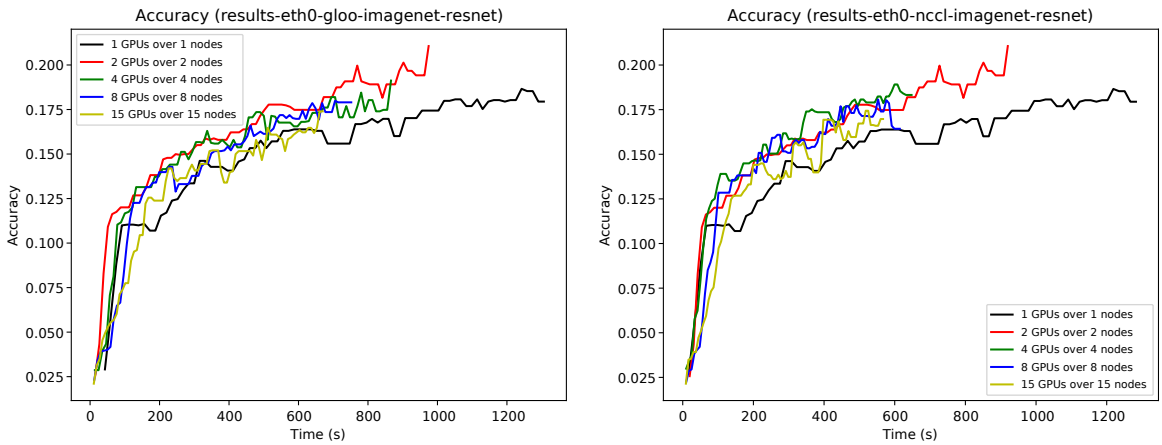
Figure 3.6 shows that NCCL scales well with respect to the total number of GPUs that were used with both CIFAR-10 and ImageNet datasets. The average training time shown above the cells in the bottom row show correlation with the total number of GPUs that were used; it decreases when more GPUs are used. In particular the training time of the second (4.37 seconds) and third column (4.65 seconds) is nearly equal, with the time of the third column slightly higher, most likely due to communication between the two nodes.

Gloo, shown in Figure 3.5, however, does not seems to scale with respect to the number of nodes available. Lower or equal training times are achieved on a single node with two GPUs (5.14 seconds) than on two nodes with one GPU each (12.27 seconds). This shows that



(a) *Gloo communication backend.* **(b)** *NCCL communication backend.*

Figure 3.3: *Accuracy as a function of time on 1, 2, 4, 8, or 15 nodes, with 1 GPU per node, using the Ethernet interconnect, and ResNet-18 model with CIFAR-10 dataset. The accuracy as a function of the epoch is shown in Figure A.1.*



(a) *Gloo communication backend.* **(b)** *NCCL communication backend.*

Figure 3.4: *Accuracy as a function of time on 1, 2, 4, 8, or 15 nodes, with 1 GPU per node, using the Ethernet interconnect, and ResNet-18 model with ImageNet dataset.*

communication between nodes adversely affects the training times with Gloo.

Table 3.3 shows the speed-ups that were achieved. Comparing the values on the second row (doubling the number of GPUs on one node), the two datasets gave similar speed-ups between the two communication backends.

For Gloo, the training times decrease as the number of GPUs per node increases, but the training time takes a severe hit when increasing the number of nodes with the CIFAR-10 dataset. The two red percentages in the third and fourth row are lower than 50% and 25%, respectively. This shows that more time was needed than for the case of one GPU on a single node. For NCCL this difference in speed-up was not observed when increasing the number of nodes used.

Figure 3.7 clearly shows a higher accuracy on the CIFAR-10 dataset is achieved more quickly with the Gloo backend when there is no communication between different nodes. Using the NCCL does not appear have a detrimental effect (compared to the case of a single node with a single GPU) on the time needed to achieve a certain accuracy as the number of nodes and GPUs increase. With the ImageNet dataset, shown in Figure 3.8, the difference between two GPUs on a single node or on two nodes is much smaller when using the Gloo

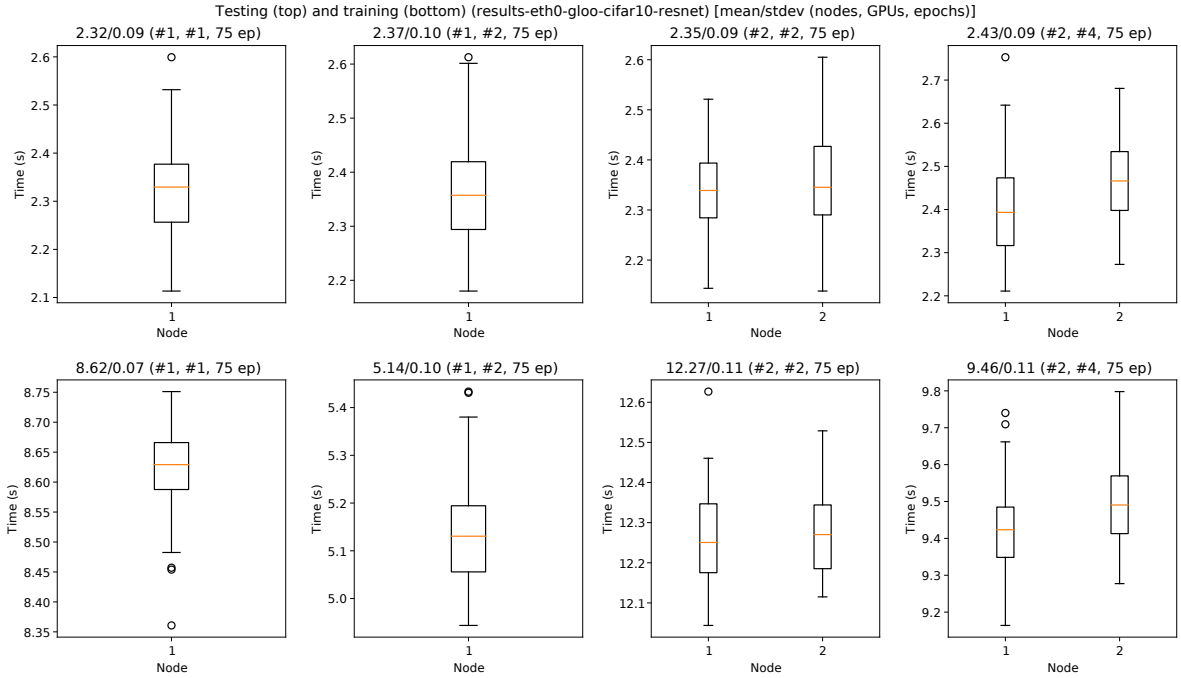


Figure 3.5: *Wallclock time for testing and training on 1 and 2 nodes when using the Gloo communication backend, with 1 and 2 GPUs per node, using an Ethernet interconnect, and ResNet-18 model with CIFAR-10 dataset.*

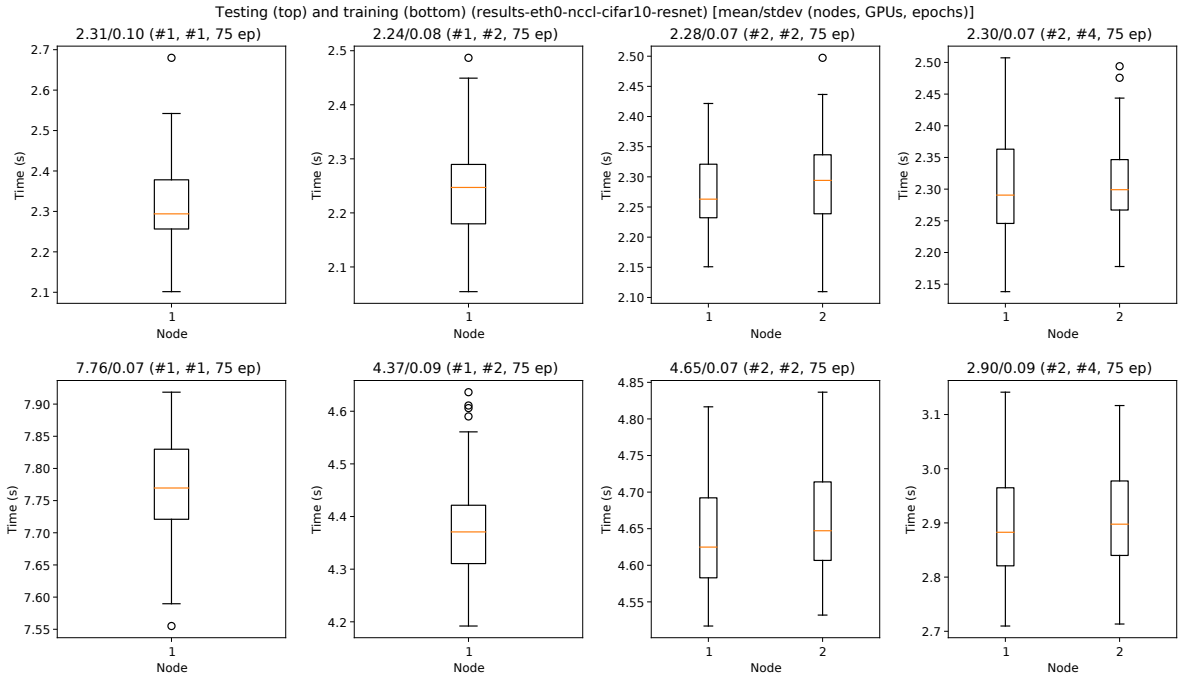
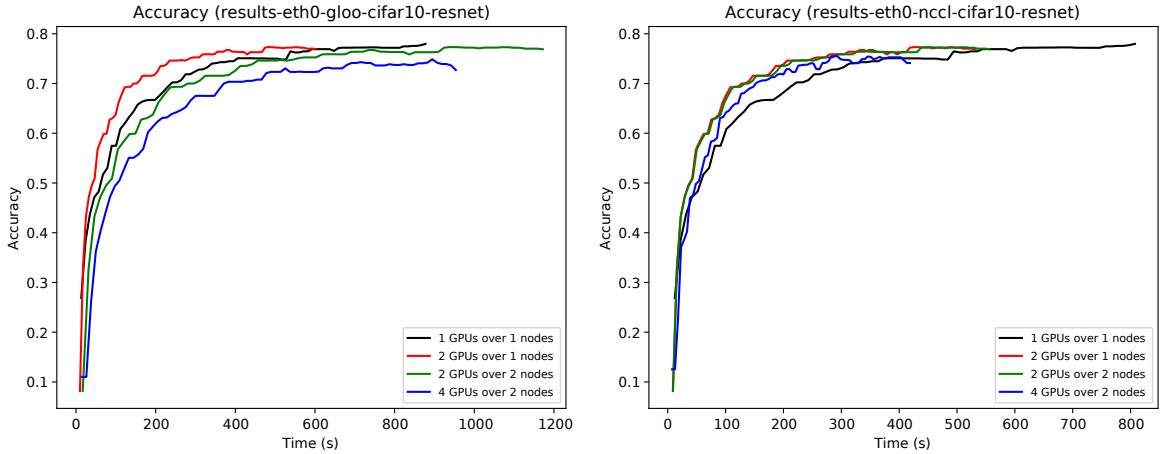


Figure 3.6: *Wallclock time for testing and training on 1 and 2 nodes when using the NCCL communication backend, with 1 and 2 GPUs per node, using an Ethernet interconnect, and ResNet-18 model with CIFAR-10 dataset.*

backend. However, the accuracy for four GPUs (over two nodes) is still smaller relative to the experiment with two GPUs (over one or two nodes), while the NCCL communication backend shows a higher accuracy in this case.

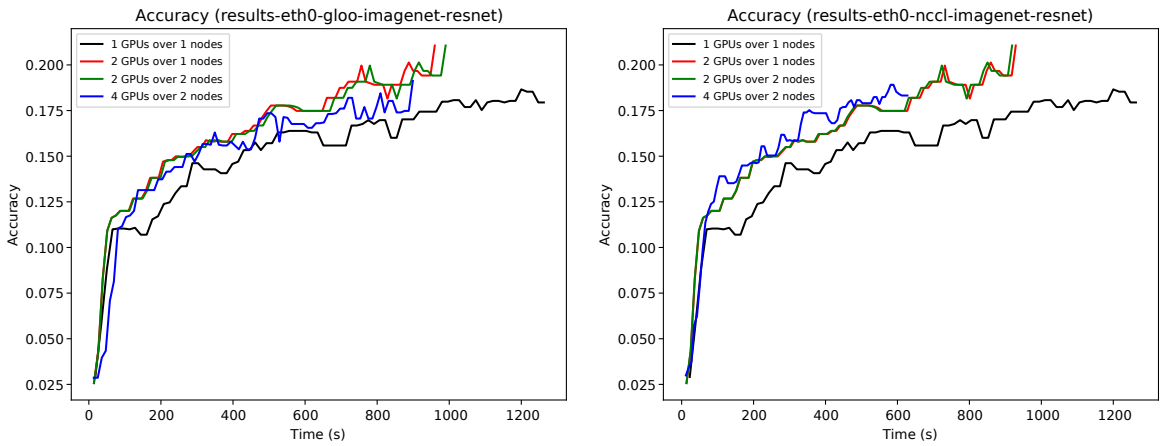
Table 3.3: *Speed-up of training time in Figures 3.5, 3.6 and A.5 compared to training on a single node with a single GPU. Each node has 1 or 2 GPUs and communication happens over the Ethernet interconnect. The column with the number of GPUs shows the total number of GPUs in the cluster used during training. The percentages indicate the percentage of the maximum speed-up (based on the total number of GPUs). Closer to 100% is better.*

		NCCL				Gloo			
Nodes	GPUs	CIFAR-10		ImageNet		CIFAR-10		ImageNet	
1	1	1.00	(100%)	1.00	(100%)	1.00	(100%)	1.00	(100%)
1	2	1.78	(89%)	1.44	(72%)	1.68	(84%)	1.40	(70%)
2	2	1.67	(84%)	1.46	(73%)	0.70	(35%)	1.33	(67%)
2	4	2.68	(67%)	2.41	(60%)	0.91	(23%)	1.53	(38%)



(a) *Gloo communication backend.* **(b)** *NCCL communication backend.*

Figure 3.7: *Accuracy as a function of time on 1 and 2 nodes, with 1 and 2 GPUs per node, using the Ethernet interconnect, and ResNet-18 model with CIFAR-10 dataset. The accuracy as a function of epoch is shown in Figure A.3.*



(a) *Gloo communication backend.* **(b)** *NCCL communication backend.*

Figure 3.8: *Accuracy as a function of time on 1 and 2 nodes, with 1 and 2 GPUs per node, using the Ethernet interconnect, and ResNet-18 model with ImageNet dataset. The accuracy as a function of epoch is shown in Figure A.4.*

3.4.2 Interconnects

Single-GPU multi-node set-up

Figure A.6 shows the wallclock time needed for testing and training a ResNet-18 model with a CIFAR-10 dataset when communicating over Ethernet and InfiniBand. Because Gloo does not support InfiniBand, NCCL is used with both interconnects. To force NCCL to use Ethernet for communication, the environment variables `NCCL_IB_DISABLE=1` and `NCCL_P2P_DISABLE=1` are used in the SLURM job scripts to prevent NCCL from detecting and switching to InfiniBand.

Comparing the top rows of the two subfigures shows there is no difference between the two interconnects when testing the model, while training the model over InfiniBand is between 1 and 3 % faster than over Ethernet depending on how many nodes take part in the training.

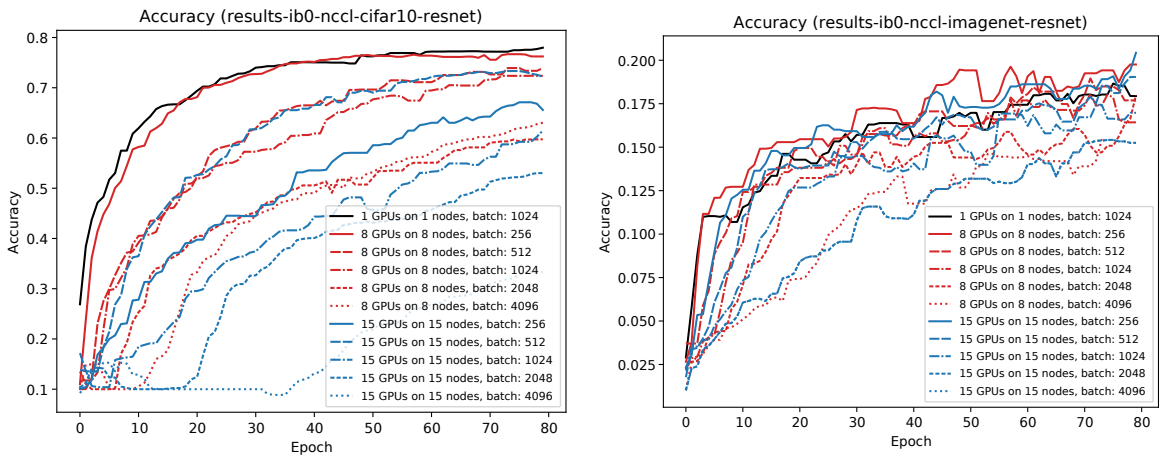
Figures showing the accuracy as a function of the elapsed time for the CIFAR-10 and ImageNet datasets when communicating over Ethernet and InfiniBand using the NCCL communication backend can be found in Appendix A.3.

Multi-GPU multi-node set-up

Figures A.9 and A.10 show that training an ImageNet or CIFAR-10 dataset with NCCL on both types of interconnect achieves a given accuracy in an almost similar time.

3.4.3 Batch sizes

Previous experiments showed that a lower accuracy was obtained as a model was trained on more GPUs. According to [65] a higher batch size would make more efficient use of the available hardware and suggests the batch size needs to be increased for each epoch. However, in PyTorch a `DataLoader` object is constructed given a fixed batch size and then iterated over for each epoch. Therefore, each experiment will use a fixed batch size for the data used to train the models for all epochs. The previous experiments all used a batch size of 1024 for the CIFAR-10 and ImageNet datasets (see Table 3.1), but in the following experiments sizes of 256, 512, 1024, 2048, and 4096 are used.

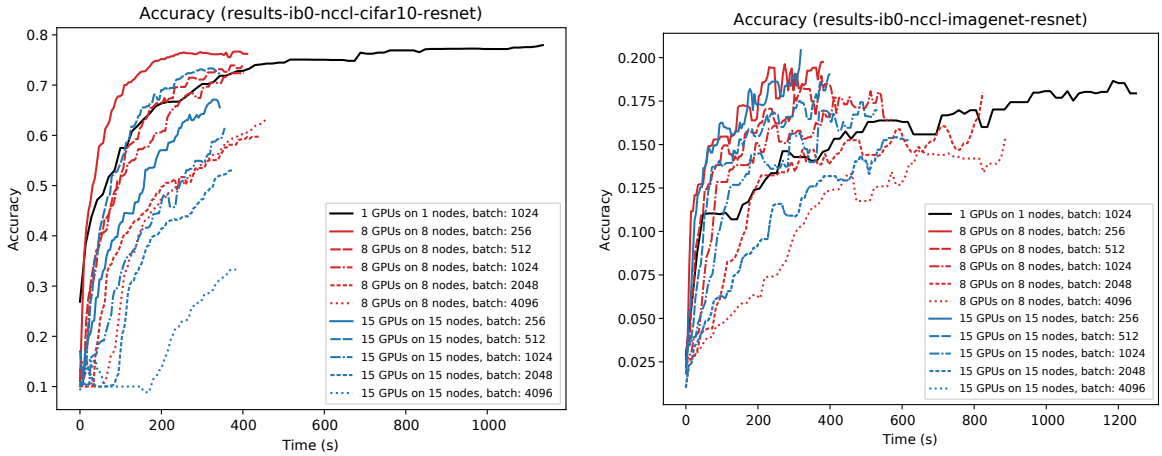


(a) *CIFAR-10 dataset.*

(b) *ImageNet dataset.*

Figure 3.9: *Accuracy as a function of the epoch on 1, 8, and 15 nodes, with a single GPU per node, using NCCL communication backend and the Ethernet interconnect, and a ResNet-18 model.*

Figures 3.9 and 3.10 show the accuracy for the CIFAR-10 and ImageNet datasets during training on 1, 8, or 15 nodes with a single GPU each. The line graphs for the experiments with 8 and 15 nodes in Figure 3.10 end much earlier than the one with a single node because in all cases an equal number of epochs were used. For both datasets, a lower batch size during training results in a higher accuracy as a function of the epoch and elapsed time.



(a) *CIFAR-10 dataset.*

(b) *ImageNet dataset.*

Figure 3.10: *Accuracy as a function of the elapsed time on 1, 8, and 15 nodes, with a single GPU per node, using NCCL communication backend and the Ethernet interconnect, and a ResNet-18 model.*

Interestingly, for the CIFAR-10 dataset, a better accuracy was obtained during the experiments with 15 nodes for a size of 512 compared to 256, but any size higher than 512 would result in a worse accuracy. This was not observed with the ImageNet dataset or with any of the experiments with 8 nodes.

While [65] uses ever increasing batch sizes for a fixed number of processors, the experiments performed on the DAS-5 showed that an increased batch size was detrimental for the obtained accuracy when training on an increasing amount of GPUs. [64] shows that a different training algorithm than synchronous SGD is needed to avoid achieving a lower accuracy as the batch size increases. The results in this section do not provide any clue to an optimal size for the given hardware and number of nodes.

3.5 Performance profiling

Instrumenting the PyTorch code base with Perfetto proved to take too much time because of the amount of code in PyTorch that needed to be patched. Additionally, functions from Perfetto's SDK that provide objects with RAII semantics would not correctly register their finalization in the trace file. To work around this issue, all exit points of an instrumented PyTorch function (the lines in the source code where a function returns) would require manually recording the exit in the trace file. The training and testing phase were recognizable in the trace viewer tool, Perfetto UI, but the trace files still showed a lot of empty gaps, caused by missing instrumentation. Profiling tools like py-spy[6] were able to visualize the loading of data with Python's `multiprocessing` package, but did not visualize the training and testing phases of PyTorch. Furthermore, these tools can only identify the utilization of the CPU, but not of the GPUs themselves. Chapter 4 will show that there are other ways to collect these metrics.

3.6 Discussion

Section 3.4.1 shows that NCCL achieves better training times than Gloo when scaling the number of nodes or GPUs. NCCL also shows little or no difference when using a number of GPUs on less or more nodes, while in the case of Gloo, it is preferred to have a higher GPUs per node ratio. For both collective communication backends, the accuracy seems to decrease when training on more GPUs. This requires one to repeatedly adjust the batch size to improve the accuracy. The figures provide no insight into why different datasets like

[6] https://github.com/benfred/py-spy

CIFAR-10 and ImageNet behave differently. No difference was observed between the Ethernet and InfiniBand interconnects when using the NCCL communication backend with the used datasets. The `nvidia-smi` tool was of little use due to only being able to report very few metrics and at a low resolution (no more than once per second).

The experiments performed in this chapter partially answer research question **RQ1**; the hardware utilization seems to depend on the used dataset and configuration of the system, including the number of nodes and number of GPUs per node, and the used collective communication backend. The results, however, do not explain why certain behavior, like the values shown in Table 3.3, was observed and what measures can be taken to reduce the time needed to train and test the used models. Successful instrumentation of the code would have helped answering research question **RQ3**, and in turn provide insight into the measures that can be taken to reduce the time spend.

4

Monitoring techniques

Chapter 3 showed that various aspects like the used communication backend and the number of nodes had a significant effect on the time needed for training and inference. However, no insight was gained into possible causes for the shown behavior. Instrumentation could have helped but would require extensive modifications to the code to be effective.

In this chapter various techniques are investigated that may help to monitor the utilization of the system. These techniques may then be used as part of a new tool that helps the user gaining insight into how the system is performing.

This chapter is structured as follows: Section 4.1 describes the research questions this chapter will focus on. Section 4.2 lists the various tools that are available to perform monitoring. Section 4.3 lists the metrics that can be gathered using these tools. Section 4.4 provides a discussion of the usefulness of these metrics.

4.1 Research questions

This chapter will focus on the following research questions of Section 1.1:

RQ2 Which parts of the system can be measured and visualized?

To answer research question **RQ2**, various tools and libraries available on the system will be investigated and checked whether they are suitable to be used for measuring how much and when certain parts of the system are utilized.

