T2R2 東京工業大学リサーチリポジトリ
Tokyo Tech Research Repository

論文 / 著書情報
Article / Book Information

題目(和文)	大規模畳み込みニューラルネットワークの階層的なハイブリッド並列学習
Title(English)	Hierarchical Hybrid Parallel Training of Large-Scale Convolutional Neural Networks
著者(和文)	大山洋介
Author(English)	Yosuke Oyama
出典(和文)	学位:博士(理学), 学位授与機関:東京工業大学, 報告番号:甲第11892号, 授与年月日:2021年3月26日, 学位の種別:課程博士, 審査員:松岡 聡,増原 英彦,遠藤 敏夫,脇田 建,横田 理央
Citation(English)	Degree:Doctor (Science), Conferring organization: Tokyo Institute of Technology, Report number:甲第11892号, Conferred date:2021/3/26, Degree Type:Course doctor, Examiner:,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Powered by T2R2 (Tokyo Institute Research Repository)

Hierarchical Hybrid Parallel Training of Large-Scale Convolutional Neural Networks
（大規模畳み込みニューラルネットワークの階層的なハイブリッド並列学習）

Yosuke Oyama (大山洋介)
oyama.y.aa@m.titech.ac.jp

A Doctoral Thesis Submitted to
School of Computing, Tokyo Institute of Technology

Supervisor: Prof. Satoshi Matsuoka

2021/02/27

Abstract

In the last decades, deep learning technology has attracted substantial research interests. Deep learning has been empowered by the advance of network architecture and training algorithms, such as the growth of available data to train deep neural networks, and the increase of the computation capability of high-performance GPUs and supercomputers. Specifically, many studies successfully have adopted data-parallelism to distribute their training workloads among dozens and even hundreds of accelerators due to its simple parallelization design. As the demand for training deep learning models with complex and massive data increases continuously, these improvements must take place in parallel to keep up research speed. However, trends in hardware, software, and model architecture for deep learning are changing rapidly. For example, recent accelerators equip specialized hardware in matrix multiplication operations that are frequently used in deep learning, but their performance is not fully investigated for various deep learning workloads. Another change is the emergence of high-resolution, large-scale models that perform end-to-end learning on scientific data, which cannot be trained with conventional data-parallel methods. For these reasons, there is a strong demand for a general-purpose method of accelerating and saving memory for more diverse model architectures, computational precisions, and large-scale models on multiple levels of parallelism.

In this thesis, we propose acceleration algorithms that maximize the parallel efficiency of CNNs at two different levels, intra-processor and inter-processor parallelism. For intra-processor optimization, we present the μ-cuDNN library, which applies loop optimization and adaptively uses different algorithms and computational precisions for convolution kernels. This library replaces the cuDNN library transparently to optimize the convolution performance, which is the de-facto standard kernel library for deep learning frameworks, and thus, our library is widely applicable for such frameworks. Since convolution is one of the most computationally-intensive parts of deep learning training and inference, accelerating convolutional computation plays a crucial role in accelerating such jobs in many fields. Our loop optimization method allows users to select a broader range of convolutional algorithms without changing computation semantics. We combine the μ-cuDNN library into two frameworks, Caffe and TensorFlow, and show that it achieves reasonable performance improvements in several different CNNs; we achieve speedups of 1.60x for AlexNet and 1.30x for ResNet-18 on an NVIDIA V100 GPU. We also show that μ-cuDNN achieves speedups of up to 4.54x, and 1.60x on average for DeepBench convolutional layers, and demonstrate that NVIDIA GPU's single-precision arithmetic units are still beneficial to accelerate convolution on half-precision float data. These results indicate that micro-batching can seamlessly increase deep learning performance while using the same overall memory footprint. Moreover, we propose an interface to use this information for multi-node training using the property that μ-cuDNN can obtain layer parameters via the cuDNN interface. Furthermore, we show that the ONNX data format can be combined with our algorithm to apply the loop optimization algorithm without changing the framework itself. We also discuss whether μ-cuDNN's algorithm can be extended to layer types other than convolution and different memory layouts.

For inter-processor optimization, we present an end-to-end hybrid-parallel training approach for strong-scaling training large-sample 3D CNNs, which applies spatial partitioning. Since data-parallel training frameworks cannot train large models beyond the memory capability of a single GPU, this approach enables users to improve the inference accuracy of such CNNs by increasing the input dimensions. Specifically, we propose various techniques to optimize the performance for better scalability; we show GPU kernels designed for 2D CNNs can be inefficient, and thus, implementation of custom kernels for high-dimensional layers are necessary. Moreover, we propose a spatial-partitioned sample I/O method to mitigate the data read overhead, which is essential to achieve practical strong scaling, such as spatial-partitioned sample I/O. We demonstrate training on full-resolution samples for the CosmoFlow network (512^3) and the 3D U-Net (256^3). Our performance results show good strong and weak scaling on up to 2048 NVIDIA V100 GPUs; we achieve 1.77x of speedup on 2048 GPUs over 512 GPUs with the same mini-batch size of 64 for the CosmoFlow network and achieve 1.42x of speedup on 512 GPUs over 256 GPUs with the same mini-batch size of 16 for the 3D U-Net. We also propose a performance model for the hybrid-parallel training framework to demonstrate that its performance is predictable with benchmark results using a limited number of compute nodes. Besides, we present a significant improvement in prediction accuracy by using full-resolution data of the CosmoFlow cosmological data.

This thesis provides a means to accelerate the training of CNNs across multiple levels of parallelism extremely through our proposed approaches. We expect this to discover new scientific knowledge by training large-scale, high-dimensional, and high-resolution CNNs in highly parallel environments.

Acknowledgment

I want to thank my advisor, Prof. Satoshi Matsuoka, for his continuous advice and suggestions. I would also thank my co-authors, Tal Ben-Nun, Prof. Torsten Hoefler at ETH Zurich, and Naoya Maruyama, Nikoli Dryden, Brian Van Essen, and other members of the LBANN development group at Lawrence Livermore National Laboratory for their dependable support to promote the research. Finally, I would like to thank all members and staff of our laboratory, my family and friends for their huge support.

Chapter 4 and any related sections of this thesis cover the contents of our paper "The Case for Strong Scaling in Deep Learning: Training Large 3D CNNs with Hybrid Parallelism" submitted to the IEEE Transactions on Parallel & Distributed Systems. I, Yosuke Oyama, is responsible for all of the contents and the experiments described in this thesis, but other co-authors of the paper are also involved in implementing and maintaining our source code as a part of the Livermore Big Artificial Neural Network Toolkit (LBANN) open-source software and its dependent libraries.

This research was supported by JSPS KAKENHI Grant Number JP18J22858, Japan.
In Chapter 3 and any related parts in other chapters:

- The work in Chapter 3 was partially supported by the ETH Zurich Student Summer Research Fellowship.
- Part of the work in Chapter 3 is conducted as research activities of AIST - Tokyo Tech Real World Big- Data Computation Open Innovation Laboratory (RWBC-OIL).

In Chapter 4 and any related parts in other chapters:

- Prepared by LLNL under Contract DE-AC52-07NA27344 (LLNL-TH-819967).
- This research was supported by the Exascale Computing Project (17-SC-20-SC).
- Experiments were performed at the Livermore Computing facility.
- This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231.

Publications

Publications submitted as the first author are highlighted with underlines.

Refereed Conference Papers

- **Yosuke Oyama**, Akihiro Nomura, Ikuro Sato, Hiroki Nishimura, Yukimasa Tamatsu, Satoshi Matsuoka, "Predicting Statistics of Asynchronous SGD Parameters for a Large-Scale Distributed Deep Learning System on GPU Supercomputers", In *Proceedings of the 2016 IEEE International Conference on Big Data (BigData 2016)*, Washington D.C., Dec. 5-8, 2016.
- Ikuro Sato, Ryo Fujisaki, **Yosuke Oyama**, Akihiro Nomura, Satoshi Matsuoka, "Asynchronous, Data-Parallel Deep Convolutional Neural Network Training with Linear Prediction Model for Parameter Transition", In *Proceedings of the 24th International Conference on Neural Information Processing (ICONIP 2017)*, Nov. 14-18, 2017.
- **Yosuke Oyama**, Tal Ben-Nun, Torsten Hoefler, Satoshi Matsuoka, "Accelerating Deep Learning Frameworks with Micro-batches", In *Proceedings of the 2018 IEEE International Conference on Cluster Computing (Cluster 2018)*, Belfast UK, Sep. 10-13, 2018.
- **Yosuke Oyama**, Naoya Maruyama, Nikoli Dryden, Erin McCarthy, Peter Harrington, Jan Balewski, Satoshi Matsuoka, Peter Nugent, Brian Van Essen, "The Case for Strong Scaling in Deep Learning: Training Large 3D CNNs with Hybrid Parallelism", In *IEEE Transactions on Parallel & Distributed Systems (TPDS)*, vol. 32, no. 7, pp. 1641-1652, Jul, 2021.
- (To appear) Jens Domke, Emil Vatai, Alexsandr Drozd, Peng Chen, **Yosuke Oyama**, Lingqi Zhang, Shweta Salaria, Daichi Mukunoki, Artur Podobas, Mohamed Wahib, Satoshi Matsuoka, "Matrix Engines for High Performance Computing: A Paragon of Performance or Grasping at Straws?", Submitted for *International Parallel and Distributed Processing Symposium (IPDPS 2021)*, 2021.

Reviewed Posters

- **Yosuke Oyama**, Tal Ben-Nun, Torsten Hoefler, Satoshi Matsuoka, "u-cuDNN: Accelerating Deep Learning Frameworks with Micro-Batches", In *GPU Technology Conference 2019 (GTC*

2019), Silicon Valley, May. 17-21, 2019.

- **Yosuke Oyama**, Naoya Maruyama, Nikoli Dryden, Peter Harrington, Jan Balewski, Satoshi Matsuoka, Marc Snir, Peter Nugent, Brian Van Essen, "Toward Training a Large 3D Cosmological CNN with Hybrid Parallelization", In *48th International Conference on Parallel Processing (ICPP 2019)*, Kyoto, Aug. 5-8, 2019.

Talk

- **Yosuke Oyama**, Naoya Maruyama, Nikoli Dryden, Peter Harrington, Jan Balewski, Satoshi Matsuoka, Marc Snir, Peter Nugent, Brian Van Essen, "Toward Training a Large 3D Cosmological CNN with Hybrid Parallelization", In *The 1st Workshop on Parallel and Distributed Machine Learning 2019 (PDML'19), in 48th International Conference on Parallel Processing (ICPP 2019)*, Kyoto, Aug. 5, 2019.

Unreviewed Papers / Workshop Manuscripts

- **Yosuke Oyama**, Tal Ben-Nun, Torsten Hoefler, Satoshi Matsuoka, "μ-cuDNN: Accelerating Deep Learning Frameworks with Micro-Batching", In *arXiv e-prints*, 2018.
- 土川稔生, **大山洋介**, 野村哲弘, 松岡聡, "機械学習による計算機トレースの自動生成", In *並列/分散/協調処理に関するサマーワークショップ (SWoPP2018)*, Jul. 30-Aug. 1, 2018.
- 八島慶汰, **大山洋介**, 松岡聡, "深層学習における BatchNormalization 使用時の計算時間と精度の関係性", In *並列/分散/協調処理に関するサマーワークショップ (SWoPP2018)*, Jul. 30-Aug. 1, 2018.
- **Yosuke Oyama**, Naoya Maruyama, Nikoli Dryden, Peter Harrington, Jan Balewski, Satoshi Matsuoka, Marc Snir, Peter Nugent, Brian Van Essen, "Toward Training a Large 3D Cosmological CNN with Hybrid Parallelization", In *並列/分散/協調処理に関するサマーワークショップ (SWoPP2019)*, Jul. 24-26, 2019.
- 土川稔生, 遠藤敏夫, **大山洋介**, 野村哲弘, 近藤正章, 松岡聡, "メモリアクセスデータを用いた機械学習によるアプリケーションの類型化", In *並列/分散/協調処理に関するサマーワークショップ (SWoPP2019)*, Jul. 24-26, 2019.
- **Yosuke Oyama**, Naoya Maruyama, Nikoli Dryden, Erin McCarthy, Peter Harrington, Jan Balewski, Satoshi Matsuoka, Peter Nugent, Brian Van Essen, "The Case for Strong Scaling in Deep Learning: Training Large 3D CNNs with Hybrid Parallelism", In *arXiv e-prints*, 2020.
- Jens Domke, Emil Vatai, Alexsandr Drozd, Peng Chen, **Yosuke Oyama**, Lingqi Zhang, Shweta Salaria, Daichi Mukunoki, Artur Podobas, Mohamed Wahib, Satoshi Matsuoka, "Matrix Engines for High Performance Computing: A Paragon of Performance or Grasping at Straws?", In *arXiv e-prints*, 2020.

Unreviewed Posters

- 大山洋介, “大規模並列環境における少精度型を用いたディープラーニングの学習精度の検証”, In *JHPCN*：学際大規模情報基盤共同利用・共同研究拠点第 *10* 回シンポジウム, Jul. 12-13, 2018.
- 大山洋介, 野村哲弘, 佐藤育郎, 松岡聡, “大規模並列環境における低精度型を用いたディープラーニングの学習精度の検証”, In 公開シンポジウム「*Co-Design* による深層学習基盤」, Nov. 27, 2018.
- **Yosuke Oyama**, Tal Ben-Nun, Torsten Hoefler, Satoshi Matsuoka, “μ-cuDNN: Accelerating Deep Learning Frameworks with Micro-Batching”, In 公開シンポジウム「*Co-Design* による深層学習基盤」, Nov. 27, 2018.

Article

- “GPU でのディープラーニングをさらに速く、東工大が NVIDIA チップ向けに高速化ライブラリ”, In 日経 *Robotics 2019* 年 *7* 月号, 2019.

Contents

1 Introduction **12**

1.1 Problem statement . . . 13

1.2 Proposal and contributions . . . 14

1.2.1 Proposal 1: Automatic optimization of computational kernels . . . 16

1.2.2 Proposal 2: Training 3D CNNs with hybrid-parallelization . . . 17

1.3 Outline of this thesis . . . 17

2 Background **19**

2.1 Neural networks . . . 19

2.1.1 Convolutional Neural Networks (CNNs) . . . 20

2.1.2 Back-propagation and mini-batch Stochastic Gradient Descent . . . 24

2.1.3 Metrics of DNNs . . . 25

2.2 Training DNNs with HPC . . . 26

2.2.1 Intra-processor parallelism . . . 28

2.2.2 Data-parallelism . . . 28

2.2.3 Model-parallelism . . . 30

3 Automatic optimization of computational kernels **33**

3.1 Motivation and background . . . 34

3.1.1 cuDNN . . . 34

3.2 μ-cuDNN and beyond . . . 37

3.2.1 μ-cuDNN Overview . . . 37

3.2.2 Performance analysis on cuDNN kernels using DeepBench . . . 41

3.2.3 Micro-batching with mixed-precision . . . 42

3.2.4 High-level optimization with μ-cuDNN . . . 47

3.3 Evaluation . . . 48

3.3.1 DeepBench 49
3.3.2 CNN performance 50
3.3.3 Case study: Heterogeneous cluster optimization 53

4 Training 3D CNNs with hybrid-parallelization 56

4.1 Motivation 57
4.1.1 The CosmoFlow network 58
4.1.2 The 3D U-Net 60
4.2 Hybrid-Parallel implementation of 3D CNNs 62
4.2.1 Extending Distconv for 3D CNNs 63
4.2.2 Kernel optimization 66
4.2.3 Spatially-partitioned I/O 67
4.2.4 Performance modeling 69
4.2.5 Architecture tuning 72
4.3 Performance evaluation 78
4.3.1 Evaluation environment 79
4.3.2 Strong scaling 79
4.3.3 Weak scaling 85
4.3.4 CosmoFlow model accuracy improvement with 512^3 data cubes 90

5 Related work 96

5.1 Optimizing intra-processor parallelism 96
5.2 Optimizing inter-processor parallelism 97

6 Discussion 100

6.1 Automatic optimization of computational kernels 100
6.1.1 Implications and possible future work 100
6.2 Training 3D CNNs with hybrid-parallelization 104
6.2.1 Predicting multi-dimensional partitioning performance 107
6.2.2 Implications 109
6.2.3 Future work 110

7 Conclusion 112

A cuDNN performance on DeepBench convolutional layers 113

List of Figures

1.1 Increasing GPU performance and per-GPU memory capacity [49, 54, 55, 56, 57] 15
1.2 An overview of our proposals 15

2.1 A neuron 20
2.2 2D convolution 21
2.3 Convolution algorithms 22
2.4 The GEMM-based convolution 23
2.5 Specifications of popular DNNs [46] 27
2.6 Three parallel strategies for deep neural networks 29
2.7 Three parallel strategies of model-parallelism on a three-layer 2D CNN 30

3.1 An example code to perform cuDNN convolution 34
3.2 Per-layer breakdowns of memory consumption and computation time of AlexNeton P100-SXM2 36
3.3 The conceptual execution timeline of μ-cuDNN 38
3.4 μ-cuDNN software stack 40
3.5 Overview of μ-cuDNN 41
3.6 An example code to perform convolution with μ-cuDNN 41
3.7 Convolution performance of DeepBench convolutional layers on V100-SXM2 44
3.8 Convolution performance of a DeepBench's 5×5 convolutional layer 46
3.9 Convolution performance of DeepBench's 20×5 convolutional layers on V100-SXM2 . . 46
3.10 Convolution performance of DeepBench's 3×3 and 7×7 convolutional layers on V100-SXM2 47
3.11 The μ-cuDNN interface with layer IDs 48
3.12 Relative speedups of DeepBench's forward convolution against cuDNN 50
3.13 Benchmark results of forward convolution of AlexNet's "conv2" layer on P100-SXM2 . . 51

3.14 Benchmark results of AlexNet on three different GPUs 52
3.15 TensorFlow benchmark results on P100-SXM2 GPU 53
3.16 Sample code for heterogeneous cluster optimization 54
3.17 Estimated time of forward-backward passes of ResNet-18 on heterogeneous GPUs 55

4.1 Visualization of one channel of a 128^3 CosmoFlow data cube 59
4.2 Visualization of a 512^3, 4 "redshift" channels CosmoFlow data sample 61
4.3 Visualization of a 256^3 LiTS data sample 62
4.4 The definition of CosmoFlow's first convolutional layer on LBANN 64
4.5 The software stack of LBANN with hybrid-parallelism 64
4.6 Overview of Distconv 65
4.7 Overview of hybrid-parallel training on LBANN and Distconv 66
4.8 Sample-parallel I/O 68
4.9 Strong scaling of the CosmoFlow network with 512^3 input cubes without spatial-parallel I/O 69
4.10 Spatially-partitioned data movement on our framework 70
4.11 Inter-GPU communication performance on Lassen 73
4.12 All-reduce collective performance among GPUs on Lassen 74
4.13 Layer-wise convolution performance of the CosmoFlow network 75
4.14 Our revised CosmoFlow network architecture 76
4.15 Sierra/Lassen node diagram 80
4.16 Strong scaling of the CosmoFlow network with 512^3 input cubes 81
4.17 Strong scaling of the 3D U-Net with 256^3 input cubes 83
4.18 Single-GPU execution timelines for training the CosmoFlow network with the 512^3 input cubes 84
4.19 Weak scaling of the CosmoFlow network with different spatial partitioning 88
4.20 Weak scaling of the two different 3D CNNs 89
4.21 The software stack of Horovod with data-parallelism 90
4.22 The software stack of Mesh-TensorFlow with model-parallelism 90
4.23 TensorFlow's GPU matrix multiplication (matmul) of two tensors 91
4.24 Timeline of Mesh-TensorFlow with data-parallelism and 4-way partitioning 92
4.25 Training of the CosmoFlow network with different input resolutions 94
4.26 True and predicted cosmological parameters from four different configurations 95

6.1 Speedups of convolutional layers with the NHWC format compared to the NCHW format 102
6.2 Making an ONNX convolution node 102

6.3 Combining the μ-cuDNN method with ONNX 103
6.4 Visualization of AlexNet on the micro-batching technique on V100-SXM2 105
6.5 Relative time to compute ResNet's batch-normalization layer with cuDNN on V100-SXM2 106
6.6 Predicted strong scaling of the CosmoFlow network with multi-dimensional partitioning 108
6.7 Predicted strong scaling of the 3D U-Net with multi-dimensional partitioning 109

A.1 Time to compute DeepBench's convolution kernels on a K80 GPU 114
A.2 Time to compute DeepBench's convolution kernels on a P100-SXM2 GPU 115
A.3 Time to compute DeepBench's convolution kernels on a V100-SXM2 GPU 116

List of Tables

2.1 Specifications of popular DNNs [46] 26
2.2 Comparison of data-, model-, and hybrid-parallel computation 32
3.1 Supported forward convolution algorithms in cuDNN 7.0.4 36
3.2 Support status of the direct algorithm in cuDNN 36
3.3 Micro-batch size policies 40
3.4 The convolutional layers of DeepBench 43
3.5 Number of convolutional layers in DeepBench 45
3.6 Available data types in cuDNN's convolution 45
3.7 cuDNN runtime parameters of various deep learning frameworks 45
3.8 Evaluation environment 49
3.9 Evaluation settings 49
3.10 TensorFlow benchmark results on P100-SXM2 GPU 53
3.11 GPU specification for heterogeneous cluster optimization 55
4.1 Two different target 3D CNNs 58
4.2 Comparison of CosmoFlow work 60
4.3 Comparison of U-Net work 61
4.4 Statistics of a CosmoFlow data sample 70
4.5 Comparison of CosmoFlow network variants 76
4.6 Our revised CosmoFlow network architecture 77
4.7 The number of operations of CosmoFlow's convolutional layers with a 128^3 cube 78
4.8 Legend specifications of the strong scaling plots 80
4.9 Achieved performance of CosmoFlow convolution layers compared to the peak performance of cuDNN 85

4.10 The number of kernels/memcpy of Mesh-TensorFlow on the CosmoFlow network in one training iteration 91
4.11 Comparison of relative prediction errors 94

6.1 GPU trace of DeepBench's 1×1 convolution layer in different memory formats 101
6.2 The support status of ONNX 101
6.3 Profiling results of ResNet's batch-normalization layer 106

Chapter 1

Introduction

In the last decades, a subset of machine learning techniques called **Deep Learning (DL)** has attracted substantial research interests, and it became one of the most successful machine learning technologies on various real-world tasks, including image classification [1, 2, 3, 4], image segmentation [5, 6, 7, 8], object detection [9, 10, 11], language understanding [12, 13], acoustic models [14, 15, 16], autonomous driving [17], and medical image analysis [6, 18, 19, 20, 21, 22]. Deep learning is a generic term for machine learning methods using **Deep Neural Network (DNN)** models, which are composed of vast numbers of learnable parameters (e.g., the GPT-3 language model has 175 billion parameters [23]) to imitate given complex non-linear objective functions. The success of the study on deep learning is supported by the effort from different orthogonal aspects: 1) the advance in network architecture and training algorithms, 2) the increase of available data that is used to train DNNs, and 3) the increase of the compute capability of high-performance computers, such as GPUs and supercomputers. To increase the generalization performance of DNNs in a realistic time, all three of these factors need to be improved simultaneously. A lot of effort has been made to improve these factors; the number of papers about deep learning submitted to arXiv [24] and conferences is increasing sharply in the past decade [25], and the data size of public datasets in various objectives is increasing as well [26]. Since both advances increase the demand for computing capability for given networks and datasets, 3) is also crucial to make progress in the deep learning field.

In this thesis, we mainly focus on 3), which significantly affects the speed of deep learning training and inference. In the **High-Performance Computing (HPC)** domain, the most powerful way to accelerate a given computation workload is parallelization, where many computers or accelerators (such as GPUs) are utilized simultaneously. As we explain in Section 2.2, this basic principle has also been applied to deep learning. Multiple research report that even a single network is now able to be trained on thousands of GPUs [27, 28, 29]. A common idea to distribute deep learning workload is to utilize **data-parallelism**, where the computation on data (data samples) are parallelized among accelerators. The typical dataset size used in deep learning is order-of-magnitude

more extensive than the degree of parallelism (i.e., number of processors) typically available. Therefore data-parallel training is regarded to be very scalable. Indeed, many studies have achieved this successful distributed training by exploiting data-parallelism in many machine learning tasks, such as image classification [27, 28, 29, 30, 31] and so on. And thus, this technique is becoming a common parallelization technique in applications and industrial domains; many deep learning frameworks such as PyTorch [32], TensorFlow [33], Chainer [34] support this technique by default. This trend leads that the computational performance of each framework converges by relying on highly-optimized bender libraries, such as NVIDIA cuDNN [35] and NCCL [36], just as many HPC applications exploit BLAS and MPI libraries.

1.1 Problem statement

Nowadays, data-parallel training is known as the easiest way to parallelize deep learning workload due to its simple structure, just like the bulk synchronous parallel model [37], and thus it has adopted many existing deep learning frameworks [32, 33, 34, 38, 39, 40, 41, 42, 43, 44]. One of the most famous examples linking deep learning and HPC is the data-parallel training of 2D CNNs. As in the example of AlexNet, proposed by Krizhevsky et al. [45] for the ILSVRC image recognition dataset [3], which was trained using multiple GPUs, CNNs have been associated with high-performance cluster and supercomputers that can meet their computational demands. Moreover, much prior work report that the key components of such software to bring good performance is computational kernels on each layer (such as convolution kernels) and collective communication to synchronize parameter gradients, both of which are mostly entrusted by bender libraries such as NVIDIA cuDNN and NCCL.

Specifically, in my previous thesis [46], we focused on the following topics that are related to training of 2D CNNs:

- We proposed a performance model for an asynchronous, data-parallel deep learning framework for training 2D CNNs [47]. We demonstrated that performance modeling is an effective way to estimate the node-hour required for training and detect potential hardware bottlenecks only with small benchmark trials.
- We proposed the basic idea of the μ-cuDNN library [48], a wrapper library for cuDNN, which performs auto-tuning on convolutional layers. We evaluated how much loop splitting is effective for cuDNN's GPU convolution kernels.
- We proposed a low-precision communication technique for synchronizing parameter gradients (Section 2.1.2) for data-parallel training [47]. We demonstrated that the communication workload could be reduced by up to a quarter with no accuracy loss.

However, since deep learning is a research area of intense research and development, **the interest is shifting to problems that are more complex and unknown than the data-parallel training**

of 2D CNNs, both in terms of software and hardware. One of the examples, which is not limited to deep learning, is that hardware architectures are expected to become more diverse in the future. For example, the computational performance of GPUs has simply improved in each generation as implied by Moore's law (Table 3.8), and this performance is well reflected when the computation of DNNs is actually performed on different generations of GPUs (as we show in Table 3.14). However, hardware vendors such as NVIDIA have been specializing their architectures to meet the growing demand for deep learning, and it is not clear how much of that can be leveraged in real workloads. In fact, in our thesis, we demonstrate that one of such new features, Tensor Cores [49], are not necessarily applicable to all types of deep learning workloads (Section 3.3.1). Another critical change in hardware trends is the end of Moore's law [50]. In general, one crucial factor in understanding the performance of HPC applications is to understand the ratio of computation, memory operations, and communication of the applications as well as those of underlying hardware. As already explained, in the case of conventional deep learning workloads, one of the best parallelization strategies is to promote data-parallelism by increasing the number of processors. However, if the computational performance of each processor slows down, there will be a need to parallelize on a larger scale than can be achieved by data-parallel training in order to keep up with the increasing model complexity.

Another problem from a non-hardware aspect is that the conventional assumptions about data-parallel training are broken down by the increasing complexity of deep learning models. Data-parallel training with mini-batch SGD assumes that at least a single processor can perform the training. However, models so large that they cannot fit into a single GPU memory, for example, cannot be practically trained by the method. Since the increase in memory size has not kept pace with the increase in model size due to manufacturing costs, it is difficult to solve this problem only with hardware evolution (Figure 1.1). This can be particularly problematic when using high-dimensional data, which we explain in Chapter 4. In the early 2010s, several model-parallel training implementations were proposed [45, 51, 52, 53], mainly for networks with fully-connected layers and for CNNs that are smaller (in terms of spatial width, the number of channels, and the number of layers) than those used today; we describe further in Section 5.2. However, few of them have actually been implemented in existing actively-developed frameworks (as we demonstrate in Chapter 4), and therefore, it is unclear to what extent they work in practice.

For the aforementioned reasons, it is essential to continue to develop efficient parallelization methods of new hardware architecture and deep learning models at various hardware layers in order to continue to develop the coupling of deep learning and HPC in the future.

1.2 Proposal and contributions

In this thesis, we broadly propose two optimization methods that exploit two levels of parallelism of CNNs (Figure 1.2); these methods are independent and can be applied in combination. In particular,

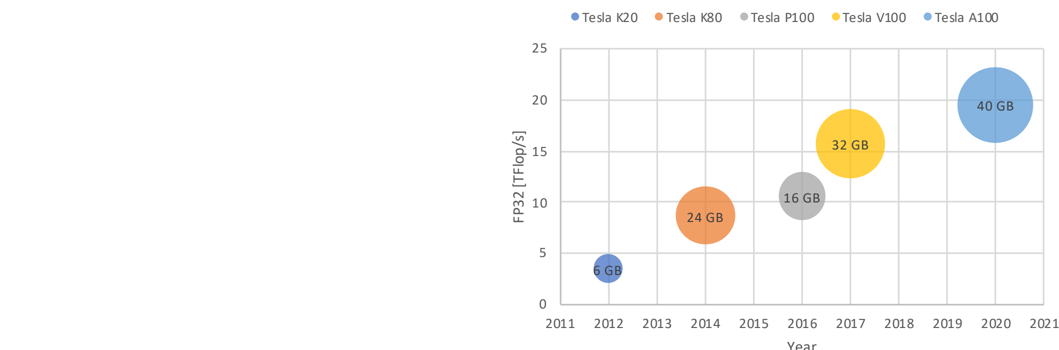


Figure 1.1: **Increasing GPU performance and per-GPU memory capacity [49, 54, 55, 56, 57]**. A K80 GPU contains two distinct GPU chips, which share 24 GiB of memory. In this paper, we handle a CNN that requires 100 GiB of memory, which is beyond the memory capacity of the latest GPU generation.

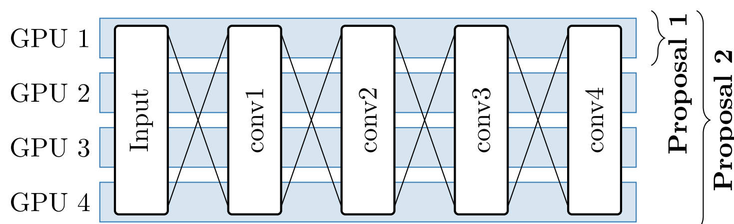


Figure 1.2: **An overview of our proposals**. We apply "Proposal 1: Automatic optimization of computational kernels" to improve per-GPU performance, and "Proposal 2: Training 3D CNNs with hybrid-parallelization" to improve scalability on multiple GPUs.

research trends in hardware, software, and model architectures change rapidly in the deep learning research domain; hence we cover such recent research topics as possible. Specifically, we cover low-precision arithmetic using Tensor Cores [49], acceleration methods that are portable to different deep learning frameworks, and high-dimensional CNNs that are intended for end-to-end learning of scientific data. Through these topics, we discuss appropriate parallelization methods for advanced training tasks and architectures.

Both of our methods use optimization problems and performance models to parallelize deep learning workloads by taking full advantage of the parallelism of DNNs. This is regarded as an extension of performance modeling for deep learning frameworks, as proposed in our previous work [47]. Our proposals allow us to quantitatively show whether any parallelization strategy is suitable for given network architecture, rather than just providing best-effort parallelization technique for a particular network. For example, in Section 3.3.1, we evaluate the acceleration of our loop optimization method for various convolutional layers, and in Section 6.2.1, we discuss the optimal hybrid parallelization strategy for currently available computational resources.

1.2.1 Proposal 1: Automatic optimization of computational kernels

Convolution is one of the most computationally-intensive parts of deep learning training and inference. Therefore, accelerating convolutional computation plays a key role in accelerating such jobs in many fields. For GPU-based convolutional computation, NVIDIA's cuDNN math kernel library is widely known for its high computational efficiency. However, its performance heavily depends on various factors such as convolutional layers' parameters (such as the filter size), the selection of convolution algorithms, the available free memory size, and the arithmetic precisions. We extend the μ-cuDNN library, which was initially proposed in my master's thesis [46], to improve the performance using loop optimization for various convolutional layers and runtime configurations. Our contributions are as follows:

- We extend the loop optimization algorithm of the μ-cuDNN library to use algorithms of different computational precisions. This technique allows us to select a broader range of convolutional algorithms without degradation of accuracy.
- We combine the μ-cuDNN library in two frameworks, Caffe and TensorFlow, and show that it achieves reasonable performance improvements in several different CNNs; we achieve speedups of 1.60x for AlexNet and 1.30x for ResNet-18 on an NVIDIA V100 GPU. We thus demonstrate that μ-cuDNN speeds up not only cuDNN's convolutional kernels but also works for end-to-end learning tasks.
- By using DeepBench's $\sim$ 100 convolutional kernels selected from CNNs designed for different machine learning tasks, we measure its speedup on multiple GPUs and computational precisions. We show that the adaptive loop optimization algorithm can achieve significant speedups even in the case of using modern reduced-precision operations (i.e., Tensor Cores); we show that μ-cuDNN achieves speedups of up to 4.54x, and 1.60x on average for DeepBench convolutional layers on an NVIDIA V100 GPU.
- Taking advantage of the property that μ-cuDNN can obtain layer parameters via the cuDNN interface, we propose an interface to use this information for multi-node training. We demonstrate its practicality by optimizing the performance of data-parallel training on a heterogeneous cluster.
- We discuss possible directions to extend the library to support a wider range of deep learning frameworks and layer types. Specifically, we show that the ONNX data format allows us to apply the loop optimization algorithm of μ-cuDNN without any changes to the framework itself. We also discuss whether μ-cuDNN's algorithm can be extended to layer types other than convolution and to layouts other than the usual NCHW data layout.

1.2.2 Proposal 2: Training 3D CNNs with hybrid-parallelization

Using deep learning models as surrogate models, which perform end-to-end inference for raw scientific data, has received much attention in recent years. Although data-parallel training methods are known to scale up to hundreds of GPUs for training a wide range of deep learning models, in the training of such surrogate model, the minimum amount of memory required for training tends to be larger than the amount of memory in one GPU, and therefore, data-parallel training is infeasible. Therefore, we focus on 3D high-resolution CNNs and train them by using model-parallelism using spatial decomposition. Our contributions are as follows:

- We present an end-to-end approach for strong-scaling training large-sample 3D CNNs. Specifically, we propose various methods to optimize computation and I/O performance for better scalability.
- We extend the LBANN framework [44] to implement our proposed approach. We then demonstrate training on full-resolution samples for the CosmoFlow network (512^3) and the 3D U-Net (256^3); we achieve 1.77x of speedup on 2048 GPUs over 512 GPUs with the same mini-batch size of 64 for the CosmoFlow network, and achieve 1.42x of speedup on 512 GPUs over 256 GPUs with the same mini-batch size of 16 for the 3D U-Net. Our performance results show good strong and weak scaling on up to 2048 GPUs. We provide detailed model-based performance analyses in order to give a comprehensive understanding of their scaling efficiencies.
- We propose a performance model for the hybrid-parallel training framework to demonstrate that its performance is predictable with benchmark results using the limited number of compute nodes.
- We demonstrate a significant improvement in prediction accuracy by using full-resolution data. The CosmoFlow model trained with 512^3 samples realizes ten times lower mean squared error than when trained with 128^3 samples, which was the largest size reported previously.