Some of the tools and libraries that appear to be suitable are then investigated in depth to see whether they impose any limitations on the capabilities of the tool developed in the next chapter.

4.2 Tools and libraries

Before a list of potentially suitable tools can be gathered, it is necessary to determine which parts of the system can be measured. Two major aspects of the system need to be measured:

1. **(Single-node)** Utilization of the hardware of a node. This includes the kernels launched on the GPU as well as the interaction between the CPU and GPU.

2. **(Multi-node)** Interaction, including the communication and exchange of data, between nodes over the network.

See Figure 4.1 for an overview of the DAS-5 cluster. Each node has one or more CPU sockets and one or two GPUs. The master node has a RAID storage controller with multiple disks attached. Because of the storage controller, the disks are visible to the system as a single disk of tens of terabytes, hiding the throughput and latency of the individual disks. Its

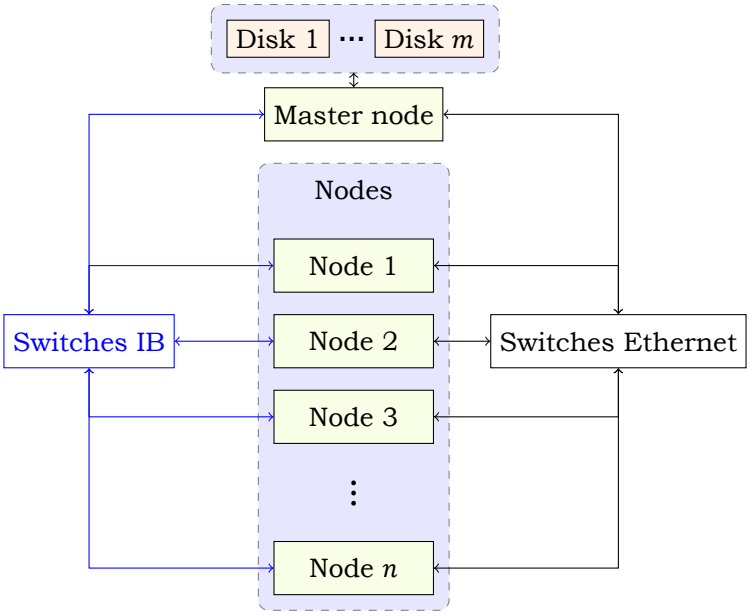


Figure 4.1: *A schematic overview of the DAS-5 cluster.*

storage is accessible to the compute nodes through the NFS filesystem over the InfiniBand interconnect. Each compute node also has a small RAID0 array for local storage. The nodes are connected to one of several InfiniBand switches, which form a star topology.

Table 4.1 shows the tools that were investigated for use on the DAS-5 cluster. Elevated privileges means that, in order to use the tool, the user invoking the tool needs to have special Linux kernel capabilities like `CAP_SYS_ADMIN`, which is an overloaded capability, unlikely to be granted by the system administrators of a cluster. Instrumentation means that the code of the machine learning framework must be modified. Most of the tools found are focused on giving insight into the utilization of the hardware of a single node.

Table 4.1: *A list of tools investigated for use on the DAS-5 cluster. Elevated privileges means that the tool requires root access or certain Linux capabilities (● = for most/all features, ◑ = for some features). Code means that either instrumentation of the C++ code is needed or Python bindings must be used (● = Python bindings/code, ◑ = thin Python bindings, ◑ = some C/C++ code required or feasible to create thin Python bindings, ● = requires C/C++ code). Type refers to the aspects mentioned in Section 4.2.*

Name	Elevated privs.	Code	Measures	Type
`nvidia-smi`	○	○	GPU device	1
NVML	○	◑	GPU device	1
Nsight Compute	○	○	GPU kernel	1
Nsight Systems	○	○	CUDA, GPU kernel	1
`perf`	●	○	Application	1
`perf_event_open`	◑	●	Application	1
eBPF	●	◑	System	1, 2
psutil	○	●	System	1, 2
Perfetto	○	●	Application	1
CUPTI	○	◑	CUDA, GPU kernel	1
Netintercept	○	○	Network traffic	2
sysfs	○	○	Network statistics	2

4.2.1 nvidia-smi

`nvidia-smi` is a utility that can be used to periodically poll the selected GPU for the following metrics:

- SM and memory utilization in percentage.
- Processor and memory clock speed.
- Temperature and power usage and limits.
- PCIe read and transfer speed in MB/s.

These metrics can be queried by executing `nvidia-smi -q -x` to produce XML output containing all metrics or a single or few metrics can be queried with `nvidia-smi -query-gpu=<metrics> -format=csv`.

4.2.2 NVML

The NVIDIA Management Library (NVML) [45] (via `nvidia-ml-py` on PyPI) is a library which can be used to programmatically query information about the GPU. It provides access to the same information as the `nvidia-smi` utility since that utility uses NVML internally. It is also used by the Slurm resource manager [57, 58].

4.2.3 Nsight Compute

NVIDIA's Nsight Compute [42] is an application that is used to profile the kernels launched by an application and gather a detailed set of metrics of up to over 160 metrics. To gather the required metrics, kernels need to be replayed individually or by restarting the whole application. The latter is not feasible when training for many epochs. Even with only a handful of metrics selected, profiling using the `ncu` application causes a tremendous slowdown, unless profiling is limited to only specific CUDA kernels or invocations.

Metrics are grouped into several sections, see Table 4.2. Sections or individual metrics can be selected for profiling. The metrics of the section LaunchStats provide similar information as the metrics provided by Nsight Systems, discussed in Section 4.2.4.

Table 4.2: *A list of sections into which metrics gathered by Nsight Compute are grouped. A detailed set can be selected and consists of all sections mentioned below. A full set includes MemoryWorkloadAnalysis_Chart and MemoryWorkloadAnalysis_Tables as well.*

Default	Name	Notable metrics
	ComputeWorkloadAnalysis	Utilization SM
	InstructionStats	Executed/issued instructions
✓	LaunchStats	Block/grid size, threads, waves/SM
	MemoryWorkloadAnalysis	Mem. throughput/bandwidth, L1/L2 hit rate
✓	Occupancy	Achieved occupancy, active warps/SM
	SchedulerStats	
	SourceCounters	
✓	SpeedOfLight	Utilization DRAM, L1, L2, SM
	SpeedOfLight_RooflineChart	
	WarpStateStats	Active threads/warp

4.2.4 Nsight Systems

NVIDIA's Nsight Systems [43] is an application to gather traces of CUDA API calls, execution of kernels, and memory operations. Because no detailed metrics per launched kernel are gathered, the overhead of profiling an application using the `nsys` application is much less

than the overhead created by the `ncu` application. On the other hand, only a small number of metrics are available:

- Grid and block size
- Registers per thread
- Memory transferred to/from the device
- Start and end time of API calls, kernels, and memory operations

Some of these metrics are similar to those of the section LaunchStats of Nsight Compute. See Table 4.2.

4.2.5 perf

The `perf` program can provide detailed statistics of the CPU such as branch misses, cache misses, as well as profiling function calls and syscalls of the kernel. It is mainly useful for performance profiling of applications running on the CPU, but not on the GPU. System-wide profiling using `perf` requires elevated privileges, which makes it largely unusable for regular users.

4.2.6 perf_event_open

The Linux system call `perf_event_open`[1] allows an application to set up performance monitoring. This can be used to measure events provided by the kernel such as context switches and page faults, kernel tracepoints, and hardware performance counters to measure the number of hardware instructions, cache usage, and cache and branch misses. Some features require kernel capabilities like `CAP_SYS_ADMIN` or `CAP_PERFMON` (Linux 5.8 or higher).

4.2.7 eBPF

The Linux system call `bpf` can be used to create and load eBPF programs into the kernel [9, 18]. This requires the system to have a recent Linux kernel (4.1 or later, and 4.9 or later for stacktraces) and the required kernel capabilities `CAP_SYS_ADMIN` or `CAP_BPF` and `CAP_NET_ADMIN` in recent versions of the Linux kernel. eBPF programs are able to trace various things, including operations on filesystems like NFS, device I/O, and network traffic. Programs are written in a restricted subset of C and then compiled to eBPF bytecode. The BCC toolkit [9] provides Python bindings to compile and load these eBPF programs and to collect the results.

4.2.8 psutil

psutil [47] is a Python library to query the state and utilization of the system. This includes the CPU utilization, used memory, sensors, running processes, and disks and network I/O counters. psutil uses the proc and sysfs filesystems mounted to `/proc` and `/sys`.

4.2.9 Perfetto

Perfetto [16] is an SDK that can be used to profile C++ code by instrumenting it. Performance profiling by Perfetto was found to be inadequate due to the large amount of code that needed to be instrumented. Trace files could be loaded in the trace viewer tool, Perfetto UI, but would show too many gaps (missing function calls) to be useful. Section 3.5 discussed this in more detail.

[1] https://man7.org/linux/man-pages/man2/perf_event_open.2.html

4.2.10 CUPTI

The NVIDIA CUPTI [44] library is a library that provides a C API to perform tracing (similar to `nsys`) and profiling (similar to `ncu`). It provides APIs which can be used to collect a large number of metrics. There appears to be no official Python bindings, which means that these need to be created or a machine learning framework like PyTorch must be modified in order to use its C-based API. This is in contrast to, for example, a Python package like `nvtx`[2] on PyPI, which can be used by end users to annotate functions and loops in their code. These annotations are recorded by CUPTI and `nsys` and displayed in various tools.

4.2.11 Netintercept

The netintercept library [49] uses the `LD_PRELOAD` environment variable to override functions used by programs to send and receive network packets. It can be used to reconstruct network packets (TCP and UDP) and write the packets to .pcap files, so that it can be read and processed by tools like Wireshark. From the .pcap files it is clear that the nodes use the Ring AllReduce operation to communicate with each other, as shown in Figure 4.2. A disadvantage of the library is that it only captures traffic over Ethernet interfaces and that it stores the reconstructed packets in a .pcap file, which can become very large, in the order of hundreds of megabytes to a few gigabytes. It does not capture traffic that uses the InfiniBand interconnect (for example when using NCCL) because applications using InfiniBand use a *verbs* API instead of the *Berkeley socket* API. The NCCL backend over eth0 interface showed very little data to be exchanged, about a few kilobytes between each two nodes in the first few seconds, followed by a few packets in the last few seconds of the file. When using the Gloo communication backend, Wireshark showed many malformed packets after 30 seconds after the start of the capture.

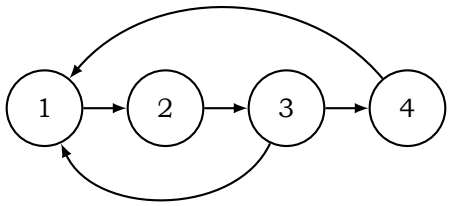


Figure 4.2: *Network packets in a .pcap file show that each node communicates with the other nodes using several rings.*

4.2.12 sysfs

Because a user on the DAS-5 has no elevated privileges, there is no way to measure how much traffic an application is sending and receiving to and from specific hosts. However, it is possible to gather system-wide aggregated statistics of traffic over specific network interface. While NFS traffic is served over InfiniBand on the *ib0* interface, collective communication can be done over the *eth0* Ethernet interface with either NCCL or Gloo.

The Linux kernel provides a filesystem called sysfs, which is normally mounted to `/sys`, and which, when mounted, contains many virtual files that can be read or written to query or modify parts of the state of the kernel.

In the virtual folder `/sys/class/net/<iface>/statistics` various files exist that can be read by regular users:

- `rx_bytes`
- `rx_packets`
- `tx_bytes`

[2] https://github.com/NVIDIA/NVTX/tree/dev/python

- `tx_packets`

These files count only traffic that goes through the TCP/IP stack in the kernel and can also be gathered programmatically using psutil, described in Section 4.2.8.

To measure data that is sent and received over InfiniBand using the *verbs* API, the files in the virtual folder `/sys/class/infiniband/<device>/ports/<port>/counters` must be read instead. This folder contains the following files that can be used:

- `port_rcv_data`
- `port_rcv_packets`
- `port_xmit_data`
- `port_xmit_packets`

4.3 Metrics

Measuring the performance of two aspects of the system, Single-node and Multi-node, requires collecting different metrics. Single-node requires metrics that deal with the hardware of a single node. This includes behavior of the program on the CPU, the GPU, system memory, and the network interfaces. Local disks are not taken into consideration since it is assumed that data is read from over the network.

- **CPU**: Total CPU utilization, thread migration, L1 and L2 cache hits/misses, branch mispredictions.
- **GPU**: Various metrics (see below), power draw and/or energy used.
- **Memory**: Available (free) memory, cached memory.
- **Network**: Throughput.

Measuring the utilization of the CPU cores may help to determine whether the CPU is a potential bottleneck. While this is unlikely to be the case given the large number of CPU cores of nodes in the DAS-5 cluster, this may matter when a CPU with a small number of cores, like in virtual machines in the cloud, is used. Thread migration and the L1 and L2 cache hit rate may indicate inefficient use of the CPU cache. Thread migration may move a thread to a core on a NUMA node that is not directly connected to various PCIe devices like network interfaces and video cards.

Many metrics for the GPU can be collected that provide insight into the utilization of the hardware and the computational efficiency of the kernels executed on the GPU. Metrics for the GPU that might be useful to the user are:

- SM and warp occupancy, stall reasons, bank conflicts.
- Amount of data read and written to device and system memory, over PCIe and NVLink.
- Throughput and utilization of device memory and L1 and L2 caches.
- Usage of specific hardware like shared memory, tensor cores, and NVLink.

These metrics can help to determine the utilization of the SM's, the usage of specific parts of the hardware like the tensor cores or instructions that operate on data with a reduced precision, potential performance problems in kernels, and the number of data transfers to different pieces of memory.

The block and grid size of a kernel launch, and the required registers per thread and amount of shared memory used by kernel, determine the theoretical occupancy, while at runtime the achieved occupancy is measured. The numbers of warps (groups of 32 threads) and blocks that can be in flight on an SM, limit the theoretical occupancy. The block size is

often a multiple of the warp size to avoid inactive threads. A large grid of many smaller blocks increases parallelism and reduces the number of inactive SM's during the last wave [11]. (A wave is a number of blocks per SM in flight times the number of SM's).

Several metrics exist which show the usage of various parts of the memory subsystem of the GPU. These include the amount of data read and written to device, peer, or system memory, the L1 and L2 throughput and cache hit rate per request. The amount of used and free device memory can indicate whether it is feasible to adjust certain hyperparameters which would require the use of more memory on the GPU. Other metrics show the amount of data sent and received over PCIe or NVLink.

Many GPUs have separate instructions that operate on data with a reduced precision like FP16, and a few also support formats like BF16 and TF32, which are more appropriate for machine learning tasks than FP16. Recent GPUs also have hardware for processing matrix multiplications and additions, known by some vendors as tensor cores. Several metrics exist that show the usage by kernels of execution units in the hardware for these formats.

CUDA kernels can have three potential performance problems:

1. Divergent warps
2. Uncoalesced memory access
3. Memory bank conflicts

[40] implements a tool that performs static analysis of the `.cu` files of the CUDA kernels. The tool provides predictions and upper bound for some of the metrics (conflicts and sectors), but requires access to the source code and is thus useful only during development. The 'conflict' metric overestimates compared to what would be measured by Nsight Compute. Furthermore, without the source code, end users do not learn how well a neural network model runs on the actual hardware.

A number of metrics count the number of warps per cycle that were stalled for various reasons. These include warps waiting for barriers due to diverging control flow, memory barriers, memory operations to complete when kernels write to device memory, waiting because of the usage of shared memory or presence of bank conflicts, or a non-balanced use of the various math pipelines. Other metrics count the number of data bank conflicts. These set of metrics can indicate the previously mentioned potential performance problems in kernels.

The throughput per used network interface indicates whether a node can send and receive data fast enough such that the network interconnect is not a bottleneck for training a model.

To measure how different nodes of the system interact with each other, Multi-node, the following metrics are needed:

- **GPU**: The total duration of various sets of launched kernels.
- **Network**: Total send/received bytes and packets, source and destination addresses of packets sent/received by the application.

Dividing the launched CUDA kernels into distinct sets can be used to show how much time the GPU uses to perform various tasks like evaluating the network, updating the gradients, and perform tasks related to collective communication.

The source and destination addresses of network packets give insight into which nodes are communicating with each other and when they communicate. While these metrics normally require elevated privileges, alternative ways exist to obtain them; one way is to override functions of the Berkeley socket API, verbs API, or the functions of the collective communication backend using the `LD_PRELOAD` environment variable. Another way is to measure the changes in the number of sent and received bytes over a network interface in separate files per node, and then find correlations in the various files. Although this would be less reliable than using the `LD_PRELOAD`.

Without using `LD_PRELOAD`, it is still possible to determine which nodes will communicate with each other, but not when they communicate and only when using the NCCL backend, by setting the environment variable `NCCL_DEBUG` to `INFO` and parsing the text printed by NCCL.

Different rings are constructed depending on how many nodes are involved in the Ring AllReduce operation to exchange the gradients during training. Training on four or eight nodes will cause NCCL to construct a single unidirectional ring, shown in Figure 4.3 (for the case with eight nodes). For 16 nodes, the graph shown in Figure 4.4 shows various smaller rings each consisting of three to five nodes.

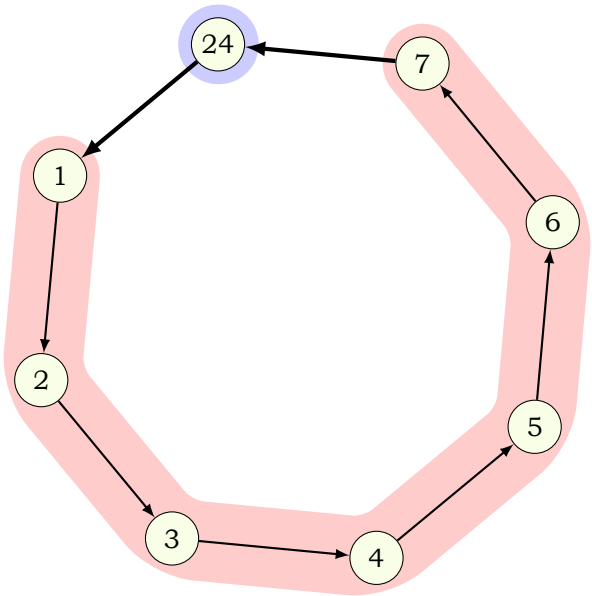


Figure 4.3: *A ring of eight nodes constructed by NCCL. The colored areas represent the IB switches (● = #2, ● = #5).*

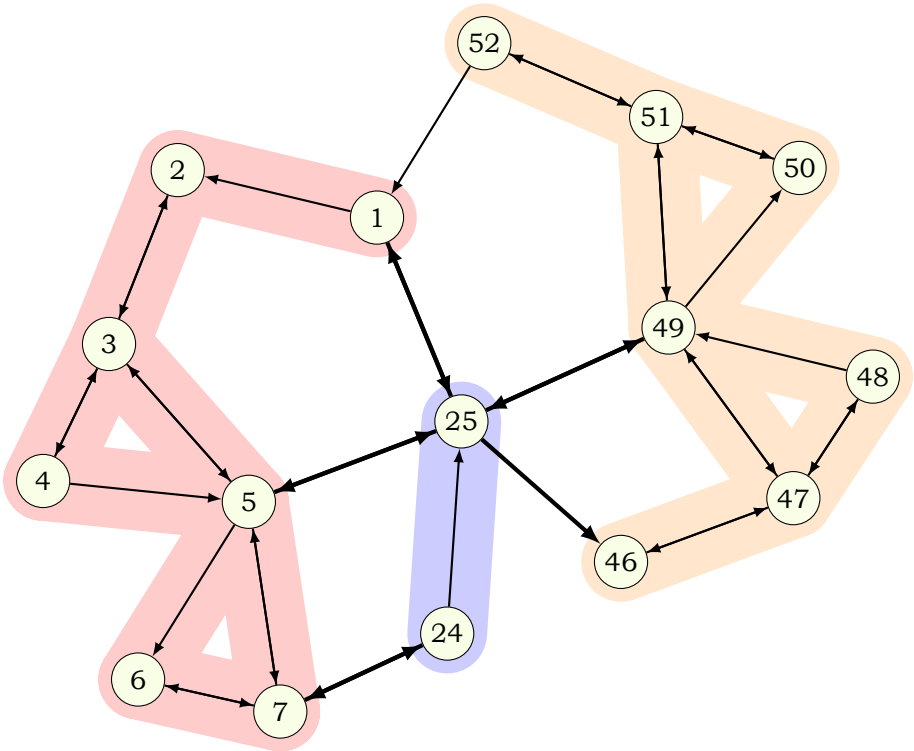


Figure 4.4: *A ring of 16 nodes and a tree constructed by NCCL. The colored areas represent the IB switches (● = #2, ● = #5, ● = #4).*

While in the case of four or eight nodes, all edges of the ring are unidirectional, in the case of 16 nodes, most edges are bidirectional and a few are unidirectional. However, the unidirectional edges are between nodes such that these edges and nodes are part of one unidirectional outer ring and the remaining bidirectional edges are part of a tree.

Each node is connected to one of various InfiniBand switches and the switches are connected to each other in a star topology. The central switch has a handful of nodes connected to it, but none of its nodes participated in the training, and is thus not shown. The red, blue, and orange colored areas represent the switch to which the nodes in an area are connected. The thick edges indicate that communication between two nodes travels along (in this case) three switches instead of one (for the nodes in the same area).

The various metrics for the GPU are measured by tools like `nsys`, `ncu`, or CUPTI. Some of the metrics for the CPU, system memory, or the network interfaces can be obtained using psutil or by reading specific files in virtual filesystems like procfs and sysfs. `nvidia-smi` and NVML can be used to query the state of the GPU. Some of these will now be investigated more in depth in the remainder of this section.

4.3.1 nvidia-smi and NVML

`nvidia-smi` and NVML can be used to query various information about the GPU. Executing the `nvidia-smi` binary, using the function `run` of Python's `subprocess` module in a separate thread, showed that it would often take more than a hundred milliseconds to complete the execution and in several occasions the execution blocked for up to a few seconds. Spawning a new process with `subprocess.run` requires the original process to be forked and this contributes to the high overhead of measuring the metrics this way.

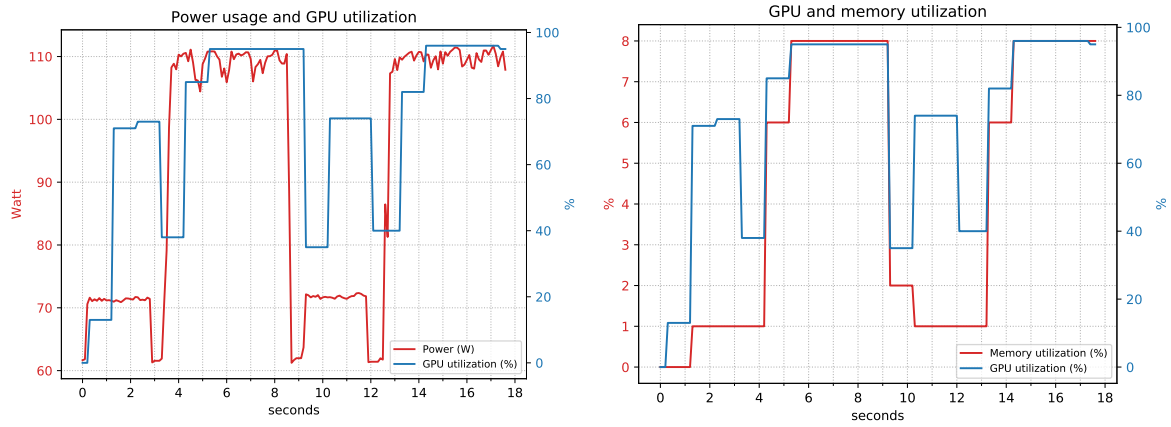
Instead of running the `nvidia-smi` command, Python bindings for the NVML library allow the library to be used directly. The package `nvidia-ml-py` on PyPI provides the Python module `pynvml`, which contains various functions whose names map directly to the names of the C functions exported by the NVML library. For this thesis, an extra Python wrapper was created in the form of a class called `NvmlDevice` which internally invokes the functions provided by `pynvml` and by providing a Python context manager to automatically initialize and shutdown NVML and to yield a list of `NvmlDevice` objects. This would allow various metrics to be taken in a separate thread in about 2 milliseconds in total per iteration.