1.3 Outline of this thesis

This thesis is organized as follows:

Chapter 2: Background

In this chapter, we first briefly summarize the background theory of deep learning. Note that the main focus of this thesis is the acceleration of computation on deep neural networks, and thus we mostly explain it from its computational aspects. We then explain various ways to parallelize training

of DNNs, from the same perspective. We also introduce some prior work to try to accelerate deep learning by exploiting its data-parallelism, model-parallelism, and the combinations of them.

Chapter 3: Automatic optimization of computational kernels

In this chapter, we propose μ-cuDNN, a library to perform auto-tuning on convolutional layers of DNNs. We briefly introduce its methodology presented in my master's thesis [46], and then extend its exploration space for optimization based on observations of the DeepBench [58]. We demonstrate that μ-cuDNN squeezes more speedups from the state-of-the-art kernel library cuDNN on various layers and precision configurations.

Chapter 4: Training 3D CNNs with hybrid-parallelization

In this chapter, we propose hybrid-parallel training of 3D CNNs by applying spatial partitioning on hundreds of GPUs. We use the CosmoFlow [59] 3D CNN and the 3D U-Net [18] as two motivative examples. We propose various optimization techniques to alleviate the overhead of the model-parallel scheme, including asynchronous halo exchange and model-parallel I/O. We demonstrate that our software achieves order-of-magnitude improvement on its generalization performance by increase the input size by up to 256 times, which cannot be done only with data-parallelism and nor any other frameworks.

Chapter 5: Related work

In this chapter, we compare prior studies about optimizing the computational efficiency of deep learning frameworks with our proposed methods. Specifically, we explain that our methods solve new types of performance issues of deep learning workloads, loop optimization of convolution by combining multiple convolution algorithms and precisions, and hybrid-parallel training of high-resolution 3D CNNs on a GPU supercomputer, which have not been studied before.

Chapter 6: Discussion

In this chapter, we discuss the future prospects of our methods proposed in Chapter 3 and Chapter 4. We mainly discuss implications to the research community, applicability to other studies, and possible future work. For μ-cuDNN work, we further explain and demonstrate how our algorithm can be generalized for other types of layers and software.

Chapter 7: Conclusion

In this chapter, we discuss future possibilities for extending our work to more general network and software architectures.

Chapter 2

Background

Deep Learning (DL) is a subset of machine learning algorithms that uses mathematical models called **Deep Neural Networks (DNNs)**. In this chapter, we briefly explain the fundamental algorithms of DNNs, and various techniques to train these models on one or more computers. We introduce the minimum amount of knowledge to understand the contents of this thesis, but more supplemental details about the basics are found in my master's thesis [46].

2.1 Neural networks

Neural Networks (NNs) are types of mathematical functions. As implied in their names, these structures resemble animal brains, which are composed of many **neurons** to represent complex functions. In the simplest type of the networks, **Multi-Layer Perceptron (MLP)**, given an input vector $x_0 \in \mathbb{R}^{N_0}$, its output $y = x_L \in \mathbb{R}^{N_L}$ is computed as follows:

$$y = x_L = \sigma(W_{L-1}x_{L-1} + b_{L-1}) \tag{2.1}$$

$$y = \sigma(W_{L-1}\sigma(W_{L-2}x_{L-2} + b_{L-2}) + b_{L-1}) \tag{2.2}$$

$$y = \sigma(W_{L-1}\sigma(W_0 \ldots \sigma(Wx_0 + b_0) \ldots + b_{L-2}) + b_{L-1}), \tag{2.3}$$

where $W_l \in \mathbb{R}^{N_{l+1} \times N_l}$ is a **weight** matrix, $b_l \in \mathbb{R}^{N_l}$ is a **bias** vector (which is optional), and σ is an activation function, which applies nonlinearity to each input element. One of many functions is used as the activation function, such as the sigmoid function, tanh, ReLU [60], PReLU [61], and leaky ReLU [62]. The computation of each x_l (**layer**) is regarded as a batch computation of each neuron

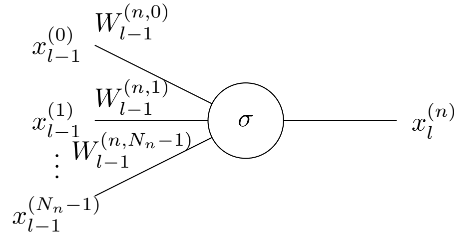


Figure 2.1: **A neuron**.

(Figure 2.1):

$$x_l^{(n)} = \sigma\left(\sum_{m=0}^{N_{n-1}} W_{l-1}^{(n,m)} x_{l-1}^{(m)} + b_{l-1}^{(n)}\right). \tag{2.4}$$

From a computational point of view, the computation of an MLP is regarded as a set of matrix-vector multiplications and lightweight element-wise (memory-intensive) operations. Moreover, as we explain Section 2.2.1, the forward computation of such networks is practically batched with more than one input data, by replacing each x_l with a matrix, each column of which represents (intermediate) data for one distinct input data. Therefore, the computation of MLPs is typically performed with General Matrix-Matrix multiplications (GEMMs), and this technique is also used in back-propagation steps (Section 2.1.2).

Deep Neural Networks (DNNs) are a subset of NNs which has multiple layers. Many papers have demonstrated that the more NNs have, the better accuracy is generally achieved [15, 23, 63, 64, 65, 66]. On the other hand, it also leads networks difficult to converge due to several reasons, including gradient vanishing [67] and so on. Also, from the computational point of view, deeper models consume more memory to store hidden activations and error signals, which causes out-of-memory problems (Section 2.2.2).

2.1.1 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are a subset of NNs where convolution is performed on all or some layers instead of multiply-add of Equation 2.3 (Figure 2.2):

$$y_{n,c',h',w'} = \sigma\left(\sum_{c,v,u} w_{c',c,v,u}\ x_{n,c,p_y+s_y h'+v,p_x+s_x w'+u}\right), \tag{2.5}$$

where

- $\mathbf{X} = (x_{n,c,h,w})$ is the input tensor,
- $\mathbf{Y} = (y_{n,c',h',w'})$ is the output tensor,

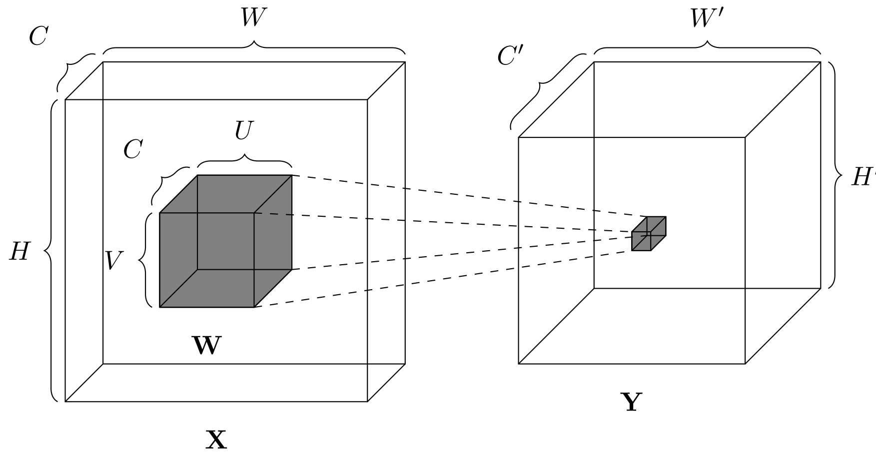


Figure 2.2: **2D convolution**.

- $\mathbf{W} = (w_{c',c,v,u})$ (in which each subscript iterates over 1 to its upper case) is the weight tensor,
- $s_x, s_y \in \mathbb{N}$ are **strides**, and
- $p_x, p_y \in \mathbb{Z}$ are **paddings**.

The strides and the paddings are used to reduce or adjust the output dimensions.

The number of spatial dimensions of a CNN depends on its target dataset. 2D layers are used for image classification [45, 63, 64, 65, 66, 68], image segmentation [5, 6, 7, 8], object detection [9, 10, 11], and even for acoustic models [14, 15, 16], and CNNs with higher-dimensions are possible for multi-dimensional data; we use two examples of 3D CNNs, the CosmoFlow network and the 3D U-Net, in Chapter 4.

A convolutional layer is regarded as a particular type of MLP layer where most weights are disconnected, and non-zero connections share their weights with other connections. In contrast to convolutional layers, each layer of MLPs is also called **fully-connected layers**. In CNNs, it is common that layers other than convolutional layers are combined to reduce the output dimensions; many CNNs have one or fully-connected layers at the end. Some networks also equip **pooling layers** to reduce the spatial dimensions (and their computational cost) while increasing the number of channels as activations are propagated, assuming that it preserves the information of input data. On the contrary, some networks that have to propagate the spatial feature of input data at the end may lack fully-connected layers.

Convolution algorithms

The Convolution operation is regarded as one of the most time-consuming, compute-intensive operations in DNNs. In general, computation on convolutional layers is believed to be dominant in typical training and inference workload due to their high degree of arithmetic operations; we actually

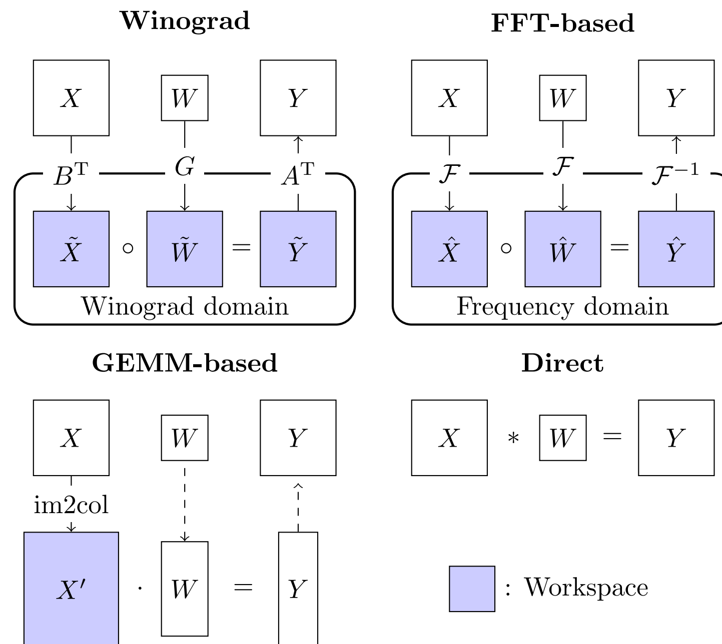


Figure 2.3: **Convolution algorithms**.

demonstrate how convolution occupies the computational time of CNNs in Chapter 3 and Chapter 4. Thus, many equivalent algorithms have been proposed to compute convolutional layers efficiently (Figure 2.3):

- **GEMM-based convolution**, or "im2col" convolution, is a straightforward implementation of convolution using GEneral Matrix Multiplication (GEMM) that is defined in the BLAS interface [69]. The algorithm is two-fold; An im2col (image-to-column) kernel copies input elements to an intermediate workspace so that the following GEMM is equivalent to convolution (Figure 2.4). The advantage of the algorithm is that it can exploit the nice computational efficiency of BLAS libraries to perform matrix multiplication. However, the downside is 1) it requires a sizeable extra memory footprint to pass the reshaped input tensor to GEMM, and 2) the copy part and the compute part cannot be overlapped as it relies on an external BLAS library.

- **Winograd's algorithm** [70, 71] is an algorithm to perform convolution with a smaller number of multiplications than the original algorithm. For example, convolution on $\mathbf{X} = [x_0, x_1, x_2, x_3]$

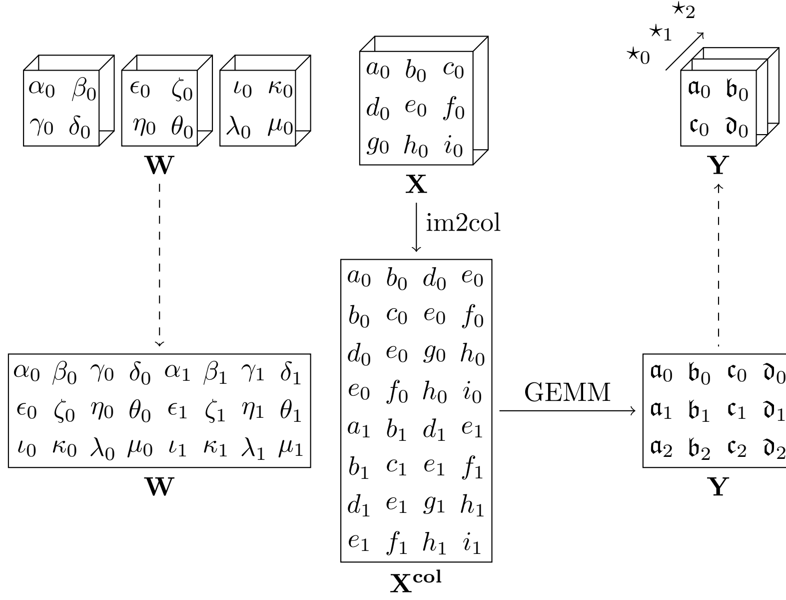


Figure 2.4: **The GEMM-based convolution**.

and $\mathbf{W} = [w_0, w_1, w_2]$ is computed with four multiplications as follows:

$$\mathbf{Y} = \mathbf{X} * \mathbf{W} = \begin{bmatrix} x_0 & x_1 & x_2 \\ x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix} \tag{2.6}$$

$$\Leftrightarrow \mathbf{Y} = A^{\mathrm{T}}[(G\mathbf{W}) \odot (B^{\mathrm{T}}\mathbf{X})] \tag{2.7}$$

$$\left(\begin{array}{lll} A^{\mathrm{T}} & = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix} & \in \mathbb{R}^{m \times (m+r-1)} \\ G & = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix} & \in \mathbb{R}^{(m+r-1) \times r} \\ B^{\mathrm{T}} & = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} & \in \mathbb{R}^{(m+r-1) \times (m+r-1)} \end{array} \right), \tag{2.8}$$

where A, B, G are constant matrices that are determined by the layer's architecture. This algorithm is decomposed into 1) matrix multiplications to transform the input data and weights into the Winograd domain, 2) element-wise multiplication on the two data, and 3) a matrix multiplication to restore the output from the Winograd domain. The advantage of

Winograd's algorithm is that it reduces the arithmetic complexity of convolutional layers, although it requires a decent-sized workspace [71].

- **FFT-based convolution** [72, 73, 74, 75, 76] performs convolution as element-wise multiplication on the frequency domain. In this algorithm, both the input data and weights are transformed into the frequency domain by performing Fast Fourier Transformation (FFT), and then element-wise multiplication is performed to the two transformed data, and Inverse FFT (IFFT) is applied to get the output data that is equivalent to the output of the layer. It is regarded as a special case of the Winograd algorithm with particular complex transform matrices; Equation 2.8 is equivalent to

$$A^{\mathrm{T}} = \frac{1}{4}\begin{bmatrix} 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix}, G = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix}, B^{\mathrm{T}} = \begin{bmatrix} 1 & 1 & 1 \\ -1 & -i & 1 \\ 1 & -1 & 1 \\ -1 & i & 1 \end{bmatrix}. \tag{2.9}$$

  It has been demonstrated that FFT-based convolution is efficient on layers with large filter sizes [73, 76]. Still, this algorithm also has the same memory footprint problem as the GEMM-based algorithm.

2.1.2 Back-propagation and mini-batch Stochastic Gradient Descent

Back-propagation is an algorithm to compute parameter gradients of each activation. In the algorithm, parameter gradients are obtained by computing gradients with respect to the input data and the parameters of each layer in the reversed order of the original computation (which is also called forward-propagation). Back-propagation of a fully-connected layer requires two GEMMs, each of which is for computing gradients with respect to input and parameters, and a similar rule is applied to convolutional layers (where two convolutions are needed).

Back-propagation is combined with **Stochastic Gradient Descent (SGD)** to optimize DNNs' parameters iteratively in most cases:

$$W^{(t+1)} = W^{(t)} - \eta^{(t)} \sum_{n=1}^{N} \nabla L\left(x_n; W^{(t)}\right), \tag{2.10}$$

where W^t are the network parameters at step t, $\eta^{(t)}$ is the learning rate at step t, N is the mini-batch size, L is the loss function, and x_n is the n-th sample randomly selected from the dataset. A wide variety of alternatives to the original SGD algorithm have been proposed, such as AdaGrad [77], ADADELTA [78], Adam [79], and so on.

In the computational perspective, "training" usually refers to a computational workload that is composed of 1) forward propagation, 2) back-propagation, and 3) weight update explained in Equation

2.10. Although 1) and 2) cannot be overlapped since 2) requires error signals of the final layer, there are various scheduling methods for 2) and 3) in terms of better computational throughput or less memory consumption, and so on (Section 5.2). Similarly, "inference," where unknown samples are inferred by using a trained DNN, usually refers to forward-propagation.

2.1.3 Metrics of DNNs

There are a variety of metrics to summarize the characteristics of DNNs:

- **The number of parameters** is the number of trainable weight values of the network. On a fully-connected layer, the number of parameters is $O(N^2L)$, where N is the number of neurons of each layer, and on a 2D convolutional layer, it is $O(K^2C^2L)$ where K is the filter size, and C is the number of channels. Note that it is not necessarily constant throughout the network but typically increases as it gets close to the output layer. The reversed principle is applied to the spatial width (W) due to convolutional layers without padding and pooling layers. Since K and C are typically independent of the input data size but so is N, MLPs tend to have more parameters than CNNs that are designed for exactly the same objective dataset.

- The number of **FLOPs** (floating-point operations) of a DNN per data sample represents the computational cost of the network. It is $O(N^2L)$ on MLPs and $O(K^2C^2W^2L)$ on 2D CNNs, where W is the spatial dimension size. And thus, typical CNNs requires more FLOPs to compute each data sample.

- The number of neurons of a DNN defines how it requires a memory footprint to store activations and error signals. It is $O(NL)$ on MLPs and $O(CW^2L)$ on 2D CNNs. CNNs tend to consume more memory than MLPs as it is square of the input dimension size. A huge memory footprint may cause a runtime error of deep learning frameworks on GPUs because conventional GPU software architecture (such as NVIDIA CUDA [80]) prohibits the total memory consumption from exceeding the physical GPU memory size. As we explain in Section 5.2, various techniques have been proposed to mitigate the memory pressure, such as offloading [81, 82, 83], recomputation [83, 84, 85], efficient memory management [86], using NVIDIA GPUs' Unified Memory [82], and the use of model-parallelism which we introduce in Section 2.2.

- The **accuracy**, loss, or generalization performance represents how networks can predict proper outputs from unseen input data. The way a network is evaluated is defined by its objective function, such as softmax cross-entropy loss for classification tasks [45], or general error functions for other types of tasks. Thus, this metric is used only for comparing the potential of networks on the same task.

- **The mini-batch size** (N in Equation 2.10) defines the update frequency of model parameters. Typically, a DNN is trained for a fixed number of epochs (how many times each sample is used

Table 2.1: **Specifications of popular DNNs [46]**. We report place and accuracy for the annual ILSVRC classification task [3] if it is officially submitted to the competition. Note that the accuracy can be those of the ensemble of the networks.

Network	Year-Place	Number of Weights [10^6]	ILSVRC-2012 Classification Top-5 Accuracy [%]
28x28-1000-10 [1]	1998	0.8	-
LeNet [1]	1998	0.07	-
AlexNet [45, 88]	2012-1st	64.0	15.3
GoogLeNet [64]	2014-1st	15.2	6.67
VGG [63]	2014-2nd	175.1 (VGG-19)	7.32
NiN [68]	2014	11.3	-
ResNet [65]	2015-1st	103.1 (ResNet-152)	3.57

in Equation 2.10), and hence the number of FLOPs itself is almost invariant to the mini-batch size. However, this size strongly correlates to the computational efficiency because it defines how much parallelism is available and how much inter-processor communication is required, as we discuss further in Chapter 3 and Chapter 4. Besides, it also affects the quality of convergence; several papers on different machine learning tasks have reported that an inappropriately large mini-batch size results in a degradation in inference accuracy [27, 28, 29, 30]. Keskar et al. [87] reported that this is because large batches decrease the stochasticity of parameter updates, and thus, parameters tend to be stuck in local minima. Therefore, the mini-batch size should be treated carefully to evaluate the performance of DNNs.

An important observation is that, although such metrics correlate with each other, it is not necessarily true that they have a positive correlation (Table 2.1, Figure 2.5). This is mainly because 1) more efficient network architecture is found (regardless theoretically of heuristically) year by year, 2) more compute- and memory-intensive networks become trainable in a reasonable time by hardware performance advances. Therefore, there are many factors involved in the performance of DNNs, and it should be carefully evaluated.

2.2 Training DNNs with HPC

The HPC technologies are inseparable from the deep learning domain advances, as a tremendous amount of computation is required to train DNNs. For example, Krizhevsky et al. [45] took several days to train a single model on two NVIDIA GTX 580 GPUs. Similarly, it is common that the training of one model takes several hours to days [15, 45, 90, 91]. This problem becomes more troublesome when one tries to tune their model to improve its convergence because no or less theoretical estimation of resulting accuracy can be made before running actual training. For this reason, researchers have

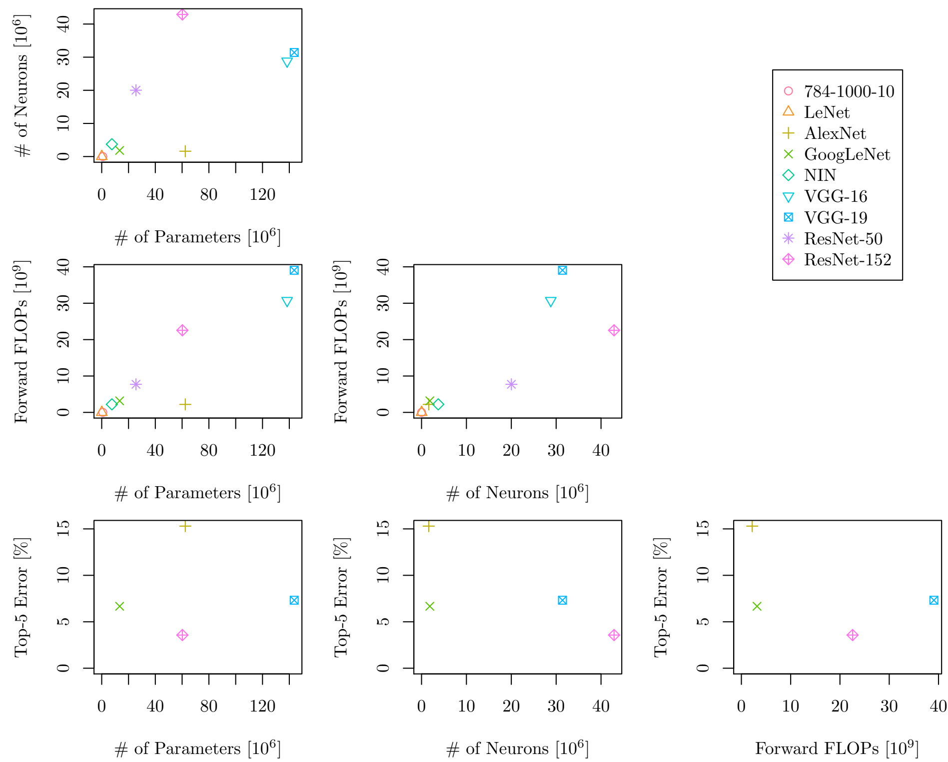


Figure 2.5: **Specifications of popular DNNs [46]**. We estimate "Forward FLOPs" as the number of multiplication and addition of convolutional layers and fully-connected layers. We estimate "# of Parameters" and "# of Neurons" from the variable sizes of the networks defined in Chainer [34] 3.2.0 and a third-party script [89].

to devote much node-hour to optimize their networks, regardless of manually or automatically, by using auto-tuning methods [92]. Therefore, it is essential to utilize efficient hardware (such as GPUs and supercomputers) and efficient software (techniques derived from the HPC domain) to accelerate research on DNNs.

The parallelism that is available in a DNN is classified into two different levels: **intra-processor parallelism**, which is available on each computing processor, and **data-, model-, and hybrid-parallelism**, which define how the network is partitioned among processors. Although the latter group is not necessarily considered when more than one processor is used (for example, batched computation of fully-connected layers with GEMM on a GPU is a kind of both data-parallelism and model-parallelism because each core computes a part of the model in parallel), we only use the group to discuss inter-node parallelism for simplicity.

2.2.1 Intra-processor parallelism

The most straightforward way to exploit parallelism on one computing node to compute DNNs is to utilize multi-core processors. For example, on fully-connected layers, the computation (GEMM) is naturally parallelized by using multi-threaded or GPU-based BLAS libraries, such as cuBLAS [93], OpenBLAS [94], Intel MKL [95] and so on. For other types of layers, many math libraries or kernel libraries designed for deep learning workloads can be used, such as cuDNN [35] and oneDNN [96]. Such libraries typically cover common layer types such as convolutional layers, pooling layers, various activation functions, and batch-normalization layers. Alternatively, such computation can be implemented by using parallel programming models such as CUDA [80] and OpenMP [97]. In fact, many deep learning frameworks adopt such programming models to implement computation kernels that are not supported by any of their underlying kernel libraries.

As explained in Section 2.1, a common technique to improve parallel computation efficiency is to batch the computation by computing multiple samples simultaneously because the computation of one sample does not necessarily keep processors busy. It also reduces memory access to parameter tensors, which are repeatedly used for each data sample. Almost all deep learning frameworks and libraries adopt this methodology. Note that this batch does not necessarily equal the mini-batch size, as this technique is combined with data-parallelism explained in Section 2.2.2.

2.2.2 Data-parallelism

Data-parallel training distributes the computation of different data samples ($\nabla L\left(x_n; W^{(t)}\right)$ of Equation 2.10) among processors. This can be done by 1) performing forward- and backward-computations of different samples on processors simultaneously, and 2) synchronizing parameter gradients among the processors before the parameters are updated. For the synchronization, all-reduce collective communication is typically used, as shown in Figure 2.6. The advantages of

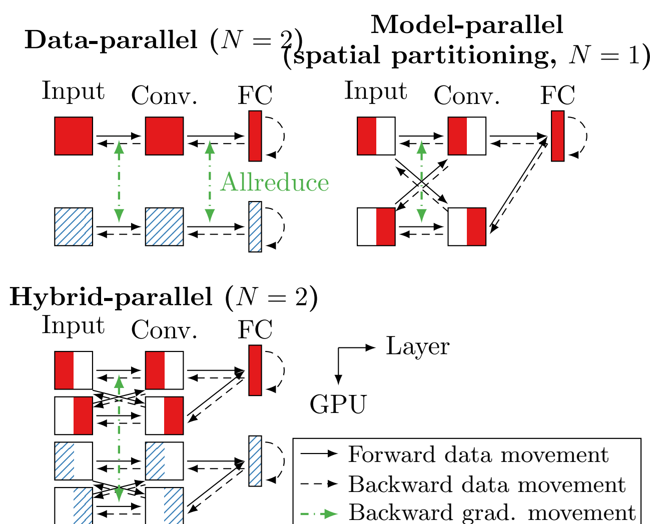


Figure 2.6: **Three parallel strategies for deep neural networks**. "Data-parallel" and "Hybrid-parallel" compute a mini-batch of two samples (■ and ▨). "Model-parallel" splits the spatial dimension of one sample (■) into two GPUs. Data movement within a single process is typically cheap.

data-parallel training than single-processor training are as follows:

- The mini-batch size, or the number of samples of a dataset, is more than one hundred or even thousands in most cases, which makes hundreds of GPUs can be utilized at the same time [27, 28, 29]. Furthermore, since the complexity of all-reduce is $O(N \log P)$, where N is the message size and P is the number of processes [98], this scheme enjoys its good scalability until at least such number of GPUs.

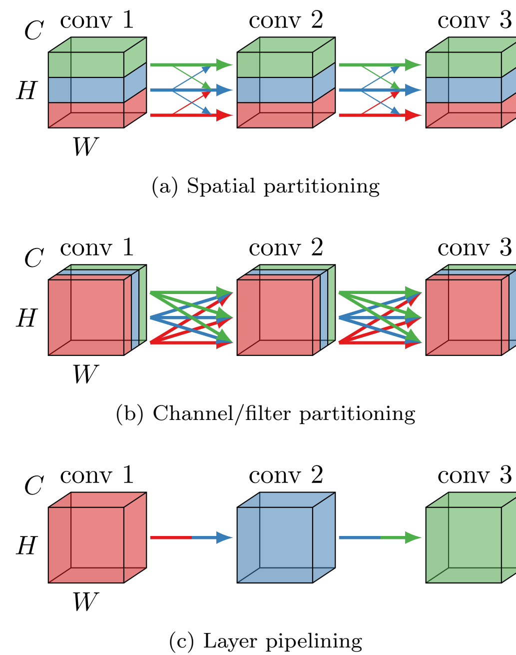
- No load-imbalance is expected as the exact same amount of workload is assigned to each processor.

- It is easily implemented only by updating underlying data readers to make data samples read by each processor don't overlap with each other and inserting all-reduce collectives. In fact, multiple software and libraries such as FireCaffe [99] and Horovod [100] adopt this methodology to extend single-processor or single-node deep learning frameworks to support data-parallelism.

Due to the reasons above, data-parallelism is considered the most popular way to do distributed training. In fact, it is implemented in almost all popular deep learning frameworks.

On the contrary, there are several problems and barriers to use this parallelism in practice:

- The number of processors cannot be larger than the mini-batch size. Considering that too large mini-batch sizes are not undesirable for better convergence, as we mentioned in Section 2.1.3, this limits parallelization on extremely large-scale environments.



(a) Spatial partitioning

(b) Channel/filter partitioning

(c) Layer pipelining

Figure 2.7: **Three parallel strategies of model-parallelism on a three-layer 2D CNN**. The computation on each color is assigned to one of three GPUs. "C" is the channel dimension and "H" and "W" and the spatial dimensions, respectively.

- Each processor must be capable of computing at least one sample. This limitation raises an issue when a large network (in terms of the number of neurons, i.e., the memory footprint) as memory overflow may happen. Since a sample is the undividable parallelism unit, this problem cannot be solved by adjusting the number of processors.

2.2.3 Model-parallelism

Model-parallelism is parallelism available inside a model; we introduce prior implementations that support model-parallelism in Section 5.2. Unlike data-parallelism, which explicitly defines how the computation is partitioned among processes, model-parallelism is a more general concept that a single model is computed on multiple processors.

Model-parallel training includes the following methods:

- **Spatial partitioning** (Figure 2.7a): Each layer is partitioned in one or more spatial dimensions among processes. This method assumes that the target network has large spatial dimensions enough to be parallelized among multiple processors, which implies that multi-dimensional CNNs are suitable to be applied. This method is associated with multi-node stencil applications where a domain is partitioned among processors [101]. As in stencil computations, halo exchange is required to compute each layer that exchanges marginal regions of adjacent processes. The advantage of this method is that a good surface (communication) to volume (computation) ratio

is expected, which efficiently hide the communication overhead. Meanwhile, all the processes have to perform all-reduce to synchronize their own copies of layer parameters, just as in data-parallel training.

- **Channel (filter) partitioning** (Figure 2.7b): Each layer is partitioned in the channel dimension among processes. While this method resembles spatial partitioning, it requires all-to-all communication instead of halo exchanges so that each process receives all channels of the previous layer to compute its own channels.
- **Layer pipelining** (Figure 2.7c): Computation of each layer is distributed among processors. In the forward passes, each process receives the activation of its previous layer, compute the layer's activation, and then sends it to the next process. Similar computation and communication are performed in the backward passes as well. The benefit of this method is that layer parameters reside in a single process, whose movement cost may be much less than that of activation and error signals. This is usually used for networks that have a large number of layers.

The common merit of model-parallelism is that it can distribute the computational and memory pressure among processors. For computation, it can theoretically use an arbitrarily large number of processes, which is infeasible only with data-parallelism, as we explained in Section 2.2.2. For memory pressure, it is capable of handling huge networks whose intermediate data (such as activation and error signals) does not fit into a single processor's memory. Therefore, model-parallelism offers the opportunity to increase both scalability and network performance.

Model-parallelism has a high degree of freedom because 1) multiple methods can be combined at the same time, 2) different methods can be applied to different parts of the target network, and 3) it can be combined with data-parallelism (which is called **hybrid-parallelism**). For example, when spatial partitioning is applied to a CNN, its fully-connected layers at the end are typically left because the number of parameters is much more than the number of activations. However, as explained above, the network placement should be carefully designed to minimize the inter-processor communication overhead and not cause memory overflow. And thus, several studies proposed methods to predict the optimal way to parallelize given networks, including performance models [102, 103, 104].

Table 2.2 shows the pros and cons of the three types of parallelism. The strong advantage of data-parallelism is its nice scalability that is derived from its simple design, while it relies on the assumption that 1) the target network fits in single GPU's memory, and 2) the mini-batch size is larger than the number of available GPUs. Meanwhile, model-parallelism works even when these assumptions do not hold, but it is not trivial how the network should be distributed among processors. Hybrid-parallelism is believed to enjoy the pros of both parallelisms, while the distribution problem still remains.

Table 2.2: **Comparison of data-, model-, and hybrid-parallel computation**. We assume spatial partitioning on multi-dimensional CNNs for model- and hybrid-parallelism. "?" means that it is the user's responsibility to achieve good performance.

	Data	**Model**	**Hybrid**
What to parallelize	Samples	Layers	Samples & layers
Available parallelism	$O(N)$	$O(W^n)$	$O(NW^n)$
GPU memory pressure	✗	✓	✓
Influence on accuracy	✗	✓	✓
Weak scaling	✓	N/A	✓
Strong scaling	✓	✓	✓
Load balance	✓	?	?

Chapter 3

Automatic optimization of computational kernels

As we explained in Chapter 2, since the training of DNNs is compute-bound, it is essential to optimize the computational performance of each accelerator for DNNs as well as parallelization methods on multiple nodes. For this reason, many of today's DL frameworks are built on top of computational kernel libraries for deep learning, just as many HPC applications rely on BLAS libraries for linear algebra operations. However, we have found in my master's thesis [46] that, these uses assume that the underlying libraries always deliver the optimal computational performance, and in practice, such frameworks often achieve only considerably less performance than the peak performance. Specifically, we have shown that such frameworks are indifferent about the choice of algorithms on compute-intensive convolutional layers, and very slow algorithms such as GEMM-based algorithms can be chosen. Therefore, we propose μ-cuDNN, a wrapper library for the NVIDIA cuDNN deep learning kernel library on NVIDIA GPUs, to mitigate this inefficiency by performing auto-tuning of convolution algorithms transparently. In this chapter, we discuss further the possibility of our auto-tuning algorithm by 1) extending μ-cuDNN to consider convolution algorithms using different precisions as a part of algorithm choices to apply to our splitting technique, and 2) proposing a μ-cuDNN interface for optimizing the performance of data-parallel training on multiple GPUs using the performance information collected by the library.