Figure 4.5 shows the measurements of several metrics obtained using NVML while training and evaluating a ResNet-18 model on the CIFAR-10 dataset for two epochs on two nodes. The various metrics, like power usage and the utilization of the GPU and its memory, were sampled at an interval of 100 milliseconds. Figure 4.5a shows the power usage in Watt and the utilization of the GPU (the SM's). While the power drawn by the GPU can be measured more than ten times per second, at least, and is therefore a good indicator of whether the GPU is executing kernels, the utilization can be measured no more than once per second and only shows the percentage of time that at least one kernel was running on the GPU. The power draw is measured with a resolution of 1 mW while the utilization is in whole percentage. The figure also shows that the power drawn during testing has a small variance while the power used during training of the neural network shows larger fluctuations. Figure 4.5b show the measured utilization of the GPU and its memory. Because of the large sampling interval, the two measured values do not necessarily always increase or decrease at the same time for all nodes.

4.3.2 psutil

Figure 4.6 shows the network traffic measured by psutil on several network interfaces. This traffic was measured while training and evaluating a ResNet-18 model on the CIFAR-10 dataset for eight epochs on a single node, using the NCCL communication backend, configured to transmit directly over InfiniBand using the verbs API. A sampling interval of 1 millisecond was used to obtain the measurements.

Text printed to standard output by the script is transmitted over the eth0 interface, but all `print()` function calls in the Python script were removed prior to running the experiment

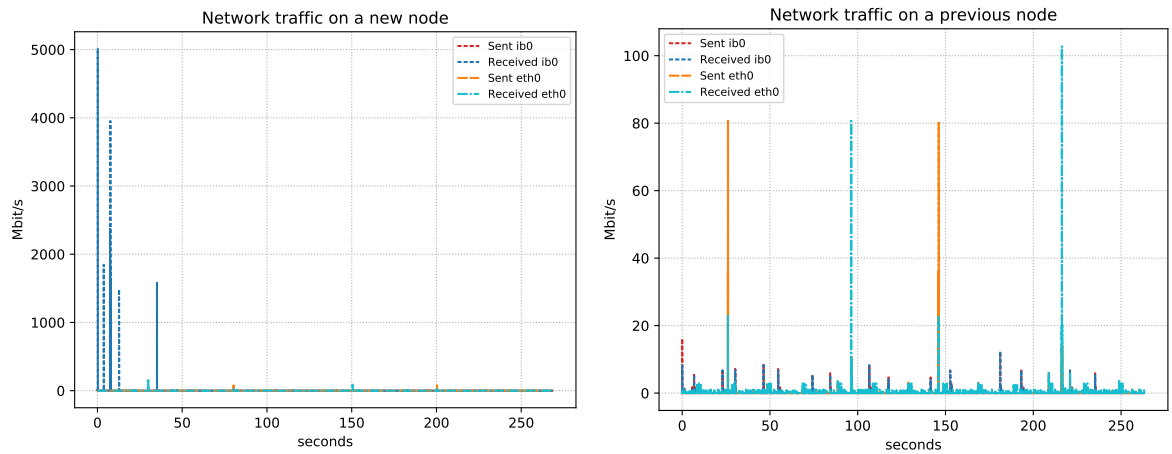


(a) *Power usage and GPU utilization.*

(b) *GPU and memory utilization.*

Figure 4.5: *Power usage and the utilization of the SM's and the memory of a single GPU of one of the nodes, measured by NVML.*

via the Slurm scheduler. Parts of an ML framework like the NCCL communication backend may print a few lines of text at the beginning of the output. The four spikes of network traffic over Ethernet in the figures are most likely transmitted by some system daemon running on the nodes. On the DAS-5 two of such daemons are listening on the eth0 interface. This shows that other processes running on the system may cause noise in the results and, if the user were to have elevated privileges, other techniques such as eBPF could be used to gather more accurate measurements.



(a) *Observed traffic on a previously unused node.*

(b) *Observed traffic on a node that has been used before.*

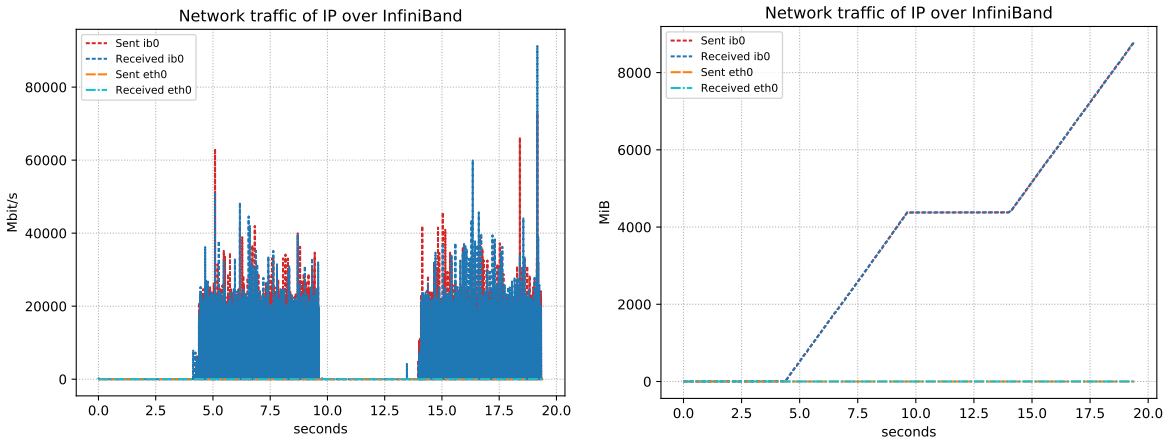
Figure 4.6: *Traffic measured by psutil when training on a single node using NCCL for communication. Because psutil has no special support for InfiniBand, it measures only IP packets that go through one of the Ethernet interfaces.*

The Python script used to perform training using PyTorch framework was wrapped in the `nsys` tool for tracing the CUDA API calls and kernels. The usage of the `nsys` tool explains why, on a single node, a very small amount of traffic over InfiniBand is observed per epoch for all eight epochs. Figure 4.6a shows the traffic observed on a new node. The relatively large amount of traffic is not observed when repeating the experiment as shown in Figure 4.6b, likely due to the files of the dataset being cached in one of the caches used by the *Virtual File System (VFS)* layer of the OS. A similar change may be observed when measuring I/O traffic to local disks. For repeatability, the caches should be flushed prior to running the experiment. Otherwise different techniques such as overloading several system calls using `LD_PRELOAD` or

overriding Python methods of the `DataLoader` from PyTorch would be necessary. The latter option is not optimal for a tool that wishes to be framework agnostic.

When running the experiment on two nodes, the observed traffic shows almost no traffic is measured by psutil over InfiniBand except about 1 kilobyte in the first few seconds of the measurements. Setting the environment variable `NCCL_DEBUG` to `INFO` and `NCCL_DEBUG_SUBSYS` to `ALL` shows that NCCL uses the `Broadcast` and `AllReduce` operations to communicate with the other nodes and that psutil is able to measure IP packets send over the InfiniBand interconnect, but not InfiniBand packets themselves, even though the information is available in the virtual files in `/sys/class/infiniband/`. This is because the IP packets go through the TCP/IP stack in the kernel and are therefore counted by the files in `/sys/class/net/ib0/statistics`.

psutil can still be used to measure collective communication using the NCCL backend when NCCL is forced to use IP packets by setting the environment variable `NCCL_IB_DISABLE` to 1. However, this will cause traffic to go through the TCP/IP kernel stack, incurring extra latency. Figure 4.7 shows the throughput in megabit per second (Mbit/s) and the accumulated size in mebibytes (MiB) of one of two nodes when training the same model and dataset as before for two epochs instead of eight. The average throughput is about 7340 Mbit/s, which is much less than the maximum speed of 56.25 Gbit/s of the 4x FDR InfiniBand interconnect.



(a) *Throughput of observed traffic.*

(b) *Accumulated size of observed traffic over time.*

Figure 4.7: *Throughput and accumulated size of IP packets over InfiniBand measured by psutil when training on two nodes for two epochs using the environment variable `NCCL_IB_DISABLE=1`. The use of `nsys` was disabled.*

4.4 Discussion

Many of the utilities and libraries listed in Table 4.1 focus on the utilization of the hardware of a single node and few can be used to monitor the interaction between nodes.

Perfetto and Netintercept were not usable because they required either instrumentation or produced large files. Netintercept could potentially be modified to transfer reconstructed packets or the headers of these packets to another process using some form of *interprocess communication (IPC)*. Other tools like `perf`, `perf_event_open`, and eBPF require elevated privileges. eBPF programs, created with the BPF Compiler Collection (BCC) toolkit [9], may be useful on systems where a user can obtain the necessary Linux kernel security capabilities. psutil, files exposed by sysfs in `/sys`, and NVML can be used to measure network traffic and query information about the GPU and have low overhead. `nvidia-smi` provides access to the same information as NVML, but cannot be used in Python to query at high frequency. psutil can measure InfiniBand traffic only when sent over the ib0 interface via IP sockets. Nsight Systems (`nsys`) and Nsight Compute (`ncu`) are powerful applications for tracing and profiling, but `ncu` has high overhead because of the many metrics it gathers. A disadvantage of tools

like `nsys` and `ncu` is that they do not allow for real-time analysis of gathered metrics. These tools periodically write their collected data to temporary files, which, after the user's Python script has finished executing, are exported to another format, which can subsequently be processed and analyzed by a tool such as the one described in the next chapter, Chapter 5. CUPTI is useful as the programmatic counterpart of `nsys` and `ncu`.

Answering research question **RQ2**, which parts of the system can be measured, depends mostly on whether Linux kernel capabilities can be obtained and whether C/C++ code or Python bindings are required. Table 4.3 shows different sets of tools and libraries that can be used together to build a new tool. Set A requires elevated privileges because of eBPF, which runs small programs in the kernel, while B requires modifying Netintercept to avoid writing large .pcap files or creating a new library which overrides various functions, and the creation of Python bindings for CUPTI. Set C uses the `nsys` or `ncu` tools, and thus require processing files exported by these tools after the training of a model has completed. Set D requires creating Python bindings for CUPTI, but may allow for collecting and analyzing metrics in real-time instead of afterwards. Due to the Linux kernel used on the DAS-5 and the missing capabilities, and the lack of Python bindings for CUPTI, set C has chosen to build the tool. However, commercial cloud services may provide virtual machines or containers and provide customers with a user with elevated privileges. In this case a tool build using the technologies of set A may be able to gather more precise metrics and thus provide more detailed analysis of the system.

Table 4.3: *Different combinations of tools and whether they require Linux kernel security capabilities or whether C/C++ code or Python bindings are required. See Table 4.1 for an explanation of the symbols.*

Set	Tools	Elevated privs.	Code
A	eBPF, NVML, CUPTI	●	◑
B	`LD_PRELOAD`, NVML, CUPTI	○	●
C	psutil, NVML, `nsys`/`ncu`	○	◑
D	psutil, NVML, CUPTI	○	◑

5

ML Board

In Chapter 4 various techniques and libraries were investigated that could be used to create a tool called ML Board [1] that provides the user with insight into how effective a neural network model is trained and used on a cluster of multiple nodes with GPUs. This chapter describes the features and architecture of the tool.

This chapter is structured as follows: Section 5.1 describes the research questions this chapter answers. Section 5.2 lists the objectives of the tool, describing which parts of the system will be measured and visualized. Section 5.3 explains how the objectives are met by several visualizations displayed by the tool. Section 5.4 provides a description of the software architecture of the tool. Section 5.5 provides a discussion of the tool.

5.1 Research questions

This chapter answers the following research questions of Section 1.1:

RQ3 How much time is spend on communication between nodes, loading data, updating parameters, and other aspects in popular large-scale machine learning frameworks?

To answer research question **RQ3**, a tool is presented that can visualize the performance of various parts of the system. The tool will gather the set of metrics mentioned in Section 4.3 and visualize these in a web application.

5.2 Objectives

There are already a few existing tools that can do performance profiling of GPUs. However, these tools often have a perspective that is focused on the hardware of the GPU of only a single node instead of having a focus on the system as a whole or on the application, the training and evaluation of a neural network model.

ML Board should be able to help the user to determine how well a neural network model makes use of the system when it is trained on GPUs in a cluster of nodes. It will focus on the following objectives:

1. **(Idleness)** Identifying periods during training and testing of the model where some or all parts of the system (on different nodes) show reduced utilization or are completely idle or unused.

   Identifies parts of the system that are less utilized at times than other parts. A low utilization may indicate that the model or the machine learning framework itself does not make optimal use of all of the available hardware. The user could then make changes

[1] The Git repository of ML Board can be found at https://github.com/markpk/mlboard

to the model, swap it for a larger model, or change the hyperparameters. If used on resources of a commercial cloud service, the user might decide to switch to a cheaper lower tier of a resource, or trade it in for a resource with different specifications which may be more suitable for training or testing the model on the given dataset. A very high utilization might indicate that some parts of the system are a potential bottleneck.

2. **(Utilization per layer)** Showing the utilization of various parts of the GPU for the layers of a neural network model.

   Helps the user to determine whether specific layers of a model utilize the various parts of the GPU like the Streaming Multiprocessors (SM's), usage of the L1 and L2 caches, PCIe bandwidth, etc. Parts of the GPU that show high utilization may indicate a bottleneck.

3. **(Common tasks)** Showing when the machine learning framework is loading data, communicating with other nodes, or performing computations, and whether there is any overlap.

   Shows the user what the parts of the system are doing. This includes evaluating the model on the GPU, exchanging and updating parameters, and loading datasets from local disks or over the network. This may help the user to identify whether the various tasks that a machine learning framework performs is done concurrently.

4. **(Power and energy)** Determining the power and energy usage of the system.

   Would show the energy required to perform training or testing a model. A high power usage would mean a higher utilization, but does not mean the hardware is used efficiently.

As mentioned in Section 4.2, when monitoring the performance of the system, one can look at utilization of the hardware of a specific node as well as the interaction between the nodes that participate in training or testing a model. Table 5.1 shows which aspects of the system are taken into account by an objective.

Table 5.1: *A list of objectives and which aspects of the system they measure and visualize.*

Objective	Single-node	Multi-node
Idleness	✓	✓
Utilization per layer	✓	
Common tasks		✓
Power and energy	✓	

5.3 Visualizations

Various visualizations displayed by the tool help to achieve the objectives mentioned in the previous section. Objective 1 is addressed in Section 5.3.1, Objective 2 in Sections 5.3.2 and 5.3.3, and Objectives 3 and 4 in Section 5.3.4.

5.3.1 Idle time between kernels

Evaluating or training a neural network requires launching multiple kernels due to the many layers of a network. Launching a kernel has overhead and gaps may exist between two kernels in the trace if API calls do not happen quickly one after another, which could cause the GPU to become idle. Thus the total duration from the first to the last kernel might be reduced if the whole graph of kernels is known in advance and spawned with a single API call instead of multiple.

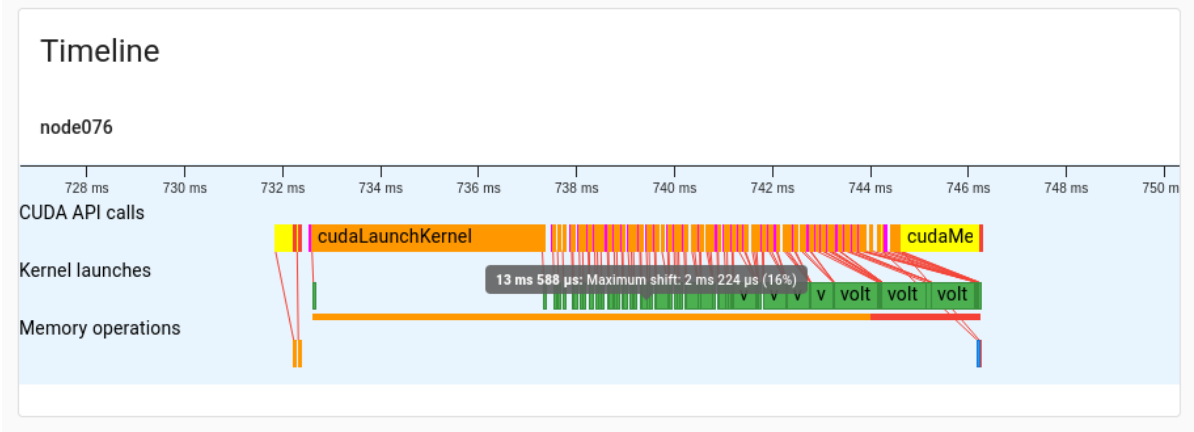
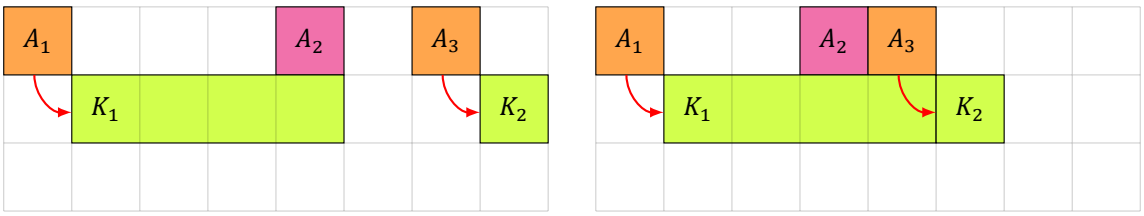


Figure 5.1: *A timeline showing several API calls and launched kernels. The text in the tooltip shows the total duration of the sequence of launched kernels that are visible and how much it can be decreased. The tooltip shows that the sequence of launched kernels that can be shorten by up to 16%.*

Figure 5.1 shows some examples of a timeline showing the execution of CUDA API calls, kernels running on the GPU, and memory operations. In the row of API calls (the first row), orange represents `cudaLaunchKernel()` call, yellow is used for calls that allocate, free, set, or copy memory, magenta is used for calls related to events, red for calls that perform synchronization, cyan for CUDA streams (not visible in the figures), and brown for anything else (also not shown in the figures).

The second row is used to display the kernels running on the GPU, represented by the green rectangles. The thin orange and red line shows the total duration from the first to last kernel that was launched in the visible part of the timeline. The red part represents that maximum fraction of this duration that could be saved (or shortened) if the overhead of launching the kernels on the CPU were to be reduced. The latency between a call to `cudaLaunchKernel()` and its corresponding kernel launch on the GPU is assumed to remain the same and only the gaps between two API calls and the two kernels are taken into account.

For example, if there are two subsequent calls to `cudaLaunchKernel()` divided by a duration of 200 µs but their kernels are divided by just 100 µs, then the API and its corresponding kernel execution can be shifted no more than 100 µs. In this case the maximum shift to the left is limited by the duration of the kernels on the GPU. This scenario is shown in Figure 5.2.

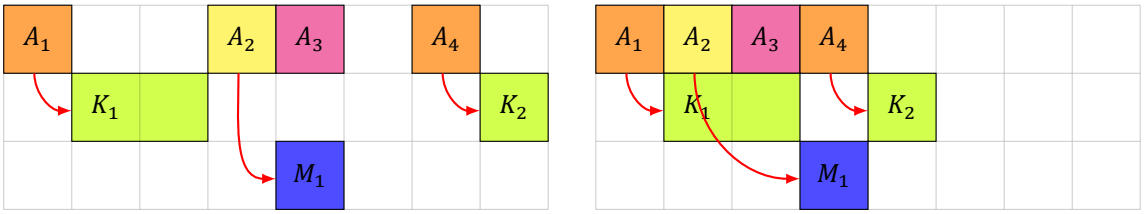


(a) *Original sequence of API calls and kernels.* **(b)** *A sequence without any gaps between the kernels.*

Figure 5.2: *The maximum shift of A_{i+1} and its kernel is limited by the duration of the kernel corresponding to API call A_i. In Figure 5.2b the presence of the additional API call A_2 has no effect on the maximum shift of the kernel K_2 launched by A_3.*

Between two calls to `cudaLaunchKernel()` there can be zero or more other API calls, which limits how much the second call to `cudaLaunchKernel()` (and its corresponding kernel launch) can be shifted to the left. This is shown in Figure 5.3. Without the API calls A_2 and A_3, the call A_4 and its kernel K_2 would be shifted further to the left.

The third and last row shows the memory operations. Copying data from the GPU's global memory to another location in its memory (called device to device) is colored green, while data transfers from the CPU to the GPU (host to device) are colored orange, and back from



(a) *Original sequence of API calls, kernels, and memops.* **(b)** *A sequence without any gaps between the API calls.*

Figure 5.3: *The maximum shift of A_{i+1} and its kernel is reduced because of other API calls between A_i and A_{i+1}.*

GPU to CPU (host to device) are shown using red rectangles.

Most of the sequences of API calls and launched kernels show a maximum shift of roughly 13 to 16%. Some of the sequences take more time than usual and thus can be shifted more than 13 to 16%, such as the sequence shown in Figure 5.4. A nonzero maximum shift suggests there is potential to decrease the total duration of an iteration and decreasing the chance that the training time of an epoch is limited by the computations instead of the communications. The maximum shift of the iterations can be summarized and displayed by some boxplots, such as the ones shown in Figure 5.9. The overhead of the multiple `cudaLaunchKernel()` call can be avoided in certain cases by using CUDA Graphs [41]. The launching of the kernels is recorded in a graph once and the whole graph can then be launched multiple times with a single call to `cudaGraphLaunch()`. There is a higher cost upfront, but any recurrence of the sequence of calls (which is the case because the neural network is executed many times for different input) would avoid a lot of the overhead of launching the individual kernels.

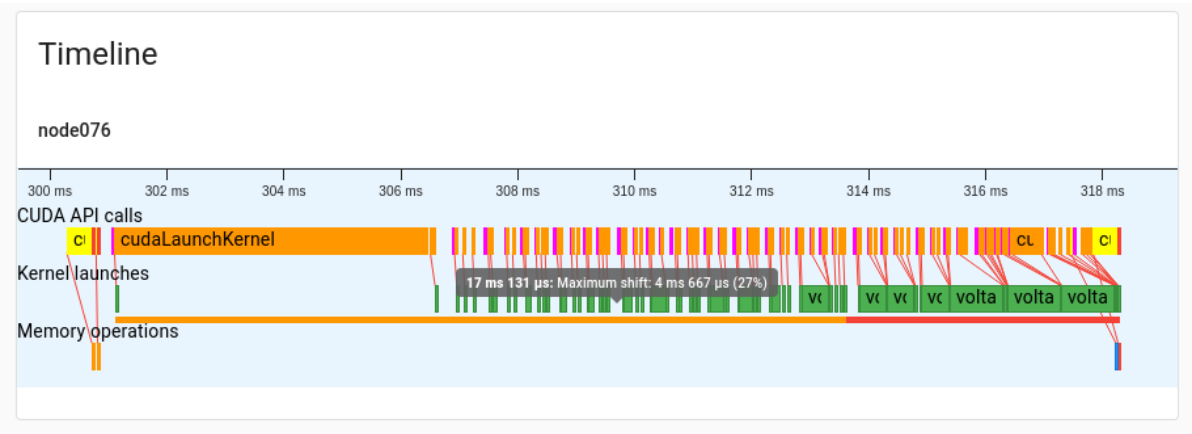


Figure 5.4: *A timeline showing several API calls and launched kernels. The tooltip shows that the sequence of kernels that can be shorten by up to 27%.*

5.3.2 Statistics of launched kernels

Figure 5.5 is a screenshot of the tool showing a list of kernels that were launched on the GPU while training or testing a model of a neural network. By default, only the top ten of kernels that spend the most time on the GPU are shown. The user can press a button in the lower right corner to expand the table so that all launched kernels are displayed on the screen. For each kernel, the number of launches, total time spend on the GPU, and the average duration per launch is shown. Different launches of a kernel can use a different number of threads (block size) or different number of thread blocks (grid size), but are grouped by the name of the kernel. Each column shows a progress bar displaying the percentage of the measured value compared to the total amount of all values in the same column, and the measured

Kernels

070 071 078 σ:

Name	Launches	Total time (s)	Average time (µs)
volta_scudnn_128x128_relu_small_nn_v1	33264	16.270	489.124
dgrad2d_alg1_1	1584	8.956	5653.735
vectorized_elementwise_kernel	332552	1.689	5.079
volta_scudnn_128x64_relu_small_nn_v1	15248	1.474	96.670
volta_scudnn_128x128_stridedB_small_nn_v1	3168	1.168	368.536
volta_scudnn_128x64_stridedB_small_nn_v1	3696	1.154	312.264
wgrad_alg1_engine	3696	1.125	304.466
volta_scudnn_128x64_stridedB_splitK_small_nn_v1	3696	1.093	295.590
bn_fw_inf_1C11_kernel_NCHW	50080	0.455	9.088
volta_scudnn_128x128_stridedB_splitK_small_nn_v1	1584	0.374	236.329

Figure 5.5: *The top ten of launched kernels that have spend the most time on the GPU. The user can use the widgets in the upper right corner to select which kernels must be displayed.*

value itself. The user can hover over a value to display a tooltip with the percentage and the number of standard deviations from the mean of all values in the same column or from the mean of the values of the same kernel on all the nodes.

Values that are sufficiently higher than other values in the same column are shown in a red color and values which are sufficiently lower are shown in green. To determine whether a value is considered to be sufficiently different, first the mean and the standard deviation of each column is computed. A value in a column is then considered to be sufficiently different if the ratio of its demeaned value to the standard deviation of a column is greater than a threshold selected by the user. If a kernel has a column with a value that is sufficiently different, then the name of the kernel is shown in a red color as well. The progress bars and the red colored values can help the user to quickly determine which kernels can have the most benefit from being optimized.

The upper right corner of the card contains several widgets, which the user can interact with to change which information is to be displayed. The first widget is a group of buttons that can be toggled. This allows the user to select which node whose kernels must be displayed in the table. Only one node can be selected at a time.

The second widget is a group of toggle buttons which represent the CUDA streams to which the traced kernels belong. When training using the PyTorch framework and the NCCL collective communication backend, there exist two CUDA streams; one for the kernels which evaluate and train the network, and one for the NCCL kernels.

The third widget shows a slider to select the threshold needed to display kernel names and values in a red or green color, and whether the standard deviation must be computed for each column (values of different kernels) or layer (values in the same cell for different nodes) using the two buttons on the right of the slider. The slider allows changing the threshold between 1 and 6 standard deviations (using a default of 3).

When choosing to compute the standard deviation per column, the slider can be set to a threshold between 1 and 6 for a few kernels to be displayed in a red color. When choosing to compute per layer, a much lower threshold of 1 must be selected. In this case quite many kernels light up in red with demeaned values which are between 1 and 2 times the standard deviation. However, this often happens only for specific nodes (and appears to correlate with other visualizations explained in Section 5.3.4). Other nodes display values in the table which do not deviate significantly according to the threshold set by the user.

In Figure 5.5 it can be clearly seen that the kernel `vectorized_elementwise_kernel` was launched many times (close to 333×10^3 times or 54.7% of all launches), but took very little time in total and per launch. In contrast, the kernel `dgrad2d_alg1_1` was launched relatively few times (close to 1.58×10^3 times or 0.3% of all launched kernels), but used nearly 25.4% of the time on the GPU and a significant amount of time per launch (compared to the other kernels). This means that any optimizations applied to the kernel `dgrad2d_alg1_1` would decrease its absolute average and total time more than optimizations applied to kernels like `vectorized_elementwise_kernel`, and therefore have the greatest effect on decreasing the time of an epoch to train the model.

5.3.3 GPU utilization per node

A graph of a ResNet-18 model is shown in Figure 5.6. The rendering of the graph is performed by Cytoscape.js [13, 14], a library to visualize and analyze graphs. The gray compound nodes representing the basic or bottleneck blocks (see Figure 5 in [21]) of a ResNet model can be collapsed. The first block of the third layer (conv3_1 in Table 1 in [21]) has been expanded and shows the extra convolution and batch normalization steps performed for blocks that downsample the size of the data in the right side of the compound node.

Some machine learning frameworks specify the graph of the model explicitly, while others, like PyTorch, construct it implicitly during the execution of the forward phase. This prevents retrieving a graph representation directly from an instance of a model like `torchvision.models.ResNet` in Python code outside of the framework. However, many frameworks can export a computation graph of a neural network model to open standards like ONNX [36] or framework-specific formats.

Without the availability of an neural network model exported to a standard format, the tool would need to contain graph representations of various popular models and then match each of the models with the traced CUDA kernels of an iteration of an epoch to detect which model is being used. The names, ordering and number of occurrences of the kernels, and various metrics like the grid size can be used to test which of the predefined models most closely matches with the traced kernels.

Using `ncu`, a recent version of `nsys` (not available on the DAS-5), or by using CUPTI, several of the metrics mentioned in Section 4.3 can be gathered for each kernel. These metrics can then be displayed in the table in Figure 5.5 or visualized in the graph of the model by making use of colors, for example to highlight nodes in the graph which have some of the mentioned potential performance problems which CUDA kernels can have, or by using a variable width for the edges between two adjacent nodes depending on the amount of data read and written to memory. However, the computational graph does not show kernels used by the backward phase like the kernels which compute the gradients, and is thus less useful than simply displaying various metrics in a table.

5.3.4 Common tasks of machine learning frameworks

All machine learning frameworks perform various common tasks to train a model on multiple nodes, including running kernels on the GPU for the forward phase, updating gradients on the GPU, performing collective communication, loading data, and idling. To train a neural network, first, input data must be loaded locally from disks or over the network and fed to the neural network in order to compute an output, the prediction. Then, a loss function is used to compute the error between the desired output and the predicted output, which is then propagated backwards through the network of the model. In distributed training on multiple nodes, the updates to the model must be exchanged by the machine learning framework with the other nodes using a collective communication backend, which uses a primitive like AllReduce to exchange data between multiple GPUs. During each epoch the whole dataset is processed in multiple iterations consisting of small batches of input samples.

ML Board should show the following common tasks performed during each epoch:

- **Communication**: Collective communication primitives on the GPU and sending and receiving data over the network.

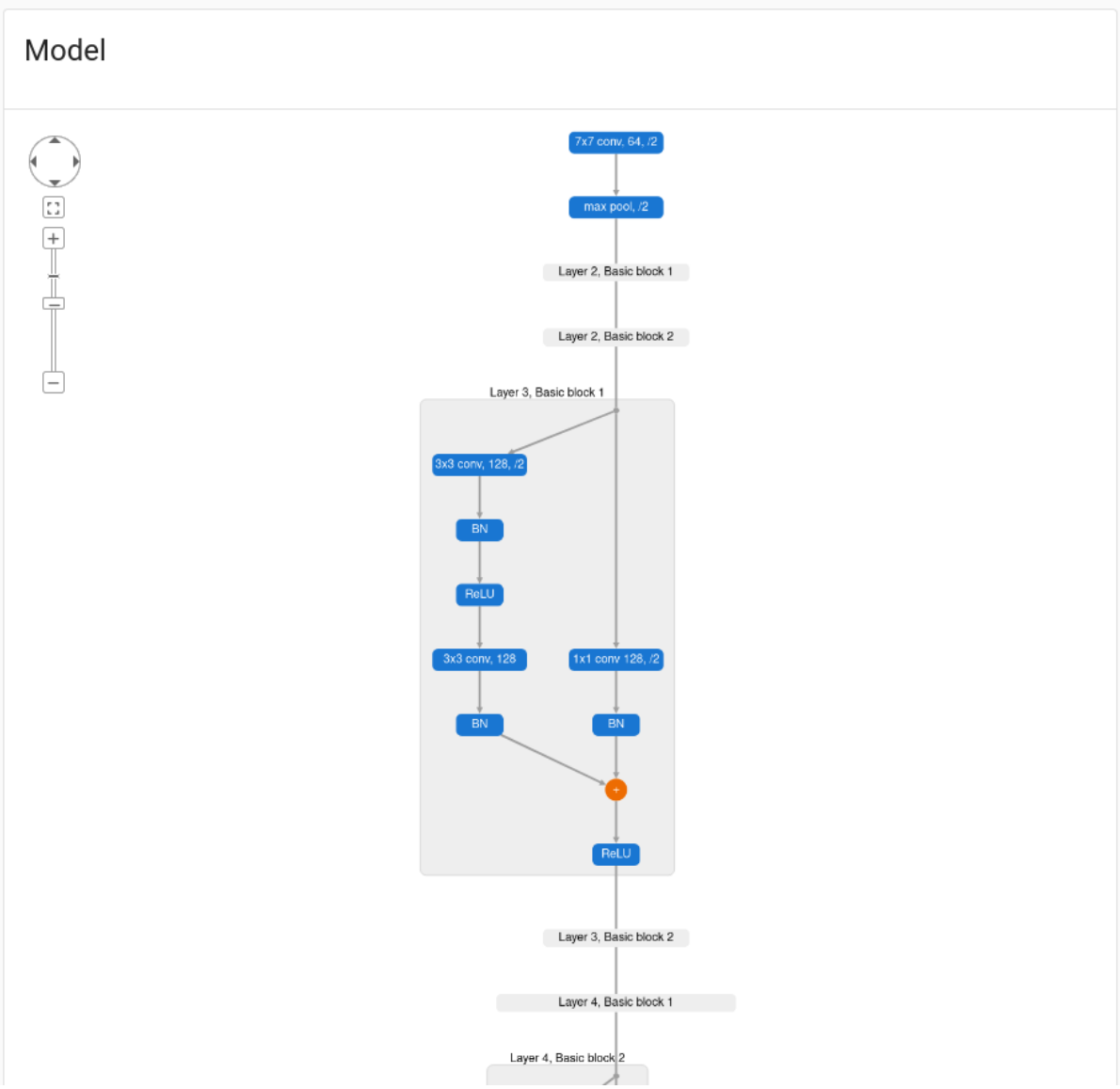


Figure 5.6: *A visualization of the graph of the network architecture of a ResNet-18 model. The basic blocks in most of the layers have been collapsed, except for the first block of the third layer.*

- **Computations**: Evaluating the neural network model and updating gradients during each epoch.
- **Loading data**: Loading data from disks or over the network.

Additionally, it should show when there is overlap between the tasks mentioned above and when the system is idle.

Figure 5.7 displays another view of the timeline, showing several iterations of one of the epochs of a run of a Slurm job. For every epoch, the Python script used by the Slurm job tests the test dataset on the network to determine its accuracy followed by training of the network on the available nodes in the cluster. The epochs can be detected using a set of heuristics such as the idle time between two API calls or the presence of specific API calls like `cudaStreamSynchronize()`. To detect the testing and training parts of the epochs, an idle gap of about 25 milliseconds is used. The iterations are detected using a smaller idle gap of about 2 milliseconds. In the bottom part of the timeline, the network traffic for collective communication is shown. The line graph shows the ratio of the throughput to the maximum

speed of the interconnect. The maximum speed of the 4x FDR InfiniBand interconnect of the DAS-5 cluster is $64/66 \cdot 56.25$ Gbit/s. The traffic was sent and received by NCCL using an IP socket so that it could be measured with psutil. A sampling time of 1 millisecond was used to obtain the measurements. The red line graph shows the throughput of the traffic that was sent by the node and the purple colored line graph shows the traffic that was received from other nodes.

Figure 5.7: *A timeline showing several iterations of one of the epochs during which the model is trained.*

Figure 5.7 shows that network traffic is sent and received concurrently with each iteration of the training phase of an epoch. Each iteration first evaluates the network (forward phase) and then updates the gradients during the backward phase.

This view shows rather detailed and raw measurements, from which it is difficult to draw conclusions regarding the high-level behavior of the workload. It is therefore needed to reduce the amount of information presented to the user by analyzing the measurements of multiple nodes and summarizing it in the tool.

Figure 5.8 shows the mean of the ratio of the time spent on each task during the epochs. The dark and light orange bars represents the total duration of the computation of the forward and backward phases of the neural network model when it is either updating the gradients or doing some other task like batch normalization or convolution, while the blue bar represents the collective communication performed on the GPU by backends like NCCL. These two tasks are performed by different CUDA streams, which means they are running concurrently on the GPU. The dark and light green bars shows the overlap of the two tasks.

Detailed statistics in the form of a boxplot and a violin plot for the probability density function for several metrics are shown in Figure 5.9. The following metrics can be chosen:

- Ratio and absolute total duration in seconds for the common tasks mentioned above.
- Idle time before epochs either evaluate or train the network.
- Total duration of an epoch (training or evaluation).
- Ratio and amount of time that the duration of an epoch can be decreased if no idle gaps would exist between kernels.
- Average power drawn in Watts and energy spent in Joules.

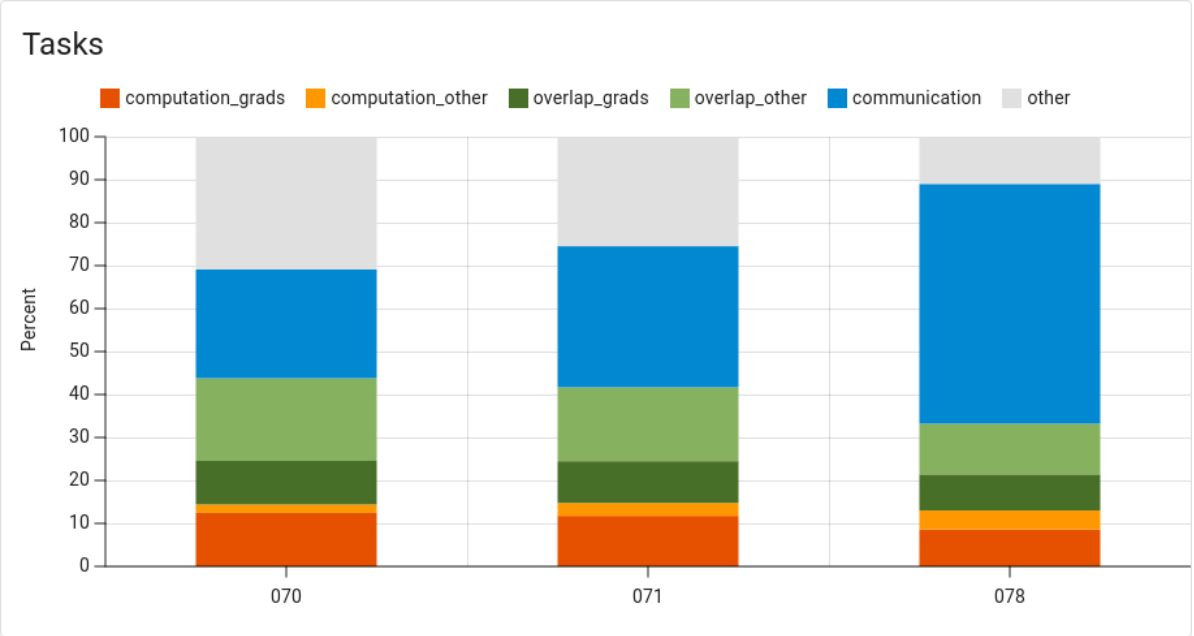


Figure 5.8: *A chart showing for each node a bar stack of the tasks performed by the machine learning framework during training.* `communication` *refers to kernels launched by the collective communication backend,* `computation_grads` *and* `computation_other` *by other kernels which either update the gradients or do any other task such as batch normalization and convolution,* `overlap_grads` *and* `overlap_other` *when these two types overlap with the communication kernels, and* `other` *when no kernels are running.*

In Figure 5.9 it is clearly visible that node078 spends more time on collective communication than the other two nodes. The box plots for the total duration of an epoch when training shows similar results. In contrast, the median of the duration of epoch which only evaluate the model are within 0.2 seconds of each other. This is likely due to no communication happening between the nodes for these epochs. The total duration of the computation tasks is within 0.1 seconds for the three nodes.

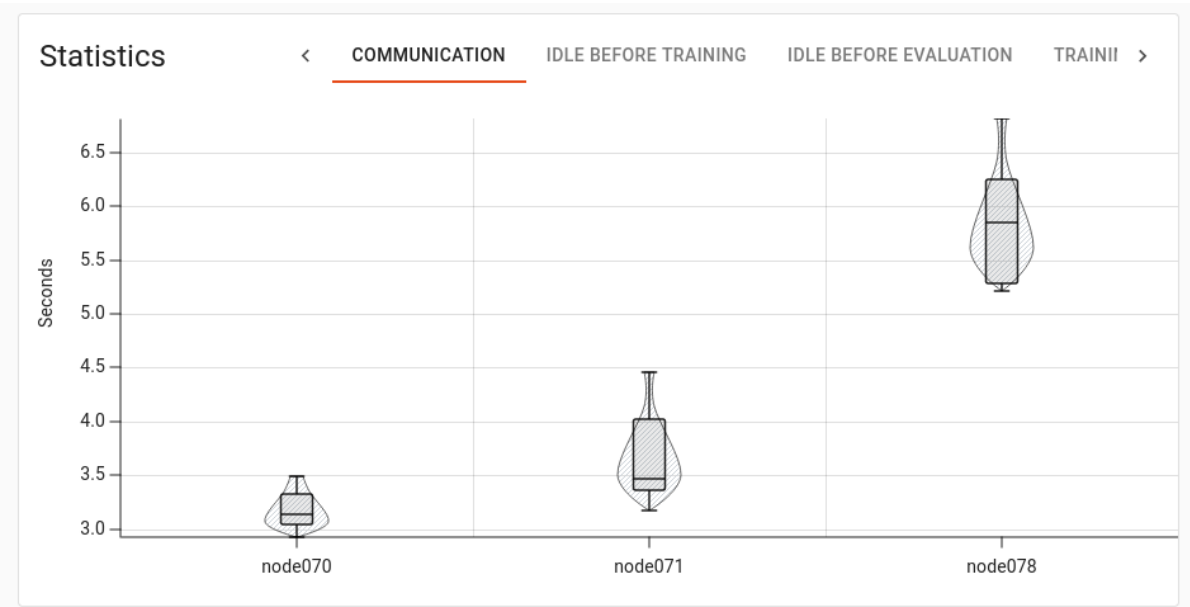


Figure 5.9: *A chart showing a box plot and violin plot for each node for one of several metrics.*

5.4 Architecture

This section provides an architectural description (IEEE Standard 1471 [38]) of the architecture of ML Board using a number of views based on the viewpoints documented by [50]. A viewpoint specifies how to describe a certain aspect of the architecture. The following viewpoints are used:

- **Context**: Documents the relationship between the system and its environment.
- **Functional**: Documents the functions of the system.
- **Concurrency**: Documents which parts of the system operate concurrently and how they communicate and synchronize.

5.4.1 Context

Scope and responsibilities

As mentioned in Section 5.2, the system, ML Board, should be able to help the user to determine how well a neural network model makes use of the system when it is trained on GPUs in a cluster of nodes. See the mentioned section for a list of objectives.

The system reads and analyzes various trace files and provides the results with statistics and other visualizations to the user.

External entities

The following external entities interact with the system:

- Trace files. A *data provider* and *consumer*. Files containing metrics described in Section 4.3. Multiple files exist for each Slurm job and node in the cluster.
- `nsys profile`. A *data provider*. Collects various metrics and stores the metrics in `.qdrep` files.
- Trace agent. A *data provider*. Entity which is invoked by the the Python code that performs the training. The entity performs collection of various metrics not collected by `nsys profile`.
- `nsys export`. A *service provider*. Converts `.qdrep` files to `.sqlite` files, which uses the SQLite format. The conversion depends on the amount of data that has been collected and can take several seconds per file. Interaction between this entity and the system occurs when the system detects new `.qdrep` files or when the user interacts with the system.

The system and the external entities are shown in Figure 5.10. The trace agent is an external entity, but is considered part of the ML Board tool.

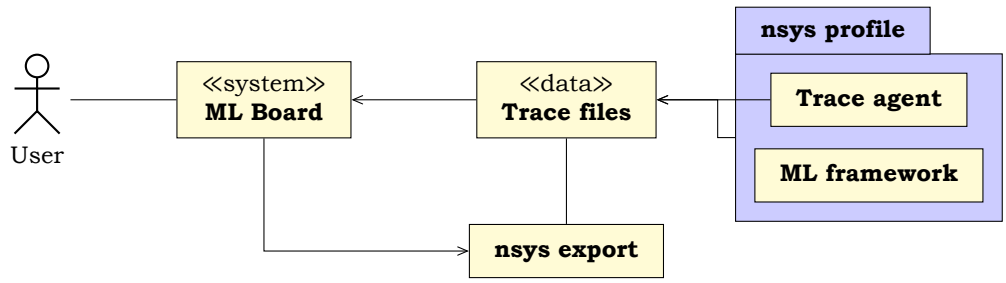


Figure 5.10: *Context diagram of ML Board, showing the system and its environment.*

5.4.2 Functional

Functional elements

The functional structure of the system is shown in Figure 5.11. The Web Browser is responsible for showing the visualization described in Section 5.3, while the Web Server is responsible for providing the data needed to display the visualizations. The Web Server asks the Trace Processor to analyze the traces and metrics it has gathered and return the data needed for the visualizations.

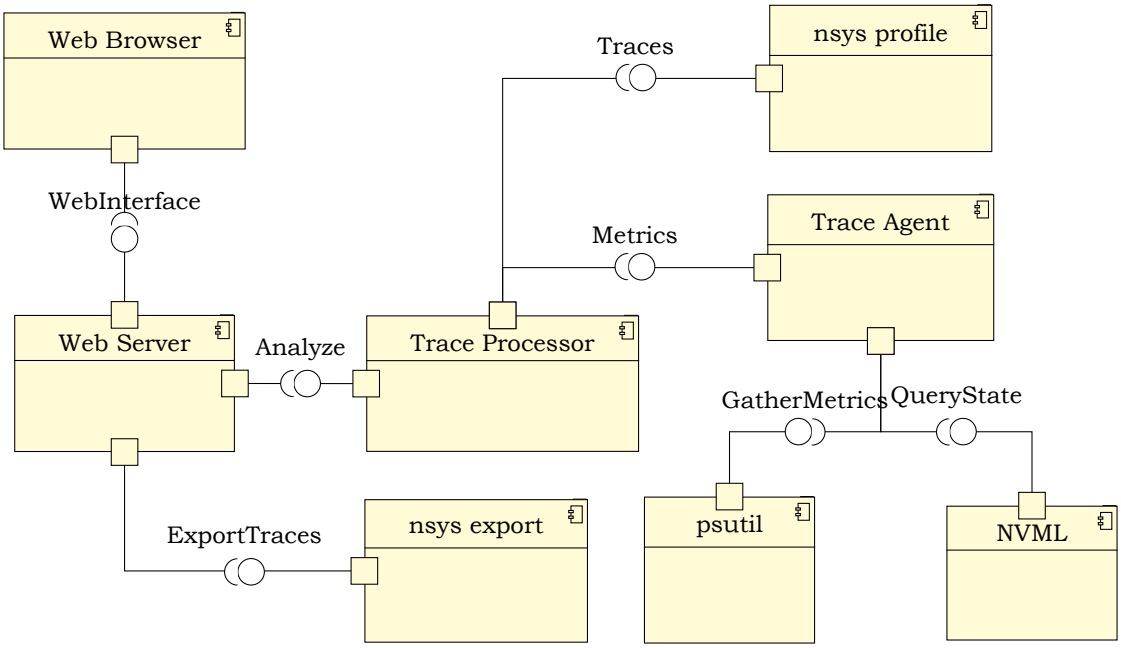


Figure 5.11: *Component diagram showing the functional structure of the system.*

Interfaces

The following interfaces (data and control flow between elements) and their operations are used:

- **WebInterface**: A *Single Page Application (SPA)* which fetches the data needed to display the various visualizations of a job, the current job reservations and a list of completed jobs.
- **Analyze**: A request which asks the Trace Processor to provide the analysis of the collected traces and metrics. The operation will cause the Trace Processor to start analyzing the data it has if needed.
- **GatherMetrics**: A request to gather and return metrics of a node using the Python psutil library.
- **QueryState**: A request to query and return the state of the GPU using the NVML library.
- **Traces** and **Metrics**: The traces and metrics collected by the Trace Agent and `nsys profile`. This data is received automatically while the respective functional elements are active.
- **ExportTraces**: A request to export the traces to a format suitable for analysis.