This chapter is organized as follows; In Section 3.1, we introduce the methodology of the cuDNN library. In Section 3.2, we propose the aforementioned μ-cuDNN algorithm. Specifically, we conduct a convolution performance benchmark using DeepBench [58] to examine the further possibility of extending the proposed algorithm. In Section 3.3, we evaluate the performance of our library from different aspects, including a per-layer benchmark and CNN performance benchmarks. We also demonstrate that the interface of the library can be used for high-level load-balance optimization on

Figure 3.1: **An example code to perform cuDNN convolution**. `#` is one of `Forward`, `BackwardData`, or `BackwardFilter`.

```
cudnnHandle_t cudnn;
cudnnCreate(&cudnn);

// Input, output, weights, and convolution descriptors
auto xDesc,yDesc,wDesc,convDesc = ...;

auto algo = cudnnGetConvolution#Algorithm(cudnn, xDesc, WS_LIMIT, ...);
size_t wsSize = cudnnGetConvolution#WorkspaceSize(cudnn, xDesc, algo, ...);
void *ws;
cudaMalloc(&ws, wsSize);

// Training loop
while(true) {
  cudnnConvolution#(cudnn, xDesc, ws, ...);
}
```

a heterogeneous GPU cluster. Further discussion about the future work is found in Chapter 6.1.

3.1 Motivation and background

In this section, we explain an overview of the cuDNN kernel library, our target library to perform auto-tuning. Since cuDNN is used for many deep learning frameworks [32, 33, 34, 35, 38, 39, 41, 42, 44], accelerating techniques on cuDNN is widely applicable to such frameworks independently from framework-specific optimization possibilities.

3.1.1 cuDNN

cuDNN is a math kernel library designed for computing DNNs on NVIDIA GPUs [35]. It implements GPU kernels for various types of DNN layers, including convolutional layers, pooling layers, recurrent layers, and so on. Just as NCCL has been adopted by many applications as a highly-optimized collective communication library for GPUs, cuDNN has been adopted by many deep learning frameworks.

In particular, for the implementation of convolutional layers, cuDNN implements different equivalent convolutional algorithms, such as GEMM-based algorithms, FFT-based algorithms, and Winograd's algorithm. Users specify one of these algorithms when the actual convolution kernel (`cudnnConvolution#` where `#` is either of `Forward`, `BackwardData`, or `BackwardFilter`) is called. At the same time, the users must provide a **workspace** (a GPU array) whose size should be equals to or more than the requirements cuDNN requests. Although it is not strictly mentioned in its document how this workspace is used during convolution, this workspace is used by convolution algorithms to store their non-persistent intermediate data, such as input, output, and weight tensors in the Winograd domain, the frequency domain, or in the im2col form. In fact, as we show in Section 3.2.2, the workspace size is determined by layer architecture and algorithm types. Typical usage of cuDNN for computing a convolutional layer is as follows (Figure 3.1):

1. The framework calls `cudnnFindConvolution#Algorithm` to get a list of available convolution algorithm for the layer. Note that while cuDNN equips various convolution algorithms, each algorithm is not necessarily available depending on the architecture of the layer (Figure 3.1).

2. The framework then chooses the fastest convolution algorithm whose workspace requirement is equal or less than the available memory for workspace. This memory is size may be set by users [34, 38, 39, 44], or the free memory size when the training begins [33].

3. When computing that layer, it executes the cuDNN kernel using the algorithm ID obtained in the previous step and the pre-allocated workspace.

The algorithm is guaranteed to provide the fastest algorithm for a single convolutional layer with a single cuDNN kernel call for the available GPU memory size. However, there are several practical problems with this algorithm:

- Users do not know the exact workspace size required by each layer and algorithm, and thus they may use an inappropriate size limit. In particular, since the performance of each algorithm and the required workspace size depends on the architecture of the layers and training configurations (e.g., mini-batch size), it is difficult to manually provide the optimal size limit for multiple layers at the same time. This can result in a slower algorithm being used, while, in fact, a more efficient algorithm can be used by setting a larger workspace size.

- It is expected that the optimal algorithms will always be used when the workspace size is sufficiently large, but in typical frameworks, it is common that GPU memory is used to the limit. This is because multiple samples of a typical CNN can be trained on a single GPU's memory, so the batch size is set to use up all the memory to increase the computational efficiency (Section 2). Such a scenario is common except in extremely parallel training, so this problem arises even if the framework efficiently uses free memory as a workspace. As we show in Chapter 4, even when performing hybrid-parallel training of a large model that cannot fit on a single GPU, the number of model partitions is minimized (i.e., GPU memory is used to the limit) to increase the computational efficiency of each GPU.

Figure 3.2 illustrates the performance issue on the AlexNet [45] convolutional neural network. As shown in the figure, when the workspace size is set to 512 MiB, faster algorithms (`FFT` and `FFT_TILING`) are used, which were not available when the workspace size was 64 MiB, resulting in about ~ 1.4x of speedup for overall network computation time. At the same time, however, the workspace size used (shown in blue) consumes as much memory as the amount of memory needed to compute the network (shown in red and green), which unrealistic in actual training.

[*] Not implemented as of cuDNN 7.1.2.

[†] $input_width - 2 \times stride$

[‡] 256 if either of the filter sizes is one, or 32 otherwise

[§] The numbers of input/output filters should be multiples of 8, and all of the tensors should be aligned with 128 bits if the filters use the NHWC format.

Table 3.1: **Supported forward convolution algorithms in cuDNN 7.0.4**. “TCs” is whether Tensor Cores [49] can be used. Note that the `DIRECT` convolution algorithm is defined but not implemented (Figure 3.2), but for convolutional layers with 1×1 filters, it is equivalent to GEMM-based convolution in principle.

Algorithm	Input	Kernel	Stride	Dilation	Group	TCs
`IMPLICIT_GEMM`						
`IMPLICIT_PRECOMP_GEMM`				1		✓
`GEMM`				1	1	
`DIRECT` *			N/A			
`FFT`	≤ 256 †	$>$ Padding	1	1	1	
`FFT_TILING`		$\leq 32, 256$ ‡	1	1	1	
`WINOGRAD`		3	1	1	1	
`WINOGRAD_NONFUSED`		3, 5	1	1	1	✓ §

Table 3.2: **Support status of the direct algorithm in cuDNN**.

Version	Status
1	`cudnnConvolutionFwdAlgo_t` does not exist
2	GEMM algorithms and `DIRECT` is defined, but unclear whether it is actually implemented
3,4	FFT algorithms are added, and it is mentioned that `DIRECT` is not implemented
5,6,7	Winograd’s algorithms are added

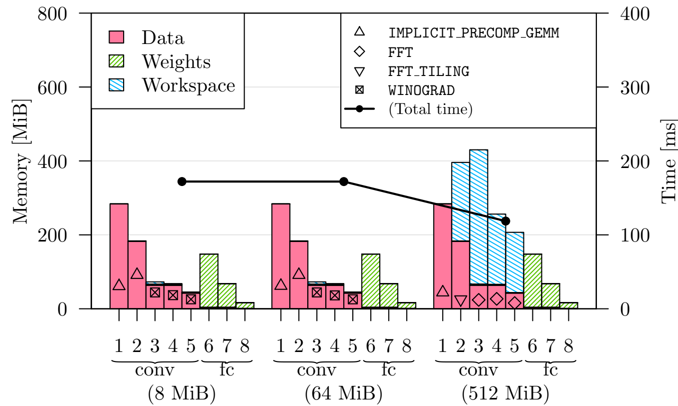


Figure 3.2: **Per-layer breakdowns of memory consumption and computation time of AlexNeton P100-SXM2**. We show the sum of forward and backward pass time. The bars and the points show the per-layer memory consumption and the time, respectively. We use three different workspace sizes (8, 64, 512 MiB), and a mini-batch of 256.

3.2 μ-cuDNN and beyond

μ-cuDNN is a wrapper library for cuDNN to improve the convolution performance transparently. In this section, we first introduce an overview of μ-cuDNN proposed in my master's thesis [46], and then extend its functionality as follows:

- We evaluate the convolution performance using Tensor Cores available in V100 GPUs and show that there is room for optimization in several aspects similar to the problem found in our μ-cuDNN work. We extend the algorithm to use mixed-precision kernels without compromising precision.
- We introduce an interface to μ-cuDNN to collect performance metrics of convolutional layers without interfering in DL frameworks. We demonstrate that this function can be used for high-level performance optimization, such as optimizing workload on a heterogeneous cluster.

3.2.1 μ-cuDNN Overview

μ-cuDNN takes layers' architecture and workspace size via the equivalent interface to cuDNN and provides optimal computational performance for the given workspace size by dividing the computation internally using the cuDNN computational kernels. Its basic algorithm does not change the workspace size itself (that managed by the framework) but internally decomposes convolution into sub-problems, thus making it possible to use a limited workspace to utilize efficient algorithms. Since it performs optimization transparently, it is theoretically applicable to all frameworks that employ cuDNN.

μ-cuDNN takes advantage of the property that the outer-most loop of convolution can be split (Algorithm 1). As shown in the code, the actual computation line (line 8) does not have dependency between itself on different n. This implies that a common loop optimization, **loop splitting**, can be applied to the loop (Algorithm 2). On the pseudo-code, this splitting is not expected to improve computational performance because it does not change the access pattern of each array. However, if we apply this loop division to convolution by cuDNN, we can expect another effect; The cuDNN computation is performed on a per-loop basis, which reduces the workspace size required for each loop. This makes it feasible to use faster algorithms that require a larger workspace. We illustrate our methodology in Figure 3.3; it intercepts cuDNN API calls on convolutional layers to internally split the outer loop of convolution so that faster algorithms are used, while it leaves other cuDNN calls. We refer to applying loop splitting to the mini-batch loop as **micro-batching** and split mini-batch as micro-batches.

Auto-tuning of micro-batch sizes

In this chapter, we mainly focus on the Workspace Reuse (WR) algorithm, one of the two different optimization algorithms of μ-cuDNN, and we refer to WR as the μ-cuDNN algorithm unless it is

Algorithm 1 Pseudo-code of two-dimensional convolution.

1: for($n = 0$; $n < N$; n++) // Mini-batch loop
2: for($k = 0$; $k < K$; k++) // Output channel loop
3: for($h = 0$; $h < H$; h++) // Height loop
4: for($w = 0$; $w < W$; w++) // Width loop
5: for($c = 0$; $c < C$; c++) // Input channel loop
6: for($v = 0$; $v < V$; v++) // Kernel width loop
7: for($u = 0$; $u < U$; u++) // Kernel height loop
8: $\mathbf{Y}[n,k,h,w]$ += $\mathbf{W}[k,c,v,u] \times \mathbf{X}[n,c,h+v,w+u]$;

Algorithm 2 A loop-split code equivalent to Figure 1.

1: for($n = 0$; $n < N_1$; n++) // Micro-batch loop 1
2: for($k = 0$; $k < K$; k++) // Output channel loop
3: for($h = 0$; $h < H$; h++) // Height loop
4: for($w = 0$; $w < W$; w++) // Width loop
5: for($c = 0$; $c < C$; c++) // Input channel loop
6: for($v = 0$; $v < V$; v++) // Kernel width loop
7: for($u = 0$; $u < U$; u++) // Kernel height loop
8: $\mathbf{Y}[n,k,h,w]$ += $\mathbf{W}[k,c,v,u] \times \mathbf{X}[n,c,h+v,w+u]$;
9: for($n = N_1$; $n < N$; n++) // Micro-batch loop 2
10: for($k = 0$; $k < K$; k++) // Output channel loop
11: for($h = 0$; $h < H$; h++) // Height loop
12: for($w = 0$; $w < W$; w++) // Width loop
13: for($c = 0$; $c < C$; c++) // Input channel loop
14: for($v = 0$; $v < V$; v++) // Kernel width loop
15: for($u = 0$; $u < U$; u++) // Kernel height loop
16: $\mathbf{Y}[n,k,h,w]$ += $\mathbf{W}[k,c,v,u] \times \mathbf{X}[n,c,h+v,w+u]$;

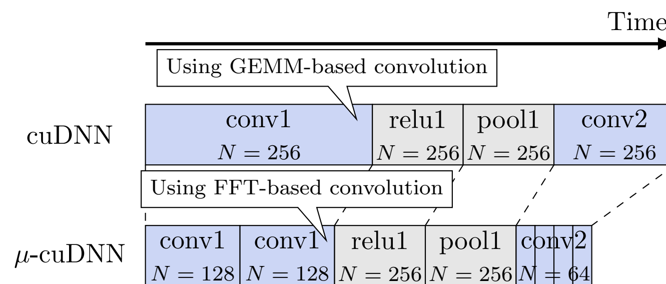


Figure 3.3: **The conceptual execution timeline of μ-cuDNN**. N represents the mini-batch size.

Algorithm 3 **μ-cuDNN's optimization algorithm**.

1: **for** $n = 1$ to N **do**
2: $\quad \hat{n_\mu} \leftarrow \underset{n_\mu=1,2,\ldots,n}{\arg\min} \{T_\mu(n_\mu) + T(n - n_\mu)\}$
3: $\quad T(n) \leftarrow T_\mu(\hat{n_\mu}) + T(n - \hat{n_\mu})$
4: $\quad c(n) \leftarrow \{c_\mu(\hat{n_\mu})\} + c(n - \hat{n_\mu})$
5: **end for**
6: **return** $c(N)$ // Configuration; a list of (algorithm ID, batch size)

explicitly mentioned. This is because the other algorithm, WD, assumes all convolution kernels share a single workspace to allow themselves to be invoked simultaneously, while WR does not make that assumption and is applicable to a broader range of applications. WR optimizes the computation time of convolution layers by splitting a mini-batch into a set of batches. In this algorithm, we use Dynamic Programming (DP) to minimize time.

The object of the algorithm is to minimize the time to compute a given convolutional layer with a mini-batch size of N, $T(N)$. By dividing the mini-batch N into one or more batches, $T(N)$ is formulated as follows:

$$T(N) = \min \left\{ \begin{array}{l} T_\mu(N), \\ \min_{n=1,2,\ldots,N-1} T(n) + T(N-n) \end{array} \right\}, \tag{3.1}$$

where $T_\mu(n)$ is cuDNN's convolution time with a batch size of n. This time is obtained by benchmarking cuDNN inside μ-cuDNN before performing the optimization. Note that it is infeasible to benchmark $T_\mu(n)$ in advance of optimization with all possible layer architecture due to its extremely large exploration space (such as the number of input/output channels, spatial input dimensions, the mini-batch size, the filter size, the filter stride, and etc.); instead, we run cuDNN kernels online to take a minimum amount of time for benchmarking. Since each cuDNN call is performed inside the μ-cuDNN and guaranteed not to be executed in parallel, we assume that each kernel has access to the full workspace size specified by users via the cuDNN interface. In this model, we don't consider any cost to split a batch because this can be done by calling cuDNN's convolution by applying different offsets to the input and output pointers.

Since n is an integer and $T(n)$ only refers $T(n')$ where $n' < n$, this problem can be solved by DP. We show the pseudo-code of the algorithm in (Algorithm 3), where $c_\mu(n)$ is a pair of a convolution algorithm ID and a batch size that is computed by the algorithm. The output of the algorithm is a list of such pairs, which we call a **configuration**. For example, if the output is $c(256) = \{(\texttt{FFT}, 100), (\texttt{FFT}, 100), (\texttt{GEMM}, 56)\}$, μ-cuDNN computes 100 samples of the first two groups with the FFT-based algorithm and the remaining 56 samples with GEMM.

We propose different micro-batch size granularities to evaluate the tradeoff between the optimization overhead and the acceleration obtained by the optimization algorithm. Since our

Table 3.3: **Micro-batch size policies**.

	Micro-batch size(s)	# of benchmark calls
`all`	$\{1, 2, 3, \cdots, N\}$	$\mathcal{O}(N)$
`powerOfTwo`	$\{2^0, 2^1, 2^2, \cdots, N\}$	$\mathcal{O}(\log N)$
`undivided`	$\{N\}$	$\mathcal{O}(1)$

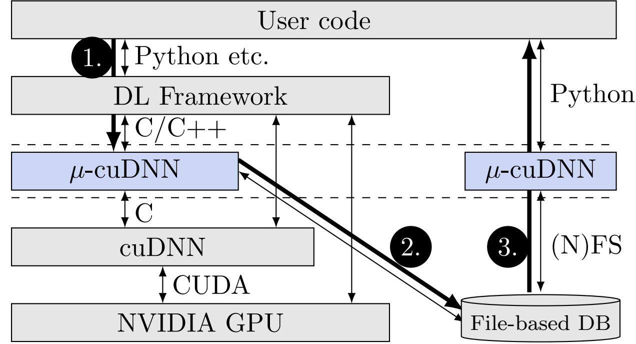


Figure 3.4: **μ-cuDNN software stack**. Users can obtain performance metrics via 1. to 3., as described in Section 3.2.4.

optimization needs actually to call cuDNN kernels to obtain $T_\mu(n)$, and the benchmark for each batch size has to run all the (potentially slow) available algorithms, it involves the overhead of an order of the mini-batch size. However, since the algorithm allows using a subset of $\{1, 2, \cdots, N\}$ as available micro-batch sizes if only there is a combination of batch sizes equal to the mini-batch size, we can reduce the overhead by using such a subset instead of all positive integers up to the mini-batch size. Users must specify either of the policies to μ-cuDNN by setting an environment variable; Note that environment variables can be easily propagated to other computes nodes by using the job scheduler and MPI runtime functions, so this feature can be used in multi-node training too. An essential point about `powerOfTwo` is that several convolution algorithms, including those of cuDNN, prefer input and output widths of two [105], which are feasible with the policy. Note that `undivided` is used for debugging purposes only.

Implementation details

Figure 3.4 shows the software stack of a deep learning framework using μ-cuDNN. Since the C language is used for the cuDNN interface, and many frameworks are implemented in C or C++, μ-cuDNN is inserted between these and being compiled as part of the framework. Users usually use Python to build DNN models on top of a framework, and thus μ-cuDNN is used transparently.

Figure 3.5 shows the workflow of μ-cuDNN. When the framework tries to retrieve the algorithm for each layer, μ-cuDNN intercepts the architecture information and workspace limit of the layer and then calculates the optimal micro-batch combinations by invoking its optimizer internally. After that, μ-cuDNN returns the workspace size and a dummy algorithm ID, but this ID is ignored when cuDNN

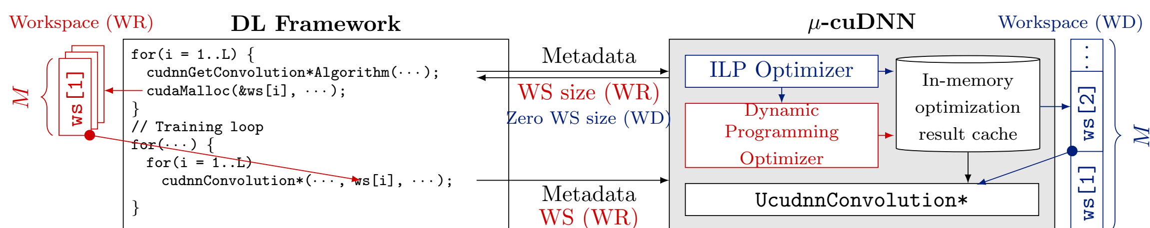


Figure 3.5: **Overview of μ-cuDNN**. μ-cuDNN optimizes micro-batch sizes and internally calls cuDNN functions, with either a per-layer workspace (WR) or a part of a global workspace (WD) of a size limit M.

Figure 3.6: **An example code to perform convolution with μ-cuDNN**. μ-cuDNN is enabled by replacing the `cudnnHandle_t` handle type with `UcudnnHandle_t`.

```
1  // --- u-cuDNN header file ---
2  cudnnStatus_t cudnnGetConvolution#Algorithm(UcudnnHandle_t cudnn, ...) {
3    return UcudnnGetConvolution#Algorithm(cudnn, ...);
4  }
5  // ---------------------------
6
7  UcudnnHandle_t cudnn;
8  cudnnCreate(&cudnn);
9
10 // Input, output, weights, and convolution descriptors
11 auto xDesc,yDesc,wDesc,convDesc = ...;
12
13 auto algo = cudnnGetConvolution#Algorithm(cudnn, xDesc, WS_LIMIT, ...);
14 size_t wsSize = cudnnGetConvolution#WorkspaceSize(cudnn, xDesc, algo, ...);
15 void *ws;
16 cudaMalloc(&ws, wsSize);
17
18 // Training loop
19 while(true) {
20   cudnnConvolution#(cudnn, xDesc, ws, ...);
21 }
```

is called at the time of training but performs convolution using its internal cached configuration.

Figure 3.6 shows an example code to replace cuDNN with μ-cuDNN. μ-cuDNN defines its handle type (`UcudnnHandle_t`) and overloads a part of cuDNN functions that are related to convolutional layers with it to intercept the control flow. Users can introduce μ-cuDNN to their deep learning framework by replacing cuDNN's handle type with that of μ-cuDNN. For other cuDNN functions, μ-cuDNN implements a cast function between their handle types to just call the cuDNN functions with a μ-cuDNN handle.

3.2.2 Performance analysis on cuDNN kernels using DeepBench

In this section, we investigate the convolution performance of NVIDIA GPUs from various aspects. We use the DeepBench benchmark suite [58] to test convolutional layers from a wide variety of CNNs for different tasks, such as ResNet [65] and Deep Speech [14]. We investigate the computational performance of various algorithms, layer architectures, and computational precisions. We show that μ-cuDNN's algorithm can be used for more general optimization than just optimizing on a limited

workspace size.

DeepBench

DeepBench is a set of benchmarks that collects various computational and communication kernels that appeared in DNNs. While many other benchmarks [106, 107, 108] are based on the computational performance of particular networks and the time to train such networks, DeepBench encompasses various layer-wise computational kernels and communication patterns of DNNs. This is because the computational performance of a network depends on various factors, such as how frameworks execute their internal computational graphs. On the other hand, per-kernel evaluation can accurately assess the computational performance of individual accelerators used for deep learning. For this reason, DeepBench is suitable for evaluating what properties of layers a kernel library for deep learning, such as cuDNN, performs efficiently.

Table 3.4 shows the list of convolutional layers of DeepBench. We summarize the characteristics of each layer on a task by task basis in Table 3.5. As shown in the tables, DeepBench encompasses convolutional layers designed for different tasks and even includes uncommon layers with anisotropic filter sizes. Since in the master's thesis, we have only used one or two types of CNNS for our evaluation, and thus, its performance in general cases has not been assessed, we use this benchmark in this thesis.

In this section, we only feature some of the DeepBench kernels exhibiting characteristic behavior for cuDNN. We show all experimental results using all DeepBench convolutional layers on three different GPUs in Appendix A.

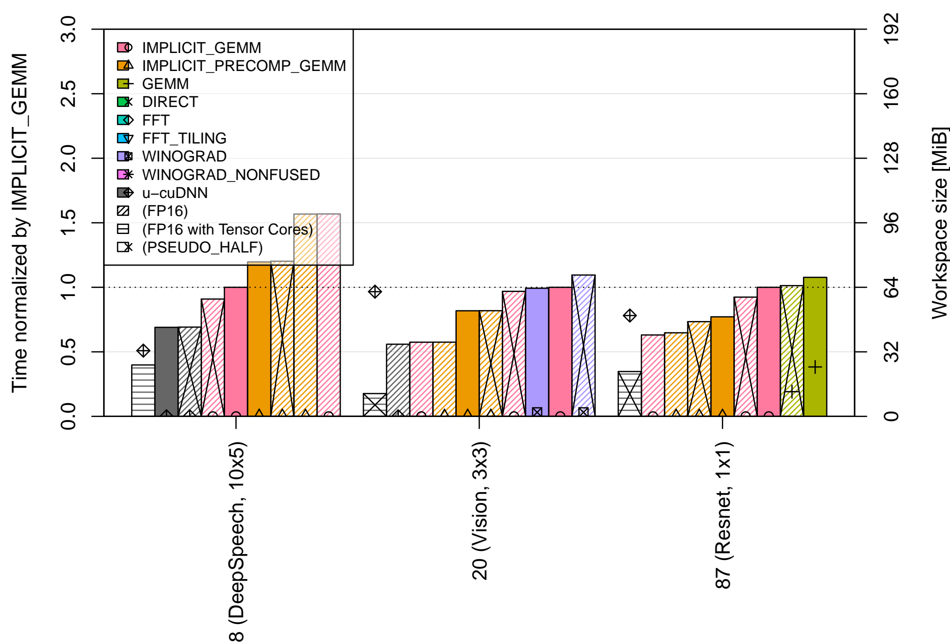
Figure 3.7a shows an example where μ-cuDNN works well even for a GPU with Tensor Cores [49]. In cuDNN, not all algorithms are necessarily available for all computational accuracies; in these layers, only `IMPLICIT_PRECOMP_GEMM` and `WINOGRAD_NONFUSED` support Tensor Cores, and comparing Figure 3.7a and Figure 3.7b, we can see that we cannot use them if the workspace limit is less than 64 MiB. On the other hand, μ-cuDNN can use these algorithms with more than half of the 64 MiB workspace, in the same principle as described in Section 3.3. This result implies that μ-cuDNN's assumption that the division of workspace usage results in faster convolution still holds on such GPUs using the specialized units.

3.2.3 Micro-batching with mixed-precision

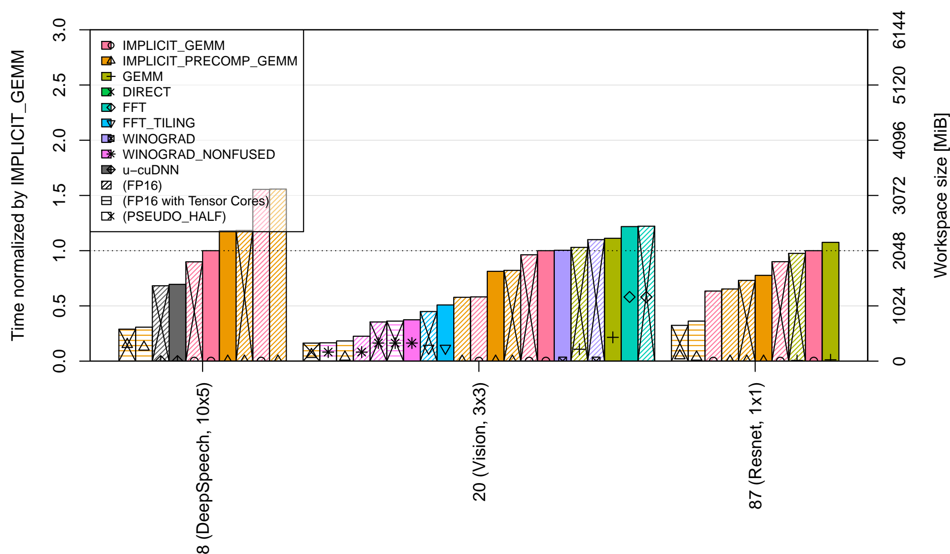
In this section, we investigate cuDNN's convolution performance with different numerical precisions. In cuDNN, users can separately specify the data type for holding the input, output, and weight tensors (or their derivatives) and the data type used for the operation (Table 3.6). The configuration for using only the half or float data type is called `TRUE_HALF` or `FLOAT`, respectively, but using half for data store and float for computation is also a valid combination called `PSEUDO_HALF`. One of the advantages of using `PSEUDO_HALF` is that convolution algorithms for single-precision that have been implemented in

Table 3.4: **The convolutional layers of DeepBench.**

id	source	W	H	C_{in}	N	C_{out}	$kernel_w$	$kernel_h$	pad_w	pad_h	$stride_w$	$stride_h$
1	DeepSpeech	700	161	1	4	32	20	5	0	0	2	2
2	DeepSpeech	700	161	1	8	32	20	5	0	0	2	2
3	DeepSpeech	700	161	1	16	32	20	5	0	0	2	2
4	DeepSpeech	700	161	1	32	32	20	5	0	0	2	2
5	DeepSpeech	341	79	32	4	32	10	5	0	0	2	2
6	DeepSpeech	341	79	32	8	32	10	5	0	0	2	2
7	DeepSpeech	341	79	32	16	32	10	5	0	0	2	2
8	DeepSpeech	341	79	32	32	32	10	5	0	0	2	2
9	OCR	480	48	1	16	16	3	3	1	1	1	1
10	OCR	240	24	16	16	32	3	3	1	1	1	1
11	OCR	120	12	32	16	64	3	3	1	1	1	1
12	OCR	60	6	64	16	128	3	3	1	1	1	1
13	Face Recognition	108	108	3	8	64	3	3	1	1	2	2
14	Face Recognition	54	54	64	8	64	3	3	1	1	1	1
15	Face Recognition	27	27	128	8	128	3	3	1	1	1	1
16	Face Recognition	14	14	128	8	256	3	3	1	1	1	1
17	Face Recognition	7	7	256	8	512	3	3	1	1	1	1
18	Vision	224	224	3	8	64	3	3	1	1	1	1
19	Vision	112	112	64	8	128	3	3	1	1	1	1
20	Vision	56	56	128	8	256	3	3	1	1	1	1
21	Vision	28	28	256	8	512	3	3	1	1	1	1
22	Vision	14	14	512	8	512	3	3	1	1	1	1
23	Vision	7	7	512	8	512	3	3	1	1	1	1
24	Vision	224	224	3	16	64	3	3	1	1	1	1
25	Vision	112	112	64	16	128	3	3	1	1	1	1
26	Vision	56	56	128	16	256	3	3	1	1	1	1
27	Vision	28	28	256	16	512	3	3	1	1	1	1
28	Vision	14	14	512	16	512	3	3	1	1	1	1
29	Vision	7	7	512	16	512	3	3	1	1	1	1
30	Vision	224	224	3	16	64	7	7	3	3	2	2
31	Vision	28	28	192	16	32	5	5	2	2	1	1
32	Vision	28	28	192	16	64	1	1	0	0	1	1
33	Vision	14	14	512	16	48	5	5	2	2	1	1
34	Vision	14	14	512	16	192	1	1	0	0	1	1
35	Vision	7	7	832	16	256	1	1	0	0	1	1
36	Vision	7	7	832	16	128	5	5	2	2	1	1
37	Face Recognition	56	56	64	8	64	3	3	1	1	1	1
38	Face Recognition	56	56	64	8	256	1	1	0	0	2	2
39	Face Recognition	28	28	128	8	128	3	3	1	1	1	1
40	Face Recognition	28	28	128	8	512	1	1	0	0	2	2
41	Face Recognition	14	14	256	8	256	1	1	0	0	1	1
42	Face Recognition	14	14	256	8	256	3	3	1	1	1	1
43	Face Recognition	14	14	256	8	1024	1	1	0	0	2	2
44	Face Recognition	7	7	512	8	512	1	1	0	0	1	1
45	Face Recognition	7	7	2048	8	512	1	1	3	3	2	2
46	Face Recognition	56	56	64	16	64	3	3	1	1	1	1
47	Face Recognition	56	56	64	16	256	1	1	0	0	2	2
48	Face Recognition	28	28	128	16	128	3	3	1	1	1	1
49	Face Recognition	28	28	128	16	512	1	1	0	0	2	2
50	Face Recognition	14	14	256	16	256	1	1	0	0	1	1
51	Face Recognition	14	14	256	16	256	3	3	1	1	1	1
52	Face Recognition	14	14	256	16	1024	1	1	0	0	2	2
53	Face Recognition	7	7	512	16	512	1	1	0	0	1	1
54	Face Recognition	7	7	2048	16	512	1	1	3	3	2	2
55	Speaker ID	700	161	1	16	64	5	5	1	1	2	2
56	Speaker ID	350	80	64	16	64	3	3	1	1	1	1
57	Speaker ID	350	80	64	16	128	5	5	1	1	2	2
58	Speaker ID	175	40	128	16	128	3	3	1	1	1	1
59	Speaker ID	175	40	128	16	256	5	5	1	1	2	2
60	Speaker ID	84	20	256	16	256	3	3	1	1	1	1
61	Speaker ID	84	20	256	16	512	5	5	1	1	2	2
62	Speaker ID	42	10	512	16	512	3	3	1	1	1	1
63	Resnet	112	112	64	8	64	1	1	0	0	1	1
64	Resnet	56	56	64	8	256	1	1	0	0	1	1
65	Resnet	56	56	256	8	64	1	1	0	0	1	1
66	Resnet	56	56	256	8	128	1	1	0	0	2	2
67	Resnet	28	28	128	8	512	1	1	0	0	1	1
68	Resnet	28	28	512	8	128	1	1	0	0	1	1
69	Resnet	28	28	512	8	256	1	1	0	0	2	2
70	Resnet	14	14	256	8	1024	1	1	0	0	1	1
71	Resnet	28	28	512	8	1024	1	1	0	0	2	2
72	Resnet	14	14	1024	8	256	1	1	0	0	1	1
73	Resnet	14	14	256	8	1024	1	1	0	0	1	1
74	Resnet	14	14	1024	8	512	1	1	0	0	2	2
75	Resnet	7	7	512	8	512	3	3	1	1	1	1
76	Resnet	7	7	512	8	2048	1	1	0	0	1	1
77	Resnet	14	14	1024	8	2048	1	1	0	0	2	2
78	Resnet	7	7	2048	8	512	1	1	0	0	1	1
79	Resnet	112	112	64	16	64	1	1	0	0	1	1
80	Resnet	56	56	64	16	256	1	1	0	0	1	1
81	Resnet	56	56	256	16	64	1	1	0	0	1	1
82	Resnet	56	56	256	16	128	1	1	0	0	2	2
83	Resnet	28	28	128	16	512	1	1	0	0	1	1
84	Resnet	28	28	512	16	128	1	1	0	0	1	1
85	Resnet	28	28	512	16	256	1	1	0	0	2	2
86	Resnet	14	14	256	16	1024	1	1	0	0	1	1
87	Resnet	28	28	512	16	1024	1	1	0	0	2	2
88	Resnet	14	14	1024	16	256	1	1	0	0	1	1
89	Resnet	14	14	256	16	1024	1	1	0	0	1	1
90	Resnet	14	14	1024	16	512	1	1	0	0	2	2
91	Resnet	7	7	512	16	512	3	3	1	1	1	1
92	Resnet	7	7	512	16	2048	1	1	0	0	1	1
93	Resnet	14	14	1024	16	2048	1	1	0	0	2	2
94	Resnet	7	7	2048	16	512	1	1	0	0	1	1



(a) 64 MiB workspace



(b) 2048 MiB workspace

Figure 3.7: **Convolution performance of DeepBench convolutional layers on V100-SXM2**. Time and required workspace sizes are shown as bars and points, respectively. Time is normalized by the time of the single-precision `IMPLICIT_GEMM` algorithm. Each label shows its ID number, original task, and filter size listed in Table 3.5. For bars where their normalized time exceeds the vertical axis, we omit the top ends and show the time in the text. We show a cross on each bar that uses the `PSEUDO_HALF` configuration. We omit a μ-cuDNN bar when it fails to find a faster configuration than other existing algorithms.

Table 3.5: **Number of convolutional layers in DeepBench**.

Network type	Kernel size	Batch size	# of layers
DeepSpeech	10×5, 20×5	4, 8, 16, 32	8
OCR	3×3	16	4
Face Recognition	1×1, 3×3	8, 16	23
Vision	1×1, 3×3, 5×5, 7×7	8, 16	19
Speaker ID	3×3, 5×5	16	8
ResNet	1×1, 3×3	8, 16	32

Table 3.6: **Available data types in cuDNN's convolution**.

Configuration	Tensors' Data Type	Compute Type	Algorithms
`TRUE_HALF`	`HALF`	`HALF`	GEMM, Winograd
`PSEUDO_HALF`	`HALF`	`FLOAT`	GEMM, Winograd, FFT
`FLOAT`	`FLOAT`	`FLOAT`	GEMM, Winograd, FFT

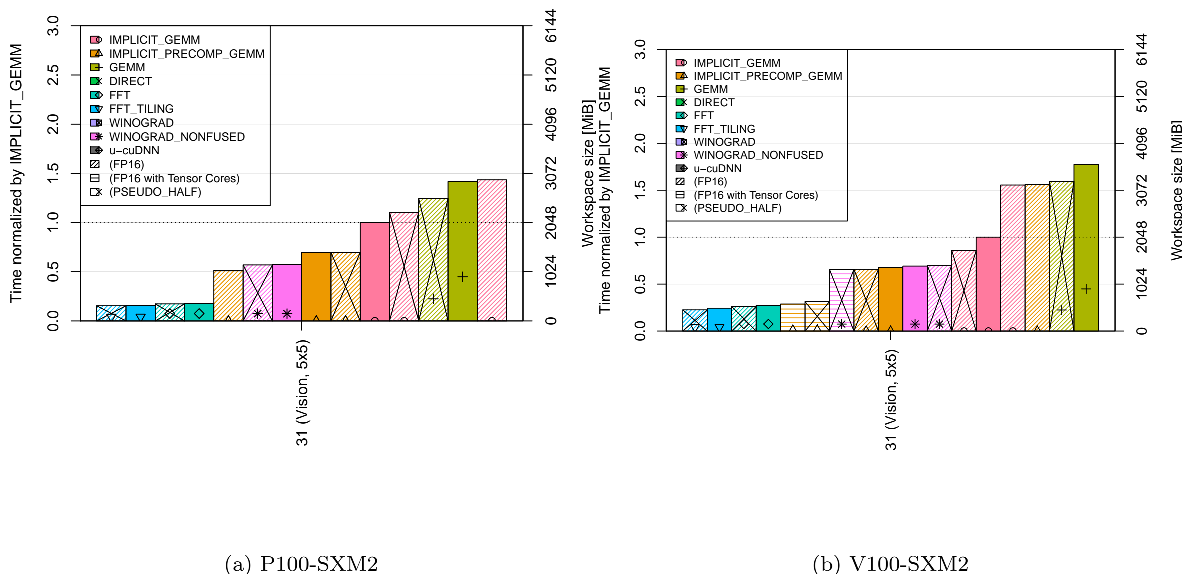
cuDNN can be used. Frameworks rarely switch between these configurations adaptively, and generally, only one of them is used at a time (Table 3.7).

We found several cases where `PSEUDO_HALF` is competitive or even faster than other configurations with a sufficient workspace limit (Figure 3.8, Figure 3.9, Figure 3.10). There are several interesting observations, as follows:

- Figure 3.8 is a typical case that `PSEUDO_HALF` works better by using an algorithm not implemented in `TRUE_HALF` (regardless of whether Tensor Cores are available or not.) However, even if an algorithm is implemented in both configurations, there are cases where `PSEUDO_HALF` works better (Figure 3.10). Note that the source code of cuDNN is not publicly available, so each algorithm's details, such as blocking sizes, are not necessarily identical for different data type configurations.
- We can get additional performance gains by using `PSEUDO_HALF` not only for uncommon filter sizes that may not be very well optimized (Figure 3.9), but also for common filters such as 3×3 (Figure 3.10).
- Since the workspace limit is set large enough in all cases, incorporating `PSEUDO_HALF`'s algorithms as choices would allow μ-cuDNN to make performance improvements that cannot be solved by just increasing the workspace limit.

Table 3.7: **cuDNN runtime parameters of various deep learning frameworks**.

Framework	Version	`cudnnConvolutionMode_t`	`cudnnTensorFormat_t`	Compute type for `HALF` Tensors
Caffe	1.0	`CROSS_CORRELATION`	`TENSOR_NCHW`	N/A
Caffe2	v0.8.1	`CROSS_CORRELATION`	`TENSOR_N[HWC,CHW]`	`TRUE_HALF`
TensorFlow	v1.7.0	`CROSS_CORRELATION`	`TENSOR_NCHW[_VECT_C]`	{`TRUE`,`PSEUDO`}`_HALF`
Chainer (CuPy)	v2.5.0	`CROSS_CORRELATION`	`TENSOR_NCHW`	`PSEUDO_HALF`



(a) P100-SXM2

(b) V100-SXM2

Figure 3.8: **Convolution performance of a DeepBench's 5×5 convolutional layer**. We use a workspace size of 2 GiB.

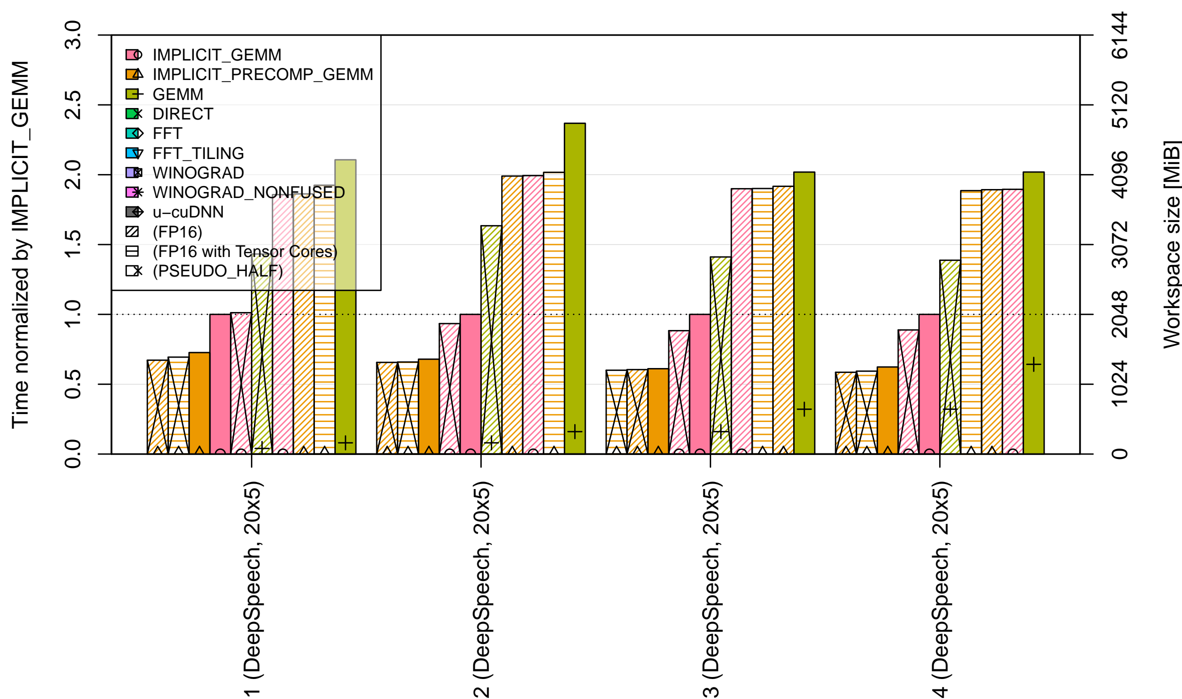


Figure 3.9: **Convolution performance of DeepBench's 20×5 convolutional layers on V100-SXM2**. We use a workspace size of 2 GiB.

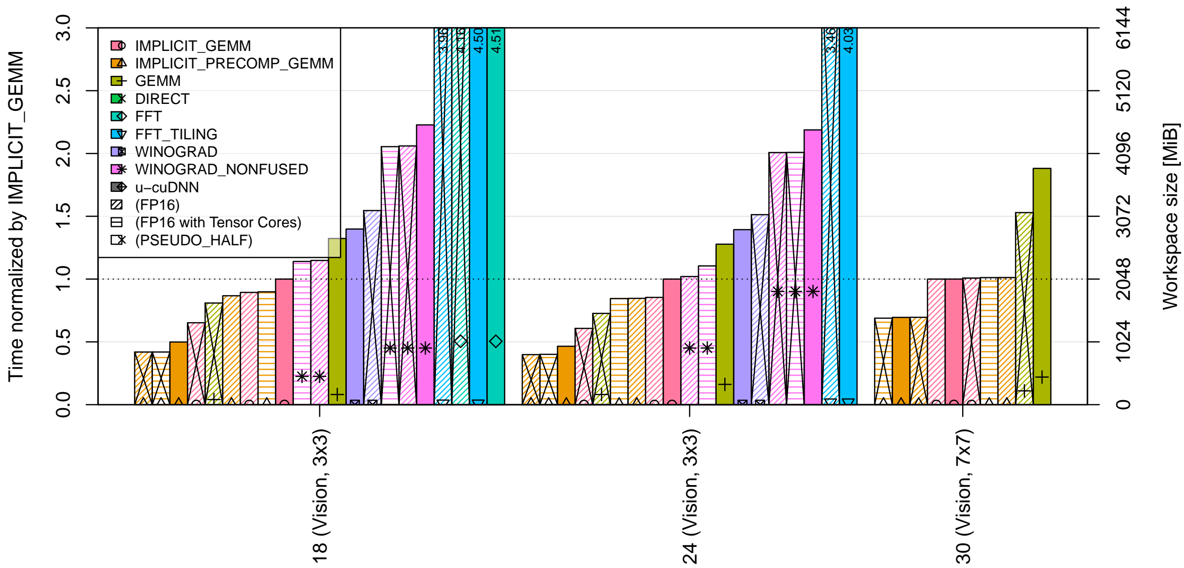


Figure 3.10: **Convolution performance of DeepBench's** 3×3 **and** 7×7 **convolutional layers on V100-SXM2**. We use a workspace size of 2 GiB.

Therefore, we extend μ-cuDNN so that when `TRUE_HALF` is specified by users μ-cuDNN ignores it but uses both the `TRUE_HALF` and `PSEUDO_HALF` algorithms instead. This change is conservative because this feature does not implicitly degrade calculation precision. Besides, if `FLOAT` is specified, we use `FLOAT` as before because using half-precision arithmetic may cause unexpected calculation errors and rounding.

3.2.4 High-level optimization with μ-cuDNN

In this section, we introduce μ-cuDNN's Python interface. Since the convolutional layers' parameters (such as the kernel shape and input and output tensor shapes) are passed through μ-cuDNN, the interface can simplify the analysis and optimization of convolutional layers, regardless of the underlying deep learning framework. Indeed, since almost all deep learning frameworks exploit cuDNN, this interface provides widely-applicable utilities to end-users. This feature is similar to the cuDNN's API logging feature, but cuDNN only records the order and arguments of its function calls and is intended to be used for debugging, so users cannot use it for performance evaluation.

In this interface, μ-cuDNN first transparently stores benchmark results, coupled with layer parameters, in an SQLite file-based database (1. and 2. of Figure 3.4). This database contains a table of performance metrics of the cuDNN's benchmarking functions for specific GPUs, a table of layer IDs, and corresponding layer parameters. Then, the Python interface loads the results from the

Figure 3.11: **The μ-cuDNN interface with layer IDs**.

```
1  UcudnnHandle_t cudnn;
2  cudnnCreate(&cudnn);
3
4  // Input, output, weights, and convolution descriptors
5  auto xDesc,yDesc,wDesc,convDesc = ...;
6
7  auto algo = cudnnGetConvolution#Algorithm(cudnn, ..., layer_id);
8  size_t wsSize = cudnnGetConvolution#WorkspaceSize(cudnn, ..., layer_id);
9  void *ws;
10 cudaMalloc(&ws, wsSize);
11
12 // Training loop
13 while(true) {
14   cudnnConvolution#(cudnn, xDesc, ws, ..., layer_id);
15 }
```

database, without involving the framework itself (3.). This scheme decouples framework-specific codes from the analysis process itself. This file-based caching is also beneficial to eliminate repetition of benchmarking for the optimization scheme described in Section 3.2.1.

One concern when extracting network parameters from the information passed through the cuDNN interface is that it does not provide unique identifiers for layers, such as its names. To solve this problem, μ-cuDNN requires an extra "layer ID" argument to be passed to the μ-cuDNN functions (Figure 3.11). This requirement can easily be fulfilled in most frameworks, as frameworks usually keep internal layer identifiers internally, as shown in the figure.

3.3 Evaluation

We evaluate the performance of μ-cuDNN on three different GPU architectures, NVIDIA Tesla K80 [55], P100-SXM2 [56], and V100-SXM2 [49] on the TSUBAME-KFC/DL, TSUBAME 3.0 supercomputers, and an NVIDIA DGX-1, respectively (Table 3.8). We also use a Tesla K20Xm and GTX 750Ti on TSUBAME-KFC/DL for our heterogeneous cluster optimization case-study (Section 3.3.3). Note that a K80 GPU contains two distinct GK210 chips, and we show performance results of a single GK210 as "K80". We use cuDNN 6.0 and 7.1.2 if available. The full hardware/software specifications on the three supercomputers are shown in Table 3.8.

Table 3.9 summarizes the experimental configurations we used. Throughout our evaluation, unless explicitly mentioned, we use the single-precision floating-point format and store tensors in the (N, H, C, W) storage order, which is the most common configuration in deep learning applications. We use three different deep learning frameworks for evaluations: Caffe [38], its NVIDIA branch (NVCaffe) [40], and TensorFlow [33]. We use a built-in benchmarking command (Caffe's "time" command) or an official benchmarking script (from TensorFlow model repository [109]) to measure the execution time of forward and backward passes and show the sum of forward and backward passes together. In the following sections, unless explicitly mentioned, each forward-backward pass is measured 50 times on both Caffe and TensorFlow[]. For CNNs, we use AlexNet [45], ResNet [65],

Table 3.8: **Evaluation environment**.

	TSUBAME-KFC/DL	TSUBAME 3.0	NVIDIA DGX-1
CPU (Intel Xeon)	E5-2620 × 2	E5-2680 v4 × 2	E5-2698 v4 × 2
GPU (NVIDIA Tesla)	K80 × 4 - 8.73 SP TFlop/s - 24 GiB GDDR5 (480 GiB/s BW)	P100-SXM2 × 4 - 10.6 SP TFlop/s - 16 GiB HBM2 (732 GiB/s BW)	V100-SXM2 × 8 - 15.7 SP TFlop/s - 16 GiB HBM2 (900 GiB/s BW)
OS	CentOS 7.3.1611	SUSE Linux Enterprise Server 12 SP2	Ubuntu 16.04.3
CUDA	8.0.61 / 9.1.85	8.0.44	9.0.176
cuDNN	6.0 / 7.1.2	6.0	7.1.2
GLPK	4.63	4.63	N/A
Caffe	1.0	1.0	NVCaffe v0.16.5 [40]
TensorFlow	N/A	1.4.1	N/A

Table 3.9: **Evaluation settings**.

	Workspace limit [MiB]	Neural Architecture	GPU(s)
Micro-benchmark	64	DeepBench	K80, P100, V100
Caffe	8, 64, 512	AlexNet, ResNet	K80, P100, V100
TensorFlow	64	AlexNet, ResNet, DenseNet	P100
Hetero. cluster opt.	64	ResNet	K80, K20Xm, 750Ti

and DenseNet [66]. For evaluations on Caffe, we use the AlexNet model defined in Caffe, ResNet-18, and ResNet-50 from NVCaffe. We modify data prefetching size from 4 to 16 for AlexNet and ResNet-18 for TSUBAME 3.0 to hide the I/O latency sufficiently. For evaluations on TensorFlow, we use the definitions in an official benchmarking repository [110]. We also use a DenseNet model definition for Caffe [111].

As for workspace limit, unless explicitly mentioned, we use 8 MiB and 64 MiB for each layer, which are the default workspace size limits of Caffe and Caffe2 [39], respectively. Also, we use 512 MiB of workspace per layer to investigate the case where sufficiently large workspace is provided. To shorten the benchmarking time, we use several GPUs on the same node with the parallel evaluation function of μ-cuDNN, as mentioned in Section 3.2.4. Its performance database allows us to automatically collect metrics for each computational kernel by simply calling cuDNN for different batch sizes in parallel. Note that this measurement gives the same results as when using a single GPU.

3.3.1 DeepBench

Figure 3.12 shows the speedup of each convolutional layer against cuDNN on three different GPUs. In this section, we use the latest version of cuDNN (7.1.2), and set the workspace limit to 64 MiB. Also,

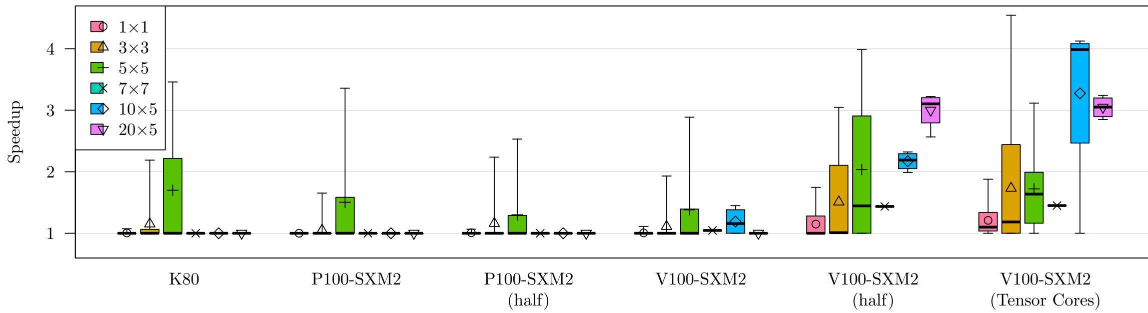


Figure 3.12: **Relative speedups of DeepBench's forward convolution against cuDNN**. The whiskers and the points represent minimum/maximum speedups and their means, respectively. We use 64 MiB workspace size.

we multiply the batch size by 4 from the original size because the batch sizes defined in DeepBench are generally smaller than the actual sizes used (e.g., its ResNet layers use 8 or 16, while many papers use 32 [27, 65]), so increasing the batch size allows for more practical performance evaluation.

In Figure 3.12, not only μ-cuDNN accelerates the frequently-used 3×3 and 5×5 kernels for all the GPUs, but it also achieves up to 4.54x speedup (1.60x on average) on a V100-SXM2 GPU when the computation is done with half-precision and Tensor Cores. In such cases, μ-cuDNN adopts the `PSEUDO_HALF` configuration for 65 kernels (69% of the kernels), and `TRUE_HALF` for the others. This observation demonstrates that μ-cuDNN successfully avoids new implementations that are potentially inefficient. Besides, Tensor Core-enabled convolutions tend to consume a large amount of memory (up to 437.73 MiB, 64.6 MiB on average on V100-SXM2) since GEMM-based convolution is more efficient but require a large workspace to rearrange the elements of an input tensor. This workspace requirement is, however, naturally resolved by μ-cuDNN's loop splitting algorithm. More importantly, μ-cuDNN can accelerate 3×3 kernels in half-precision, which are usually adopted in recent CNNs, by 1.16x on P100-SXM2 and 1.73x on V100-SXM2, on average, respectively.

3.3.2 CNN performance

We evaluate the performance of our library by benchmarking various CNNs on two different frameworks, Caffe and TensorFlow. Since CNN computation involves not only convolution but also other layers of computation (such as pooling and activation functions) and inter-GPU communication to gradient synchronization, evaluating deep learning frameworks allow us to assess how effective the acceleration of μ-cuDNN is in actual training. Note that a part of our experiments (Figure 3.13, Figure 3.14a, and Figure 3.14b) has been shown in my master's thesis [46].

Figure 3.13 shows the execution time of forward convolution (`cudnnConvolutionForward`) of the "conv2" layer of AlexNet on P100-SXM2. When the workspace size limit is 64 MiB, and the mini-batch is not divided, the GEMM-based algorithm is selected by cuDNN, requiring only 4.3 KiB for workspace.

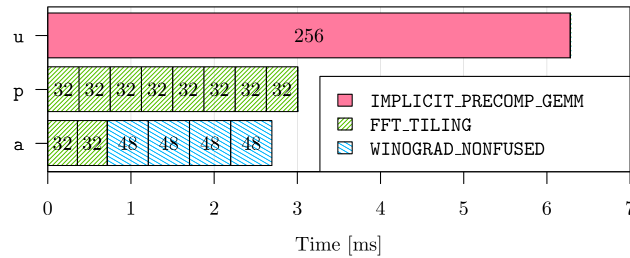


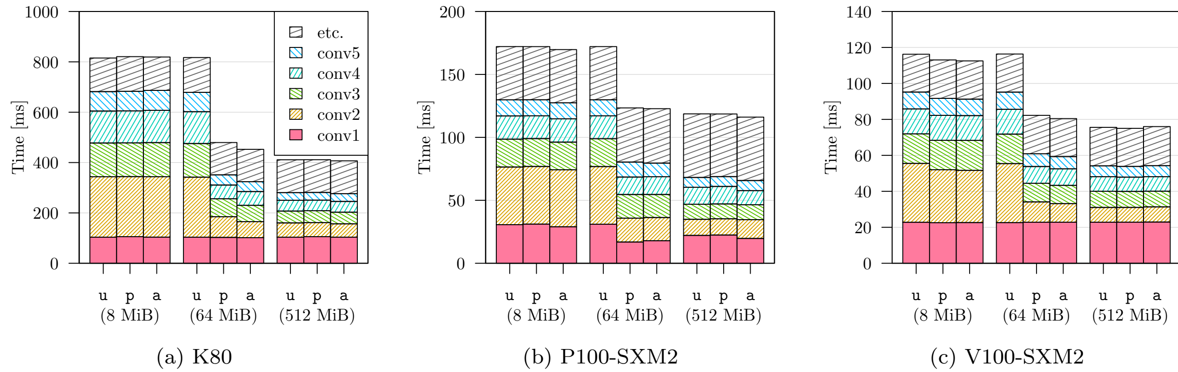
Figure 3.13: **Benchmark results of forward convolution of AlexNet's "conv2" layer on P100-SXM2**. We use 64 MiB workspace size and a mini-batch size of 256. Numbers on each rectangle represent micro-batch sizes.

On the other hand, FFT-based convolution is more efficient, although it requires an excessive amount of workspace (213 MiB) to store the images and filters in the frequency domain. μ-cuDNN with the `powerOfTwo` option successfully enables the use of the FFT-based algorithm within the same workspace constraint, using 48.9 MiB over micro-batches of size 32. Besides, the `all` option enables μ-cuDNN to use Winograd's convolution algorithm, an algorithm that is especially efficient for small convolution kernels, achieving 2.33x speedup over `undivided` in total. Note that in our work, we assume that CNNs are robust against numerical errors introduced by changing a convolution algorithm, as supported by previous work [71].

Figure 3.14 shows the timing breakdowns of Caffe on AlexNet on three different GPUs. We only highlight convolutional layers since the performance of other layer types (e.g., pooling) are not changed by our library.

One important observation from Figure 3.14 is that the performance improvement of μ-cuDNN over cuDNN (which is equivalent to `undivided`) is significant when the moderate amount of workspace is set by users. For instance, if the workspace size per kernel is 64 MiB, μ-cuDNN with the `all` option achieves 1.81x speedup with respect to the entire iteration and 2.10x with respect to convolutions alone than `undivided` on K80. This is because μ-cuDNN successfully enables cuDNN to use faster algorithms, as in Figure 3.13. Also, a similar speedup is achieved on P100-SXM2 (1.40x for the entire iteration, and 1.63x for convolutions alone) and on V100-SXM2 (1.45x for the entire iteration, and 1.60x for convolutions alone).

On the other hand, when the workspace size is limited to 8 MiB, μ-cuDNN cannot attain any performance improvement because the workspace is too small to utilize even if WR's per-layer workspace allocation scheme is applied. Indeed, on P100-SXM2, only one kernel of `all` option increases the workspace's utilization over `undivided`. On the contrary, when the workspace size limit is too large (512 MiB) on K80 and P100-SXM2 GPUs, the performance difference between cuDNN and μ-cuDNN is negligible. This is because there is no benefit to divide the mini-batch, as the best algorithms for the layers fit into the workspace constraint. However, this workspace limit consumes a considerable amount of workspace memory. In contrast, the `undivided` option consumes 2.87 GiB in



(a) K80 (b) P100-SXM2 (c) V100-SXM2

Figure 3.14: **Benchmark results of AlexNet on three different GPUs**. We use workspace sizes of 8, 64, and 512 MiB. The labels "`u`", "`p`" and "`a`" represent `undivided`, `powerOfTwo`, and `all`, respectively. We use a mini-batch size of 256.

total, `all` with 64 MiB limit only consumes 0.70 GiB, although with 4% overhead caused by choice of micro-batch algorithms.

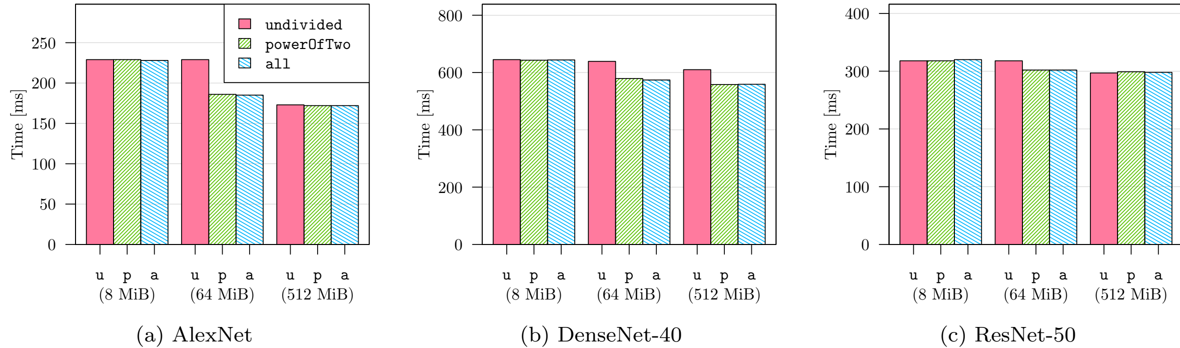
From the viewpoint of the time to optimization, including kernel benchmarking and solving DP, `powerOfTwo` considerably outperforms `all`. In particular, with 64 MiB workspace on P100-SXM2, `all` takes 34.16 s, and `powerOfTwo` takes 3.82 s. This result and Figure 3.14 imply that `powerOfTwo` is a reasonable choice to test the computation efficiency of new CNNs quickly. Note that we can reuse the same benchmarking results for different hyperparameters to save time since the hyperparameters do not affect the computational performance of the convolution operations; indeed, μ-cuDNN has a function to reuse already recorded kernel time as we explained in Section 3.2.4. Generally, the overhead of μ-cuDNN is negligible with respect to the entire training time since the benchmarking pass runs only once, whereas the forward and backward passes are repeated hundreds of thousands of times.

TensorFlow

Figure 3.15 shows the μ-cuDNN performance of the second version of AlexNet [88], ResNet-50, and DenseNet-40 on P100-SXM2. We summarize the speedups in Table 3.10. We use a mini-batch size of 256 for AlexNet and DenseNet, and 64 for ResNet-50.

We set the (input dimensions, output dimensions) to $(3 \times 224 \times 224, 1000)$ for AlexNet and ResNet-50, or $(3 \times 32 \times 32, 10)$ for DenseNet-40, which are used for training the ILSVRC2012 dataset [3] or the CIFAR-10 dataset [2] , respectively. We also set k of DenseNet-40, the number of feature maps of each convolutional layer, to 40 to obtain better computational efficiency.

Since TensorFlow 1.4.1 does not provide any workspace limits to μ-cuDNN via cuDNN's benchmarking functions before actual convolutions, we manually provide a workspace limit of 64 MiB to μ-cuDNN. μ-cuDNN achieves 1.24x speedup for AlexNet, 1.05x for ResNet-50, and 1.11x for DenseNet. Comparing μ-cuDNN's speedups on Caffe and TensorFlow for AlexNet on P100-SXM2



(a) AlexNet (b) DenseNet-40 (c) ResNet-50

Figure 3.15: **TensorFlow benchmark results on P100-SXM2 GPU**. We use a mini-batch size of 256 for AlexNet and DenseNet-40, and 64 for ResNet-50.

Table 3.10: **TensorFlow benchmark results on P100-SXM2 GPU**.

Policy	Time [ms] (Speedup)		
	AlexNet	ResNet-50	DenseNet
`undivided`	229.0	318.0	639.0
`powerOfTwo`	186.0 (1.23x)	302.0 (1.05x)	579.0 (1.10x)
`all`	185.0 (1.24x)	302.0 (1.05x)	574.0 (1.11x)

(Figure 3.14b and Figure 3.15a respectively), we can see that μ-cuDNN reduces almost the same time (about 50ms) with a 64 MiB workspace. These results prove that μ-cuDNN has good performance portability between different deep learning frameworks only if the framework depends on cuDNN. Note that we do not expect the same speedups between Caffe (Figure 3.14) and TensorFlow (Table 3.10); this is because TensorFlow uses different parameters that μ-cuDNN cannot control (such as padding widths), and a considerable part of the time is spent on non-convolutional computation, which is implemented differently in each framework.

3.3.3 Case study: Heterogeneous cluster optimization

As a part of the Python interface, we provide a function to minimize training time by assigning different micro-batch sizes to heterogeneous GPUs (Figure 3.16). The motivation behind this function is that researchers tend to have access to clusters of heterogeneous accelerators, and highly utilizing such clusters may speed up training considerably. In Figure 3.16, the function in line 8 provides an uneven batch size for each GPU so that the time to perform forward and backward passes of synchronous SGD becomes uniform among GPUs, increasing load balancing among GPUs. Since gradient synchronization is typically overlapped with computation, especially in large batch training [28, 29, 31], we omit the extra communication time from the objective, formulating the problem as an Integer Linear Programming (ILP) problem as follows:

Figure 3.16: **Sample code for heterogeneous cluster optimization**.

```
import ucudnn
import framework as f

mb = 256
gpus = ["K80", "K80", "K20Xm"]
f.CNN(gpus=gpus, minibatch=[mb, mb, ...]).run_once()
f.CNN(gpus=gpus,
      minibatch=ucudnn.best_batch_size(mb, gpus)).run()
```

$$
\begin{aligned}
\text{min} \quad & \max_{g \in \mathcal{G}} \left\{ \sum_{n \in \mathcal{B}} t_{g,n} x_{g,n} \right\} \\
\text{subject to} \quad & \sum_{n \in \mathcal{B}} x_{g,n} \leq 1 \ (\forall g \in \mathcal{G}) \\
& \sum_{g \in \mathcal{G}} \sum_{n \in \mathcal{B}} n x_{g,n} = N \\
& x_{g,n} \in \{0, 1\} \ (\forall g \in \mathcal{G}, \forall n \in \mathcal{B}),
\end{aligned}
$$

where $\mathcal{G}$ is a set of GPUs, $\mathcal{B}$ is a set of available batch sizes for each GPUs, N is the mini-batch size, and $t_{g,n}$ is time to perform forward and backward passes on GPU g with a batch size of n. μ-cuDNN uses a micro-batch size of n on GPU g if and only if $x_{g,n} = 1$. It is reasonable to restrict n to `powerOfTwo` to reduce the number of configurations. If $\sum_{n \in \mathcal{B}} x_{g,n} = 0$ for a given GPU g, the ILP failed to find a fast configuration with g and it will not participate in training. We use the GLPK ILP solver [112] to solve the problem.

We demonstrate the μ-cuDNN Python interface by combining three different GPUs from the TSUBAME-KFC/DL supercomputer: Tesla K20Xm, Tesla K80, and GTX 750Ti (Table 3.11). The K20Xm and K80 are Kepler generation GPUs, whereas the 750Ti is a Maxwell generation GPU, not intended for high-performance computing.

We first run Caffe's "time" command on each node to collect performance metrics to our database. Since we employ a file-based database, it is easily collected on a Networked File System (NFS). Then, we use μ-cuDNN's optimization function in Python, which is explained in Section 3.2.4.

Figure 3.17 shows the estimated time of forward-backward passes of ResNet-18 on heterogeneous GPUs. By combining two GK210 chips of a K80 GPU and a K20Xm GPU, forward-backward passes become 2.12x faster than that of a single GK210. When a K20Xm and a K80 are combined, μ-cuDNN assigns uneven batch sizes, 8 and $\{12, 12\}$ respectively. In addition, if a user sets even batch sizes for GPUs, 750Ti and K20Xm for example, the 750Ti will become a bottleneck (168.43 ms vs 83.03 ms), and thus, it will incur a slowdown of 1.37x than μ-cuDNN. Furthermore, a combination of all the GPUs yields 2.20x speedup against the baseline. Note that the time to perform MPI all-reduce over MVAPICH2 2.3a with a message size of 1 MiB on 3 nodes takes 2.63 ms, which can be easily hidden by

Table 3.11: **GPU specification for heterogeneous cluster optimization**.

	FP32 TFlop/s	Memory size [GiB]
Tesla K20Xm	3.95	6
Tesla K80 (GK210 × 2)	8.73	24
GTX 750Ti	1.31	2

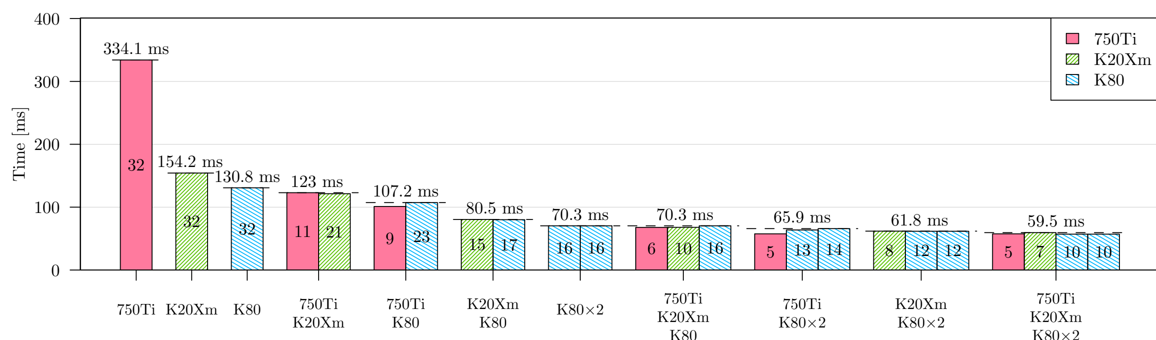


Figure 3.17: **Estimated time of forward-backward passes of ResNet-18 on heterogeneous GPUs**. Numbers on each bar represent batch sizes. The objective is to minimize a maximum of GPUs' time, shown as dashed lines.

the computation. Therefore, this example illustrates the potential speedups by heterogeneous GPUs for the training of a single CNN.

Chapter 4

Training 3D CNNs with hybrid-parallelization

As we described in Section 2.2, many studies have proposed distributed algorithms for computing deep neural networks. These algorithms are roughly classified into data-parallel training, model-parallel training, and the combination of the two other algorithms, hybrid-parallel training. Although these algorithms do not change the computation's semantics, several work using performance modeling [102, 103, 104, 113] have demonstrated that each algorithm's computational performance heavily depends on the target network architecture. For instance, it is proved that convolutional neural networks for image classification tasks can be scaled to thousands of GPUs [27, 28, 29] only with data-parallelism, due to their high computation-to-communication ratio. Although this scaling requires increasing the mini-batch size, which affects the generalization performance of the networks, prior studies have demonstrated that the degradation on the performance can be mitigated by using advanced techniques such as new learning rate scheduling [31]. Because of this, the maximum amount of parallelism for such networks has reached the theoretical upper bound, where their generalization performance does not improve further by increasing parallelism (i.e., the mini-batch size).