Connectors

The functional elements, providing the previously mentioned interfaces, use several connectors to exchange data and service requests:

- **REST API calls over HTTP**: used by the *WebInterface* of Web Server. The response contains data in the JSON format.
- **Procedure calls**: used by *Analyze* of Trace Processor, *GatherMetrics* of psutil, and *QueryState* of NVML. The latter two requests are expected to have submillisecond response times, while the first may take several seconds.
- **File transfers**: used by *Metrics* of Trace Agent and *Traces* of `nsys profile`. Files may be written during or only just before termination of the respective functional element. The Trace Processor is notified of the response (the files) through the filesystem events using the inotify API of the Linux kernel or by querying the presence of the files on the file system at a regular interval.

5.4.3 Concurrency

Processes and threads

The concurrency model of the system is shown in Figure 5.12. The system consists of several processes running on different nodes of the cluster; the main application of ML Board runs on a node accessible to the user, such as the master node. The other part of the system consists of a group of processes, one for each compute node. Each compute node runs a Slurm job which runs `nsys profile` and a Python user script. This script runs the ML framework and uses a thread pool for the functional element Trace Agent, which collects metrics using several other functional elements like psutil and NVML. The Trace Agent uses Python's `threading.Event` and `queue.Queue` to be notified by the main thread that the framework has started and stopped training in a new epoch.

The process of the main application on the master node contains the embedded web server to handle HTTP requests from the SPA running in the user's browser. The web server may use one or more threads, but the exact details are hidden by the web server library. The functional element Trace Processor is called by the web server and reads and analyzes the trace files written by `nsys` and the Trace Agent. The Trace Processor uses one or more threads to process the files.

Scalability

Due to training of many nodes results in many data files (one per node written by `nsys` and one or more by the Trace Agent), the Trace Processor must easily scale to all available CPU cores on the machine on which it is running. The data files of each node can be analyzed in parallel. For each node, several queries to obtain the traced CUDA API calls, GPU kernels, their names, and information about the used GPUs can be executed concurrently. Subsequently, detection of each epoch must be done sequentially. Detailed analysis of each epoch can then happen in parallel.

While an earlier prototype of the main application of ML Board was written in Python, the software was ported to the Ada 2012 programming language to obtain the needed performance for the Trace Processor element. Ada 2012 is a systems programming language often used in safety-critical and real-time systems, defining concurrency profiles for increased determinism, absence of deadlocks and priority inversion, and various scheduling algorithms. None of these features, meant for real-time systems, are of interest for the application, however, and it is rather the strong typing, design-by-contract, and its general concurrent programming constructs for which the language is used. (The 2022 revision of the language adds support for parallel programming (using syntax instead of pragmas or library calls), but the latest version of GCC at the time of writing does not yet support this) For example, variables of an array of a task type can be defined for parallel computations. When the variable goes

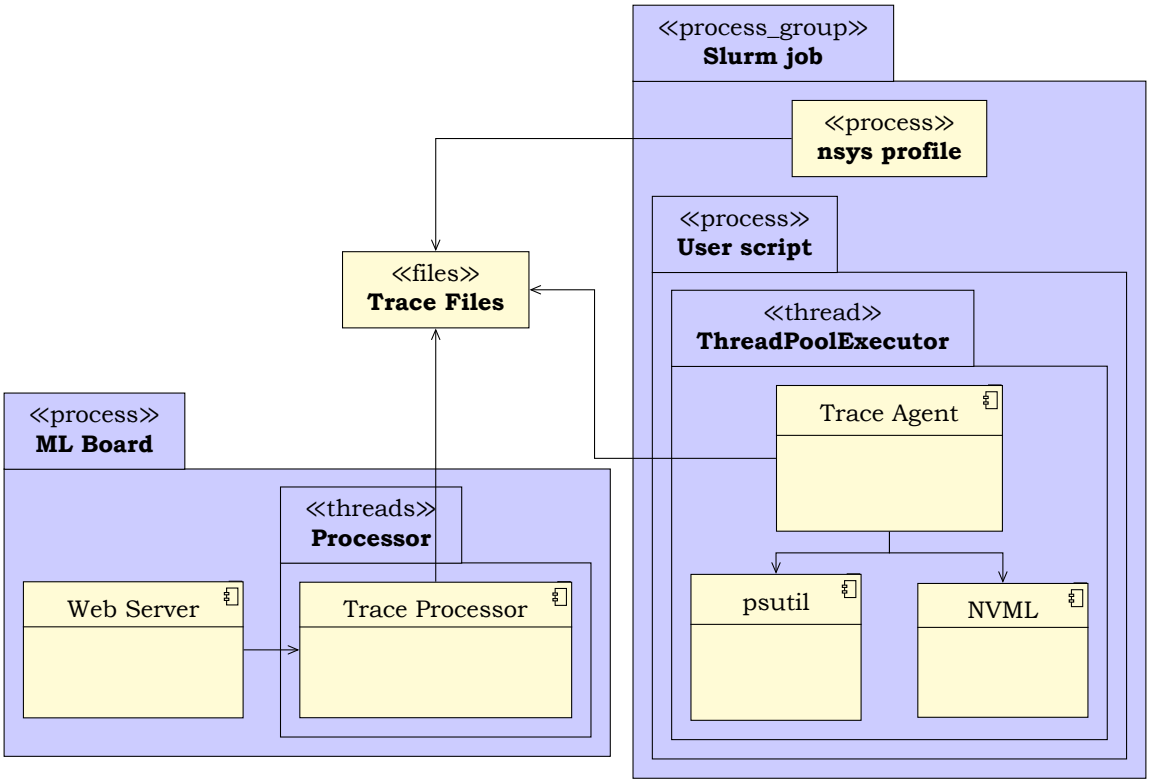


Figure 5.12: *Concurrency diagram showing the processes, threads, and IPC.*

out of scope, the tasks are synchronized; the run-time system waits for all tasks to terminate before the task which defined the variable is allowed to continue. Tasks may be created in the declarative part of packages (a namespace), procedures and functions, blocks, and tasks. Creating tasks inside other tasks can be used for recursion. On Linux tasks are implemented using POSIX Threads. The variable `Processors` on line 12 in Listing 5.1 creates 32 tasks (each stored in a record (a struct)), which are activated when the `begin` keyword is reached on line 13. At line 15 the main task waits for all 32 local tasks to finish processing their data and terminate. This fork-join model is used to process files, some queries, and the detected epochs in parallel. In ML Board the record type `Processor` contains additional fields for input and output data. A function is used to initialize and return a `Processor_Array`.

Listing 5.1: *Creating 32 tasks in a block to process data in parallel.*

```
1  declare
2     type Processor;
3     task type Processor_Task (Data : not null access Processor);
4
5     type Processor is limited record  --  Input/output fields not shown
6        T : Processor_Task (Processor'Access);
7     end record;
8
9     task body Processor_Task is begin ... end Processor_Task;
10
11    type Processor_Array is array (Positive range <>) of Processor;
12    Processors : Processor_Array (1 .. 32);
13 begin
14    null;
15 end;
```

Interprocess communication

The Trace Agent and `nsys profile` 'communicate' with ML Board through files on shared file storage. This mechanism is imposed by `nsys` and requires the file storage to be able to store many large files (several per node).

Replacing `nsys` with using CUPTI directly in the Trace Agent would avoid this file storage requirement and would allow part of the Trace Processor moved to the compute nodes. This would severely decrease the needed bandwidth between the master nodes and the compute nodes because only the results of the analysis need to be transmitted. Furthermore, this would allow real-time continuous monitoring of the training instead of after the user script has finished executing. Replacing the use of `nsys` with CUPTI would be future work and the prototype of ML Board for this thesis uses `nsys`.

5.5 Discussion

The tool fulfills the objective Idleness (Identifying when parts of the system show a reduced utilization or idle) in the user interface mostly for the total amount of idleness between GPU kernels and API calls. This indicates potential increased performance when using, for example, CUDA graphs. The time that it does not make use of the network, CPU utilization, or system and video memory is not considered. Objective Utilization per layer (Showing the utilization of parts of the GPU for the layers of a model) is implemented by showing various metrics in a table and a visualization of the graph of a neural network model. Because `ncu` cannot be used because it causes a severe slowdown, no detailed metrics of CUDA kernels can be obtained and displayed in the user interface. ML Board can display various statistics of common tasks required by objective Common tasks (Showing when a ML framework is loading data, communicating with other nodes, or performing computations, and whether there is any overlap). The power drawn and energy used, as required by objective Power and energy (Determining the power and energy usage of the system can be displayed for the forward and backward phases.

During this thesis, a competing tool [2] specifically for PyTorch was developed by a team at Facebook and Microsoft. It has a similar scope as ML Board, but can show a bit more detailed metrics for kernels due to its use of CUPTI and its integration with PyTorch.

[2] https://pytorch.org/blog/pytorch-profiler-1.9-released/

6 Evaluation

In Chapter 5 the ML Board tool was developed to visualize the utilization of the hardware of a cluster when training a neural network model.

This chapter will evaluate the tool and determine which recommendations can be made to improve the performance of training a model.

This chapter is structured as follows: Section 6.1 describes the research questions this chapter will focus on. the tool is evaluated against different sets of nodes in Section 6.2, datasets in Section 6.3, models in Section 6.4, and batch sizes in Section 6.5. Section 6.6 explains one or more use cases for which ML Board can be used. Section 6.7 provides a discussion of the evaluations.

6.1 Research questions

This chapter will evaluate ML Board to determine in which degree the following research questions of Section 1.1 can be answered:

RQ1 Does the hardware utilization depend on the dataset, the model, hyperparameters, or configuration of the system like number of GPUs and used interconnect?

RQ3 How much time is spend on communication between nodes, loading data, updating parameters, and other aspects in popular large-scale machine learning frameworks?

RQ4 Which recommendations can be given to speed up training of neural network models?

For the evaluation, experiments are run in which different datasets, models, batch sizes, and nodes are used.

6.2 Sets of nodes

ML Board will now be evaluated for two sets of nodes, in two different partitions on the DAS-5: three nodes, each with an RTX 2080 Ti, and sixteen nodes, each with a Titan X. Both sets of nodes trained a ResNet-18 model on the CIFAR-10 dataset and communicated using NCCL over InfiniBand.

3 nodes

When training on three nodes, 70, 71, and 78, each with a single RTX 2080 Ti, Figure 6.1a shows node 78 spends much more time on the `ncclAllReduceRingLLKernel_sum_f32` and `ncclBroadcastRingLLKernel_copy_i8` than the other nodes, like node 71 shown in Figure 6.1b.

To highlight kernels for a node whose average or total duration deviate from the mean, the user must select computing the mean and standard deviation across nodes instead of kernels

(by clicking the horizontal arrow pointing to the right in the user interface) and decrease the standard score (or z-score) threshold by moving the slider to the lowest value of 1.0. Experiments show that the z-score of red colored values are often between 1.0 and 2.0.

These differences are visible in the Communication tab in the Statistics card in Figure 6.2b, where the median time spend on the collective communication operations is 5.5 seconds on node 78, while nodes 70 and 71 require between 3.0 and 3.5 seconds. The time spent on communication is greater than the time needed for evaluating and updating the weights (shown in Figure 6.2a), meaning that the training time, shown in Figure 6.2d, is bound by the (kernels for) communication, not the computation. The idle time before the training phase of the epoch, shown in Figure 6.2c, is inversely related to the training time.

Kernels 070 071 078 σ:

Name	Launches	Total time (s)	Average time (µs)
ncclAllReduceRingLLKernel_sum_f32	1584	41.793	26384.622
ncclAllReduceRingLLKernel_sum_i32	528	0.332	629.399
ncclBroadcastRingLLKernel_copy_i8	1061	10.425	9825.526

(a) *Node 78.*

Kernels 070 071 078 σ:

Name	Launches	Total time (s)	Average time (µs)
ncclAllReduceRingLLKernel_sum_f32	1584	27.704	17489.811
ncclAllReduceRingLLKernel_sum_i32	528	0.187	354.462
ncclBroadcastRingLLKernel_copy_i8	1061	2.678	2524.400

(b) *Node 71.*

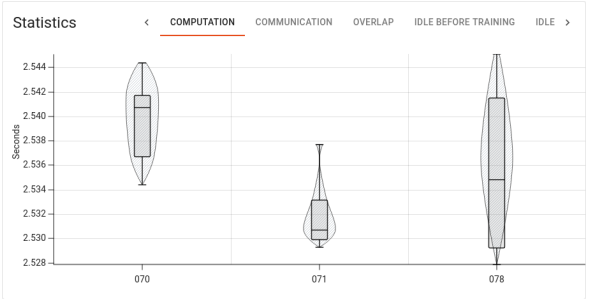
Figure 6.1: *GPU kernels launched by NCCL on nodes 71 and 78. Two of the kernels launched by NCCL on node 78 require significant more time than on the other nodes (z-score > 1.0).*

When selecting the computation kernels in the Kernels card, many kernels for node 78 will be displayed in a red color (except for `dgrad2d_alg1_1` which is shown in green), indicating that node 78 also spends more time than the other two nodes on most computation kernels. However, the absolute difference in seconds is very small and the time spent per epoch on computations, shown in Figure 6.2a, is very similar for all three nodes.

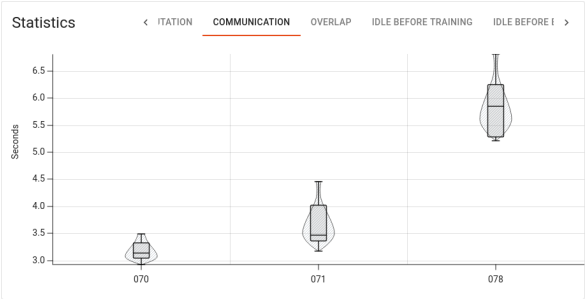
16 nodes

Figure 6.3a shows the time spent executing kernels on the GPU (excluding those launched by NCCL). Most nodes spent less than 0.8 seconds, except node 2, which needs an additional 0.3 seconds. This node repeatedly needed more time for computations in several Slurm jobs.

Figure 6.3b shows the communication time and clearly shows the nodes can be grouped into different sets with a similar median time. Most of the nodes in the sets can be mapped to different parts of the ring constructed by NCCL (see Figure 4.4), except node 4, which spent much less time on communication compared to its neighbors (node 3 and 5) in the ring, and node 47, which needs more time than its neighbors (node 46 and 48). The conclusion that node 1, 2, 3, and 47 are relatively slow is confirmed by the lower idle time before training (shown in Figure 6.3c) and the higher training time (shown in Figure 6.3d).

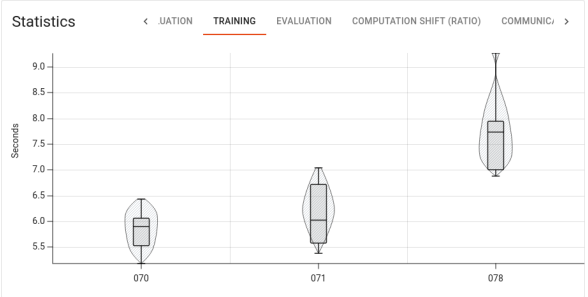


(a) *Computation time.*



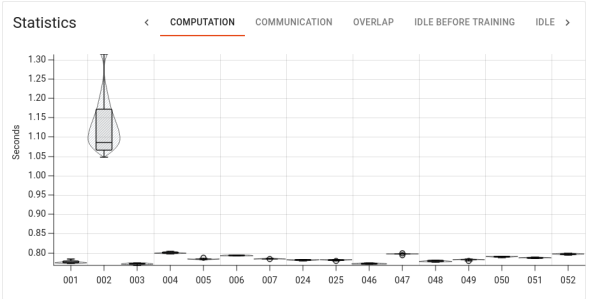
(b) *Communication time.*

(c) *Idle time before training.*

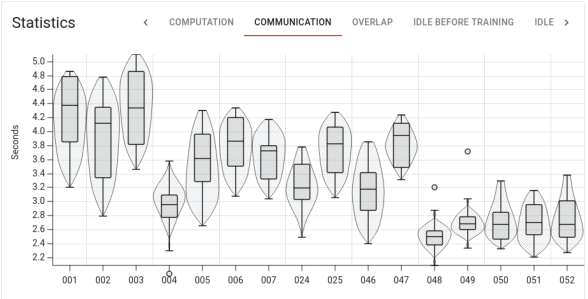


(d) *Training time.*

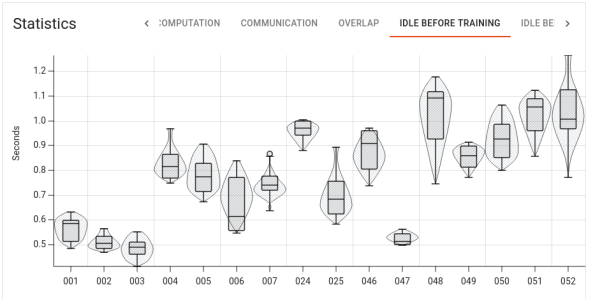
Figure 6.2: *Statistics of 3 nodes, of which node 78 requires more time to train an epoch due to it spending more time on communication than the other two.*



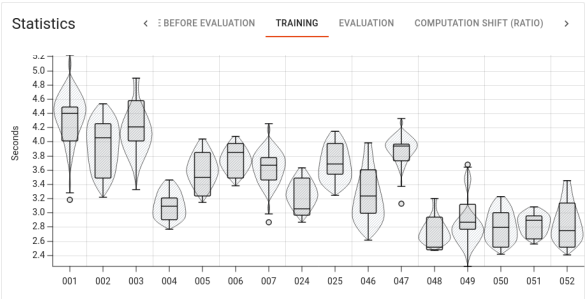
(a) *Computation time.*



(b) *Communication time.*



(c) *Idle time before training.*

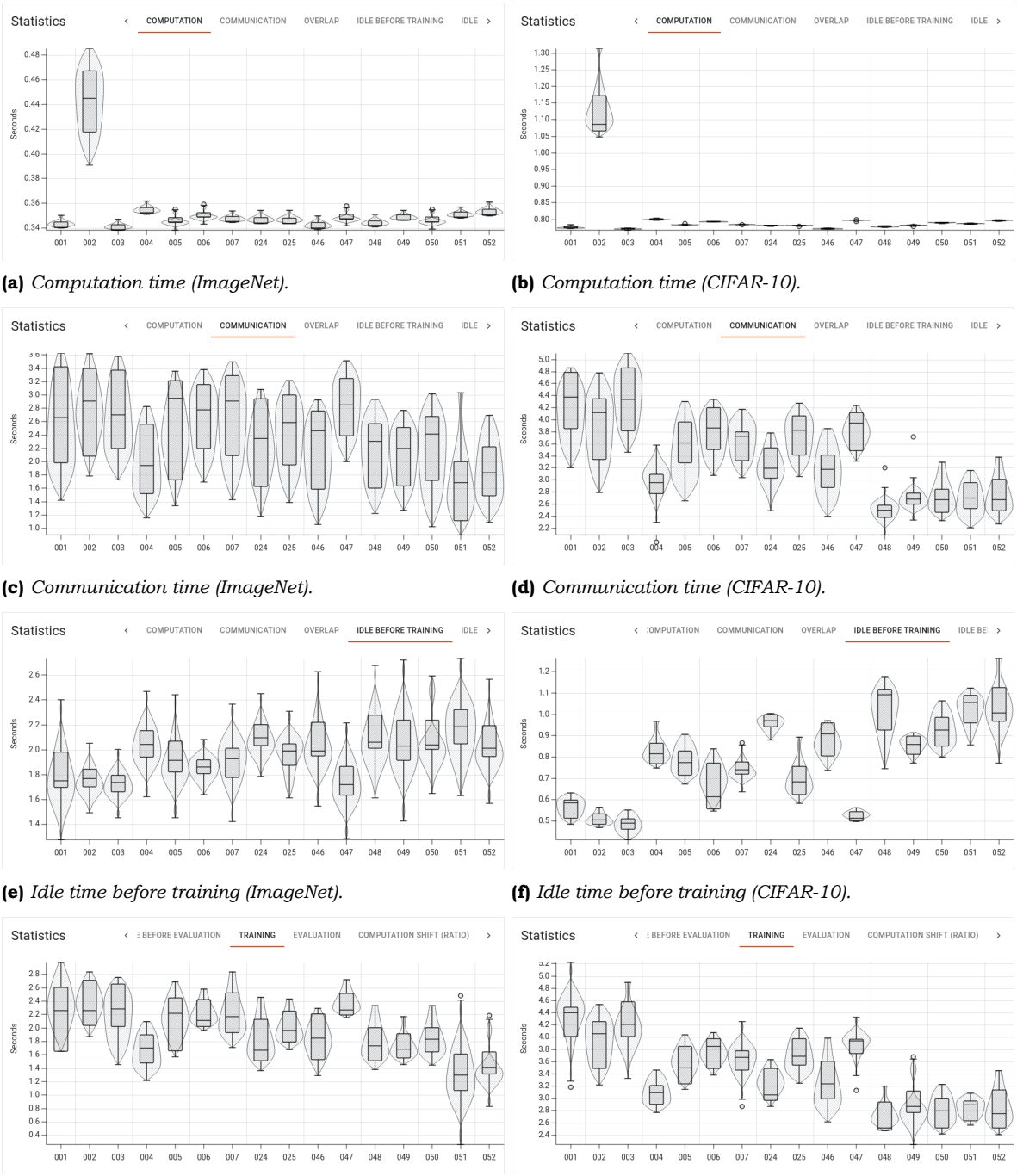


(d) *Training time.*

Figure 6.3: *Statistics of 16 nodes, which can be divided into 3 groups based on the time spend on communication. Node 2 spends more time on evaluating and updating the network, but has no effect on the node's training time.*

6.3 Datasets

Figure 6.4 shows the Statistics card for two jobs training the ResNet-18 model using the ImageNet and CIFAR-10 datasets for 8 epochs on 16 nodes with a Titan X GPU. Both jobs use the NCCL communication backend over InfiniBand. 6.4a and 6.4b clearly show that for both datasets node 2 takes between 30 to 40% more time for the computations. For CIFAR-10, in 6.4d, 6.4f and 6.4h it is easy to divide the nodes into three groups, while for ImageNet, in 6.4c, 6.4e and 6.4g, the separation is less clear, especially between nodes 4-46 and 48-52.



(a) *Computation time (ImageNet).*

(b) *Computation time (CIFAR-10).*

(c) *Communication time (ImageNet).*

(d) *Communication time (CIFAR-10).*

(e) *Idle time before training (ImageNet).*

(f) *Idle time before training (CIFAR-10).*

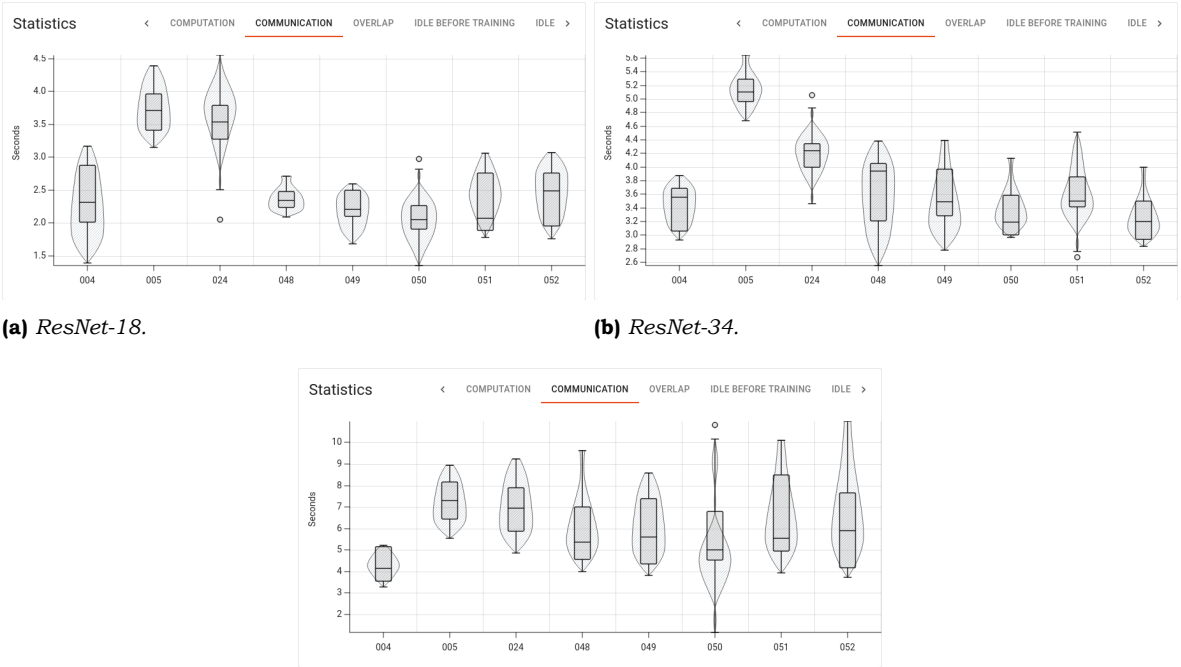
(g) *Training time (ImageNet).*

(h) *Training time (CIFAR-10).*

Figure 6.4: *Statistics card of two jobs using the ImageNet (left column) and CIFAR-10 (right column) datasets.*

6.4 Models

Figure 6.5 shows that when training a CIFAR-10 dataset on 8 nodes with a Titan X GPU using the NCCL backend, the difference between the nodes for the communication kernels slightly differs depending on the used variant of the model: the relative difference between faster and slower nodes when training a ResNet-50 model (6.5c) is slightly less than when training the ResNet-18 (6.5a) or ResNet-34 (6.5b) models. The absolute difference of nodes 5 and 24 is roughly 2 seconds compared to the other nodes for all variants.