However, this technique cannot necessarily be applied to more complex networks for several performance reasons. As we explain in Section 4.1, performing end-to-end training for complex scientific data is recently getting a lot of attention [5, 59, 114, 115, 116, 117]. This idea has great potential to find in-depth knowledge of scientific data without prior domain knowledge only with a few amounts of computing resources and time. However, such DNNs can often require more computational capability than previous conventional networks to process raw scientific data in an end-to-end fashion. For instance, as we explain in Section 4.2.5, well-known two-dimensional CNNs such as ResNet can be trained on a data-parallel framework, whereas the CosmoFlow network [59] is

too huge in terms of memory usage to perform training with such conventional GPU frameworks. In this case, the network cannot enjoy data-parallelism because even a single replica of the network cannot reside even on a single GPU.

In this chapter, we choose 3D CNNs as examples of the above surrogate model types, to demonstrate how we can accelerate training by introducing hybrid-parallelism. Specifically, we focus on **spatial partitioning** due to its good affinity with high-dimensional CNNs. As we explained in Section 2.2.3, spatial partitioning is believed to be preferable for 3D CNNs due to its nice surface-to-volume ratio, and thus we expect that the training is accelerated without significant overhead. We use the CosmoFlow network and the 3D U-Net [18] as the target networks, and propose an extension to the LBANN deep learning framework [44] to fulfill high-performance hybrid-parallel training on a GPU supercomputer.

This chapter is organized as follows; in Section 4.1, we introduce the two 3D networks as our target applications. Both of the prior work has a common performance issue that the networks have to be scaled down due to the memory pressure issue than the original spatial resolution. Although these studies improved the scalability or the computational efficiency for specific models, it is still unclear that how we can accelerate high-resolution 3D CNNs by exploiting hybrid-parallelism on GPU supercomputers and the extent how it improves the generalization performance of such networks by using high-resolution 3D data. In Section 4.2, we explain our effort to accomplish hybrid-parallel training on the LBANN framework. In Section 4.3, we demonstrate the computational efficiency of our framework. We also show training results using high-resolution 512^3 CosmoFlow data cubes that are infeasible to use without hybrid-parallel deep learning frameworks.

4.1 Motivation

Recent developments in deep learning techniques have led to substituting conventional scientific applications and simulations with DNNs, such as climate prediction [115, 116], biomedical image analysis [6, 18], and electron microscopic analysis [5, 114, 117], and so on. In such an application, a DNN is trained with the original problem's input data along with the corresponding ground-truth, which is computed by the original application. This training results in a DNN that performs the computation that is equivalent to the original applications without explicitly knowing the underlying domain-specific knowledge. This technique is also useful when it is unfeasible to compute the exact solution to a given problem, but only inputs and corresponding ground-truth outputs are known; in this scenario, the model to predict outputs are called **surrogate models**. From the computational point of view, this technique is also capable of reducing the computational requirements, such as the end-to-end latency, the memory footprint, and possibly the programming cost, to get desired solutions. Further discussion about the future work is mentioned in Chapter 6.2.

In this section, we introduce two emerging 3D CNNs, the CosmoFlow network and the 3D U-Net, which are challenging to train on available high-resolution 3D datasets. These networks estimate

Table 4.1: **Two different target 3D CNNs**. We use the LiTS image segmentation dataset for the 3D U-Net.

Network	**CosmoFlow**	**3D U-Net**
# of conv.+deconv. layers	7	18
Objective	Regression	Segmentation
Dataset	CosmoFlow	LiTS
Input	4×512^3	1×256^3
Output	4	3×256^3
Min. # of V100 GPUs per sample	8	16

several cosmological parameters concerned with input 3D data or perform segmentation on given 3D data, respectively. The networks have the domain-agnostic architecture for regression or segmentation, and indeed, 3D U-Net has been applied to a wide range of medical image analysis problems [19, 20, 21]. However, it is also reported that the problem size, especially the 3D input size, is limited to a smaller size than that of datasets currently widely available [18, 59]. Table 4.1 summarizes the specifications of the two networks.

4.1.1 The CosmoFlow network

CosmoFlow [59] is one of such applications that empowers deep learning to estimate cosmological parameters from the mass distribution of the universe. As explained by Ravanbakhsh et al. [118], estimating such parameters play a crucial role in the cosmology domain. In the prior work, a seven-layer CNN (the CosmoFlow network) is trained with a set of 3D data cubes (Figure 4.1) and few types of corresponding ground-truth cosmological parameters (the CosmoFlow dataset). The name of the project is derived from the underlying deep learning framework, TensorFlow [33]. Although the network was originally proposed by Ravanbakhsh et al., we call the network "the CosmoFlow network" throughout this thesis because the network is known by that name.

The network has a common CNN architecture; it has seven convolutional layers with optional batch-normalization layers [119] and Leaky ReLU activation [62], two or three average pooling layers at the initial stage of the network, and three fully-connected layers with dropouts [120] at the end. The output size of the network is the same to the number of parameters to predict, which is two or three in the prior work. Mean Absolute Error (MAE) or Mean Squared Error (MSE) is used as the loss function, and the Adam optimizer [79] is used to train the network.

Each of the input 3D data cubes is a 3D histogram of dark matter distribution; the value of each voxel (a discrete cell composing a 3D domain) means the number of particles in the sub-domain. The ground-truth includes cosmological parameters to describe the matter distribution of the universe, such as the matter density Ω_m, the matter power spectrum σ_8, and the spectral index N_s.

Since dozens of data samples are required to train the CNN, the authors adopted dark matter N-body simulation to generate the synthetic dataset. They first ran the simulation with a fixed number

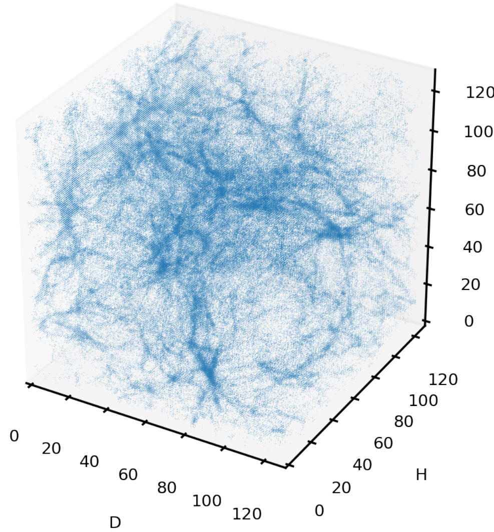


Figure 4.1: **Visualization of one channel of a 128^3 CosmoFlow data cube**. We first select voxels whose values are more than a certain threshold, and then plot a point to each voxel in the logarithmic scale. The ground-truth parameters of this sample is $(\tilde{\Omega}_m, \tilde{\sigma}_8, \tilde{N}_s, \tilde{H}_0) = (-0.242, 0.145, -0.489, 0.465)$, where each parameter is normalized to $[-1, 1]$.

of particles and with different random cosmological parameters in feasible uniform distribution (such as $0.25 < \Omega_m < 0.35$), to generate multiple possible mass distributions of the universe. Then, each set of produced particles is histogrammed into a fixed size of 3D voxels, which is suitable to feed to deep neural networks. However, in the prior work, the 3D histograms are split further into multiple sub-cubes in advance of the training to make the memory footprint and the computational requirement of the CNN practically small. When a data cube is split into multiple sub-cubes, each of the sub-cubes inherits the ground-truth of the original cube. The original work also uses random flipping ($2^3 = 8$ ways) and rotating ($3! = 6$ ways) for data augmentation, which produces 48 different data samples from one sample in total. We summarize the specifications of the datasets used in the prior work and our work in Table 4.2.

In this work, we further increase the complexity of the input data by introducing the **redshift** parameter (Figure 4.2). Redshift defines how the light that comes from distant galaxies is stretched, and thus it affects how the universe is observed. While the prior work used redshift $z = 0$, the latest CosmoFlow dataset [121] provides four different snapshots for each universe by using different redshift parameters. These simultaneous snapshots can naturally be fed into a CNN by increasing the first convolutional layers' input channels, just like providing RGB images instead of grayscale images to 2D CNNs. At the same time, the dataset also provides one additional parameter, **the Hubble constant** H_0 to predict. Therefore, the input and the output size of our work is $4 \times 512 \times 512 \times 512$ and 4 respectively, which is much more complex than those of the original work ($1 \times 64 \times 64 \times 64$

Table 4.2: **Comparison of CosmoFlow work**. "KNL 7250" and "V100" are Intel Xeon Phi 7250 (Knights Landing) on the Cori supercomputer and NVIDIA Tesla V100-SXM2 on the Lassen supercomputer, respectively. †Each of 500 256^3 3D histograms is split into 64 64^3 sub-cubes, and augmented into $64 \times 6 \times 8 = 3072$ cubes by rotating and mirroring. ‡Each of 12632 256^3 3D histograms is split into 8 128^3 sub-cubes. "N/A" is the information not shown in the papers.

	Ravanbakhsh et al. [118]	Mathuriya et al. [59]	This work
# of particles	512^3	512^3	512^3
# of simulations	500	12,632	10,017
Input size	64^3	128^3	4×512^3
# of samples	1,536,000 †	101,056 ‡	10,017
Total dataset size	N/A	1.4 TiB	9.77 TiB
# of params. to predict	2	3	4
Batch-norm. layers	✓		✓
# of machines (benchmark)	N/A	8192 × KNL 7250	2048 × V100
# of machines (training)			512 × V100

and 2 respectively). We explain in Section 4.2.5 that we also modify the network architecture to get better computational performance. We report that our modification does not degrade the quality of convergence, but even improve the original network.

4.1.2 The 3D U-Net

Image segmentation is one type of machine learning tasks in which each pixel of one or multiple channels is assigned to the ground-truth label. This type of task can be used in a wide range of real-world applications, including biomedical image analysis [6, 18, 19, 20, 21].

Ronneberger et al. [6] proposed U-Net, a 2D CNN that performs image segmentation. The network mainly consists of two parts; in the first down-sampling path, a grayscale input image is down-sampled from 572×572 to 28×28, 1024 channels by using convolutional layers and max-pooling layers. And in the following up-sampling path, the resolution of the down-sampled image is restored using up-convolutional layers (also known as deconvolutional layers) and several convolution blocks with skip connections from the former path. Finally, the network outputs a segmentation map of the central part of the image (388×388, 2 labels). They demonstrated that the U-Net overcomes other competitive models on multiple biomedical image segmentation datasets, including a sliding-window based CNN [114].

Çiçek et al. [18] extends the dimension of each layer of the U-Net to 3D as the 3D U-Net for volumetric segmentation. The 3D U-Net follows the architecture of the U-Net, but has a smaller number of blocks and channels; for instance, while U-Net has four convolutional blocks (each of which has two sequential convolutional layers) in the down-sample and the up-sample paths, the 3D U-Net has only three blocks, and each of the first convolutional layers has 2x less number of channels. It is also worth mentioning that the width of each input dimension was shrunk by a factor of four ($572^3 \rightarrow 132 \times 132 \times 116$), while the authors used 2x down-sampled volumes than the original sample

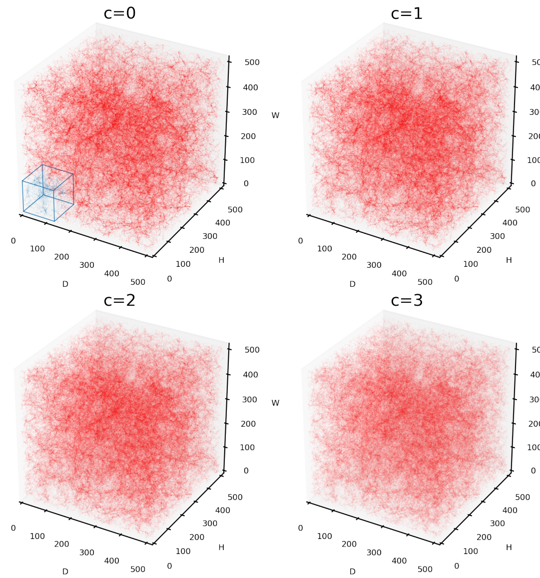


Figure 4.2: **Visualization of a** 512^3**, 4 "redshift" channels CosmoFlow data sample**. The region of Figure 4.1 is shown in blue. We use a different threshold and a scaling factor for each channel for better visibility.

Table 4.3: **Comparison of U-Net work**. "Titan" and "TitanX" are NVIDIA Titan and TitanX GPUs, respectively.

	Ronneberger et al. [6]	Çiçek et al. [18]	This work
Input size	1×572^2	$3 \times 132 \times 132 \times 116$	256^3
Output size	2×388^2	$3 \times 44 \times 44 \times 28$	3×256^3
Dataset	ISBI 2012 [5]	Microscopic 3D volumes	LiTS [22]
# of machines	Titan	TitanX	$512 \times$ V100

size. Although it is not explicitly mentioned in the paper, this implies that the model with the full-resolution data cannot fit into the single NVIDIA TitanX GPU they used, as we illustrate in Section 4.2.5. Table 4.3 summarizes the original U-Net and 3D U-Net work and our work.

The LiTS dataset

The Liver Tumor Segmentation benchmark dataset [22] is a medical volumetric dataset. This dataset aims to classify each voxel of 131 Computed Tomography (CT) volumes of the human body into one of the liver, tumors, or other. The dataset contains diverse CT images, each of which consists of a variable number of 512×512 CT image slices, from 42 to 1026. The voxels' scale also varies from 0.56 mm to 1 mm in axial and 0.45 mm to 6 mm in the z-direction, because different CT scanners and measurement protocols are used. It also includes varying tumor contrast levels, from 38 mm^3 to 349 mm^3. Label annotation was performed by human experts. Figure 4.3 visualizes one LiTS data sample. Although the original U-Net and 3D U-Net has smaller output dimensions than their inputs

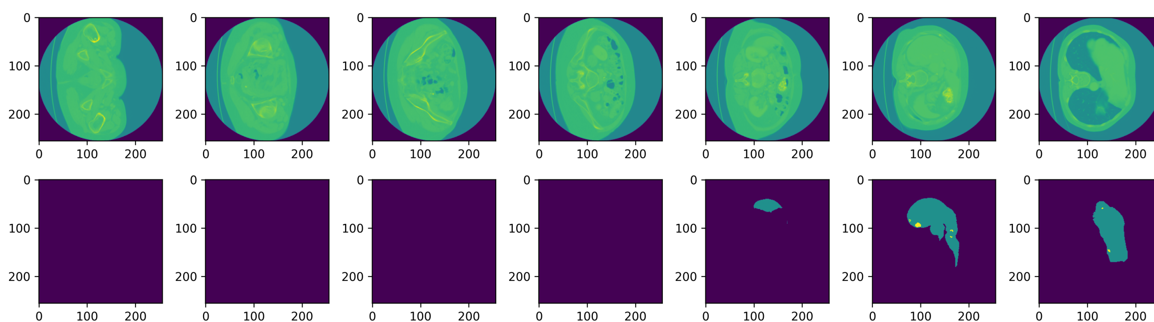


Figure 4.3: **Visualization of a 256^3 LiTS data sample**. The top row and the bottom row show input and ground-truth data, respectively. We show seven different 2D cross-sections of different positions. Black, green and yellow pixels are empty (label 0), liver (label 1), and tumor (label 2) voxels, respectively.

to infer the central parts of images and volumes, we add paddings to our model to use the same input and output size because the LiTS dataset provides per-voxel ground-truth labels for all of the voxels.

4.2 Hybrid-Parallel implementation of 3D CNNs

In this section, we introduce our modifications to the LBANN framework [44] to perform hybrid-parallel (spatial partitioning) training of 3D CNNs on a GPU supercomputer. Initially, the framework did not support spatial partitioning, and the Distconv extension for the framework has extended the framework to support spatial partitioning on only 2D CNNs. Hence, we applied the following additional changes to the framework:

- We extend Distconv to support 3D CNNs. This requires to extend layer kernels, tensor shufflers, and halo exchanges to 3D.
- We identify inefficient computation and communication kernels that are specific to high-dimensional networks and thus haven't appeared in the prior work [104, 122]. We implement our own optimized kernels or tune the architecture of the target networks to obtain reasonable computational performance without losing networks' prediction accuracy.
- We demonstrate the conventional I/O pipeline cannot hide the read latency of 3D datasets due to their huge per-sample data size. And thus, we introduce model-parallelism not only the computation but I/O to distribute the workload among all of the processes. In this method, each process loads its own responsible parts of the 1D-partitioned data samples (**hyperslabs**) from the Parallel File System (PFS). We additionally introduce distributed data-space to cache once loaded hyperslabs in the host memory using the Conduit library [123].
- We propose an empirical performance model to predict the best parallelization strategy for given network architecture, mini-batch size, and supercomputer specifications. The model digests the

benchmark performance of the target supercomputer and then predict the mini-batch time of a given network. By using this model, we demonstrate that the performance of hybrid-parallel training is fairly predictable. Thus, it is possible to provide the optimal parallelization strategy for framework users and researchers without the knowledge of HPC applications.

4.2.1 Extending Distconv for 3D CNNs

The LBANN framework works on one or more CPU or GPU nodes. When it runs on a GPU cluster, it performs data-parallel training by using MPI where each process is assigned to one GPU. Each process

1. reads its own local batch by using indexes that do not overlap with those of other processes,
2. compute parameter gradients with respect to the local batch without communicating with other processes (except for batch-normalization layers where all-reduce is required to compute the average and the variance of each channel),
3. issues asynchronous all-reduce to synchronize gradients once the backward computation of each layer is done, and
4. updates its local copy of network parameters by using the aggregated gradients.

The prior work, Distconv [104], extends LBANN by the following changes (Figure 4.5);

- it adds a new **parallel-strategy** property to each layer, which defines the number of processes for each spatial dimension and the sample dimension to parallelize the layer. Users can set arbitrary parallel-strategy to each layer (Figure 4.4).
- It inserts appropriately tensor shuffling communications before and after the computation of each layer to apply user-specified parallel-strategy. It supports multiple communication backends, including CUDA-aware MPI [124] and Aluminum [125].
- It replaces the implementation of spatial layer types (such as convolutional layers and pooling layers) to perform the equivalent computation on multiple GPUs by using halo exchanges (Figure 4.6). It first computes on the central part of the layer that does not need halo regions of other processes and perform halo exchanges on a different CUDA stream at the same time. Once the communication is done, it enqueues the layer's computation kernel around the halo to complete the computation of the layer. This parallelization is done in a similar manner used in parallel stencil applications on GPU clusters [101].

In this work, we extend Distconv to support 3D networks. Since Distconv initially assumes only 4D tensors (the sample, channel, and two spatial dimensions), we make a significant effort to update the foundation of the library to support 5D tensors along with the Hydrogen linear algebra backend [126] and LBANN itself as well. Our extension supports an arbitrary number of dimensions to be

Figure 4.4: **The definition of CosmoFlow's first convolutional layer on LBANN**. We use 128 nodes (512 GPUs) and apply 8-way partitioning on the depth dimension ("depth_groups").

```
1  layer {
2    name: "cosmoflow_module1_conv1_conv_instance1"
3    parents: "layer2"
4    children: "cosmoflow_module1_conv1_bn_instance1"
5    data_layout: "data_parallel"
6    weights: "weights1"
7    convolution {
8      num_dims: 3
9      num_output_channels: 16
10     num_groups: 1
11     conv_dims_i: 3
12     conv_pads_i: 1
13     conv_strides_i: 1
14     conv_dilations_i: 1
15   }
16   parallel_strategy {
17     sample_groups: 64
18     height_groups: 1
19     width_groups: 1
20     channel_groups: 1
21     filter_groups: 1
22     depth_groups: 8
23   }
24 }
```

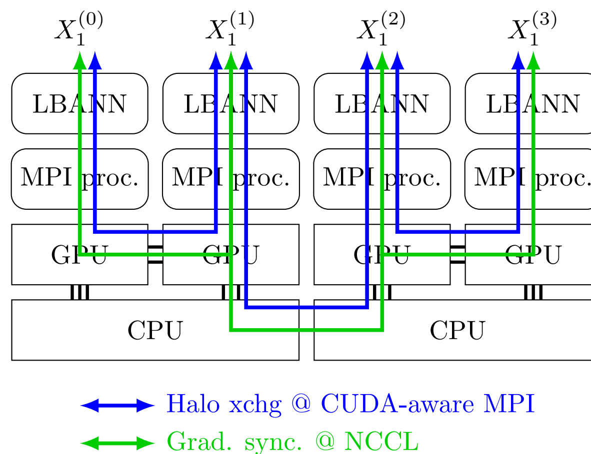


Figure 4.5: **The software stack of LBANN with hybrid-parallelism**. $X_i^{(j)}$ is the j-th partition of i-th data sample.

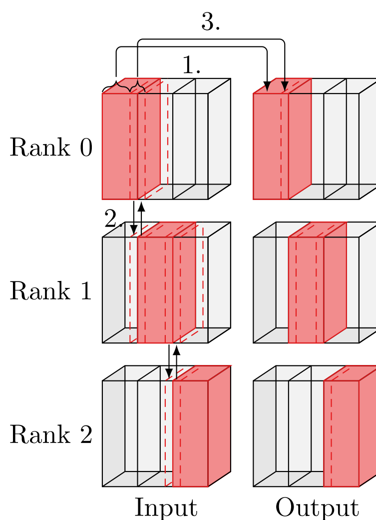


Figure 4.6: **Overview of Distconv**. Distconv 1) perform convolution on the central part of each process's assignment, 2) perform halo exchanges asynchronously, and 3) compute the boundary parts once halo exchanges are done.

partitioned simultaneously, which we call 1D-, 2D-, and 3D-partitioning. However, since we found that 1D-partitioning is enough to hide the halo exchange latency and achieve good scalability on up to hundreds of GPUs, we mainly use 1D-partitioning throughout this study. We refer to "D-way partitioning" as the strategy where the depth dimension of each sample is partitioned among D GPUs.

Figure 4.7 shows an overview of our framework. The framework repeats the following steps to perform training;

1. partial domains that are needed for the following computation (hyperslabs) are read from the Parallel File System (PFS) by using MPI I/O via the HDF5 library [127]. The hyperslabs are cached in the host memory by using the Conduit data exchange library [123] for the second epoch and beyond.

2. Each GPU compute the forward pass with appropriate halo exchanges that are issued by Distconv. When the pass reaches the first fully-connected layer (if available) or the first layer whose partitioned spatial domain is smaller than the number of processes of each group, all activation data is gathered to the root GPUs (rank 0 and 4 in Figure 4.7), and they proceed with the computation until the end of the pass.

3. The framework then performs a backward pass just as the reversed order of the forward pass. All-reduce on parameter gradients is asynchronously performed during the backward pass once gradients of each layer are computed.

4. Each GPU updates its local copy of parameters using synchronized gradients.

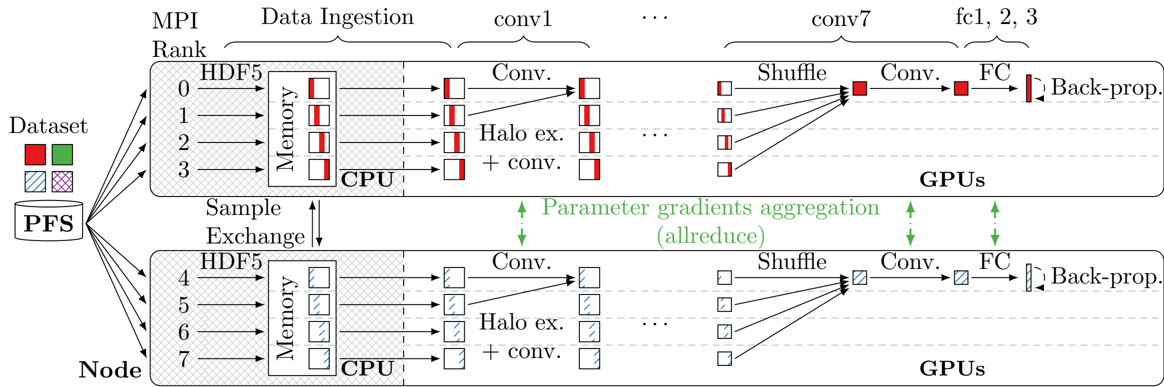


Figure 4.7: **Overview of hybrid-parallel training on LBANN and Distconv**. Each node contains four processes that partition a single data sample. Processes 0 to 3 processes the red sample, and 4 to 7 the blue shaded sample, respectively.

4.2.2 Kernel optimization

At the time of extending Distconv to 3D networks, most of the Distconv and LBANN codes, including GPU kernels, were assuming 4D tensors. We found that, however, simply extending them for the target 3D CNNs leaves several performance issues that have never been seen in 2D networks; we explain that a similar problem exists even in a highly-optimized deep learning kernel library, cuDNN in Chapter 3. This, we applied the following additional optimization to achieve reasonably good computational performance:

- Exchanging halo domains require packing before the exchange and unpacking after the exchange to convert scattered data between a fully-packed message to communicate. For efficient halo exchanges, we develop custom CUDA kernels that are dedicated to specific filter sizes (such as 3^3 and 5^3).

- The cross-entropy layer of the 3D U-Net is 3D, in which each voxel has multiple channels to output the probability of each class. However, typical frameworks, including LBANN, assume that cross-entropy layers are used for classification tasks, which only requires a spatial dimension of one. Thus, we apply spatial partitioning to the layer type as well so that the computation on each voxel is computed among multiple GPUs in parallel. We also extend its GPU kernel to use integer ground-truth voxels directly instead of one-hot voxels to eliminate a conversion kernel between them.

- We also tune CUDA kernels of different types of layers (such as batch-normalization layers) for 3D layers, where the spatial dimension size is much larger than 2D layers. We illustrate that not doing this kind of tuning even for negligible kernels in terms of time results in significant performance loss in Section 4.2.5.

4.2.3 Spatially-partitioned I/O

I/O of deep learning applications typically refer to the online data load of input samples to every training processor, such as GPUs. Its workload has peculiar characteristics than general HPC applications;

- the data is composed of uniform data files, each of which contains one data sample or more. For instance, many frameworks process the ILSVRC image dataset [3] as a list of files as the dataset follows this assumption.
- Each data sample is read exactly once in a fixed number of steps (an epoch). Then epochs are repeated until it reaches the predefined number of steps or terminated manually.
- The subset and the order of samples each processor reads are usually randomized (data shuffling). Thus, each data file might be read by each processor throughout the training.
- A dataset for deep learning might be beyond the capability of each node's disk space. For instance, the CosmoFlow dataset is about 10 TB, which is larger than the capacity of a local SSD of the Lassen supercomputer (1.6 TB).

Therefore, prior studies have proposed techniques to accelerate this heavy random read, such as staging data to local storage [116], using an efficient parallel file system [128], dedicated I/O servers (burst buffers) [59]. Note that, in this thesis, we refer to the cost to load data samples from any type of storage or memory to GPU memory as I/O, even though actual I/O is not performed.

In this section, we introduce our spatially-partitioned I/O technique. As already mentioned, many deep learning frameworks assume data samples are managed by files, which naturally enables random sample read. This approach is suitable for data-parallel training, where each process reads more than one data sample, so the I/O workload per step is naturally distributed by all of the processes. In hybrid-parallel training, however, this doesn't suit because each process does not necessarily read the entire data sample. Although the prior work have not discussed this problem in detail [104, 122], we identified this is the primary performance bottleneck of training of extremely large 3D CNNs. We then propose splitting the I/O workload of each sample into multiple processes that process the sample in upcoming forward and backward passes. We demonstrate that this technique is required to achieve good strong-scaling of our target 3D CNNs with hybrid-parallelism.

In the prior Distconv work, the I/O pipeline has not been extended to read each data sample in parallel. As we explained in Section 4.2.1, Distconv performs tensor shuffling when the parallel strategy is changed during forward and backward passes, root processes (each of which is responsible for computing its local batch when spatial partitioning is not applied) first read local batches, and then shuffling is performed to distribute them to non-root processes (Figure 4.8). Specifically, when the local batch size is one, each root process reads only one sample and then scatter it to other processes that share the local sample in the following forward and backward passes.

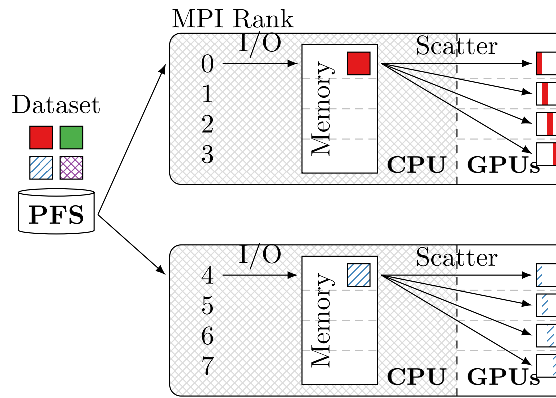


Figure 4.8: **Sample-parallel I/O**. Process 0 and 4 read the red and the blue sample, respectively, and then scatter each sample to other processes.

However, although this approach can use an existing implementation that does not intend to be combined with hybrid-parallelism, this incurs extra overhead that is not necessarily required as follows;

- most part of the data root processes read is not used for the computation of the first layer because we assume that the first layer is convolution. Even though processes share the bandwidth between the PFS, non-root processes had to be idle while the actual I/O.
- For the same reason, the scatter operation is not essential if there is a way that all of the processes read their responsible parts of data.

One trick we introduce to mitigate the I/O overhead is to cache already loaded data samples to the host memory so that they will be reused for the second epoch and beyond. This function ("data-store") is included as a part of LBANN by combining MPI with the Conduit data exchange library that provides efficient ways to exchange scientific data. For instance, we use a mini-batch size of 64 for the CosmoFlow network in actual training runs (Section 4.3.4), and our target system has 240 GB/s of PFS bandwidth, which requires 256 ms of mini-batch loading time at each step. Therefore, these types of caching are inevitable to accomplish efficient hybrid-parallel training.

However, we found in a preliminary evaluation that even this technique cannot hide the I/O latency for huge 3D data cubes. Figure 4.9 shows the iteration time of the CosmoFlow network with a mini-batch size (N) of 1, 2, or 4. We apply 1D partitioning for all of the configurations; for example, when 16 GPUs are used for $N = 2$, each sample is distributed among 8 GPUs (8-way partitioning). Note that each data series is asynchronous, and LBANN performs I/O asynchronously to the GPU computations, so the I/O time is more than seen in the figure. Figure 4.9 clearly shows that we cannot achieve strong scaling at all just by increasing the number of GPUs. This is because while the computation complexity scales as the number of GPUs is increased, the I/O workload per root process remains constant, so it quickly becomes the bottleneck on a large number of GPUs. Therefore, this result implies that I/O should scale as well as hybrid-parallelism is utilized.

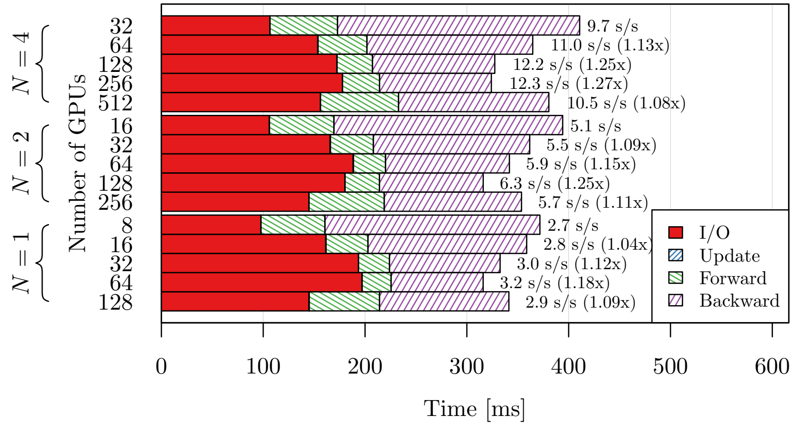


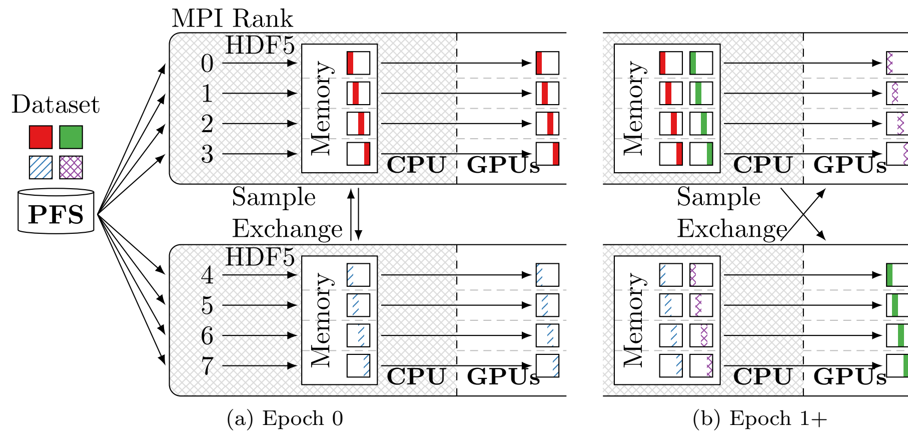
Figure 4.9: **Strong scaling of the CosmoFlow network with 512^3 input cubes without spatial-parallel I/O**. We only use distributed caching with Conduit. We only use $N = 1, 2, 4$ as it does not strong-scale clearly.

To mitigate this overhead, we propose a parallelized I/O pipeline where all of the processes are involved in reading their own responsible fragments of a mini-batch (Figure 4.10a). In Figure 4.10a, 4-way 1D partitioning is applied on two groups of 4 processes, and each process reads a quarter part of each data sample (which we call hyperslabs) from the PFS. We adopt the HDF5 file format [127] to store 3D data cubes in the PFS, whose interface library utilizes MPI I/O to read data files in parallel efficiently. We further combine this approach with the data-store function to cache hyperslabs on the host memory; for example, in Figure 4.10a, process 0 and 4 share only the left-most quarters of the data samples, so each process does not hold hyperslabs that are not used by itself, unlike the sample-based approach. After the first epoch, all samples are read and cached on the host memory, so any processes do not perform actual I/O between the PFS after the first epoch (Figure 4.10b). Note that we don't preload the datasets from the PFS to local storage because the data transfer cost is paid only on the first epoch.

Another trick specific to the CosmoFlow dataset we found is that it offers nice characteristics to reduce its data size. We found that most of the values of the dataset are one or less (Table 4.4), and all of the values are under $2^{15} - 1 = 32767$. Therefore, we convert the dataset in int16 rather than float64, which reduces the disk space by 4x than the previous work [59]. It is also possible to apply data compression to each data sample to reduce the size, but we leave it as future work as we prove that using int16 is enough to hide the I/O latency, as we show in Section 4.3. Similarly, the LiTS dataset is stored in int16 for the same reason.

4.2.4 Performance modeling

As explained in our previous work [113] and other prior studies [102, 103, 104], it is not trivial to predict the computational performance of deep learning frameworks due to their complexity. The



(a) Epoch 0

(b) Epoch 1+

Figure 4.10: **Spatially-partitioned data movement on our framework**. During epoch 0, HDF5 ingests hyperslabs in parallel into the data store. During epoch 1 and more (“1+”), the data store distributes the hyperslabs for each sample in the mini-batch that is about to be trained on.

Table 4.4: **Statistics of a CosmoFlow data sample**. We use the same data sample to Figure 4.1. Note that all of the elements are non-negative integers.

Minimum	0
2nd quartile	0
Median	0
Mean	0.925
3rd quartile	1
99% quantile	11
Maximum	916

number and types of kernel calls are determined by the target network architecture, each of which does not necessarily achieve the theoretical peak performance [113]. The performance also depends on several training configurations, such as the mini-batch size, which affects not only the computational performance but the quality of convergence. Also, benchmarking such frameworks on real large-scale environments requires an excessive amount of node-hour, or even impossible because reserving the entire supercomputer can take a very long wait time; we actually use up to 512 nodes of 792 available nodes of the Lassen supercomputer in our experiments. Therefore, applying performance modeling techniques reduces much effort to find the best configurations for a given DNN and available hardware only with limited hardware resources.

In this section, we propose an empirical performance model for our hybrid-parallel training framework. The motivation here is to use the model to investigate whether our actual performance achieves the expected peak empirical performance of the underlying supercomputer and discuss whether the model can predict the best machine configurations for the same mini-batch size. The model takes network architecture, mini-batch size, and the parallelization strategy for the network and predicts the time to perform one training step. We choose an empirical way to construct the model because our aim is to find the best achievable performance with achievable highly-optimized computation and communication libraries (such as cuDNN and NCCL).

We first collect the runtime of cuDNN convolution, deconvolution, pooling, and batch-normalization on a single GPU. We run a cuDNN benchmark suite with every possible layer parameters of the layer types. We also measure the performance of NCCL and the underlying MPI library with a limited number of nodes to construct GPU-to-GPU and global all-reduce communication models. We then combine the benchmark results with the communication models to predict iteration runtime on multiple GPUs.

First, we define the time to perform forward-computation of convolutional or pooling layer l as FP_l:

$$FP_l = \max\left\{Comp_l\left(D_l^{main}\right), \sum_{d=0}^{2} 2SR\left(D_{l,d}^{halo}\right)\right\} + Comp_l\left(D_l^{halo}\right), \tag{4.1}$$

where $Comp_l(D)$ is time to compute layer l on a given domain D, and $SR(D)$ is time to perform peer-to-peer send-receive communication between two GPUs via NVLink or InfiniBand depending on the location of the two processes. The shape of D_l^{main}, the domain which can be computed without halo communication, and $D_{l,d}^{halo}$, the domain which requires halo region to be computed, are defined by the partitioning of the layer. We also define BD_l and BF_l, the time to perform the backward-data and the backward-filter passes on layer l in a similar manner.

To estimate $Comp_l(D)$, we benchmark each layer type on a single GPU. For convolutional layers, we use the largest cuDNN workspace possible, and enable cuDNN's auto-tuning function to find the fastest convolution algorithms. We use the median of three trials after warmup throughout this study.

The time for a batch-normalization layer, which requires global all-reduce communication of the local sum and squared-sum of each channel instead of halo communication, is the sum of the computational time and time to perform the communication.

To estimate $SR(D)$, we use Aluminum's ping-pong benchmark. We use the median time of 100 trials with float vectors of $1, 2, 4, \ldots, 2^{30}$ elements for each configuration. We then apply linear regression to estimate the time for arbitrary message sizes. However, we found that the linear model cannot follow the actual performance with both small and large message sizes, and it even outputs negative time for too small message sizes (Figure 4.11a). Another conservative way is to apply linear interpolation to the measured runtime, but this might be sensitive to the measurement error (Figure 4.11b). Therefore, we use the linear model but uses one percentile of the measured runtime as its intercept (Figure 4.11c). This heuristic provides reasonable predictions, especially for small message sizes.

We ignore the cost to compute non-3D parts of the 3D CNNs, such as fully-connected layers and loss layers since their costs are negligible compared to other costs. We also ignore the cost of I/O for loading data samples from the PFS or between processes, as our optimized pipeline mitigates I/O costs drastically for the two networks we use in this work.

Finally, the total time of the network is

$$Cost = \sum_l FP_l + \max\left\{\sum_l BD_l + BF_l, \sum_l AR(\theta_l)\right\}, \tag{4.2}$$

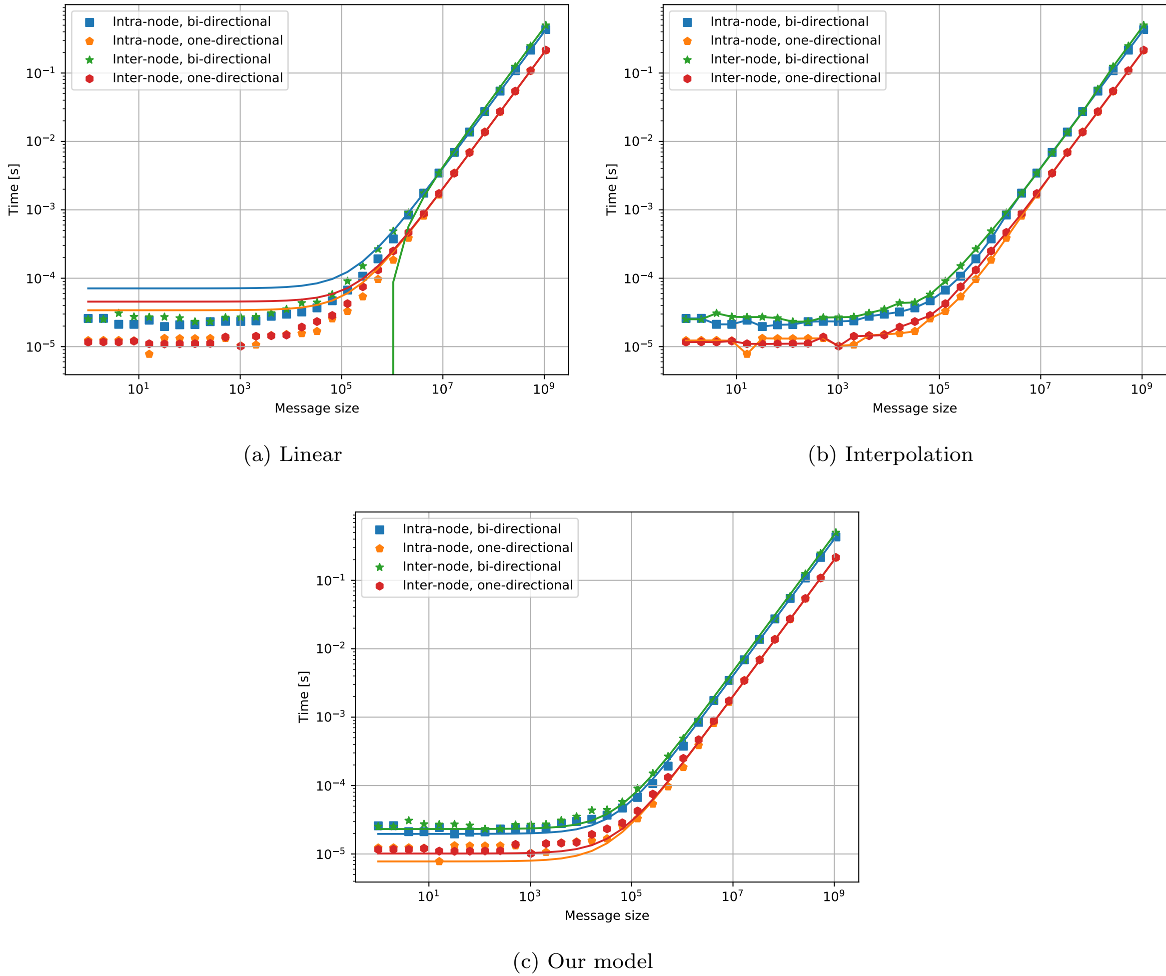
where AR is time to perform all-reduce among all of the GPUs and θ_l is the number of parameters of layer l. To measure the runtime of AR, we measure the performance on one node (4 GPUs) to 128 nodes (512 GPUs), with float vectors of 1 to 2^{24} elements. We measure 40 trials for message size and each configuration. We assume a simple ring-based algorithm [98],

$$AR(n, p) = \alpha p + \beta n + \gamma \frac{n}{p} + \delta, \tag{4.3}$$

where n is the message size, p is the number of processes, and $\alpha, \beta, \gamma, \delta$ are the model parameters. However, we found a similar prediction error to the send-recv model (Figure 4.12a). Furthermore, we cannot use the linear interpolation technique for this model because the model must accept the number of nodes that is larger than the number we use for benchmarking. Therefore, we apply the logarithmic transformation to the model to solve this problem (Figure 4.12c).

4.2.5 Architecture tuning

In this section, we conduct a manual neural architecture search to tune the network for our software and evaluation environments. For example, when we increase the input size from 128^3 to 512^3, we must carefully update the network's design not to make any unnecessary computational performance bottlenecks. Also, Distconv assumes that all the convolutional layers have odd size filters and use the



(a) Linear

(b) Interpolation

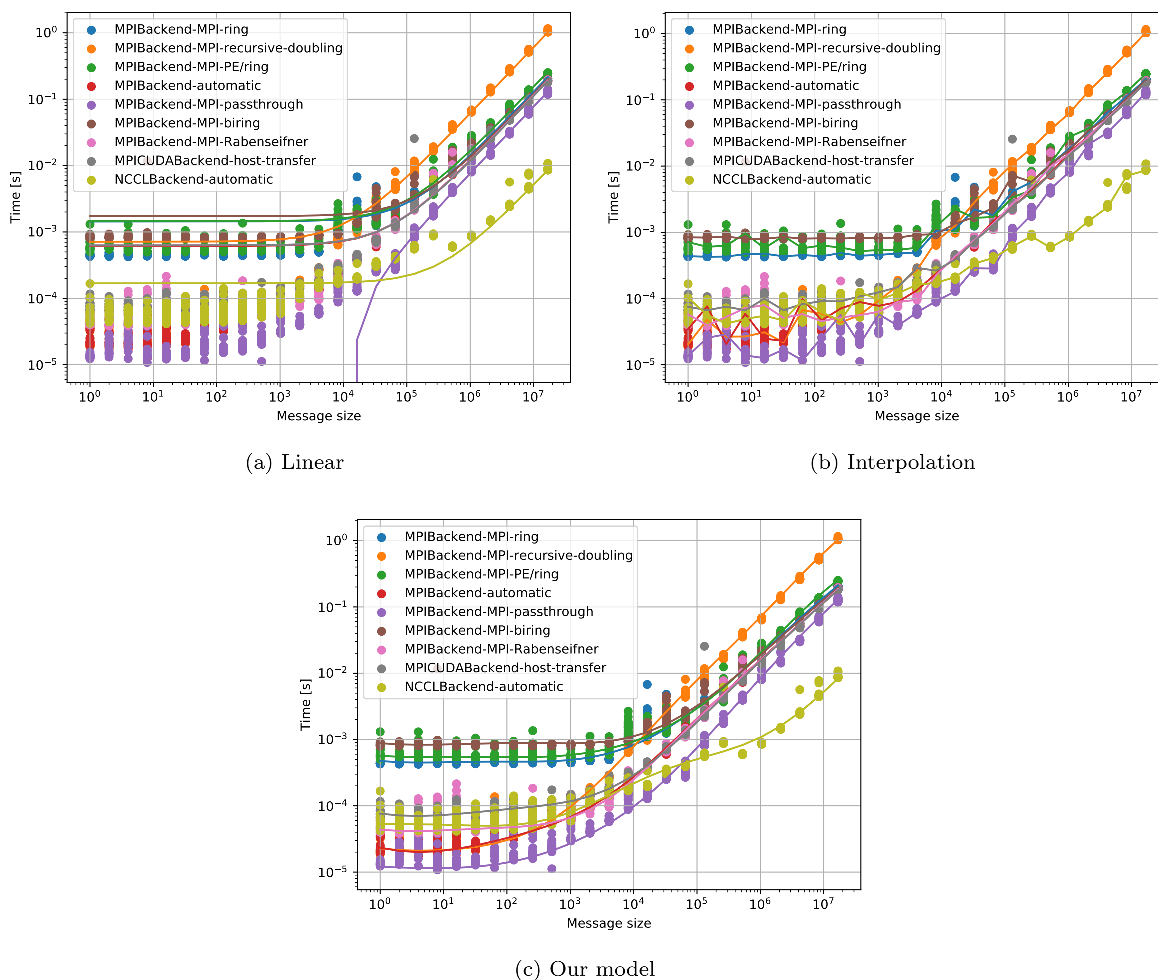
(c) Our model

Figure 4.11: **Inter-GPU communication performance on Lassen**. The points and the lines show measured and predicted time, respectively. We show the median time for each configuration and message size.

"same" padding so that the halo communication pattern is symmetric among any number of processes, while the CosmoFlow network uses $2 \times 2 \times 2$ filters. Although our primary goal is to accelerate the computation of 3D CNNs by introducing hybrid-parallelism, we must ensure that our changes do not make their convergence worse.

Table 4.5 summarizes preliminary training results of our CosmoFlow network variants using the "2parB" dataset [121]. This is a smaller, simpler variant of the main dataset we report final results in Section 4.3.4, which we use for hyperparameter tuning and architecture search. The dataset consists of 1k samples, every 512^3 voxels with a single channel. We split each sample into 128^3 samples in advance. We test the following network variants:

- **CO**: All of the 2^3 and 4^3 convolution filters are replaced by 3^3 and 5^3 filters, so all filter sizes



(a) Linear

(b) Interpolation

(c) Our model

Figure 4.12: **All-reduce collective performance among GPUs on Lassen**. The points and the lines show measured and predicted time, respectively. We show all of the trials for each message size and each configuration.

are odd.

- **C3**: All of the convolution filter sizes are replaced by 3^3.
- **C3P3**: All pooling filter sizes are also replaced by 3^3.
- **C3P3P2**: Two additional pooling layers are added to C3P3, and biases are removed from all of the convolutional layers.

We apply padding to all of the convolution and pooling layers whose filter size is odd.

The motivation of C3P3P2 is as follows:

- When each input dimension is doubled without changing the network architecture, the number of neurons of the last convolutional layers is multiplied by $2^3 = 8$. This results in increasing the

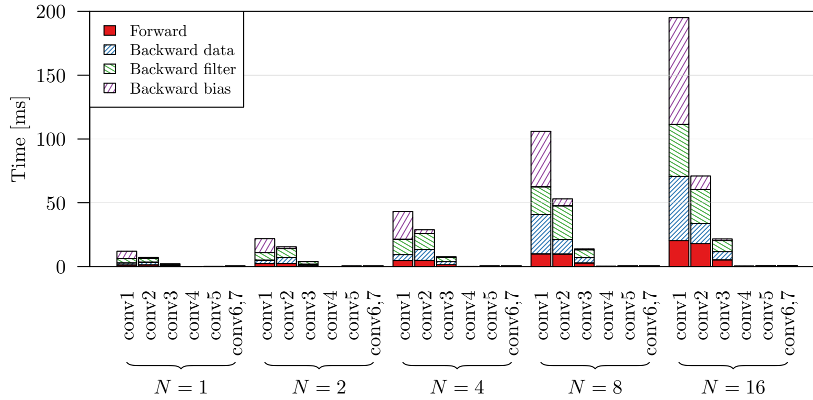


Figure 4.13: **Layer-wise convolution performance of the CosmoFlow network**.

number of parameters of the first fully-connected layer by eight times, which enlarges the communication cost of the gradient synchronization (i.e., the total number of learnable parameters) with almost the same proportion. This expansion is a possible bottleneck of strong scaling because the communication is constant to the number of processes unless fully-connected layers are also partitioned. Thus, we introduce additional pooling layers to make the dimensions of the last convolutional layer constant regardless of the input dimension size.

- As a result of benchmarking cuDNN, we found that its backward kernels for convolutional layers' biases are inappropriately slower than other kernels, especially on the first layer (Figure 4.13). This is unnatural because the kernel computes the sum of output errors for each channel, whose complexity should be order-of-magnitude smaller than other convolution kernels. Although cuDNN's source code is not publicly available, we found the number of runtime threads obtained by the nvprof profiler is determined by the number of channels. Hence, we assume that the kernel parallelizes each channel's computation, which algorithm does not work well the first convolutional layer whose number of channels is much smaller than the spatial dimensions. Therefore, we examine how biases affect the convergence quality by removing from the model.

In Table 4.5, our best network, C3P3P2, achieves better accuracy than the baseline model, and hence we use this network throughout this chapter. In addition, the test loss of the original network is degraded by increasing the mini-batch size, even if the learning rate is increased in the same proportion to the mini-batch size to keep the stochasticity of parameter gradients [87], as shown in the previous work [59]. This result implies that there is a hidden scaling limit for this network in the data-parallel scheme around hundreds of GPUs. Our hybrid-parallel scheme can accelerate training by using hundreds of GPUs while keeping the same mini-batch size (Section 4.3.4).

Figure 4.14 and Table 4.6 shows the revised CosmoFlow network architecture. In this thesis, we

Table 4.5: **Comparison of CosmoFlow network variants**. $\eta^{(0)}$ is the initial learning rate. The C3P3P2 network achieves the best test loss.

Network	# of nodes	N	$\eta^{(0)}$	Test loss
Baseline	8	512	1×10^{-4}	0.0051
Baseline	32	2048	2×10^{-4}	0.0077
Baseline	128	8192	5×10^{-4}	0.0184
CO	64	2048	1.5×10^{-4}	0.0073
C3	64	2048	3×10^{-4}	0.0077
C3P3	32	1024	1×10^{-4}	0.0057
C3P3P2	32	1024	2×10^{-4}	**0.0041**

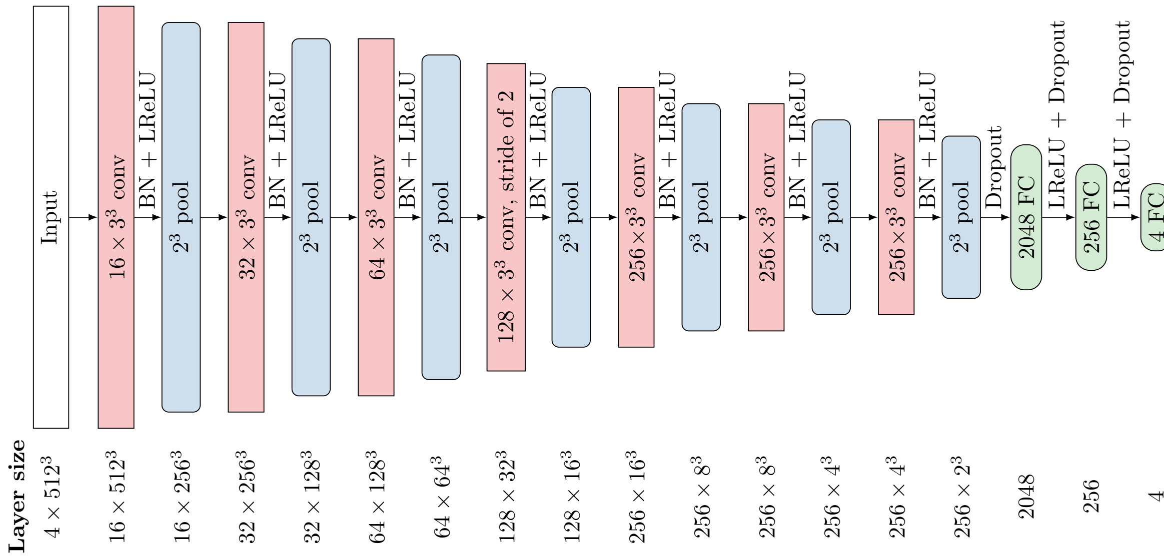


Figure 4.14: **Our revised CosmoFlow network architecture**.

use the latest 2019_05_4perE CosmoFlow dataset, each sample of which is a 4-channel, 512^3 voxels along with four different cosmological parameters.

We also propose smaller variants of the network, which takes 128^3 or 256^3 sub-volumes of the data samples. We follow the prior work by splitting each original cube into 128^3 or 256^3 sub-volumes for the smaller networks. The only modification to the networks is that we optionally add pooling layers after the 6th and 7th convolutional layers so that the output size of the last (7th) convolutional layer is always $256^3 \times 2^3$. By applying this approach, the number of parameters of all the networks is constant, which prevents itself from being the performance bottleneck. On the other hand, both the number of floating-point operations on the convolutional layers and the memory requirement per data sample is nearly proportional to the number of input elements (see Section 4.3.3). This implies that 3D CNNs offer great scalability once model-parallelism is utilized, where the complexity of the computation is $O(W_i^3)$ where that of the communication is $O(\log W_i)$ for all-reduce and $O(W_i^2)$ for halo exchange, where W_i is the input dimension size. In fact, our smallest CosmoFlow network has 9.44 M parameters and requires 18.5 GFlops for forward computation, while ResNet-152 [65], a 2D CNN, has O(10M)

Table 4.6: **Our revised CosmoFlow network architecture**. W_i is the spatial input width. cN→pN is convolution followed by pooling, and fc$N$ are fully connected layers. Convolution is stride one and pooling stride two unless noted. All of the layers use "same" padding. The last three rows show per sample requirements.

Layer(s)		**Output width**		
Name(s)	**Filter**	$W_i = 128$	$W_i = 256$	$W_i = 512$
c1→p1	16×3^3	$128^3 \to 64^3$	$256^3 \to 128^3$	$512^3 \to 256^3$
c2→p2	32×3^3	$64^3 \to 32^3$	$128^3 \to 64^3$	$256^3 \to 128^3$
c3→p3	64×3^3	$32^3 \to 16^3$	$64^3 \to 32^3$	$128^3 \to 64^3$
c4→p4	128×3^3 (stride of 2)	$8^3 \to 4^3$	$16^3 \to 8^3$	$32^3 \to 16^3$
c5→p5	256×3^3	$4^3 \to 2^3$	$8^3 \to 4^3$	$16^3 \to 8^3$
c6→p6	256×3^3	$2^3 \to$N/A	$4^3 \to 2^3$	$8^3 \to 4^3$
c7→p7	256×3^3	$2^3 \to$N/A	$2^3 \to$N/A	$4^3 \to 2^3$
fc1	2048	2048	2048	2048
fc2	256	256	256	256
fc3	4	4	4	4
# parameters $[10^6]$		9.44	9.44	9.44
# conv. ops. [GFlops]		55.55	443.8	3550
(Forward) [GFlops]		18.52	147.9	1183
Memory [GiB]		0.824	6.59	52.7

parameters and 1.13 GFlops for computation.

As shown in the table, the per-sample memory requirements for the 128^3 and 256^3 data are under the memory size of the latest GPUs: A NVIDIA Tesla V100 GPU with 16 GB memory is capable of holding one or more samples from these datasets. Thus, data-parallel training is sufficient to train these networks. For 512^3, however, the memory requirement exceeds the memory size. Hence it is not feasible to perform data-parallel training. On the other hand, our approach resolves this problem by introducing the hybrid-parallel scheme in LBANN where we spatially partition each sample among four or more GPUs.

We then add batch-normalization after each convolutional layer to evaluate its effects on both computational performance and inference accuracy. Although it was removed from the network in the prior work [59] due to a performance issue, we found that our framework can keep GPUs busy even though additional per-layer collective communication is introduced. We also demonstrate that the additional batch-normalization layers improve the inference accuracy further, and it results in order-of-magnitude improvement than the original network without batch-normalization layers. Note that this change requires approximately twice the GPU memory size to hold activation and error signals before and after each batch-normalization layer, so we use 8 GPUs at least per sample on 512^3 cubes.

Table 4.7 shows the number of multiply-and-add operations required to compute each layer of the networks. As shown in the table, the total number of operations is nearly dominated by the first three layers. While Distconv does not partition layers that partitioned spatial dimension is smaller than the number processes but gather to the root processes of each process group, it is expected that the

Table 4.7: **The number of operations of CosmoFlow's convolutional layers with a 128^3 cube**.

Name	Forward [GFlops]	Backward data [GFlops]	Backward filter [GFlops]
conv1	1.81	1.81	1.81
conv2	7.25	7.25	7.25
conv3	3.62	3.62	3.62
conv4	0.03	0.03	0.03
conv5	0.91	0.91	0.91
conv6,7	1.81	1.81	1.81

convolution operations are distributed efficiently as the first half of the layers are parallelized on up to 128 GPUs.

4.3 Performance evaluation

In this section, we evaluate the performance of our framework using the CosmoFlow network and the 3D U-Net from various aspects. Our evaluation is three-fold:

- Scalability: We evaluate strong scaling (data sample throughput with a fixed mini-batch size on different numbers of GPUs) and weak scaling (throughput with increasing both the mini-batch size and the number of GPUs in the same proportion) to measure the computational performance of our framework. Specifically, we provide a detailed performance breakdown to show our framework is reaching the peak practical computational performance that can be achieved with the current software stack. In these experiments, we only focus on the performance from the HPC aspect, so we ignore the effect of changing the mini-batch size to the quality of convergence. Furthermore, we compare the actual performance with a prediction by the performance model to discuss how performance modeling cooperates in designing training configurations with hybrid-parallelism.

- Comparison with another framework: We compare the performance of our framework with 1) TensorFlow with Horovod that supports data-parallel training on multiple GPU nodes, and 2) TensorFlow with Mesh-TensorFlow that supports spatial partitioning on a single GPU node. We demonstrate that our framework achieves better due to its well-optimized halo exchanges and also work on multiple nodes. To the best of our knowledge, LBANN is the only framework that supports spatial partitioning on multiple GPU nodes as its built-in function.

- Training with 512^3 CosmoFlow data cubes: Our ultimate goal is to improve the convergence of 3D CNNs by increasing their spatial resolution. Hence, we verify it by training the CosmoFlow network with 4-channel 512^3 data cubes, which is $4 \times (512/128)^3 = 256$ times larger input size than that of the prior work [59]. We demonstrate that improving input resolution brings order-

of-magnitude accuracy improvement. To the best of our knowledge, this work is the first attempt to train the network with that input size successfully.

4.3.1 Evaluation environment

We use Lassen, a GPU supercomputer at LLNL composed of 792 nodes. Each node has two IBM POWER9 CPUs with 256 GB memory and four NVIDIA V100 GPUs with 16 GB memory (Figure 4.15). Every two GPUs are connected to one CPU with NVLink, and the two GPUs are also directly connected via the NVLink. Each computing node equips dual-rail EDR InfiniBand NIC. Note that each node is also equipped with a 1.6 TB NVMe SSD, but we don't use them because both the CosmoFlow dataset and the LiTS dataset can be stored into the CPU memory of the minimum number of nodes required to train.

We use GCC 7.3.1, CUDA 10.1, cuDNN 7.6.4, NCCL 2.4.2 and IBM Spectrum MPI 10.2.0.11rtm2. We use cuDNN's auto-tuning function to select the best convolution algorithms with available GPU memory size. We use single-precision for computation throughout the experiments. Throughout this chapter, we do not use half-precision or mixed-precision training (regardless of the use of Tensor Cores), because the impact of applying low-precision training to CosmoFlow has not yet been evaluated.

The LBANN framework relies on two communication libraries, Aluminum and Distconv. LBANN perform all-reduce across devices by using Aluminum, which internally utilizes NCCL and MPI to perform. We use the NCCL backend of Aluminum to perform gradient synchronization. Distconv performs halo exchange by using CUDA-aware MPI (Figure 4.5).

4.3.2 Strong scaling

We refer to the scalability when varying the number of processes in the spatial dimensions with a fixed mini-batch size as strong scaling. The strong scalability of a hybrid-parallel framework is important in terms of how much the framework can be parallelized without changing the semantics of the training.

Figure 4.16 shows the strong scaling performance of the CosmoFlow network with the 512^3 dataset. We use global mini-batch sizes (N) of 1, 2, 4, 16, and 64, and split the network in the depth dimension. We run the framework for four epochs with a 128-sample subset of the dataset (if the mini-batch size is smaller than 128) or the full dataset and show the median iteration time except for the first epoch. Note that the computational workload per epoch is constant regardless of the progress of training, so we run the framework for few epochs to exclude variance in time. We also show predicted times by our performance model, which largely match with the actually measured times, confirming our implementation performed as expected. Since our data reader caches hyperslabs, which are loaded in the past training iterations, we separately show the performance on the first epoch ("Epoch 0") and beyond ("Epoch 1+").

As shown in Figure 4.16b, when the mini-batch size, N, is 16 and 64, we achieve speedups of 1.98x

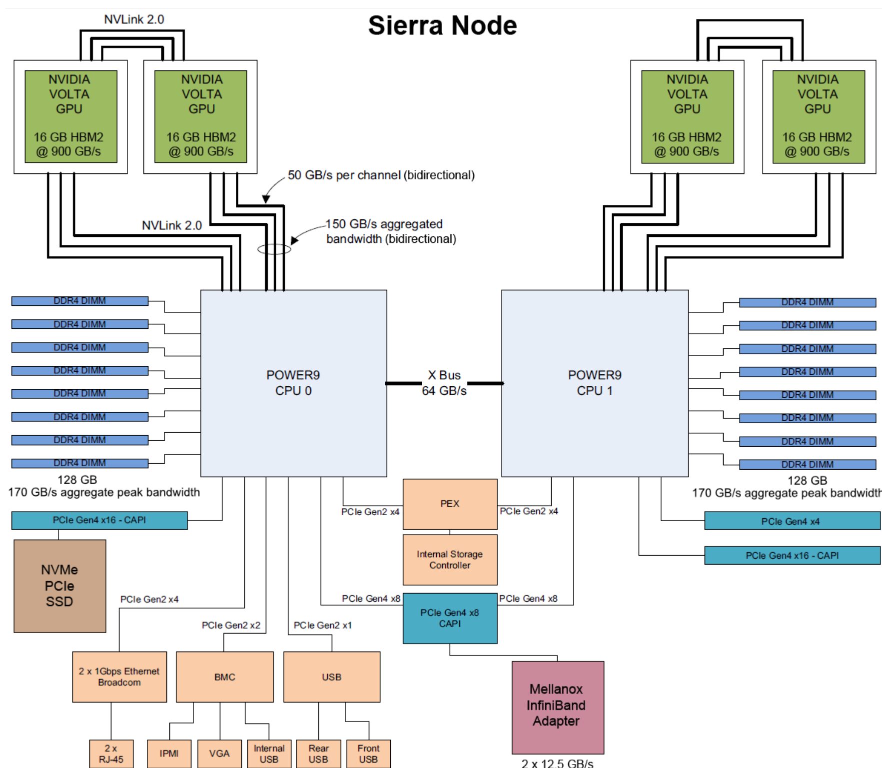
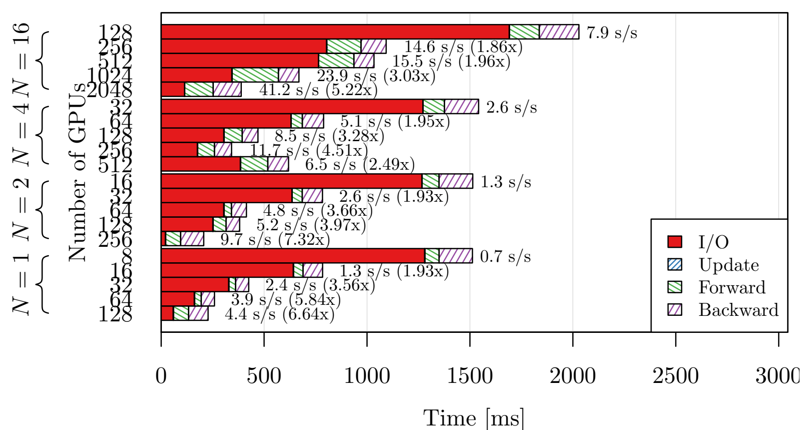


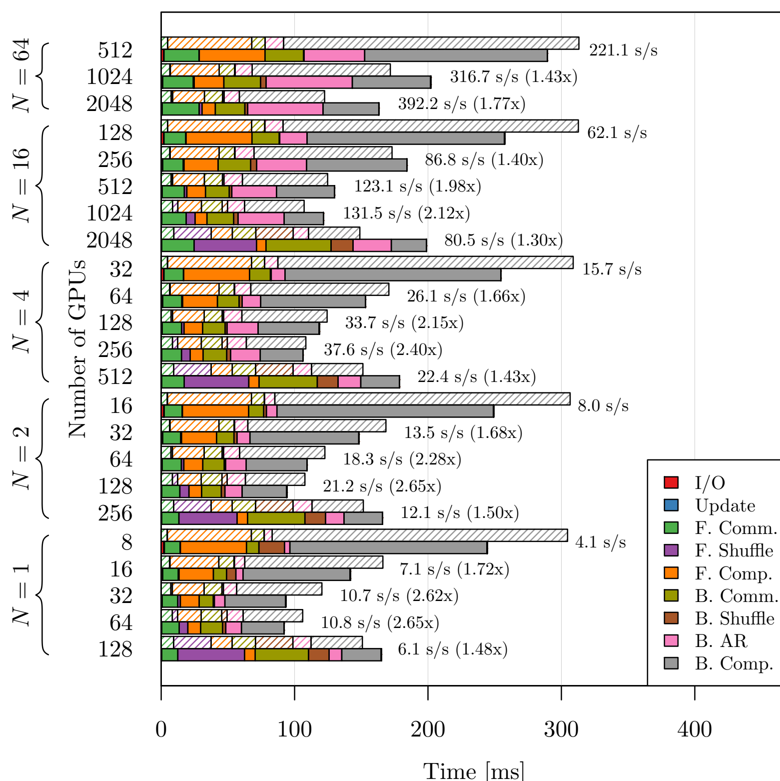
Figure 4.15: **Sierra/Lassen node diagram**. This figure is cited from the Sierra documentations [129].

Table 4.8: **Legend specifications of the strong scaling plots**. We show the time when both computation and asynchronous kernels are performed simultaneously as "Comp.".

Label	Meaning	Async. to "Comp."
I/O	Data load	✓
Update	Updating local copies of weights	
{F,B}. Comm.	Halo exchanges	✓
{F,B}. Shuffle	Gather/scatter to change the parallel strategy (see Section 4.2.1)	
{F,B}. Comp.	GPU computation	–
B. AR	All-reduce collectives for gradient synchronization	✓



(a) Epoch 0



(b) Epoch 1+

Figure 4.16: **Strong scaling of the CosmoFlow network with 512^3 input cubes**. Shaded bars of Figure 4.16b show the time predicted by the performance model. "F." and "B." are forward and backward passes, respectively. The full specifications are shown in Table 4.8. N is the mini-batch size. Bars are annotated with throughput (samples/s) and speedup relative to the minimum setting with the same N. A part of the time of each task is hidden when it is overlapped with other tasks.

with 512 GPUs (128 nodes) compared to 128 GPUs (32 nodes), and 1.77x with 2048 GPUs (512 nodes) compared to 512 GPUs (128 nodes), respectively. We note that when the mini-batch size is 16, the performance gain for going to 2048 GPUs falls off because the problem (i.e., the computation required for each data sample) becomes over-decomposed. However, the computational throughput can still be scaled further by increasing the batch size to 64. As we show in Section 4.3.4, this mini-batch size is a reasonable choice for actual training, and thus we prove that we successfully scale the training of the CosmoFlow network to thousands of GPUs. Furthermore, the I/O time is almost invisible in the figure since it is almost completely overlapped with computations in our optimized I/O pipeline. This makes a significant contrast to conventional I/O methods in terms of strong scaling performance, as their I/O parallelism is limited by the mini-batch size. In fact, as we already showed in Figure 4.9, the iteration time does not scale without our spatially-parallel I/O approach due to I/O overhead. This demonstrates the necessity of strong-scaling I/O along with computation to efficiently parallelize training.

Comparing Figure 4.16a and Figure 4.16b, we can see that the data-store caching reduces the I/O overhead considerably (note that the scale of the x-axes is different). Even though the first epoch is approximately ten times slower than epochs after that, 1) one training trial repeats hundreds of epochs to converge, and 2) it is inevitable overhead to read the dataset from file system due to its enormous dataset size, so we consider this is acceptable overhead.