(a) *ResNet-18.*

(b) *ResNet-34.*

(c) *ResNet-50.*

Figure 6.5: *Communication time per epoch when training one of the variants of the ResNet model.*

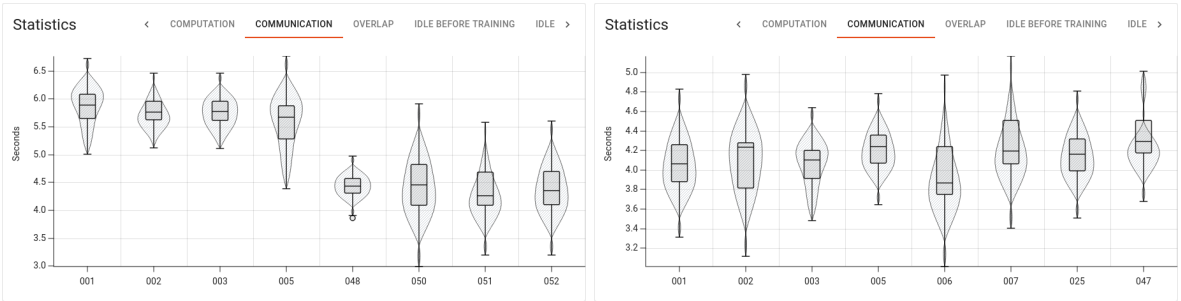
Table 6.1 shows the wallclock time when training several ResNet models on different sets of nodes for the CIFAR-10 and ImageNet datasets. For CIFAR-10, the extra time needed for the nodes of set B, compared to set A, to train a model increases much more for the larger models than for the smaller models, while for ImageNet the effect is more modest. Thus it is important and beneficial to select the nodes of set A when submitting a job to the Slurm job scheduler.

Interestingly, another set of nodes, set C, gives a similar decrease in performance com-

Table 6.1: *Wallclock time in seconds after training the CIFAR-10 and ImageNet datasets on 8 nodes for 8 epochs, using NCCL over InfiniBand interconnect. Set A = 4, 24, 49, 5, 48, 50, 51, 52, B = 1, 2, 3, 5, 48, 50, 51, 52, and C = 1, 2, 3, 5, 6, 7, 25, 47. Percentages indicate the extra time relative to set A.*

		Model					
Dataset	Nodes	ResNet-18		ResNet-34		ResNet-50	
CIFAR-10	A	75.1		108.7		144.7	
	B	85.8	(+14.2%)	155.6	(+43.1%)	263.7	(+82.2%)
	C	76.5	(+1.86%)	150.3	(+38.3%)	263.2	(+81.9%)
ImageNet	A	53.4		59.4		69.2	
	B	57.5	(+7.67%)	71.4	(+20.0%)	80.0	(+15.6%)
	C	51.7	(-3.18%)	62.3	(+4.71%)	71.2	(+2.89%)

pared to set A for the larger models, ResNet-34 and ResNet-50, with the CIFAR-10 dataset, while for ImageNet, the ResNet-18 is slightly faster than set A and only a few percent slower for the larger models. Figure 6.6 shows that the durations of ResNet-50 with ImageNet for set B and C in Table 6.1 is caused by the difference in the time needed for communication, which is surprisingly lower for nodes 1 to 5 in set C (6.6b) than in set B (6.6a). This shows that, although in nearly all cases the set of nodes A results in the lowest wallclock time, set A is not necessarily the only 'fast' set.



(a) *Set B.* **(b)** *Set C.*

Figure 6.6: *Communication time of ResNet-50 using ImageNet, trained on nodes of set B and C used in Table 6.1.*

6.5 Batch sizes

In the next experiments, a ResNet-18 model was trained using the ImageNet dataset with the NCCL backend over InfiniBand on two sets of 3 nodes, each with a single RTX 2080 Ti GPU. The first set consists of the nodes 69, 70, and 71, and the second set of the nodes 70, 71, and 78. Training occurred with a batch size between 64 and 4096 and the energy, power usage, and memory and GPU utilization were measured with NVML at an interval of 10 milliseconds. In both cases the results for nodes 70 and 71 were very similar and are therefore shown only once in the tables below.

6.5.1 Energy

Table 6.2 shows the energy spent per epoch and the efficiency, the energy per second. 6.2a shows that the time and energy needed to train a model for one epoch increases sharply for very small batch sizes like 64 and 128. For larger batch sizes, the energy and time decrease quickly and reach the lowest value for a batch size of 1024.

6.2b shows that, for larger batch sizes, the energy efficiency improves for certain nodes

Table 6.2: *Energy (J) and time (seconds) per epoch, and energy per second (J/s). The least amount of energy or used time is shown in* ***black*** *and the most in red.*

(a) *Energy (J) and time (seconds) per epoch.*

	Nodes				
Size	69	70	71	78	Time
64	2224	1395	1559	1306	67.8
128	1159	741	819	685	35.1
256	659	430	472	382	19.3
512	397	285	317	259	12.2
1024	316	257	279	**229**	**10.6**
2048	333	294	323	258	12.6
4096	428	432	469	372	20.0

(b) *Energy per second (J/s).*

	Nodes			
Size	69	70	71	78
64	32.866	20.559	23.058	**19.407**
128	33.225	21.345	23.286	19.558
256	34.078	22.095	24.615	20.381
512	32.855	23.382	25.640	21.466
1024	29.841	23.916	26.342	21.717
2048	26.416	23.566	24.906	20.527
4096	20.089	21.165	22.914	19.569

Table 6.3: *Total energy (J) per node over 16 epochs and the sum of the total energy of all 3 nodes that participated in the training. The least energy spent is shown in* ***black*** *and the most in red.*

	Nodes				Sum		
Batch size	69	70	71	78	{69, 70, 71}		{70, 71, 78}
64	38103	25536	28435	23560	92074	(+18.6%)	77635
128	21048	14726	16413	13465	52187	(+18.3%)	44130
256	12795	9632	10522	8625	32949	(+15.1%)	28620
512	8690	7278	8020	6552	23988	(+10.3%)	21742
1024	7299	6651	7382	**6075**	21332	(+6.33%)	**20063**
2048	7389	7347	8107	6500	22844	(+5.66%)	21621
4096	9030	9955	11024	8683	30009	(+4.84%)	28624

like node 69, but not for others like nodes 70 and 78, whose measured values have a fairly low standard deviation (less than 1.35 J/s). Table 6.3 shows the total energy of each node over 16 epochs, as well as the sum of the total energy of all 3 nodes. A batch size of 1024 will result in the least amount of energy spent by the GPUs and a lower total energy usage can be obtained (for any batch size) by replacing node 69 with 78. However, a larger batch size will result in a worse obtained accuracy (0.143, 0.145, 0.135, 0.118, 0.112, 0.135, and 0.094 for a batch size from 64 to 4096). Thus an appropriate batch size would be a size between 256 and 1024 (in the case of ImageNet).

6.5.2 Power

The minimum and maximum power drawn by the GPU during training (per epoch) is displayed in Table 6.4. 6.4b shows that for all nodes the maximum power that is drawn increases as the batch size increases. In the case of node 69, the node draws between 50 to 110 W extra compared to the other nodes.

Table 6.4: *Minimum and maximum power (W) per epoch.*

(a) *Minimum power (W) per epoch.*

	Nodes			
Size	69	70	71	78
64	94.8	61.5	68.3	60.8
128	94.3	61.6	68.1	59.3
256	94.2	61.7	68.0	59.0
512	59.5	61.6	68.2	58.6
1024	57.8	61.6	67.9	58.9
2048	56.7	61.5	68.0	58.2
4096	23.1	61.2	68.0	58.5

(b) *Maximum power (W) per epoch.*

	Nodes			
Size	69	70	71	78
64	123.1	72.2	81.4	70.8
128	135.2	77.1	86.5	76.5
256	161.2	87.7	97.0	87.4
512	199.3	108.8	120.0	110.0
1024	248.9	142.5	154.7	146.9
2048	262.0	152.3	164.6	155.8
4096	261.3	156.2	166.2	157.7

The increase in power, as the batch size increases, does not occur for the average power drawn, shown in Table 6.5. For nodes 70, 71, and 78, the highest average power drawn occurs for a batch size of 512, while for node 69 the highest average occurs for a smaller batch size of 256. In the case of node 69, the average generally decreases as the batch size increases.

The values in Tables 6.2b and 6.5 show a similar curve; the peak value (J/s or W) is obtained for a batch size between 512 to 1024, while the lowest and highest batch sizes give a lower value.

Table 6.5: *Average power (W) per epoch.*

	Nodes			
Batch size	69	70	71	78
64	111.417	66.669	75.619	65.192
128	113.184	67.468	76.334	66.057
256	115.224	69.187	77.454	67.369
512	108.844	70.441	79.409	68.580
1024	95.552	70.206	78.481	68.349
2048	78.549	68.150	75.668	65.533
4096	60.994	64.598	71.887	62.085

6.5.3 Memory and GPU utilization

The average memory utilization (data read or written) per epoch is shown in Table 6.6. The maximum of the measured memory utilization is 1% for batch sizes between 64 and 512, and 2, 3, and 5% for a batch of 1024, 2048, 4096. As explained in Section 4.3.1, the reported memory and GPU utilization has a low resolution of whole percentages. The measured values still indicate that the maximum utilization increases as the batch size increases. However, the average utilization is highest for a batch size of 512.

Table 6.6: *Average memory utilization (%).*

	Nodes			
Batch size	69	70	71	78
64	0.556	0.563	0.544	0.985
128	0.619	0.609	0.605	0.971
256	0.652	0.705	0.692	0.892
512	0.737	0.705	0.735	1.174
1024	0.681	0.701	0.636	1.026
2048	0.499	0.468	0.449	0.683
4096	0.276	0.218	0.237	0.379

Table 6.7 shows the average and minimum GPU utilization (percentage of sample period at least one kernel was running) per epoch. (The maximum utilization is between 98 to 100% for any batch size and node). 6.7b shows that for a batch size of 512 or larger, the GPUs become idle at times during training.

Table 6.7: *Average and minimum GPU utilization per epoch (%).*

(a) *Average GPU utilization per epoch (%).*

	Nodes			
Size	69	70	71	78
64	93.0	94.7	95.5	94.6
128	92.7	92.4	93.2	96.6
256	88.8	86.4	88.1	91.4
512	76.2	74.4	74.4	78.1
1024	52.8	49.0	49.1	52.4
2048	24.7	25.4	23.3	26.4
4096	8.8	13.3	11.2	12.4

(b) *Minimum GPU utilization per epoch (%).*

	Nodes			
Size	69	70	71	78
64	71.5	42.5	54.0	86.5
128	53.0	33.5	41.5	64.5
256	15.5	0.0	15.0	33.0
512	0.0	0.0	0.0	0.0
1024	0.0	0.0	0.0	0.0
2048	0.0	0.0	0.0	0.0
4096	0.0	0.0	0.0	0.0

6.6 Use cases

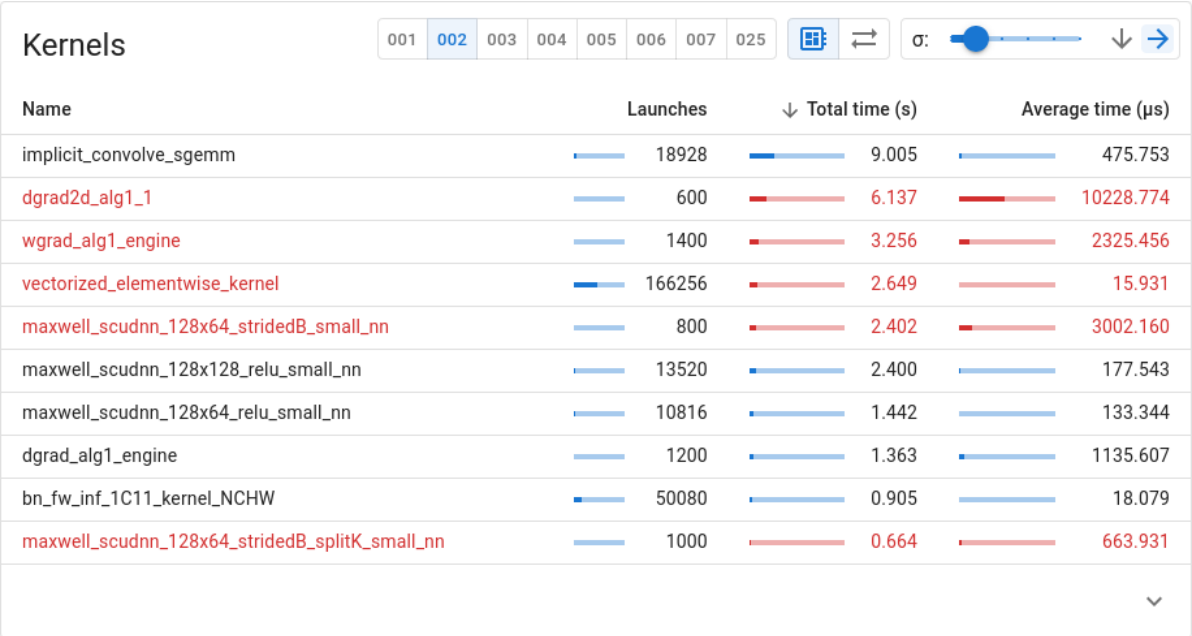
In this section ML Board is evaluated for some use cases: identifying straggling nodes, determining the power and energy usage of individual GPUs, and recommending an appropriate batch size.

6.6.1 Identifying straggling nodes

In the following experiments, ML Board is evaluated to see whether enough information can be obtained to identify straggling nodes, nodes of which the duration of the training phase of each epoch is higher than for other nodes. Excluding these nodes and replacing them with other nodes should then decrease the total duration of training a model for several epochs. Two experiments will be run on eight nodes, with some of the nodes, those which are identified as straggling nodes, replaced by others in the second experiment.

In the following two experiments (called job A and job B), a ResNet-18 model is trained on a CIFAR-10 dataset for eight epochs on eight nodes with one GPU per node (an NVIDIA GeForce GTX Titan X GPU (Maxwell generation)), using the NCCL communication backend over the InfiniBand interconnect. Both jobs evaluate a neural network model, followed by performing distributed training of the model. Only the training phase of each epoch is detected and displayed in the Tasks and Statistics cards in ML Board. For both jobs, boxplots of the computation time, communication time, idle time before the training phase of an epoch, and the duration of the training phases can be viewed in the Statistics card.

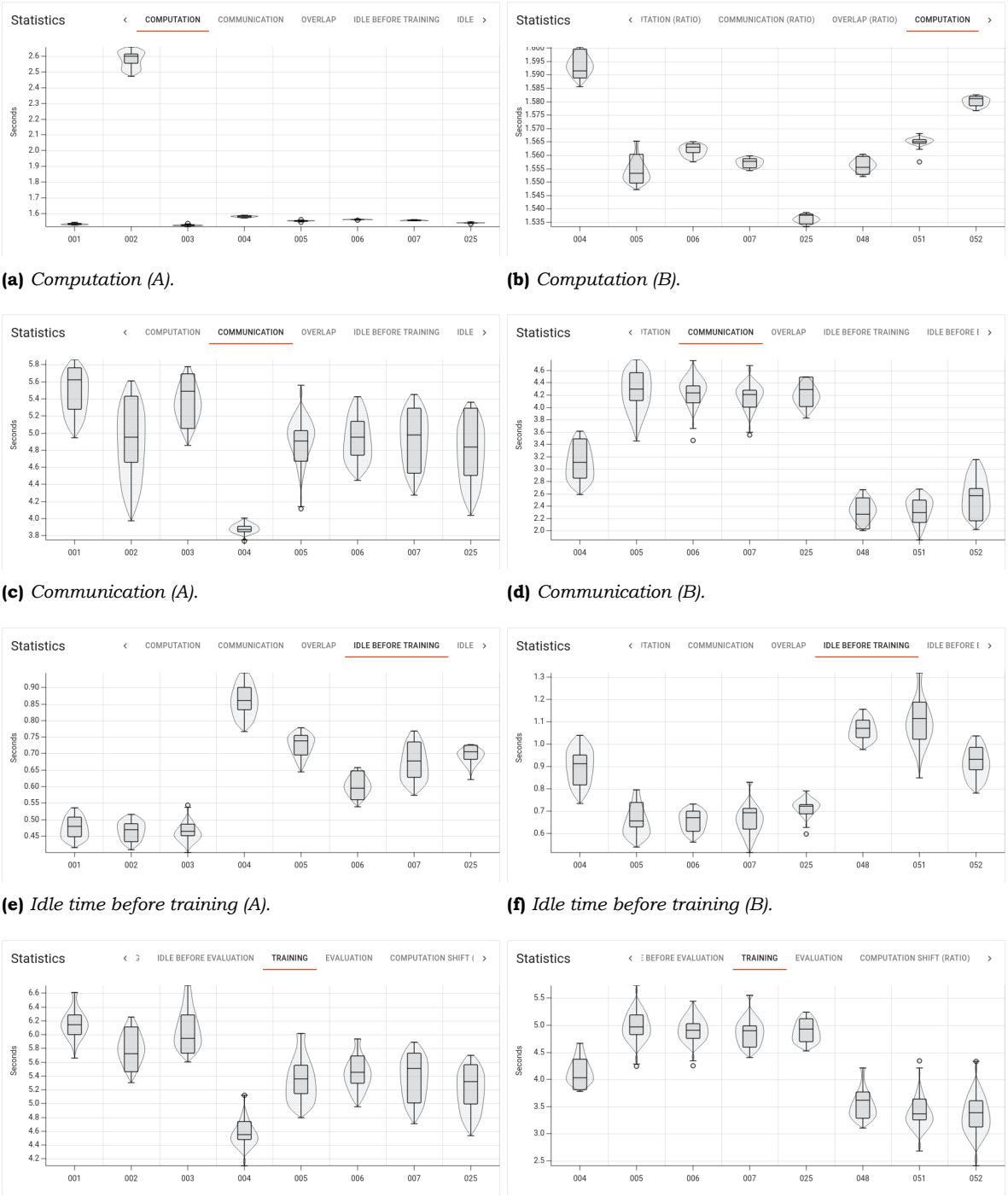
The results of job A are shown in the left column of Figure 6.8. Job A uses nodes 1 up to 7 and 25. In 6.8a it can be clearly seen that node 2 spends about 2.6 seconds per epoch on evaluating and training the network, while the other nodes need roughly 1.5 seconds per epoch. (Kernels responsible for the longer duration of node 2 are shown in the Kernels card in Figure 6.7) However, this does not necessarily mean the duration of the training phase of an epoch is bound by the time spend on computation. In fact, the other subfigures (in particular 6.8c and 6.8g) suggest it is bound by the duration of the GPU kernels launched by the NCCL communication backend of PyTorch. The durations shown in 6.8e and 6.8g are often inversely related; the nodes which need less time to train a model are often more idle before the start of a training phase.



Name	Launches	Total time (s)	Average time (µs)
implicit_convolve_sgemm	18928	9.005	475.753
dgrad2d_alg1_1	600	6.137	10228.774
wgrad_alg1_engine	1400	3.256	2325.456
vectorized_elementwise_kernel	166256	2.649	15.931
maxwell_scudnn_128x64_stridedB_small_nn	800	2.402	3002.160
maxwell_scudnn_128x128_relu_small_nn	13520	2.400	177.543
maxwell_scudnn_128x64_relu_small_nn	10816	1.442	133.344
dgrad_alg1_engine	1200	1.363	1135.607
bn_fw_inf_1C11_kernel_NCHW	50080	0.905	18.079
maxwell_scudnn_128x64_stridedB_splitK_small_nn	1000	0.664	663.931

Figure 6.7: *Node 2 spends (in particular) more time than other nodes on kernels* `wgrad_alg1_engine` *and* `vectorized_elementwise_kernel` *(≈ 2.6 and 1.5 seconds extra compared to the other nodes).*

From 6.8g it can be concluded that nodes 1, 2, and 3 need more time to train the model (between 5.7 and 6.2 seconds) than the other nodes (most have a median between 5.3 and 5.5 seconds), and thus they are good candidates for replacement by three other nodes. For job B the nodes 1, 2, and 3 have been replaced by nodes 48, 51, and 52. (These three nodes were selected using Figure 6.3d) Results of job B are displayed in the right column of Figure 6.8.



(a) *Computation (A).*

(b) *Computation (B).*

(c) *Communication (A).*

(d) *Communication (B).*

(e) *Idle time before training (A).*

(f) *Idle time before training (B).*

(g) *Time of training phase (A).*

(h) *Time of training phase (B).*

Figure 6.8: *The Statistics card of jobs A and B, each with eight nodes with one GPU per node. Three nodes (1, 2, and 3) of job A are replaced by three other nodes (48, 51, and 52) for job B.*

The measured wallclock time of the last epoch shows that the time has decreased by about 5.8% from 81.4 seconds (job A) to 76.6 seconds (job B). Repeating the experiments with the same two sets of nodes consistently shows the time decreases by between 5 and 6%.

6.6.2 Reducing total energy usage

A second use case for the tool is to determine the power and energy usage of the GPUs in order to compute the total energy spent on training a model. Being able to determine if certain GPUs draw much more power than others, can help the user to select a different node or to replace a GPU to lower the total energy spent on training. For this use case the experiments in Section 6.5 are used.

Table 6.2 shows that the energy and energy efficiency can be determined after every epoch and that the GPU of node 69 requires much more energy than the other GPUs, especially for batch sizes of 1024 and lower. Table 6.3 shows that the combination of nodes 69, 70, and 71 requires 4.8% to 18.6% more energy when training a ResNet-18 model on the ImageNet dataset compared to the set in which node 69 was replaced by node 78.

On GPU of the old generations, like the Titan X GPUs, the energy cannot be measured. In this case it is still possible to compare the average power (Table 6.5) and maximum power (Table 6.4b), which is much higher for node 69 than the other nodes.

6.6.3 Selecting a batch size

A third use case is to recommend an appropriate batch size for training a model. This can be done based on the energy spent per epoch (Table 6.2), which recommends a batch size between 512 and 2048 for the lowest amount of energy spent. However, the obtained accuracy after a fixed number of epochs may be lower for larger batch sizes, thus some heuristic should be used to select a lower one.

One such heuristic could be the average or minimum GPU utilization. Table 6.7b shows a batch size of 256 as the largest batch size for which no node has a zero minimum utilization rate, while Table 6.7b shows the average utilization is greater than 74% for batch sizes of 512 or lower. Using this information, ML Board could recommend a batch size of 256 or 512.

6.7 Discussion

Answering research question **RQ3**, several of the previous experiments shows that training time is determined by the communication time, not the time needed to evaluate the neural network and updating the gradients. Most nodes have a fairly consistent (among nodes) duration of the computation kernels, except node 2, which repeatedly needed 30 to 60% extra time (See Figures 6.3a, 6.4a and 6.8a) because it spends more time than other nodes on certain kernels (See Figure 6.7).

To answer **RQ1**, Section 6.3 showed that, for the CIFAR-10 dataset, the nodes can be clearly split into groups using the time needed for communication or the full training time (per epoch), while for the ImageNet dataset the distinction is less clear. Comparing NCCL to Gloo was not possible since Gloo does not execute CUDA kernels on the GPU. Since the current prototype of ML Board does not show the network and GPU memory utilization, it is not yet possible to determine whether model or batch size has any effect.