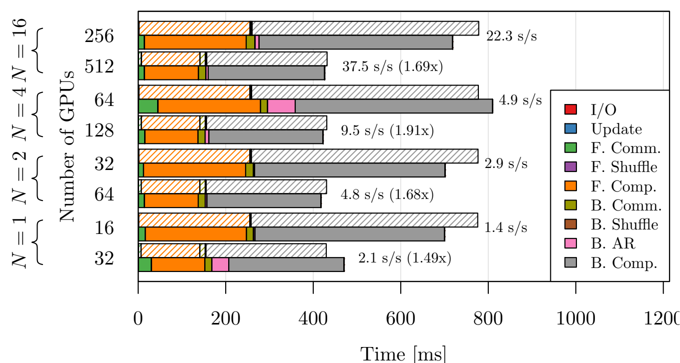
Further, our performance model successfully predicts the fastest model distribution for all of the mini-batch sizes we evaluated; significantly, it predicts the performance fall-off by over-decomposition we mentioned above. This result implies that performance modeling can choose suitable configurations for hybrid-parallel frameworks without actual runs on hundreds of nodes.

Figure 4.17 shows the strong scaling performance of 3D U-Net with 256^3 data cubes on a similar evaluation methodology. With this network, we have to use at least 16 GPUs per sample due to memory requirements. We achieve good strong scaling performance between 16-way and 32-way partitioning, such as 1.42x on 512 GPUs over 256 GPUs with a mini-batch size of 16. As shown in the figure, similarly to the CosmoFlow network, most of the iteration time is spent in computation. In fact, we achieve almost the same performance on the first epoch (Figure 4.17c) and beyond (Figure 4.17d). This implies that we achieve near-peak performance, despite the communication overheads of hybrid-parallelism compared to data-parallelism.

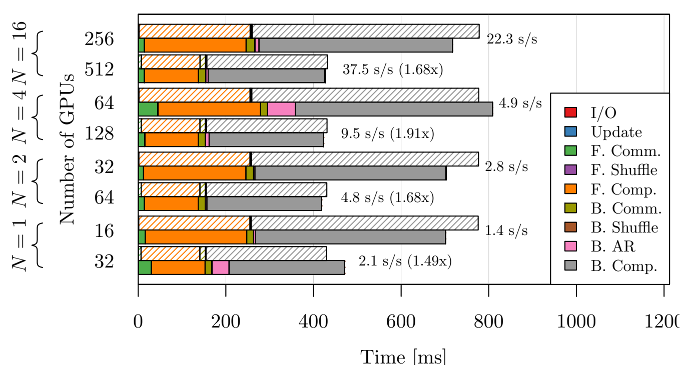
Detailed performance analysis of the CosmoFlow network

To understand the parallel efficiency of our implementation and identify potential bottlenecks, we conduct profiling on the CosmoFlow network by using nvprof [130].

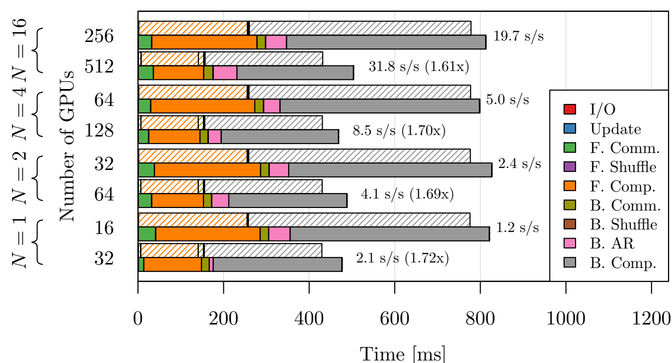
Figure 4.18 shows the GPU execution timeline of a mini-batch iteration when 32 and 64 GPUs are used to train the 512^3 model with a mini-batch size of 4. The “Main” row corresponds to the CUDA stream where compute kernels are launched; the “Halo xchg” row is an asynchronous stream to perform



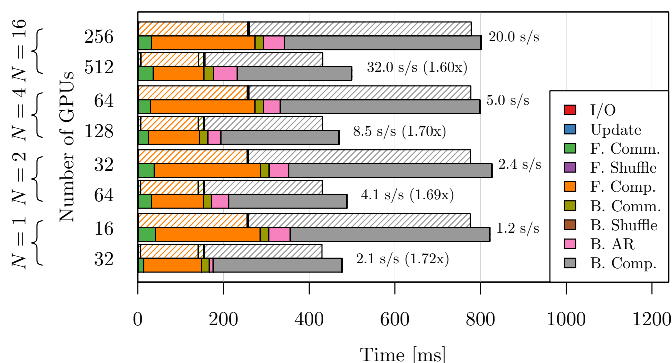
(a) Without batch-normalization layers (epoch 0)



(b) Without batch-normalization layers (epoch 1+)

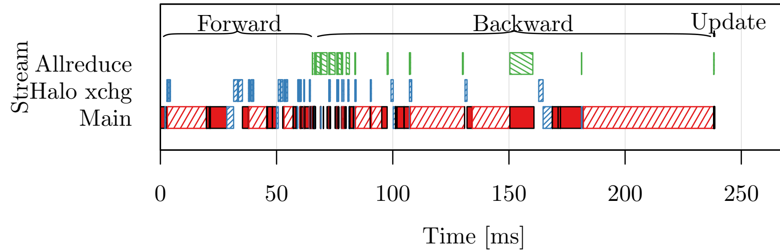


(c) With batch-normalization layers (epoch 0)

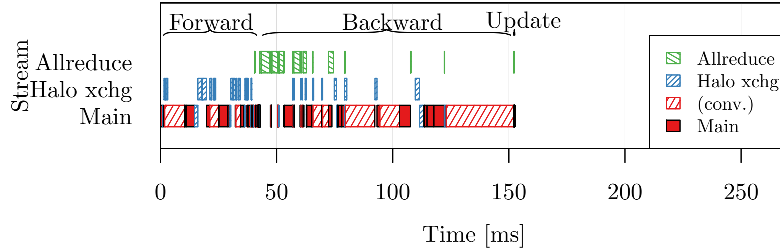


(d) With batch-normalization layers (epoch 1+)

Figure 4.17: **Strong scaling of the 3D U-Net with 256^3 input cubes**. Shaded bars show iteration time predicted by the performance model. The figure format follows that of Figure 4.16.



(a) $N = 4$, 512^3, 8-way, 32 GPUs



(b) $N = 4$, 512^3, 16-way, 64 GPUs

Figure 4.18: **Single-GPU execution timelines for training the CosmoFlow network with the 512^3 input cubes**. We use the mini-batch size of 4. We show one iteration of the root process's GPU of each run.

on-device halo exchanges; the "Allreduce" row corresponds to a stream used by the asynchronous all-reduce operations by NCCL. From the beginning of back-propagation, NCCL starts to communicate computed parameter gradients among processes asynchronously to the main computation stream. Since the communication of gradient updates is done asynchronously, this does not block the compute kernels. In both cases, the main streams are nearly fully packed, indicating the GPU compute units are fully occupied. Similarly, the timelines indicate that the cost of our optimized halo exchanges is almost negligible in these scenarios.

As shown in the figure, a speedup of approximately 1.66x is achieved using 2× the number of GPUs. We see that the speedup from the 8-way to 16-way parallelization is mostly determined by the speedups of the individual convolution kernels in the cuDNN library. In this work, we have exclusively relied on cuDNN for optimized convolution kernels. These results indicate that they may not be well-tuned for non-cube domains, as we only achieved a 1.66x speedup going from 8-way to 16-way parallelization. Identifying the local compute kernels as the bottleneck to better scaling is also corroborated by our performance model, shown in Figure 4.16b and Figure 4.17, which was generated by profiling cuDNN. Nevertheless, the convolutional layers take up most of the GPU's computation time, although the model includes layer types other than the convolutional layers. Therefore, spatial partitioning is an extremely suitable parallelization method for such a network, where convolutional layers are evenly distributed across GPUs.

We further evaluate the computational efficiency by examining the performance of each cuDNN

Table 4.9: **Achieved performance of CosmoFlow convolution layers compared to the peak performance of cuDNN**.

Depth	N	Layer	Time [ms]	Performance [TFlop/s]	Peak perf. [TFlop/s]	Relative [%]
8-way	64	All	142.9	22.6	23.6	95.6
32-way	64	All	48.8	89.9	109.1	82.4
8-way	64	conv1	73.9	12.2	13.0	93.8
32-way	64	conv1	23.5	34.6	53.4	64.7

convolution kernel. Table 4.9 shows the achieved performance of each convolutional layer of CosmoFlow with 512^3 cubes and its peak performance estimated from cuDNN runtime. The "Time" and "Performance" columns give the measured performance of our code (including halo communication, etc.), measured with nvprof. In the "Peak perf." column, we report the TFlop/s achieved by running only the local cuDNN kernel for that configuration. This gives an effective upper bound of the performance we can achieve using cuDNN in our configuration. Finally, we report the achieved percent of this peak in the "Relative" column.

We observe that for CosmoFlow, we achieve 95.6% and 82.4% of this peak performance for 8- and 32-way partitioning, respectively. This indicates that the overhead of our distributed convolution is relatively small than the computation itself. We also observe another benefit of strong scaling that the potential peak performances exhibit super-linear scaling slightly. This is because a larger aggregated memory space are available distributing the workload among more number of GPUs, which allows cuDNN to use more efficient convolution algorithms.

We also examined which layer dominates runtime, and for CosmoFlow, we find that the first convolutional layer accounts for almost half of the entire network runtime. This is due to the layer processing the largest spatial dimensions. For 8-way partitioning, we achieve excellent scaling efficiency; for 32-way partitioning, communication overheads limit increases but still enable overall performance improvements.

While these results show good scaling efficiency for the achievable peak with cuDNN, the TFlops/s achieved is relatively low compared to the theoretical peak of the hardware. This indicates that there is significant potential for further optimizing 3D convolution kernels, as implied by other literature [59].

4.3.3 Weak scaling

Weak scaling, the performance with a varying mini-batch size proportionally to the number of processes, is a commonly-used metric to evaluate the performance of distributed deep learning frameworks, even though it does not necessarily deal with the large-batch problem [87]. In this section, we evaluate the weak scaling performance of our framework to that demonstrate that hybrid-parallel training of 3D CNNs can scale on hundreds of GPUs. Note that the large-batch

problem might not happen in our case because we apply spatial partitioning, which drastically decreases the mini-batch size for a fixed amount of available computing resources (i.e., the number of GPUs). Indeed, even we use 512 GPUs for a single training experiment in Section 4.3.4, the mini-batch size is still 64, which is more than 100x smaller than the mini-batch size reported to be problematic by the previous study [59].

Figure 4.19 shows our preliminary experiments using smaller CosmoFlow cubes (128^3 and 256^3). At the time of this experiment, our CosmoFlow model did not have batch-normalization layers, and spatially-partitioned I/O is not applied; yet, it achieves good weak scaling performance because the time cost of I/O is constant to the number of GPUs. An important observation is that 2D partitioning using 4 GPUs per sample ("4-way partitioning") does not work better than 1D partitioning using the same number of GPUs ("2×2-way") for both cases. This is because when two different parallel strategies use the same number of GPUs per sample, per-GPU computation workload (the number of neurons per GPU) is almost the same, but the number of halo exchange steps is the number of dimensions to be partitioned, which increases the communication cost. > 1 D partitioning is beneficial for the case where each spatial width is too small to partition among a large number of GPUs. However, our models, the CosmoFlow network and the 3D U-Net, have enough widths to utilize hundreds of GPUs, and we observe a considerable performance drop when each sample is over-decomposed, so we adopt 1D partitioning in this study.

Figure 4.20 shows the weak scaling performance of the two 3D CNNs with different input sizes. For the CosmoFlow network with 128^3 cubes, the data size used in the original paper, we use per-GPU batch sizes of 8 and increase the global mini-batch size as we increase the number of GPUs. We evaluate the performance using 4-way and 8-way partitioning for reference. In the 512^3 case, we only evaluate hybrid parallelization where each data sample is partitioned among 8, 16, or 32 GPUs as nearly 53 GB of memory is required per sample as mentioned in Section 4.2.5.

In the case of CosmoFlow with 128^3 cubes, our implementation achieves a nice speedup up to 512 GPUs (128 compute nodes) because of the high compute-to-communication ratio of the 3D CNNs, and the asynchronous overlapped communication implementation of Aluminum. We achieve a 65.4x speedup on 512 GPUs compared to 4 GPUs with the 128^3 cubes. In this case, the highest efficiency is achieved with the data-parallel scheme since the hybrid parallelization involves additional communications due to halo exchanges.

With 512^3, however, hybrid parallelization is required because the model is too large to fit into the device memory of one GPU. We evaluate three configurations, 8-, 16-, and 32-way, and the global mini-batch size is linearly increased as the number of GPUs is increased, resulting in 147.31x, 71.32x, and 37.2x of speedup on 2048 GPUs over 8, 16, 32 GPUs (where the mini-batch size is 1), respectively. With the 3D U-Net, we achieve good weak scalability (28.4x on 1024 GPUs over 32 GPUs with 32-way partitioning) as well.

In all cases, increasing the spatial parallelism results in lower throughput due to the additional

communication overhead as well as the decreased compute efficiency of the cuDNN kernels. Still, we emphasize that the hybrid parallelization enables further speedups for a given fixed mini-batch size, as is also shown in Section 4.3.2.

Comparison between prior work

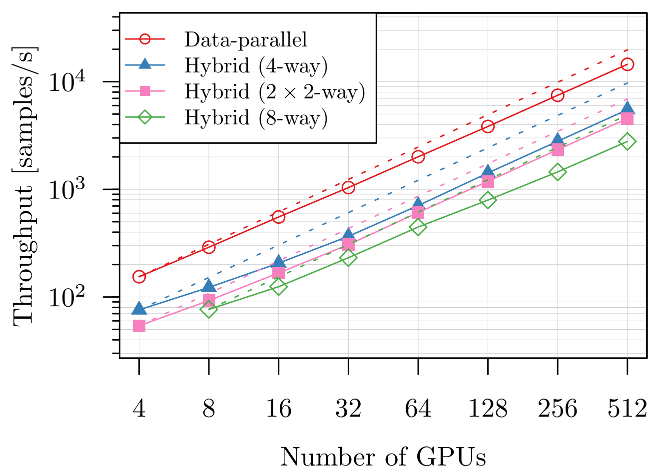
Although many previous studies have evaluated hybrid parallel training of DNNs [45, 51, 52, 53, 131], many of those codes have either not been incorporated into practical or publicly available frameworks or have adopted methods other than spatial partitioning that are suitable for 3D CNNs [45, 132]. In this section, we compare our performance with Mesh-TensorFlow [133], an extension library to a well-known framework TensorFlow, which applies spatial partitioning to DNNs. Note that Mesh-TensorFlow only supports a single CPU or GPU node and TPU [134] clusters, but not multiple GPU nodes.

In Figure 4.16, we compare with TensorFlow 2.2.0 [33] with Horovod 0.19.1 [100] (TF) for data parallelism and Mesh-TensorFlow 0.1.16 (MeshTF) for hybrid parallelism. We convert the CosmoFlow dataset into a set of TFRecord [135] files, each of which is a 128^3 data cube in 16-bit integers along with corresponding ground-truth parameters. We combine `tf.data.TFRecordDataset` to create a TensorFlow dataset, and then apply `shuffle` and `prefetch` with the `tf.data.experimental.AUTOTUNE` buffer size to mimic the behavior of our framework as much as possible. We use the original NCDHW file format for the TensorFlow+Horovod experiment, but we use the NDHWC format for the Mesh-TensorFlow experiment because it only supports the format for convolutional layers.

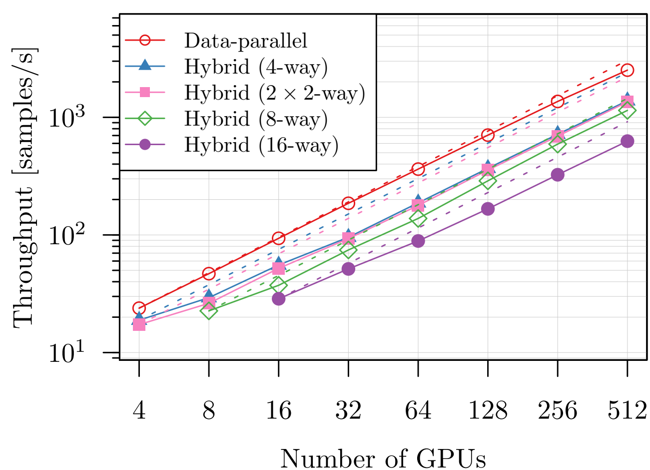
TensorFlow with Horovod achieves similar performance to LBANN in the data-parallel regime for 128^3 samples (Figure 4.20a). This is due to both frameworks using the same underlying computation and communication libraries, cuDNN and NCCL (Figure 4.21), and indeed 88.8 % and 71.8 % of the time are spent on cuDNN or NCCL on 16 nodes in LBANN and TensorFlow, respectively. However, as we demonstrate in Section 4.3.4, increasing the input resolution is crucial to improve accuracy, and thus enabling hybrid-parallel training is much more important for such applications.

On the other hand, as shown in Figure 4.20a, when hybrid-parallel is enabled, our framework significantly outperforms Mesh-TensorFlow on one computing node that has 4 GPUs. Our framework achieves 2.99x (52.86 samples/s on LBANN and 17.67 samples/s on Mesh-TensorFlow) of throughput compared to Mesh-TensorFlow. Moreover, as we already explained, our framework is capable of scaling to hundreds of GPUs, which makes it much more efficient than Mesh-TensorFlow. For the same reason, 512^3 data cubes cannot be used with Mesh-TensorFlow because at least 8 GPUs are needed.

As a result of closer profiling on Mesh-TensorFlow, it turned out that Mesh-TensorFlow is implemented on a wrong level of abstraction that creates an inefficient communication pattern. Mesh-TensorFlow is implemented on top of the TensorFlow interface, which creates appropriate communication operations to compute a given computational graph on multiple devices. Although

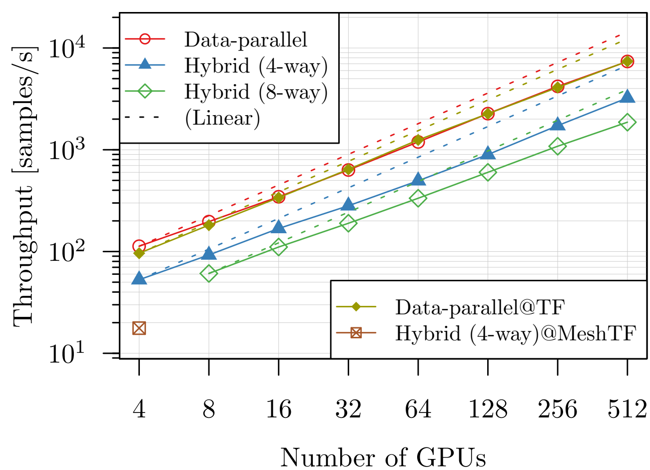


(a) CosmoFlow without BN layers, 128^3

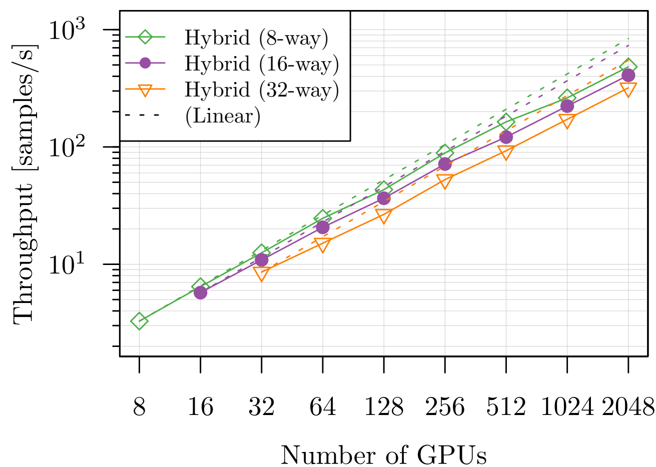


(b) CosmoFlow without BN layers, 256^3

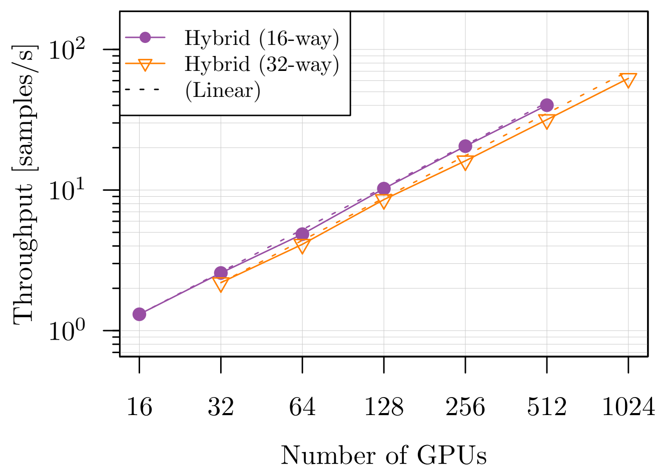
Figure 4.19: **Weak scaling of the CosmoFlow network with different spatial partitioning**. We increase the global mini-batch size as we increase the number of GPUs. In the hybrid results, we partition a single sample by multiple GPUs in its spatial domain.



(a) CosmoFlow, 128^3



(b) CosmoFlow, 512^3



(c) U-Net, 256^3

Figure 4.20: **Weak scaling of the two different 3D CNNs**. "Data-parallel" is executable only in Figure 4.20a due to the GPU memory requirements. "@TF" and "@MeshTF" are TensorFlow with Horovod and Mesh-TensorFlow, respectively.

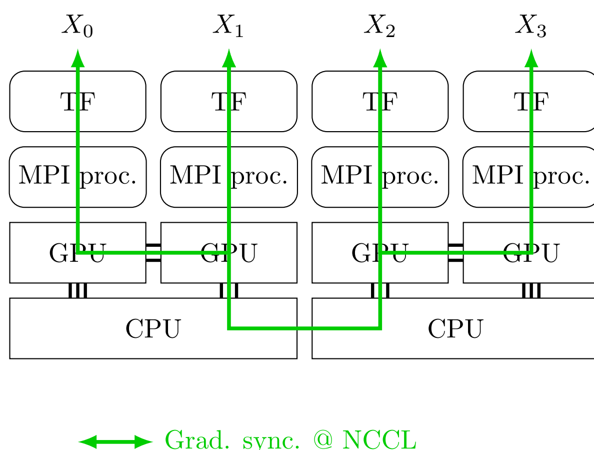


Figure 4.21: **The software stack of Horovod with data-parallelism**. X_i is the i-th data sample.

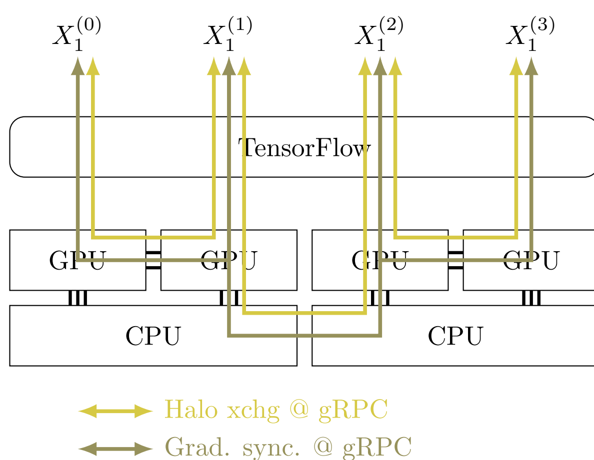


Figure 4.22: **The software stack of Mesh-TensorFlow with model-parallelism**. TensorFlow launches a single process for each computing node, and it manipulates available GPUs on the node. $X^{(j)}$ is the j-th partition of the data sample.

TensorFlow uses gRPC as the default communication backend and it is not explicitly mentioned how this communication is done in the CUDA environment [33] (Figure 4.22), as we profile TensorFlow processes, it is implemented by using P2P memory copy in the current version of TensorFlow (Figure 4.23). In fact, Mesh-TensorFlow's P2P memory copy only occurs when model-parallelism is enabled (Table 4.10). However, we observe that TensorFlow stalls at every memory copy operation, which makes its performance significantly inefficient (Figure 4.24). We expect this inefficient communication is why Mesh-TensorFlow performs poorly.

Figure 4.23: **TensorFlow's GPU matrix multiplication (matmul) of two tensors**.

(a) matmul on a single GPU

```
1 import tensorflow as tf
2
3 with tf.device('/GPU:0'):
4     A = tf.random.uniform([1024, 1024], dtype=tf.float32)
5     B = tf.random.uniform([1024, 1024], dtype=tf.float32)
6     C = tf.matmul(A, B)
```

(b) Executed CUDA kernels on a single GPU

```
1 [CUDA memset]
2 void tensorflow::..::UniformDistribution..
3 void tensorflow::..::UniformDistribution..
4 [CUDA memcpy HtoD]
5 [CUDA memset]
6 volta_sgemm_128x32_nn
7 [CUDA memcpy DtoH]
```

(c) matmul on two GPUs

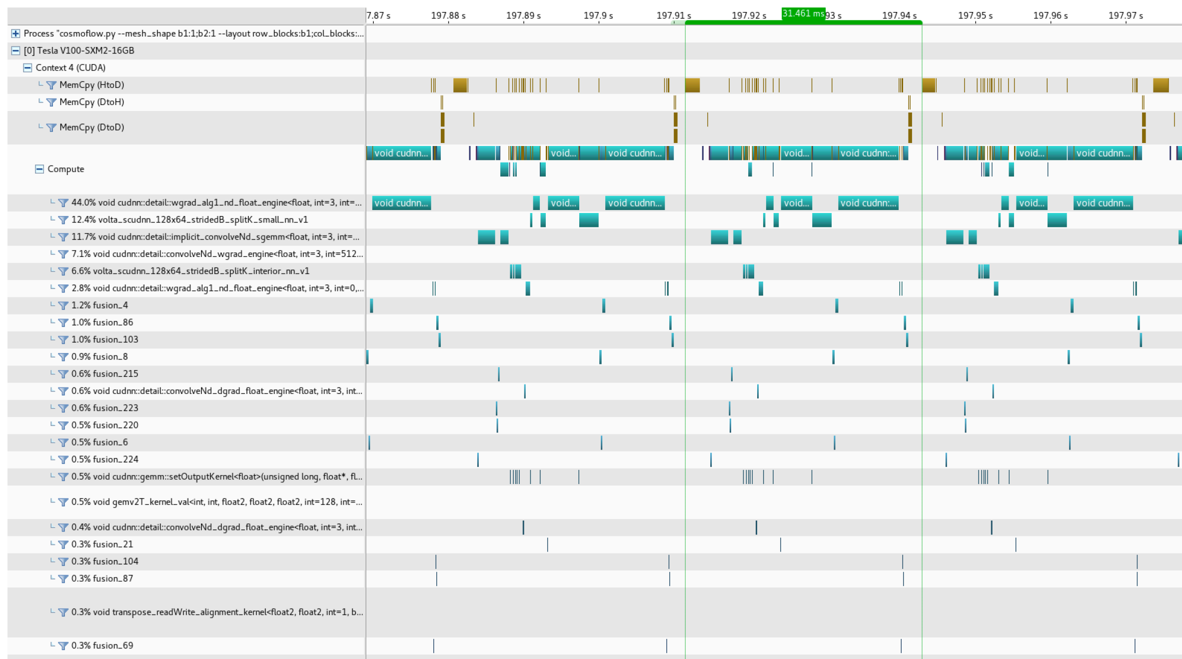
```
1 import tensorflow as tf
2
3 with tf.device('/GPU:0'):
4     A = tf.random.uniform([1024, 1024], dtype=tf.float32)
5 with tf.device('/GPU:1'):
6     B = tf.random.uniform([1024, 1024], dtype=tf.float32)
7 with tf.device('/GPU:0'):
8     C = tf.matmul(A, B)
```

(d) Executed CUDA kernels on two GPUs

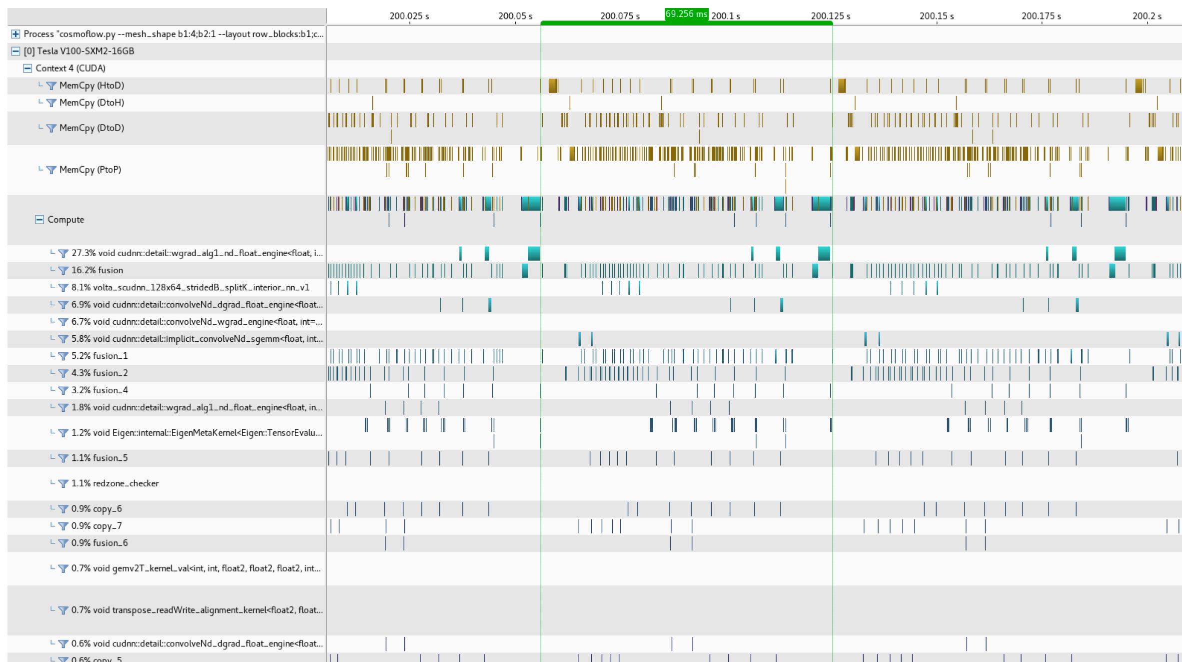
```
1 [CUDA memset]
2 void tensorflow::..UniformDistribution..
3 [CUDA memset]
4 void tensorflow::..UniformDistribution..
5 [CUDA memcpy HtoD]
6 [CUDA memcpy PtoP]
7 [CUDA memset]
8 volta_sgemm_128x32_nn
9 [CUDA memcpy DtoH]
```

Table 4.10: **The number of kernels/memcpy of Mesh-TensorFlow on the CosmoFlow network in one training iteration**. We show the modes of the numbers in 128 iterations. We use a mini-batch size of one.

	Total	HtoD	DtoH	DtoD	PtoP
DP	351	22	2	96	0
4-way	596	24	2	85	122



(a) data-parallelism



(b) 4-way partitioning

Figure 4.24: **Timeline of Mesh-TensorFlow with data-parallelism and 4-way partitioning**. We use a mini-batch size of one. Each green range shows a rough estimation of the duration of one training iteration. Note that the percentage of each kernel shown is calculated from the entire run, which is not necessarily the same as the percentage of each training iteration.

4.3.4 CosmoFlow model accuracy improvement with 512^3 data cubes

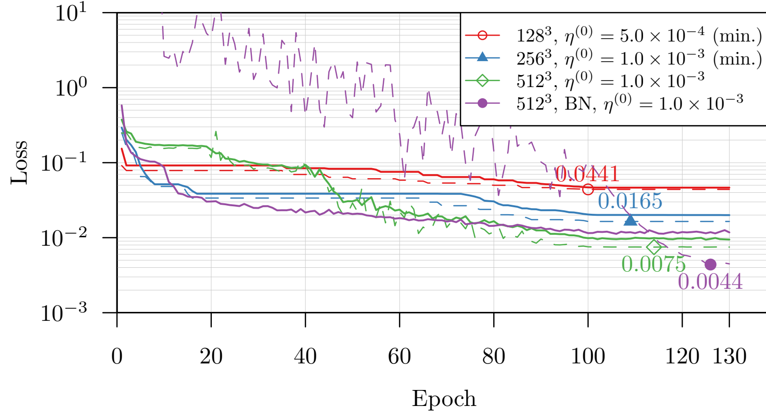
Our original motivation is to improve the convergence of 3D CNNs by providing high-resolution 3D data that cannot be used on GPUs without hybrid-parallelism. Therefore, in this section, we demonstrate that this assumption holds for the CosmoFlow dataset by conducting end-to-end training using the original data size, four-channels, 512^3 3D cube.

Figure 4.25 shows the training results of our CosmoFlow network variant with the full-resolution dataset (512^3) and its split versions (128^3 and 256^3). We swept the initial learning rate from 10^{-4} to 10^{-2} logarithmically and show the results with the best configuration. We train all networks for 130 epochs with a mini-batch size of 64 in every configuration, and use the 4-way partitioning (256 GPUs in total) for the networks without batch normalization layers, or 8-way (512 GPUs in total) for networks with batch normalization, due to the increased memory requirements. To account for training variance, we show the median result of five trials with different initial random seeds.

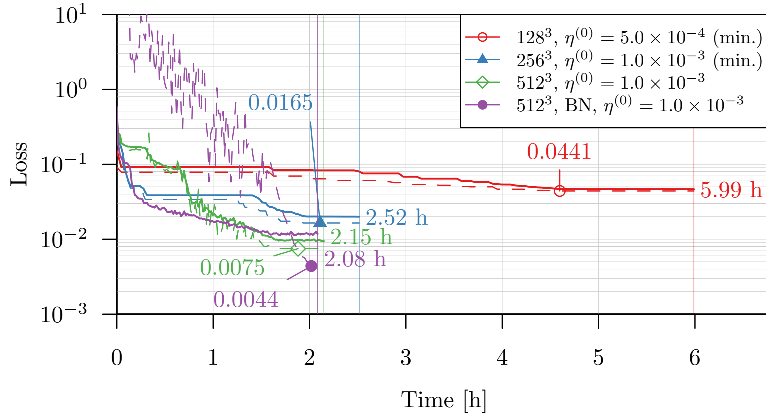
We observe that the test loss significantly decreases as we increase the dataset size to 0.0169 MSE with 256^3 and 0.00727 MSE with 512^3 data. Adding batch normalization improves this result further, to 0.00445 MSE, achieving an order-of-magnitude improvement compared to the baseline 128^3 data. At the same time, we get 2.79x of speedup from 128^3 to 512^3 with the same number of GPUs and the same mini-batch size. This result implies that CNNs can be trained with the same computing resources and dataset size, but with a smaller mini-batch and small overheads (see Section 4.3.3). This brings an opportunity to keep mini-batch sizes fixed and strong-scale onto more GPUs for speedup.

Figure 4.26 shows the correlation between the predicted and actual cosmological parameters of our networks with different input sizes. We clearly demonstrate improvements in the quality of predictions with increasing data volume size. In particular, the prediction of H_0 (the Hubble constant) shows the most improvement in accuracy with increasing data volume size. We assume that this is because the variable is related to the large-scale expansion of the universe. As cosmological simulations move to sub-percent measurements, being able to test the quality of the surrogates via a greatly improved CosmoFlow network, with an order of magnitude improvement in the measurement of the cosmological parameters, is the only way to validate the quality and precision of the models quickly.