Section 6.6.1 showed that straggling nodes can be identified and the training time can be decreased between a few percent for smaller models up to roughly 14 to 45% for larger models (depending on the dataset, see Table 6.1) by choosing the right set of nodes. Section 6.6.2 showed that by replacing node 69 with node 78, the total amount of energy needed to train a small model is lowered by 4.6% to 15.7%. In Section 6.6.3 it was shown that based on the energy spent per epoch and the GPU utilization, a batch size of 256 or 512 should be chosen. These are some of the recommendations the tool can make to answer **RQ4**.

7 Conclusion

The last few years has seen a large amount of research in the field of machine learning. However, most research is focused on increasing the accuracy of neural network models, and less on reducing the inefficiencies of distributed training with some of the popular machine learning frameworks.

This thesis has focused on examining and assessing various techniques to measure the performance of the hardware of a HPC cluster while training neural network models with the PyTorch machine learning framework. Some techniques like instrumentation require extensive modifications to a particular framework, making it unsuitable for an application that intends to be framework-agnostic. Other tools may generate files that have a well-known format, making it easy to parse and extract the information that is needed. Often though, these generated files can become very large, which puts a limit on the maximum duration or number of epochs when monitoring the system while training a model. The command line interface is often, but not necessarily, the only way to specify the behavior of a tool and must be done before starting the tool, while libraries can provide functions to change the behavior of the library during runtime. A library which gathers metrics can use callbacks instead of exporting its data to a file. This makes libraries more flexible than external tools and makes it feasible to perform real-time monitoring of the hardware of the system. Another possible limitation of a technique, is that some metrics can only be gathered when a process has the required kernel security capabilities. Without it, only aggregate metrics of some parts of the system can be gathered.

Several of the tools and libraries have been used in ML Board [1], a prototype of a framework-agnostic tool that measures and visualizes the utilization of parts of the system. ML Board uses the NVML library to measure GPU memory utilization, power draw, and energy usage of each GPU, and `nsys` to trace CUDA kernels used to evaluate a neural network model, update the gradients, and collective communication when using the NCCL communication backend. However, it was not able to measure communication when using the Gloo backend, because Gloo did not launch any CUDA kernels. ML Board provides various visualizations to give insight into the utilization of the system. First of all, it can display a table which shows the launched CUDA kernels per node and which of the launched kernels needed the most time in total or on average. The user interface provides the means to compare kernels launched by a node with the same kernels launched by other nodes and to highlight kernels which need much more or much less time than most other nodes. This makes it easy for a user to find nodes which spent more time than others on computations and communication, and which kernels are responsible for the extra time needed by a node. If ML Board were to use CUPTI or a recent version of `nsys` (which was not available on the DAS-5/VU), it would be able to gather additional and more detailed metrics per kernel and display these in the table, and it would be able to avoid having to export large files (created by `nsys`) and perform real-time monitoring of the training instead of providing an analysis of a limited number of epochs after

[1] The Git repository of ML Board can be found at https://github.com/markpk/mlboard

training has completed. Furthermore, the ratio of common tasks performed during training, such as evaluating a model, updating gradients, and performing collective communication can be displayed in ML Board. Additionally, statistics in the form of a boxplot and a violin plot (which shows the distribution) can be displayed per node for several metrics, including, among others, the computation and communication durations, and the total duration to train a model per epoch. This gives the user another way to compare how different nodes perform with respect to a specific metric.

ML Board was evaluated to determine which recommendations to the user could be presented using the information gathered about the system. One of the use cases is that the tool can be used to identify straggling nodes. Selecting different nodes to be used by subsequent Slurm jobs can decrease the training time at least a few percent up to between 14 to 45% depending on the used dataset and model. Another use case is measuring the energy used by the GPUs when training a model. Replacing a GPU by selecting a different node can reduce the energy needed for training by between 5 to 16%. A third use case is identifying an appropriate batch size based on the used energy and the GPU utilization. With ML Board it is shown that the performance of the hardware of the system can be measured and that analysis of the measurements can be used to adapt the training and the usage of the hardware (by changing hyperparameters or by replacing hardware) to each other, potentially dynamically at runtime (in future work).

7.1 Discussion of research questions

The following is a summary of the conclusions of the research questions from Section 1.1:

RQ1 *Does the hardware utilization depend on the dataset, the model, hyperparameters, or configuration of the system like number of GPUs and used interconnect?*

The hardware utilization depends on the dataset, number of nodes, the number of GPUs per node, and the used collective communication backend. Tables 3.2 and 3.3 shows that the collective communication backend has a large effect on the achieved speed-up (based on the number of nodes or total number of GPUs). Furthermore, NCCL is able to use NVLink, which would speed up communication between two GPUs on the same node. For some datasets like CIFAR-10, some nodes spend more time on communication than other nodes, while this effect is less strong for other datasets like ImageNet. The difference is also less for larger models like ResNet-50 than for a smaller model like ResNet-18. The used interconnect did not appear to cause any difference when using the NCCL backend because of the efficient bandwidth usage of the Ring AllReduce operation.

RQ2 *Which parts of the system can be measured and visualized?*

The use of `nsys` or CUPTI provides the ability to trace CUDA kernels launched on the GPU, including those launched by NCCL. CUPTI should be able to provide detailed metrics of launched kernels, while NVML can be used to measure the power usage and memory utilization. Because Gloo uses the CPU instead of executing kernels on the GPU, its usage is not so easy to measure, and may require instrumentation of the specific ML framework in order to be able to measure it.

Whether parts of the system like the network interfaces and the file systems can be measured and visualized depends partly on whether Linux kernel capabilities can be obtained.

Without kernel capabilities, aggregated statistics of the network interfaces can still be gathered using sysfs, but not which nodes communicate with each other and when. More detailed statistics can be measured by using `LD_PRELOAD` to intercept calls to shared libraries like NCCL. sysfs can also be used to measure main memory usage.

With kernel capabilities and a recent version of Linux, eBPF can be used for detailed collection of metrics related to network traffic as well as file systems for measuring the

loading of files. While the needed capabilities for eBPF are unlikely to be given to regular users on a shared system like the DAS-5, virtual machines provided by commercial cloud services may provide a root user. Therefore, a future version of ML Board would benefit from using the combination of tools in set A instead of C in Table 4.3.

RQ3 *How much time is spend on communication between nodes, loading data, updating parameters, and other aspects in popular large-scale machine learning frameworks?*

Most time during training of ResNet models is spent on collective communication, rather than the evaluation of the network and updating of the gradients itself.

RQ4 *Which recommendations can be given to speed up training of neural network models?*

ML Board is able to show which nodes that participate in the training are straggling and can be candidates for replacement by other nodes in future scheduled jobs. Table 6.1 shows that selecting the right nodes can have a profound effect on the runtime of a job, with some sets of nodes having a runtime which is between 14.2% and 82.2% greater than other sets. However, not all environments allow the user to manually select the used nodes. In this case, if real-time monitoring is supported (see Section 7.2), ML Board is at least able to show whether certain nodes are straggling after a few epochs. Using this information, the user can then quickly decide whether to continue running the job or to restart it, depending on the scheduler of the environment to select different set of nodes.

7.2 Future Work

This thesis has investigated various techniques and proposed a tool to measure the performance of a system running machine learning workloads. There are several aspects of the tool that could be improved in the future:

1. One limitation of the current prototype of ML Board is that it makes use of various technologies from NVIDIA due to the omnipresence of NVIDIA GPUs in data centers such as the DAS-5/VU. To extend the tool to systems using AMD GPUs or Intel's new discrete GPUs, support for AMD's ROCm and Intel's oneAPI software stacks could be added. Alternatively, direct support for Vulkan or OpenCL could be added if they have the necessary extensions to measure the performance of the GPUs.

2. ML Board visualizes the utilization of GPUs only. It would be useful to also be able to visualize the utilization of specialized hardware like Google's Edge TPUs. These can do inferencing of some neural network models and can be connected to the system through USB or as a PCIe accelerator.

3. On modern systems with a recent Linux kernel, ML Board could be modified to use eBPF through the use of the BCC toolkit [9]. This would allow ML Board to more precisely measure the performance of the network, block devices like hard drives, and (network) file systems. However, eBPF requires elevated privileges, which may require architectural changes to ML Board such as splitting part of it into a separate process that runs as a daemon or with setuid access rights.

4. The current prototype of ML Board requires offline processing of some of the collected data. As explained in Section 5.4.3, using CUPTI would avoid storing large files on disks and would enable real-time continuous monitoring of the training of a model. Furthermore, with real-time monitoring, an API could be provided that would give applications feedback so that they can make adjustments during training.

5. Machine learning frameworks which use AllReduce for collective communication do not provide fault-tolerance to handle straggling or failing nodes. Support for the Parameter Server (PS) architecture could be added to ML Board. This would allow it to measure and

visualize communication and staleness of nodes which run frameworks which use PS instead of AllReduce. With support for Parameter Servers and CUPTI, ML Board would be able to provide more detailed metrics of the performance of machines and their GPUs when running frameworks such as KubeML [2] on Kubernetes[2] clusters.

[2] https://kubernetes.io/

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al., TensorFlow: A system for large-scale machine learning, In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[2] Diego Albo Martinez, Kubeml: An efficient serverless platform for scalable deep learning, Master's thesis, TU Delft, 2021, URL: http://resolver.tudelft.nl/uuid:5fe8d907-a98c-4364-b594-69ebb044767e.

[3] Manish Arora, The architecture and evolution of CPU-GPU systems for general purpose computing, *By University of California, San Diago*, 27, 2012.

[4] Henri Bal, Dick Epema, Cees de Laat, Rob van Nieuwpoort, John Romein, Frank Seinstra, Cees Snoek, and Harry Wijshoff, A medium-scale distributed system for computer science research: Infrastructure for the long term, *IEEE Computer*, 49(5):54–63, may 2016.

[5] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. YOLOv4: Optimal speed and accuracy of object detection, 2020, arXiv:2004.10934 [cs.CV].

[6] Guoguo Chen, Carolina Parada, and Georg Heigold, Small-footprint keyword spotting using deep neural networks, In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4087–4091. IEEE, 2014.

[7] François Chollet, Xception: Deep learning with depthwise separable convolutions, In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017, arXiv:1610.02357 [cs.CV].

[8] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter, Fast and accurate deep network learning by exponential linear units (elus), *ICLR*, 2015, arXiv:1511.07289 [cs.LG].

[9] IO Visor Project contributors, BPF compiler collection (BCC), 2022, GitHub:iovisor/bcc.

[10] Andrew Danowitz, Kyle Kelley, James Mao, John P Stevenson, and Mark Horowitz, CPU DB: Recording microprocessor history: With this open database, you can mine microprocessor trends over the past 40 years., *Queue*, 10(4):10–27, 2012.

[11] Julien Demouth, Cuda pro tip: Minimize the tail effect, 6 2014, URL: https://developer.nvidia.com/blog/cuda-pro-tip-minimize-the-tail-effect/.

[12] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei, ImageNet: A large-scale hierarchical image database, In *2009 IEEE conference on Computer Vision and Pattern Recognition*, pages 248–255. IEEE, 2009, doi:10.1109/CVPR.2009.5206848.

[13] Max Franz, Christian T. Lopes, Gerardo Huck, Yue Dong, Onur Sumer, and Gary D. Bader, Cytoscape.js: a graph theory library for visualisation and analysis, *Bioinformatics*, 32(2):309–311, 09 2015. ISSN 1367-4803, doi:10.1093/bioinformatics/btv557.

[14] Max Franz, Manfred Cheung, Harry Adel, Alexander Li, YikaJ, Ethan Keller, Sam Kortchmar, Bjørn Fjukstad, Jascha Ephraim, N8th8n8el, and Zachary Blumenfeld, cytoscape/cytoscape.js-dagre, January 2022, doi:10.5281/zenodo.5911364, GitHub:cytoscape/cytoscape.js-dagre.

[15] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014, arXiv:1406.2661 [stat.ML].

[16] Google, Perfetto - system profiling, app tracing and trace analysis, 2019, URL: https://perfetto.dev/docs/.

[17] Dan Graur, Evaluating and improving large-scale machine learning frameworks, Master's thesis, TU Delft, 2019, URL: http://resolver.tudelft.nl/uuid:ea9d655d-eabd-4409-9d06-0e10dd7124ef.

[18] Brendan Gregg, Linux 4.9's efficient BPF-based profiler, 2016, URL: https://www.brendangregg.com/blog/2016-10-21/linux-efficient-profiler.html.

[19] Song Han, Huizi Mao, and William J. Dally, Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, *ICLR*, 2016, arXiv:1510.00149 [cs.CV].

[20] Horace He, The state of machine learning frameworks in 2019, *The Gradient*, 2019.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015, arXiv:1512.03385 [cs.CV].

[22] Lionel Heng, Benjamin Choi, Zhaopeng Cui, Marcel Geppert, Sixing Hu, Benson Kuan, Peidong Liu, Rang Nguyen, Ye Chuan Yeo, Andreas Geiger, Gim Hee Lee, Marc Pollefeys, and Torsten Sattler, Project AutoVision: Localization and 3D scene perception for an autonomous vehicle with a multi-camera system, In *2019 International Conference on Robotics and Automation (ICRA)*, pages 4695–4702. IEEE, 2019, arXiv:1809.05477 [cs.RO], doi:10.1109/ICRA.2019.8793949.

[23] Brody Huval, Tao Wang, Sameep Tandon, Jeff Kiske, Will Song, Joel Pazhayampallil, Mykhaylo Andriluka, Pranav Rajpurkar, Toki Migimatsu, Royce Cheng-Yue, Fernando Mujica, Adam Coates, and Andrew Y. Ng. An empirical evaluation of deep learning on highway driving, 2015, arXiv:1504.01716 [cs.RO].

[24] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size, 2016, arXiv:1602.07360 [cs.CV].

[25] Sergey Ioffe and Christian Szegedy, Batch normalization: Accelerating deep network training by reducing internal covariate shift, In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[26] Sylvain Jeaugey, NCCL 2.0, *GTC*, 2017, URL: http://on-demand.gputechconf.com/gtc/2017/presentation/s7155-jeaugey-nccl.pdf.

[27] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, Tiegang Chen, Guangxiao Hu, Shaohuai Shi, and Xiaowen Chu. Highly scalable deep learning training system with mixed-precision: Training ImageNet in four minutes, 2018, arXiv:1807.11205 [cs.LG].

[28] Daniel A. Jiménez and Calvin Lin, Dynamic branch prediction with perceptrons, In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pages 197–206. IEEE, 2001, doi:10.1109/HPCA.2001.903263.

[29] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, Jiyan Yang, Jongsoo Park, Alexander Heinecke, Evangelos Georganas, Sudarshan Srinivasan, Abhisek Kundu, Misha Smelyanskiy, Bharat Kaul, and Pradeep Dubey. A study of BFLOAT16 for deep learning training, 2019, arXiv:1905.12322 [cs.LG].

[30] Tero Karras, Samuli Laine, and Timo Aila, A style-based generator architecture for generative adversarial networks, *CVPR*, 2019, arXiv:1812.04948 [cs.NE], GitHub:NVlabs/stylegan.

[31] Alex Krizhevsky, Geoffrey Hinton, et al., Learning multiple layers of features from tiny images, 2009.

[32] Geoff Langdale and Daniel Lemire, Parsing gigabytes of JSON per second, *The VLDB Journal*, 28(6):941–960, Oct 2019. ISSN 0949-877X, arXiv:1902.08318 [cs.dB], doi:10.1007/s00778-019-00578-5.

[33] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel, Backpropagation applied to handwritten zip code recognition, *Neural computation*, 1(4):541–551, 1989.

[34] Yann LeCun, Lawrence D Jackel, Léon Bottou, Corinna Cortes, John S Denker, Harris Drucker, Isabelle Guyon, Urs A Muller, Eduard Sackinger, Patrice Simard, et al., Learning algorithms for classification: A comparison on handwritten digit recognition, *Neural networks: the statistical mechanics perspective*, 261(276):2, 1995.

[35] Charles E Leiserson, Neil C Thompson, Joel S Emer, Bradley C Kuszmaul, Butler W Lampson, Daniel Sanchez, and Tao B Schardl, There's plenty of room at the top: What will drive computer performance after moore's law?, *Science*, 368(6495):eaam9744, 2020.

[36] LF AI & Data Foundation, Open neural network exchange, 2022, GitHub:onnx, URL: https://onnx.ai/.

[37] Qing Li, Weidong Cai, Xiaogang Wang, Yun Zhou, David Dagan Feng, and Mei Chen, Medical image classification with convolutional neural network, In *2014 13th International Conference on Control Automation Robotics & Vision (ICARCV)*, pages 844–848. IEEE, 2014, URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7064414.

[38] Mark W Maier, David Emery, and Rich Hilliard, Ansi/ieee 1471 and systems engineering, *Systems engineering*, 7(3):257–270, 2004.

[39] Michael Meyer and Georg Kuschk, Automotive radar dataset for deep learning based 3D object detection, In *2019 16th European Radar Conference (EuRAD)*, pages 129–132. IEEE, 2019.

[40] Stefan K Muller and Jan Hoffmann, Modeling and analyzing evaluation cost of CUDA kernels, *Proceedings of the ACM on Programming Languages*, 5(POPL):1–31, 2021, doi:10.1145/3434306.

[41] NVIDIA, Getting Started with CUDA Graphs, 2019, URL: https://developer.nvidia.com/blog/cuda-graphs/.

[42] NVIDIA, NVIDIA Nsight Compute, 2021, URL: https://developer.nvidia.com/nsight-compute.

[43] NVIDIA, NVIDIA Nsight Systems, 2021, URL: https://developer.nvidia.com/nsight-systems.

[44] NVIDIA, NVIDIA CUDA profiling tools interface (CUPTI) - CUDA toolkit, 2021, URL: https://developer.nvidia.com/cupti.

[45] NVIDIA, NVIDIA Management Library (NVML), 2022, URL: https://developer.nvidia.com/nvidia-management-library-nvml.

[46] OpenAI, :, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning, 2019, arXiv:1912.06680 [cs.LG].

[47] Giampaolo Rodola, psutil, 2022, GitHub:giampaolo/psutil.

[48] Andres Rodriguez, Eden Segal, Etay Meiri, Evarist Fomenko, Y Jim Kim, Haihao Shen, and Barukh Ziv, Lower numerical precision deep learning inference and training, *Intel White Paper*, 3, 2018.

[49] Élie Roudninski, netintercept, 2021, GitHub:marmeladema/netintercept.

[50] Nick Rozanski and Eoin Woods. *Software systems architecture: working with stakeholders using viewpoints and perspectives*. Addison-Wesley, 2012.

[51] Karl Rupp, Microprocessor trend data, 2020, GitHub:karlrupp/microprocessor-trend-data.

[52] Daniel Rákos, Efficient gaussian blur with linear sampling, 2010, URL: https://www.rastergrid.com/blog/2010/09/efficient-gaussian-blur-with-linear-sampling.

[53] Daniel Rákos, SIMD in the GPU world, 2022, URL: https://www.rastergrid.com/blog/gpu-tech/2022/02/simd-in-the-gpu-world.

[54] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. MobileNetV2: Inverted residuals and linear bottlenecks, 2019, arXiv:1801.04381 [cs.CV].

[55] Grigory Sapunov, FP64, FP32, FP16, BFLOAT16, TF32, and other members of the zoo, 2020, URL: https://medium.com/@moocaholic/fp64-fp32-fp16-bfloat16-tf32-and-other-members-of-the-zoo-a1ca7897d407.

[56] Johan Schalkwyk, Doug Beeferman, Françoise Beaufays, Bill Byrne, Ciprian Chelba, Mike Cohen, Maryam Kamvar, and Brian Strope, "your word is my command": Google search by voice: A case study, In *Advances in speech recognition*, pages 61–90. Springer, 2010, URL: https://research.google/pubs/pub36340.pdf.

[57] SchedMD, Slurm workload manager, 2021, URL: https://slurm.schedmd.com.

[58] SchedMD, Generic resource (GRES) scheduling, 2022, URL: https://slurm.schedmd.com/gres.html.

[59] L Sifre and S Mallat, *Rigid-motion scattering for image classification*, PhD thesis, 2014.

[60] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al., Mastering the game of go with deep neural networks and tree search, *nature*, 529(7587):484–489, 2016.

[61] Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, and Jan S. Rellermeyer, A survey on distributed machine learning, *ACM Computing Survey*, 53(2):1–33, mar 2020. ISSN 0360-0300, doi:10.1145/3377454.

[62] Zijie J Wang, Robert Turko, Omar Shaikh, Haekyu Park, Nilaksh Das, Fred Hohman, Minsuk Kahng, and Duen Horng Polo Chau, Cnn explainer: Learning convolutional neural networks with interactive visualization, *IEEE Transactions on Visualization and Computer Graphics*, 27(2):1396–1406, 2020, arXiv:2004.15004 [cs.HC], doi:10.1109/TVCG.2020.3030418.

[63] Chhavi Yadav and Léon Bottou, Cold case: The lost MNIST digits, In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, arXiv:1905.10498 [cs.LG], GitHub:facebookresearch/qmnist.

[64] Yang You, Igor Gitman, and Boris Ginsburg, Large batch training of convolutional networks, 2017, arXiv:1708.03888 [cs.CV], doi:10.48550/ARXIV.1708.03888.

[65] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. ImageNet training in minutes, 2018, arXiv:1709.05011 [cs.CV].

[66] Yi Zhu, Analyze driving scenes with GluonCV 0.8, 2020, URL: https://medium.com/apache-mxnet/analyze-driving-scenes-with-gluoncv-0-8-56ac37664977.

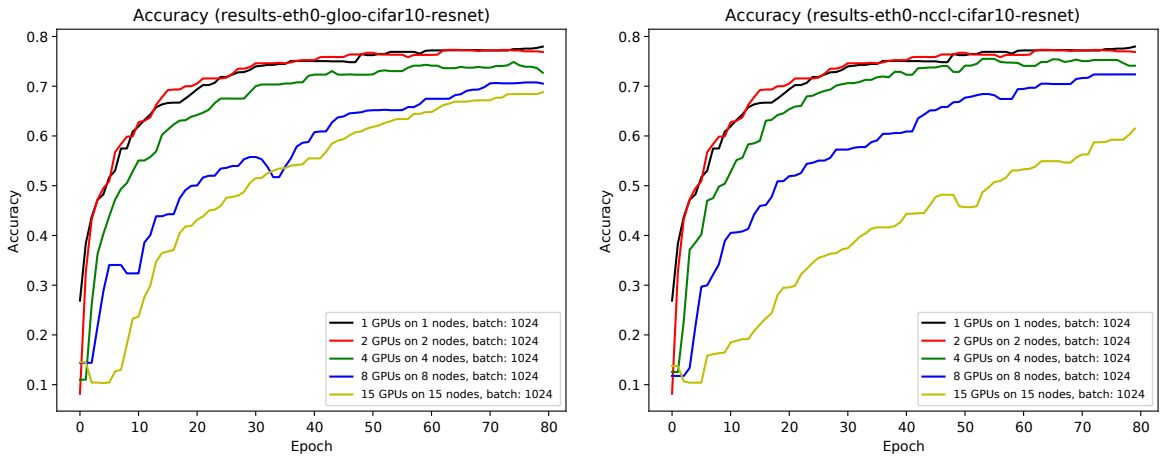
[67] Yi Zhu, Zhongyue Zhang, Chongruo Wu, Zhi Zhang, Tong He, Hang Zhang, R. Manmatha, Mu Li, and Alexander Smola. Improving semantic segmentation via self-training, 2020, arXiv:2004.14960 [cs.CV].

A

Results

A.1 Communication backends single-GPU multi-node

CIFAR-10

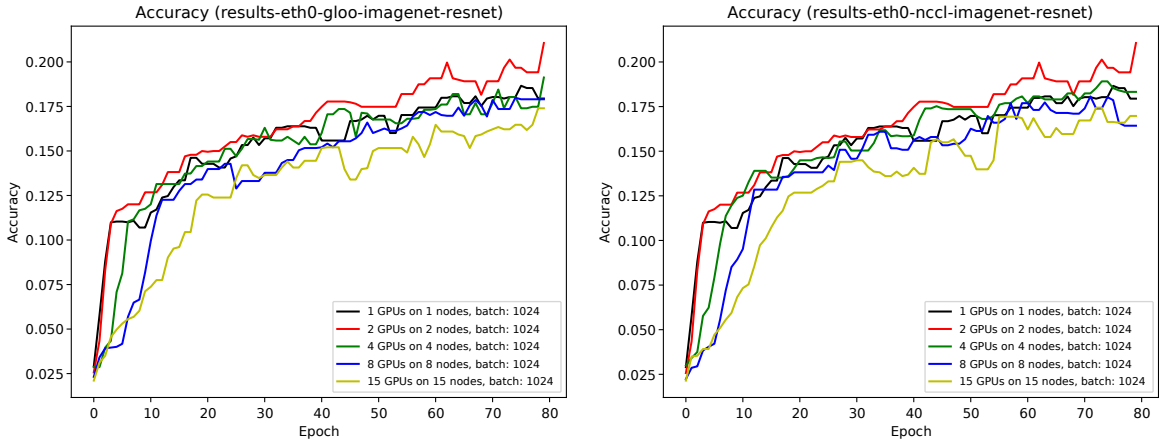


(a) *Gloo communication backend.*

(b) *NCCL communication backend.*

Figure A.1: *Accuracy as a function of epoch on 1, 2, 4, 8, or 15 nodes, with 1 GPU per node, using the Ethernet interconnect, and ResNet-18 model with CIFAR-10 dataset.*

ImageNet



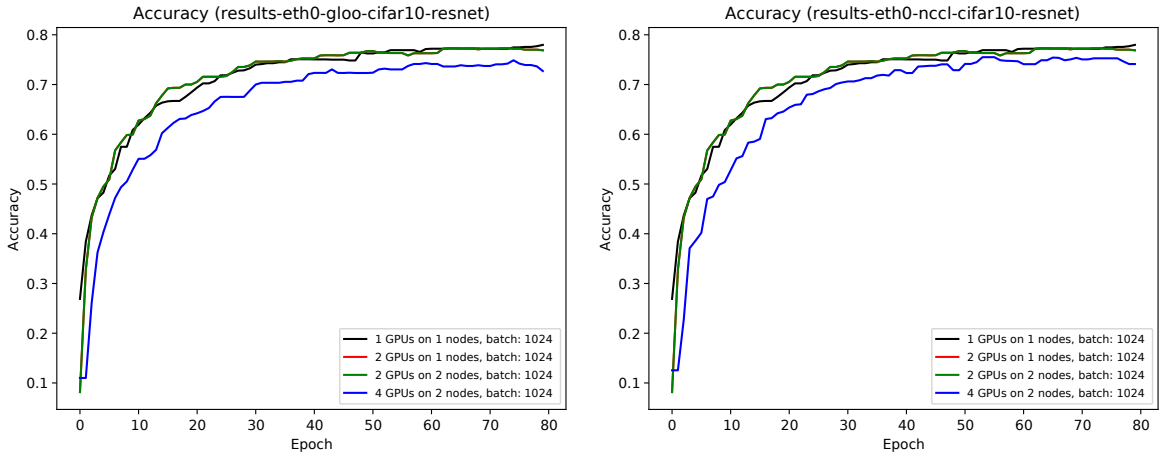
(a) *Gloo communication backend.*

(b) *NCCL communication backend.*

Figure A.2: *Accuracy as a function of epoch on 1, 2, 4, 8, or 15 nodes, with 1 GPU per node, using the Ethernet interconnect, and ResNet-18 model with ImageNet dataset.*

A.2 Communication backends multi-GPU multi-node

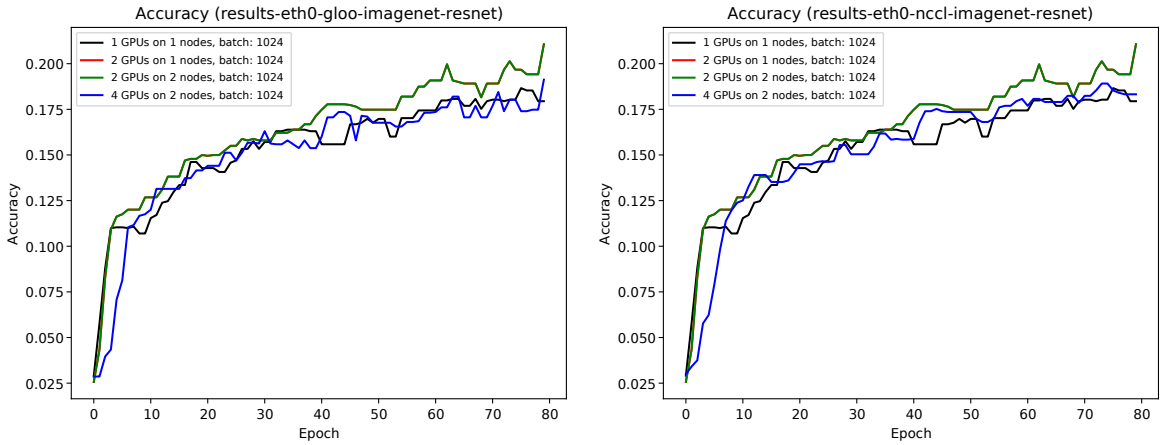
CIFAR-10



(a) *Gloo communication backend.* **(b)** *NCCL communication backend.*

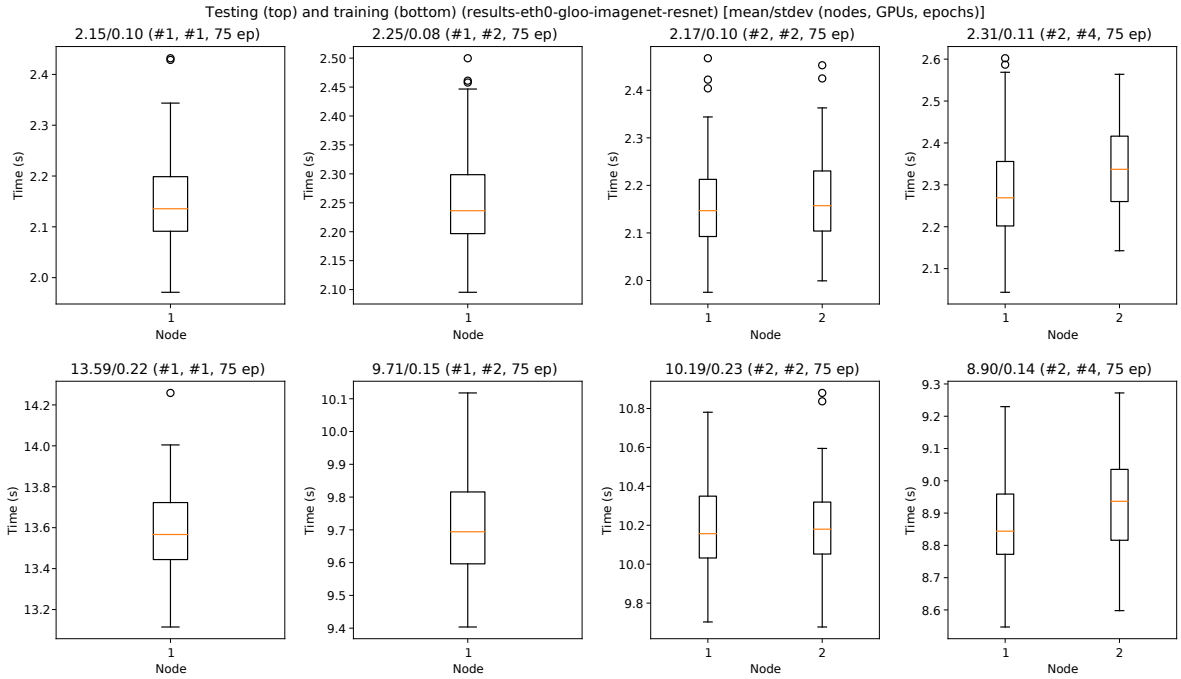
Figure A.3: *Accuracy as a function of epoch on 1 and 2 nodes, with 1 and 2 GPUs per node, using the Ethernet interconnect, and ResNet-18 model with CIFAR-10 dataset.*

ImageNet

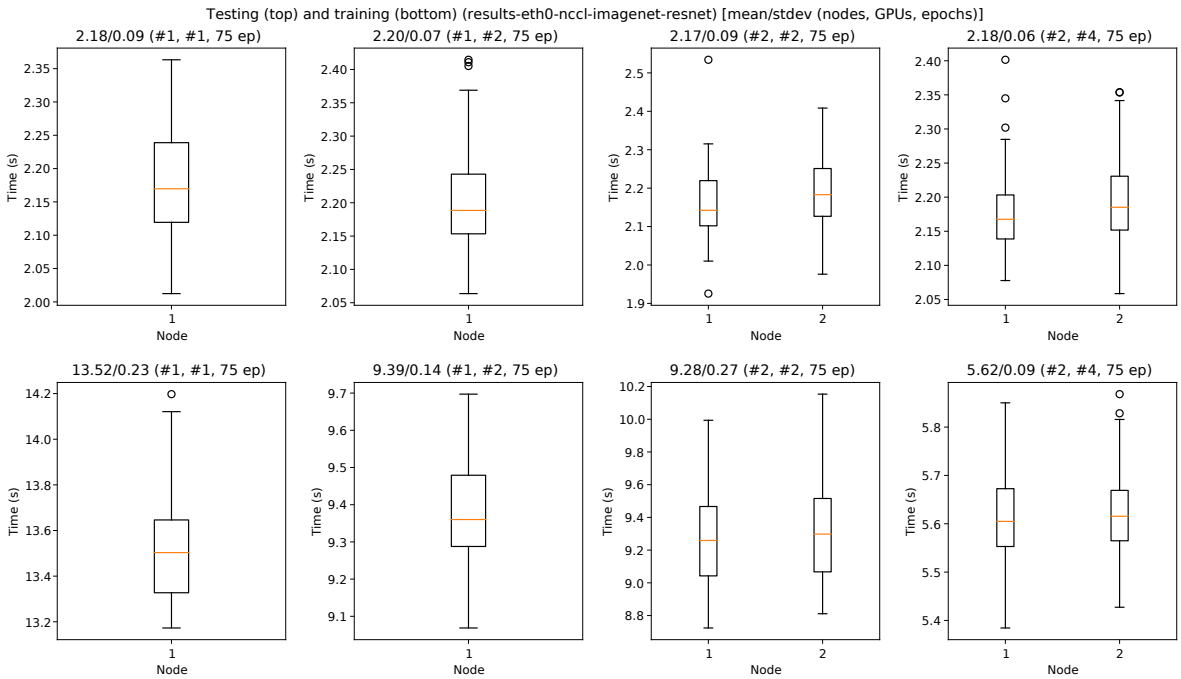


(a) *Gloo communication backend.* **(b)** *NCCL communication backend.*

Figure A.4: *Accuracy as a function of epoch on 1 and 2 nodes, with 1 and 2 GPUs per node, using the Ethernet interconnect, and ResNet-18 model with ImageNet dataset.*



(a) *Gloo communication backend.*

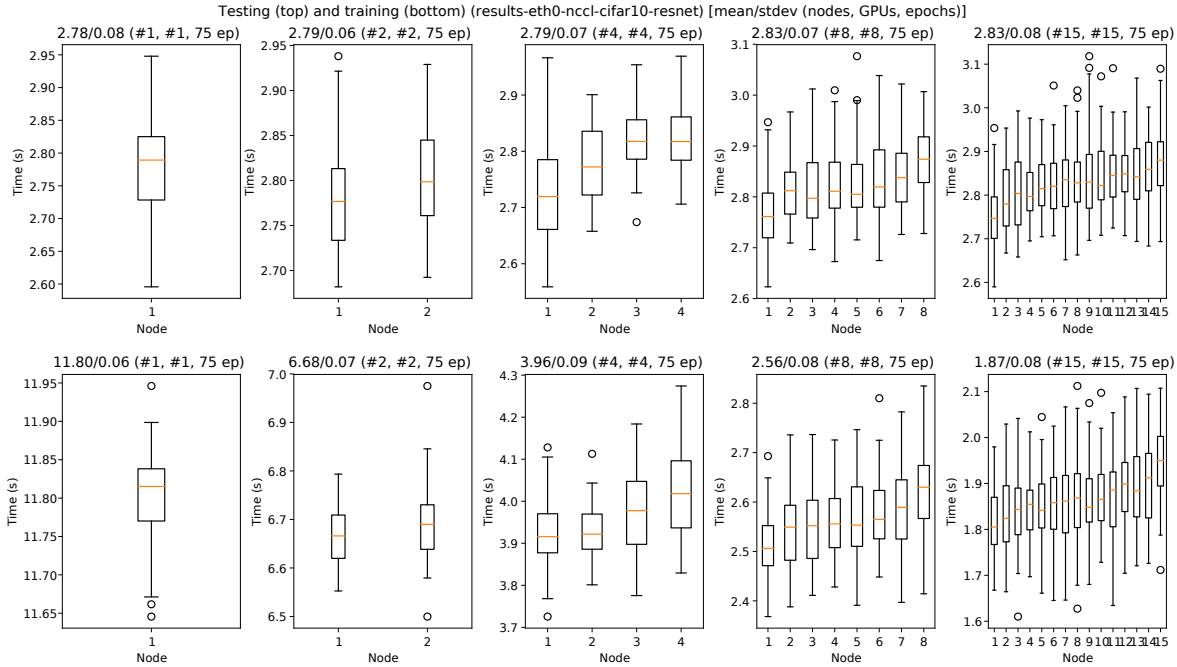


(b) *NCCL communication backend.*

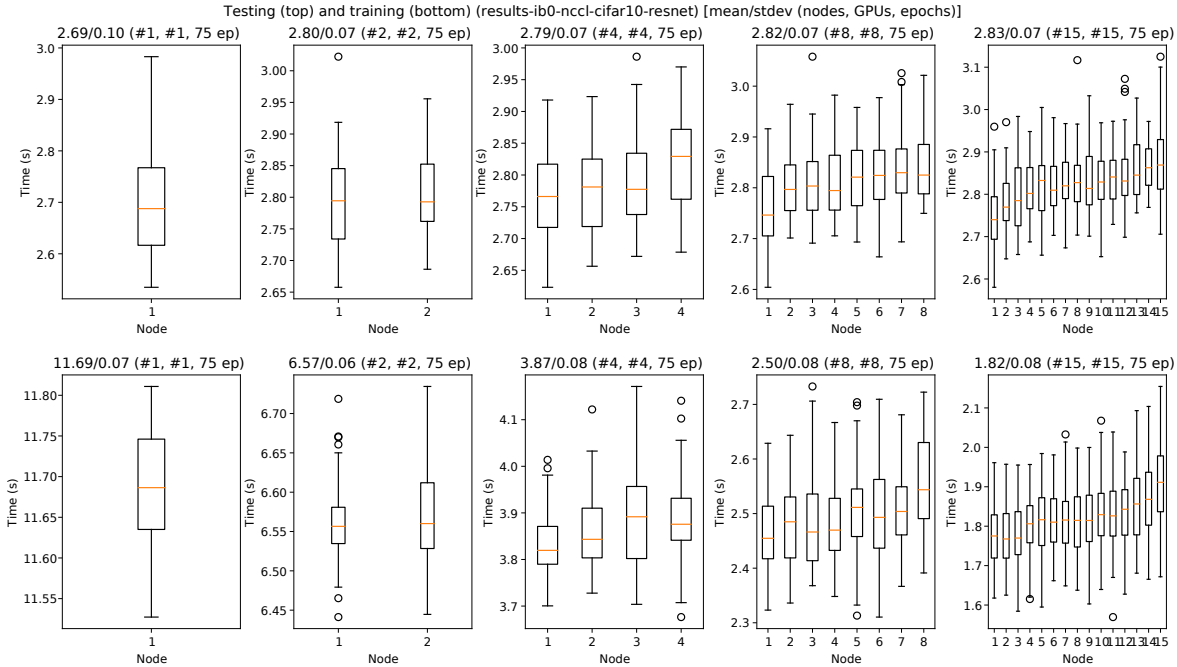
Figure A.5: *Wallclock time for testing and training on 1 and 2 nodes, with 1 and 2 GPUs per node, using an Ethernet interconnect, and ResNet-18 model with ImageNet dataset.*

A.3 Interconnects single-GPU multi-node

CIFAR-10

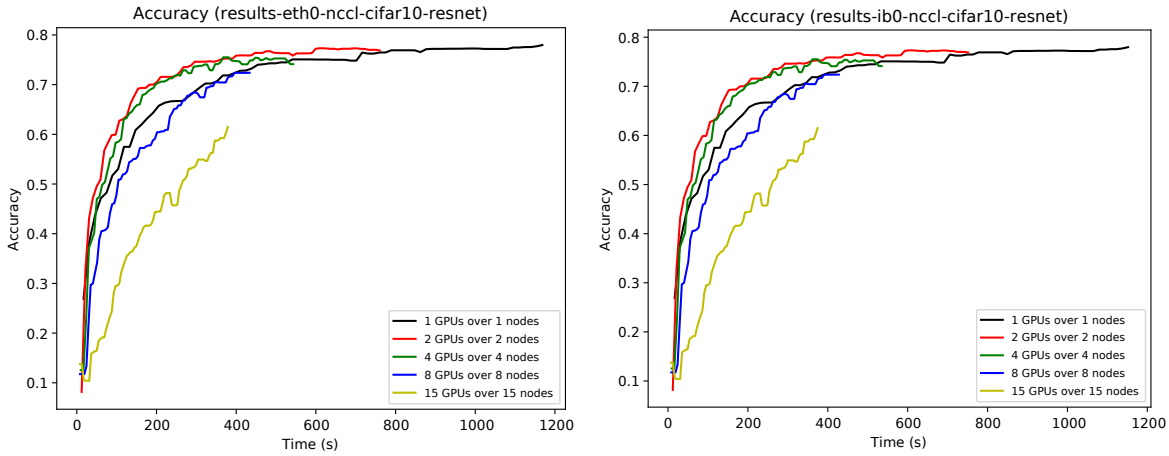


(a) *Ethernet interconnect.*



(b) *InfiniBand interconnect.*

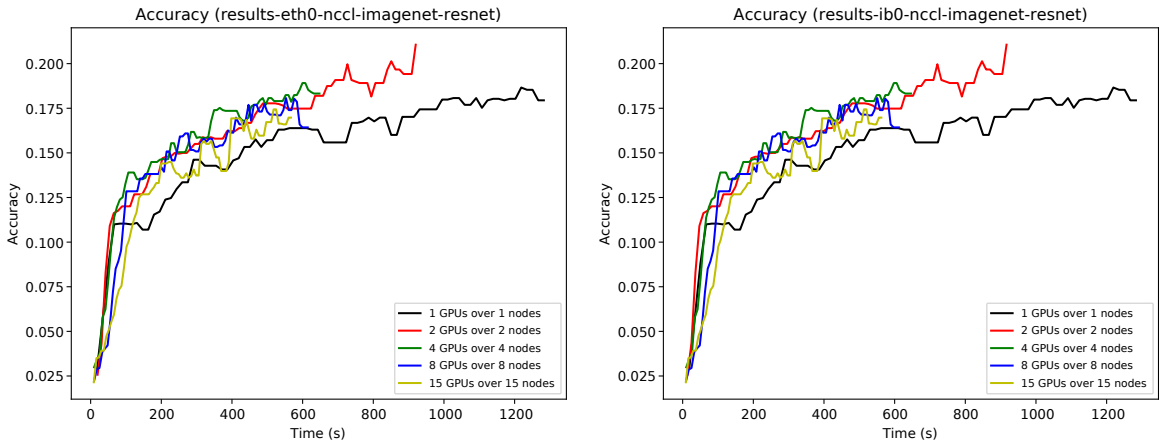
Figure A.6: *Wallclock time for testing and training on 1, 2, 4, 8, or 15 nodes, with 1 GPU per node, using the NCCL library for communication, and ResNet-18 model with CIFAR-10 dataset.*



(a) *Ethernet interconnect.* **(b)** *InfiniBand interconnect.*

Figure A.7: *Accuracy as a function of time on 1, 2, 4, 8, or 15 nodes, with 1 GPU per node, using the NCCL library for communication, and ResNet-18 model with CIFAR-10 dataset.*

ImageNet

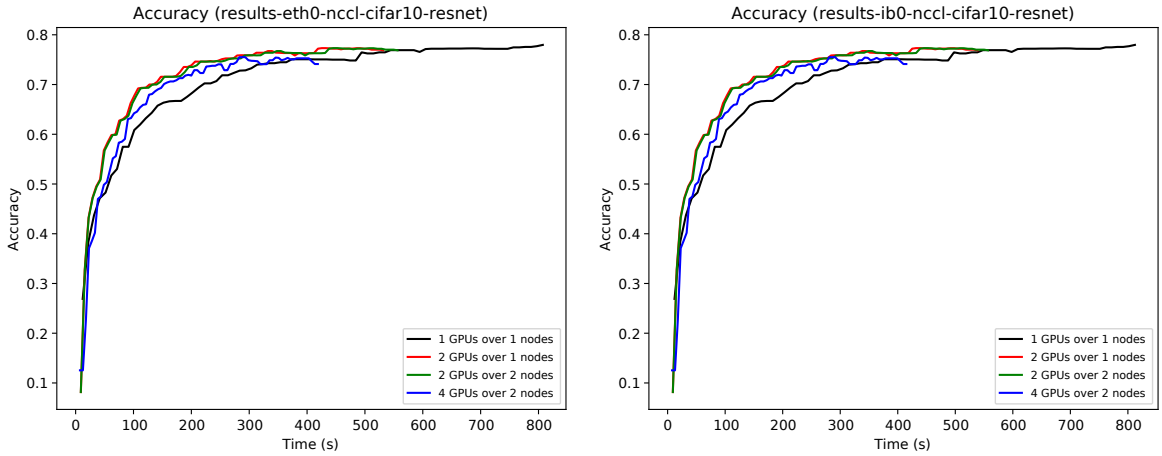


(a) *Ethernet interconnect.* **(b)** *InfiniBand interconnect.*

Figure A.8: *Accuracy as a function of time on 1, 2, 4, 8, or 15 nodes, with 1 GPU per node, using the NCCL library for communication, and ResNet-18 model with ImageNet dataset.*

A.4 Interconnects multi-GPU multi-node

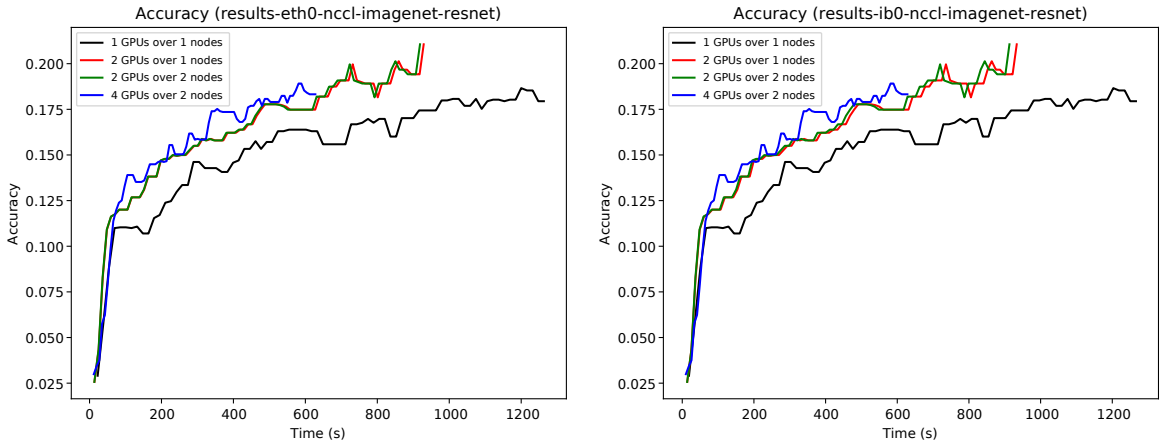
CIFAR-10



(a) *Ethernet interconnect.* **(b)** *InfiniBand interconnect.*

Figure A.9: *Accuracy as a function of epoch on 1 and 2 nodes, with 1 and 2 GPUs per node, using the NCCL library for communication, and ResNet-18 model with CIFAR-10 dataset.*

ImageNet



(a) *Ethernet interconnect.* **(b)** *InfiniBand interconnect.*

Figure A.10: *Accuracy as a function of time on 1 and 2 nodes, with 1 and 2 GPUs per node, using the NCCL library for communication, and ResNet-18 model with ImageNet dataset.*