Table 4.11 shows the relative prediction errors of each parameter. This again clearly shows that using larger data volumes improves the prediction quality for all parameters. Note that, while the prior work on CosmoFlow achieved a lower absolute error for Ω_m, σ_8, n_s cosmological constants, it is not directly comparable because the datasets used are different; this updated dataset has a broader dynamic range for each of the cosmological parameters, plus an additional parameter to predict (H_0), increasing the parameter space by two orders of magnitude. Hence, the difficulty of predicting the four parameters is much more serious than with the previous dataset. Nevertheless, we achieve similar errors with our models trained on 512^3 data.



(a) Epoch



(b) Time

Figure 4.25: **Training of the CosmoFlow network with different input resolutions**. The solid and dashed lines show training and validation losses, respectively. The points show the smaller validation losses. For 128^3 and 256^3, we show the minimum loss values at each point in time for visibility.

Table 4.11: **Comparison of relative prediction errors**.

	Ω_m	σ_8	n_s	H_0
Previous work [59]	0.0022	0.0094	0.0096	N/A
128^3	0.0380	0.0380	0.0394	0.0751
256^3	0.0234	0.0241	0.0256	0.0468
512^3	0.0158	0.0167	0.0188	0.0285
512^3, BN	0.0102	0.0126	0.0142	0.0241

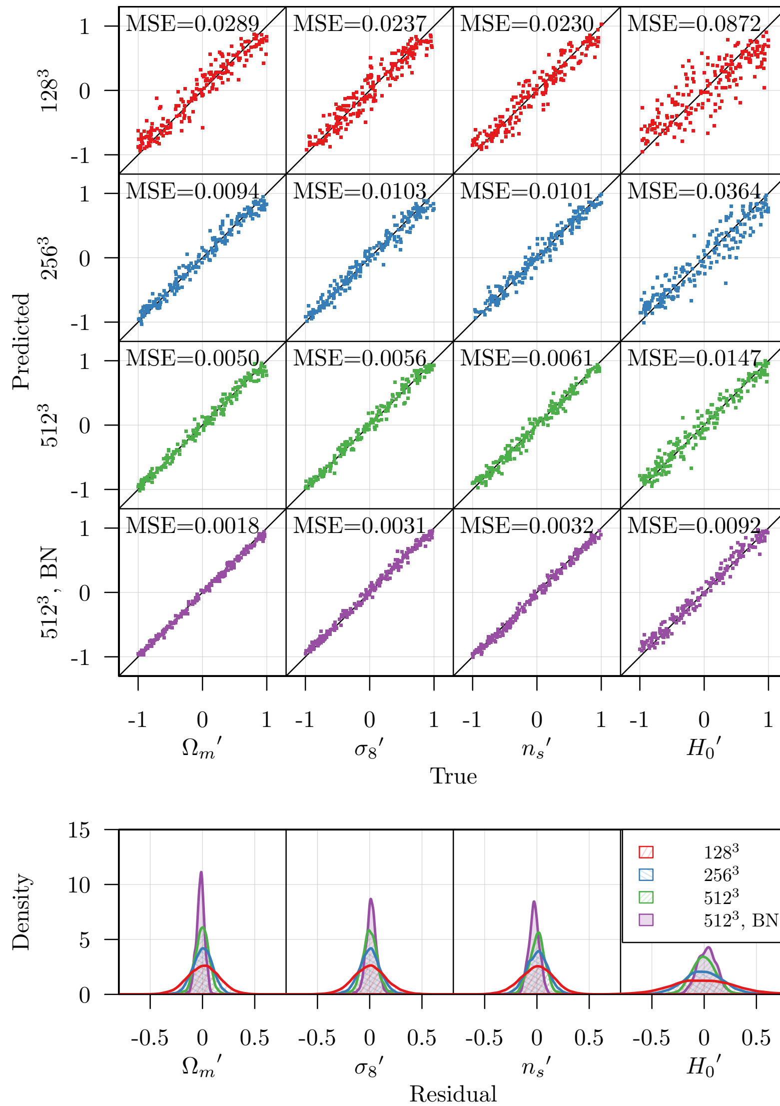


Figure 4.26: **True and predicted cosmological parameters from four different configurations**. The bottom figure shows the distribution of the residuals. The parameters are normalized to $[-1, 1]$. In the top figure, we show 200 randomly chosen data points for visibility.

Chapter 5

Related work

The issue of GPU memory capacity often comes up in deep learning applications. Although our two proposed methods optimize intra-processor and inter-processor performance, respectively, their common objective is to maximize the computational efficiency within limited GPU memory constraints. Even though hybrid-parallel training partially resolves the problem by distributing per-sample data among multiple GPUs, it is still common to maximize the use of GPU memory by increasing the local batch size, and thus the workspace issue still arises. In fact, the original study of the CosmoFlow network [118] also reported that the trade-off between memory usage and computation speed by FFT-based convolution appeared in the real application, which implies that there is a great demand for methods that automatically tune convolution algorithms independent of the users' tuning skills and knowledge of computer science.

In this chapter, we classify prior studies about optimizing the computational efficiency of deep learning frameworks and applications into two classes, intra-processor and inter-processor optimization, and discuss their advantages and disadvantages compared to our work.

5.1 Optimizing intra-processor parallelism

Li et al. [136] propose a heuristic to tune each tensor's memory layout of CNNs to utilize either GEMM-based or FFT-based convolution efficiently. The proposed heuristic is, however, based on the authors' performance observation using several conventional convolutional layers and specific GPU architecture. Thus there is no guarantee that the algorithm always provides the best memory layout for unseen CNNs and GPU architecture. On the other hand, μ-cuDNN uses dynamic programming and integer linear programming/ Thus it is guaranteed that μ-cuDNN provides the best performance that the library can produce for a given GPU architecture.

Rhu et al. [81] propose a memory management technique that offloads neuron activations, errors, and network parameters from the GPU memory to the CPU memory during forward-/backward-

propagation so that users can train larger models with the same memory constraint. However, as we show in Section 3.3 even in such memory-efficient implementations or similar memory management techniques [86] μ-cuDNN is expected to save the peak memory usage of each layer. Moreover, we revealed that there are cases where careful algorithm selection is needed to improve the performance even when sufficient workspace is given, which is solved by μ-cuDNN.

Zlateski et al. [75] propose ZNNi, an FFT-based convolution algorithm, and they mention a technique similar to μ-cuDNN to reduce the temporal memory usage by the algorithm. In our work, we generalize the schema so that loop splitting can be applied to any convolution algorithm, obtain the best computational performance for the given layer configurations, and maintain high portability between different deep learning frameworks.

The issue of GPU memory capacity often comes up in deep learning applications. We discuss another strong approach to mitigate the memory pressure, hybrid-parallel training, in Chapter 4. Still, it is common to maximize the use of GPU memory even when hybrid-parallelism is utilized, and thus the workspace issue still arises.

5.2 Optimizing inter-processor parallelism

As we explained in Section 2.2, there are various ways to parallelize the computation of deep learning frameworks; Tal et al. [25] conducted a comprehensive survey about various parallelization strategies for DL frameworks. The idea of dividing a problem into multiple processors is very common in the computer science domain. Thus even before the release of early DL frameworks such as Caffe [38] and TensorFlow[33], model-parallel model training using multiple GPUs was already proposed. For example, Krizhevsky et al. [45] trained a CNN known as AlexNet on two NVIDIA GTX 580 GPUs. They applied channel partitioning to reduce the memory footprint per GPU and allowed only a few layers to exchange their channels for reducing the communication overhead. Dean et al. [51] proposed DistBelief, which employs thousands of CPU cores to apply arbitrary partitioning on a neural network and adopts parameter server synchronization to update local parameters asynchronously. Coates et al. [52] implemented spatial partitioning for 2D networks on a GPU cluster. Chilimbi et al. [53] proposed the Adam framework similar to DistBelief that can adjust multiple parallelism degrees using hybrid-parallelism and the parameter server mechanism.

The TensorFlow framework [33], one of the most well-known machine learning frameworks, has a similar concept to DistBelief; it regards a model as a computational graph and assigns each processor to one or more nodes of the graph so that users can split the model freely without worrying about implementation limitations. Indeed, this philosophy has the advantage of exploiting hybrid-parallelism more quickly than other frameworks such as LBANN that were designed with the intention of data-parallelism. For instance, GPipe [132] implements layer pipelining on the top of the TensorFlow interface. Similarly, Mesh-TensorFlow [133, 137] implements spatial partitioning; it

allows users to specify how each tensor is split in each dimension/ It then inserts appropriate communication automatically to perform model-parallel training. The authors evaluated its performance on the Transformer [91] model [133], and the 3D U-Net [137], using hundreds of TPU [134] cores. They also reported that they successfully trained the 3D U-Net with an input resolution of 512^3. In Section 4.3.3, however, we find that the timing of communication in this library is dependent on the implementation of TensorFlow and that it is not fully optimized to perform spatial partitioning for 3D CNNs on a GPU cluster, so the LBANN framework with our hybrid-parallel extension still outperforms Mesh-TensorFlow. We also emphasize that Mesh-TensorFlow does not support multiple GPU nodes that is a severe limitation on parallelism, while our framework runs on hundreds of GPUs.

Several studies applied performance modeling for hybrid-parallel training on deep learning frameworks [102, 103, 104]. While these studies focus on relatively small models where data-parallel training is possible, we show in this work that performance modeling is still useful for estimating training throughput for huge models where data-parallel training is infeasible.

FlexFlow [131] is another deep learning framework that considers the sample, operation, attribute, and parameter dimensions distributed among GPUs. The framework can minimize the execution time of a given DNN architecture by using a simulator to estimate the execution time of a given computational graph, and the Markov Chain Monte Carlo (MCMC) search algorithm to explore the parallelization strategy design space. However, it is still limited to 2D networks, even though 3D models truly require hybrid-parallelism, as we show in this chapter.

The Distconv library [104, 122] implements spatial and channel partitioning for 2D CNNs. We extend the library to support various types of 3D CNNs, as we explain in this chapter. We also demonstrate that appropriate hybrid parallelization of I/O is also necessary to handle high-dimensional models, which has not been done in the prior work.

Several methods have been proposed to manage GPU memory efficiently to handle huge models without explicitly dividing them [81, 82, 83]. One type of such approaches is to copy tensors from/to the CPU memory during forward and backward passes, not to overflow the GPU memory. This memory copy often overlaps with computation to minimize the overhead. The advantage of this method is that the framework itself does not need to be modified fundamentally. In fact, ooc_cuDNN [82] proposes a transparent implementation using the cuDNN interface. Also, NVIDIA's Unified Memory [138] allows frameworks to use CPU memory almost entirely transparently. Specifically, it is suited to architectures with high CPU-GPU communication bandwidth, such as Lassen nodes (Figure 4.15). Another technique is to reacquire intermediate data by discarding tensors calculated at the beginning of the pass and recomputing it later in the pass [83, 84, 85].

While these methods have proven to be effective for several different models, we believe that spatial partitioning is the most suitable per-GPU data size reduction method for 3D CNNs. This is because it cannot scale with the model size in the same way as data-parallelism; for example, the CosmoFlow

network with batch-normalization layers example requires about 100 GiB of memory to compute one sample, and Lassen's architecture cannot allocate this size of memory per GPU in the combined CPU memory. Furthermore, the 3D U-Net requires a larger memory size than that. Also, the amount of data moved by offloading methods or computed by recomputation methods is W^3, where W is the spatial width, while that by spatial partitioning is W^2. Even if we consider that the CPU-GPU memory bandwidth and the communication bandwidth between two GPUs on different nodes differ by a factor of ten, the communication overhead of the offloading method is not very small compared to spatial partitioning. Therefore, model-parallelism is still necessary to train such extremely high-resolution 3D CNNs.

Chapter 6

Discussion

In this chapter, we discuss implications and future work of our main two research topics explained in Chapter 3 and Chapter 4.

6.1 Automatic optimization of computational kernels

In Chapter 3, we have shown that loop splitting can significantly speed up convolutional computation, which accounts for the majority of deep learning computations. In this section, we discuss further whether it is possible to extend this algorithm for more general software and computational content.

6.1.1 Implications and possible future work

As we mentioned, our method is applicable to a wide range of CNNs and DL frameworks, and thus our method is expected to accelerate the training and inference of various CNNs. In our basic loop algorithm, μ-cuDNN requires no additional information than that passed through the cuDNN interface, it is possible that the proposed algorithm is incorporated into cuDNN itself.

Applying the μ-cuDNN method for different tensor layout

In this section, we discuss the speedup using the NHWC memory layout. Many frameworks use the NCHW format (Table 3.7), i.e., each element of the tensor is stored sequentially in the order of the spatial, channel, and sample dimensions. For example, in FFT-based convolution, since the transformation is performed for each channel, the NCHW format is regarded to be efficient because the elements in the spatial dimension are contiguous. On the other hand, in GEMM-based convolution, since the multiply-add operation is performed on all input channels, the NHWC format is considered to be efficient. In fact, cuDNN recommends using the NHWC format when using Tensor Cores that are specialized in multiply-add [139]. In addition, when performing convolution using Tensor Cores in

Table 6.1: **GPU trace of DeepBench's 1×1 convolution layer in different memory formats**.

(a) NCHW

Duration [us]	Name
37.12	nchwToNhwcKernel<__half, __half, float, bool=1>
2.176	cudnn::gemm::computeOffsetsKernel
126.431	volta_h884cudnn_256x128_ldg8_relu_exp_interior_nhwc_tn_v1
11.2	nhwcToNchwKernel<__half, __half, float, bool=1>

(b) NHWC

Duration [us]	Name
2.208	cudnn::gemm::computeOffsetsKernel
127.264	volta_h884cudnn_256x128_ldg8_relu_exp_interior_nhwc_tn_v1

Table 6.2: **The support status of ONNX**.

Framework	Exporting	Importing
Caffe2	✓	✓
PyTorch	✓	
CNTK	✓	✓
MXNet		(✓)
Chainer	✓	
TensorFlow		(✓)

the NCHW format, extra kernels are executed before and after the main convolution kernel to perform the format conversion (Table 6.1), and thus, NHWC is inevitably faster.

We evaluate the relative speedups of DeepBench's convolution kernels by switching the data layout from NCHW to NHWC in Figure 6.1. The speedup is significant when using Tensor Cores, especially for the 1×1 kernels, which can run efficiently without redundant memory copies. On the other hand, the 3×3 and 5×5 kernels, which are frequently used as much as 1×1 kernels, do not necessarily get faster. This implies that there is room to accelerate convolution by adaptively switching the memory layout in a similar way μ-cuDNN tunes the batch size and convolution algorithms. Furthermore, cuDNN's workspace interface allows μ-cuDNN to do such optimization on memory layouts by considering the memory conversion overhead in terms of both time and workspace size.

Implementation-independent micro-batching using ONNX

Open Neural Network eXchange (ONNX) [140] is a data format to exchange model definitions between different deep learning frameworks. ONNX provides a way to describe computational graphs and defines primitives as nodes of the graphs. Since each deep learning framework has different advantages (such as support for specific accelerators or support for recently proposed layer types), ONNX aims at allowing users to exploits such advantages by exchange models between frameworks. Many frameworks support importing and exporting ONNX models (Table 6.2). Figure 6.2 shows an example code to define a convolution node with the Chainer framework and ONNX.

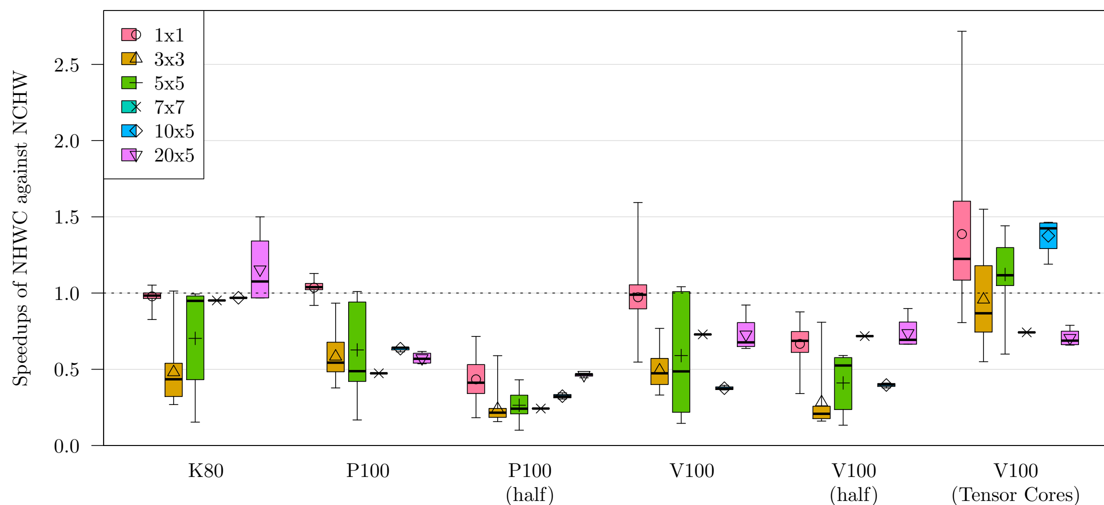


Figure 6.1: **Speedups of convolutional layers with the NHWC format compared to the NCHW format**. The bold line, edges, and whiskers of each box show the median, 1Q/3Q, and the minimum/maximum values, respectively.

Figure 6.2: **Making an ONNX convolution node**.

```
>>> import onnx

>>> node_with_padding = onnx.helper.make_node(
      'Conv',
      inputs=['x', 'W'],
      outputs=['y'],
      kernel_shape=[3, 3],
      pads=[1, 1, 1, 1],
    )

>>> node_with_padding
input: "x"
input: "W"
output: "y"
op_type: "Conv"
attribute {
  name: "kernel_shape"
  ints: 3
  ints: 3
  type: INTS
}
attribute {
  name: "pads"
  ints: 1
  ints: 1
  ints: 1
  ints: 1
  type: INTS
}

>>> type(node_with_padding)
onnx_pb2.NodeProto
```

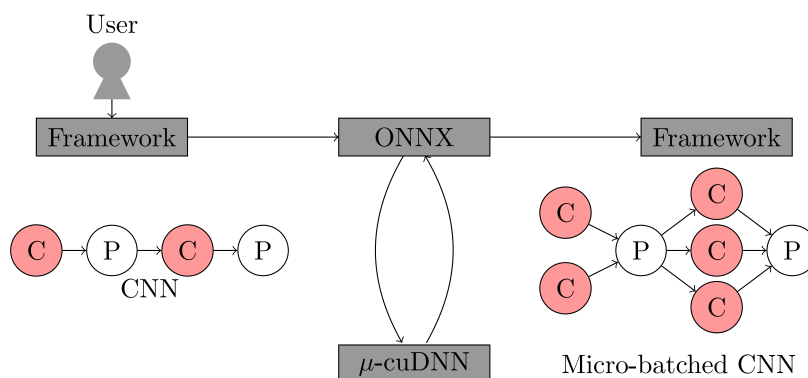


Figure 6.3: **Combining the μ-cuDNN method with ONNX**. Note that "μ-cuDNN" represents the WR algorithm, but not the C++ library itself.

In this section, we prototype an ONNX-based Python auto-tuning library to apply μ-cuDNN's loop splitting algorithm to CNNs in ONNX, but without integrating the μ-cuDNN library with the framework (Figure 6.3). The motivation here is to reduce users' effort to adopt our method and provide performance portability among frameworks. Although our library can be combined with any deep learning frameworks that employ cuDNN, and users don't have to modify the logic of the frameworks, it is troublesome to compile a deep learning framework from scratch due to its complicated software dependencies. In our new proposal, users no longer modify the framework and compile to incorporate μ-cuDNN by using the common model exchange format. Note that we only assume optimizing CNNs in terms of forward computation time because ONNX does not support backward computation,

This library assumes the following scenario to apply μ-cuDNN's WR algorithm to CNNs:

1. Users construct a CNN model on a deep learning framework. The data type of the model depends on the framework, so distinct implementation is required to apply auto-tuning to its convolutional layers.

2. Users convert the model to the equivalent ONNX model. Specifically, its convolutional layers are converted to ONNX's `Conv` nodes.

3. The library applies the WR algorithm to the ONNX version of the model. It is the users' responsibility to implement a function that takes an ONNX model and the mini-batch size as input and output time to compute the forward pass. Since it is not possible in ONNX to apply convolution for a subset of a mini-batch, we insert `Split` and `Concat` nodes between split `Conv` nodes so that each `Conv` node computes a subset.

4. The library returns the transformed ONNX model to the users, and users re-transform the model to the framework.

Figure 6.4 shows AlexNet before and after the WR transformation on Chainer v4.4.0. We achieve a 1.41x speedup (55.7 ms to 39.4 ms), including all other operations on V100-SXM2 GPU, which is

similar to the speedups we observed in Section 3.3.2. This implies that the WR algorithm does not necessarily require modifying the framework code but can be fulfilled by using ONNX. Since deep learning frameworks usually rely on many dependency libraries for computation, communication, and file accesses, and thus much effort is required to compile from scratch, the ONNX-based approach has an advantage that users can easily enjoy speedups just by using the library externally.

Applying the μ-cuDNN method for other layer types

Our proposed method does not depend on the computational content of the layer itself but is applicable as long as the computational efficiency varies nonlinearly with the workspace limit or batch size. Thus, we conduct a sensitivity analysis of the computational performance using ResNet's batch-normalization layers. Figure 6.5 shows the change in execution time when architecture parameters (the number of channels (C) and spatial width/height (W, H)) of the last batch-normalization layer of ResNet's "conv2" block are varied. We add $\pm 1, \pm 2, \pm 4, \cdots$ to the default parameters. When the mini-batch size is 32, it gets 1.25x faster by increasing the layer width, W is increased from 55 to 56, which is counterintuitive. This is because the global memory throughput is reduced because the 2D tensor size of each channel is not a multiple of the cache line size, 128 bytes (Table 6.3). Although μ-cuDNN utilizes the characteristics of cuDNN's convolutional layers that various algorithms that have different performance advantages are available, this type of performance nonlinearity is quite common in computer science; for example, we previously reported that the performance of cuBLAS's GEMM in a deep learning framework varies with different matrix sizes drastically [113], which implies that it internally selects different GEMM algorithms (in fact, the `cublasGemmEx` routine allows users to specify a GEMM algorithm explicitly [141]). This result motivates us to split the dimensions other than the sample dimension. As we mentioned, the runtime configurations of convolutional layers, especially on NVIDIA GPUs, are complex and could allow for a broader scope of exploration of μ-cuDNN's optimization algorithm. However, when dividing dimensions other than the sample dimension, an additional memory copy is required because the memory addresses are not continuous, and thus this overhead needs to be considered in the optimization algorithm. In addition, since there is room for speeding up layer types other than convolutional layers by such a method, a more general method can be considered, which applies graph transformations to divide arbitrary dimensions to improve the kernel performance.

6.2 Training 3D CNNs with hybrid-parallelization

In Chapter 4, we demonstrated that end-to-end training of high-resolution 3D data, which was infeasible in previous studies, can be achieved by implementing hybrid-parallel training. Although the method of splitting 3D data between GPUs is already known in the context of parallel stencil computations, we implemented it in a practical deep learning framework and evaluated its

(a) Before transformation

(b) After transformation

Figure 6.4: **Visualization of AlexNet on the micro-batching technique on V100-SXM2**.

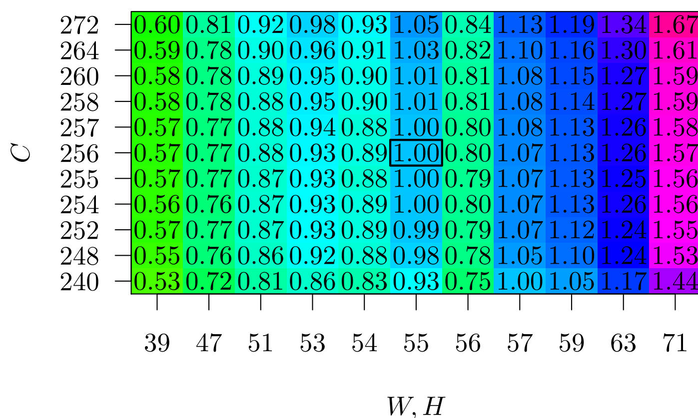


Figure 6.5: **Relative time to compute ResNet's batch-normalization layer with cuDNN on V100-SXM2**. We use Chainer v4.4.0, CuPy 5.0.0b4, cuDNN 7.1.4, CUDA 9.2. The number in each cell shows the time relative to the center cell. We measure each configuration for 100 times after three times of warmup runs and report relative median time.

Table 6.3: **Profiling results of ResNet's batch-normalization layer**. We use the same software and libraries to Table 6.5.

	Global Transactions		Shared Transactions		
	Load	Store	Load	Store	# of FLOPs
	(Throughput [GB/s])		(Throughput [GB/s])		
$W = 55$	6290498 (360.91)	3779584 (216.85)	39850 (9.15)	8788 (2.02)	342299136
$W = 56$	6439424 (488.47)	3212288 (243.67)	32085 (9.74)	8813 (2.67)	354906624

performance to show that it contributes to the improvement of learning accuracy so that the practicality of this approach is clearly demonstrated. In this section, we further discuss the possible implications of this work, a comparison between prior work and our future work.

6.2.1 Predicting multi-dimensional partitioning performance

In this section, we estimate the benefits of splitting 3D CNNs into multiple dimensions. Although the current proposed framework supports only 1D partitioning, decomposition in multiple dimensions may further speed up the training by utilizing a greater number of processes. Therefore, we study the effect on the two models using our performance model.

Figure 6.6 and Figure 6.7 show the predicted time of the two networks using 1D to 3D partitioning strategies. Since the time of each computation kernel on a single GPU can be greatly mispredicted by unnecessarily slow algorithms (as we observed in Chapter 3), we use the linear heuristic model that we used to estimate $SR(D)$ (Section 4.2.4) to predict the computation time on unseen spatial domains. In both figures, we assume the number of available nodes is 512 nodes (2048 GPUs) of Lassen's 792 nodes; for example, $8 \times 8 \times 1$-way partitioning, where the two spatial dimensions of each sample is partitioned among 8 GPUs, respectively, is infeasible with a mini-batch size (N) of 64 on the CosmoFlow network because it requires 1024 nodes (4096 GPUs).

In Figure 6.6, we can see that when the mini-batch size is sufficiently small, 2D partitioning is slightly faster than 1D partitioning. This is because the kernel computation time ("F." and "B. Comp.") is distributed to more number of GPUs, but also the shuffle time ("F." and "B. Shuffle") is reduced because the timing at which LBANN aggregates the layer data to the root GPUs is shifted backward in the network. When the mini-batch size is 64, however, 1D partitioning achieves the best throughput because the number of GPUs per sample is limited to up to 32 to keep the total number of GPUs below the limitation; in this case, $32 \times 1 \times 1$ outperforms $8 \times 4 \times 1$ and $4 \times 4 \times 2$ as we already discussed in Section 4.3.3. A similar result is found in Figure 6.7, but in the case of the 3D U-Net, the spatial width of the layer (up to 256) is smaller than the CosmoFlow network (up to 512), so as the number of GPUs per sample increases the overhead increases and a good speedup is not obtained. Considering that $N = 64$ is a mini-batch size that is actually used for training (Section 4.3.4), these results imply that if the amount of available computing resources is fairly limited (for example, 2048 GPUs in our case), the 1D partitioning method is the most reasonable strategy for practical training. Even on the Sierra supercomputer [142], which shares the same node architecture as Lassen but has 4320 nodes in total, we can still keep the mini-batch size smaller than thousands (the upper limit of which has been reported to degrade accuracy in the previous work) only with 1D partitioning. We also note that multi-dimensional partitioning techniques will be necessary for much more massive parallel environment such as the Fugaku supercomputer [143]; Fugaku is composed of 158976 nodes, each of which equips an A64FX CPU chip, which has computational performance and amount of memory of a similar order to a V100 GPU. Moreover, the CPUs are connected with Tofu Interconnect D, which is

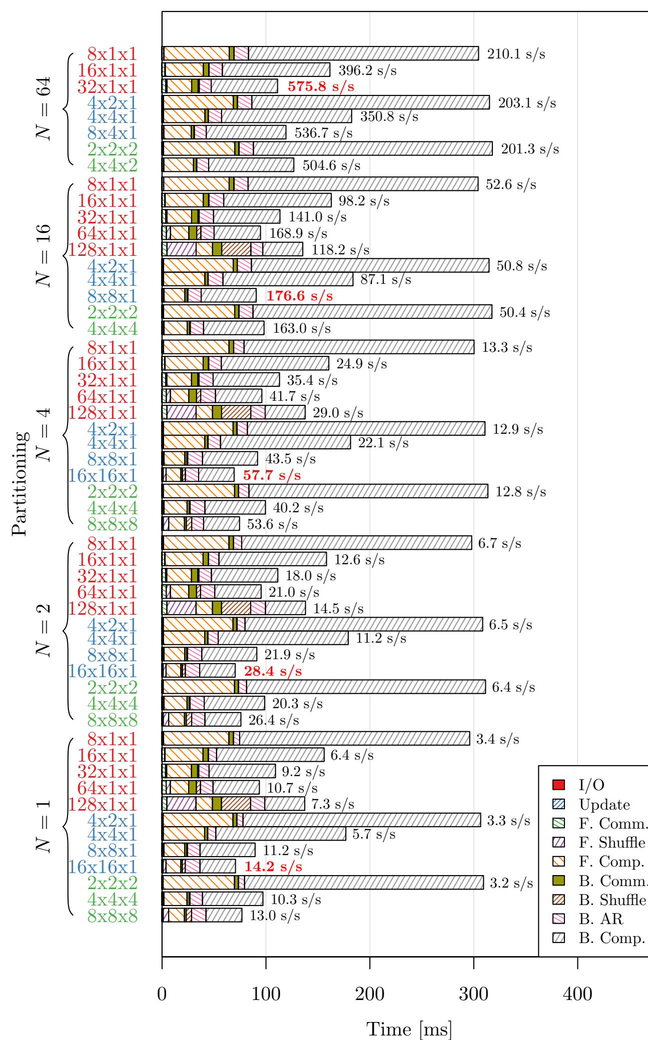


Figure 6.6: **Predicted strong scaling of the CosmoFlow network with multi-dimensional partitioning**. The figure format follows that of Figure 4.16. Labels are colored by the number of dimensions to be split. Each number in red and bold font represents the fastest partitioning in the same mini-batch size. We only show our prediction after the first epoch when I/O is hidden efficiently.

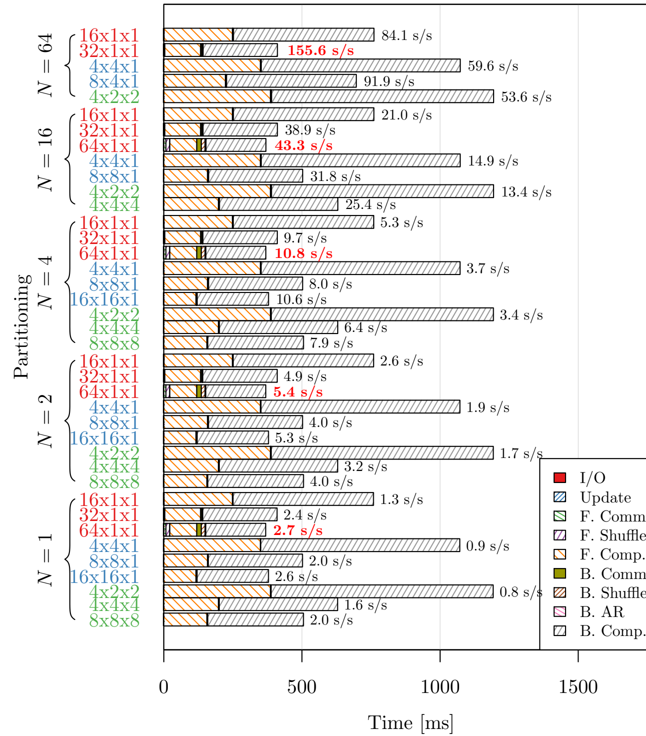


Figure 6.7: **Predicted strong scaling of the 3D U-Net with multi-dimensional partitioning**.

suitable for multi-dimensional mesh communication. In such an environment, the multi-dimensional decomposition method is considered to be the best way to perform system-wide training.

6.2.2 Implications

In this study, we have shown that hybrid parallelization enables the training of DNNs with high-resolution, high-dimensional data that could not be used before. This finding makes it possible to discover domain-specific knowledge in various fields that could not be discovered by data-parallel training alone; indeed, in Section 4.3.4, we showed that our framework improves the inference accuracy in parameter estimation of the cosmological data. By using our framework, domain scientists can not only train such models with increased spatial resolutions but also increase the number of channels per channel and the number of layers unless the per-layer memory footprint exceeds the GPU memory capacity.

As we introduced in Section 5.2, a number of studies have been done to train specific types of DNNs using hybrid parallelism. However, as explained in Section 2.2 and Section 5.2, model-parallel training is still not commonly used because it is a method that does not work well without careful consideration of the number of available processors and the characteristics of the model to determine how to distribute the model. In fact, as we have shown in Section 4.3.3, to the best of our knowledge,

our proposed framework is the most efficient among deep learning frameworks to train high-resolution 3D CNNs on a GPU cluster by a spatial partitioning. However, as we have explained in Section 2.2.3, the optimal model parallelization strategy depends on the model's architecture. For example, our spatial decomposition method is not appropriate for an MLP or LSTM network with fully-connected layers since it requires all-to-all collective communication instead of light neighbor communication. Therefore, in order to implement a general-purpose hybrid parallel training framework for a wide range of network architectures, it is necessary to integrate these methods. In this respect, LBANN supports spatial decomposition of 2D and 3D networks and 2D channel decomposition, which makes hybrid-parallel training easier than other frameworks. In addition, when training multi-modal models, such as a model that generates caption from images [144], it is expected that different parallelization methods will need to be applied to different parts of the network.

6.2.3 Future work

In this section, we discuss possible future work that extends our work in various ways, such as improving computational performance and training more complex deep learning models.

Multi-dimensional partitioning support

As we mentioned earlier, Our I/O pipeline assumes only 1D partitioning because we found in our preliminary evaluation that increasing the number of spatial partitions worsens the computational efficiency of each computation kernel and makes it difficult to obtain a good speedup (which is also implied in Figure 4.18). However, it is assumed that this strategy is not optimal for higher-dimensional data or data whose degree is higher than 3D, which requires a higher number of spatial partitions. Also, this does not support the case where the channel dimension is partitioned. Therefore, it is necessary to support such more complex parallelization methods to deal with more diverse scientific data in massively parallel environments.

Using a dataset beyond CPU memory

As we have shown in this thesis, if a large, high-resolution scientific dataset such as the CosmoFlow dataset is used for training, users need to pay close attention to the I/O performance. Benchmarks such as the ILSVRC dataset, which has been used as a benchmark for DNNs in the HPC field, are much smaller than the dataset used in this study and need to be accelerated by a qualitatively different method than simply staging or parallel I/O; in fact, Figure 4.9 uses MPI I/O and caching with CPU memory, but it does not scale at all. Our method is able to hide the I/O overhead for up to a few hundred GPUs for the CosmoFlow dataset, but if a larger dataset is used on a few computing nodes, CPU memory caching may be infeasible due to the limitations of the total memory capacity; in fact, when we ran a training job on less than 64 nodes, we encountered a runtime error due to lack of

memory. In such cases, further performance optimization is required, such as partial shuffling of the data samples [31] and using efficient methods to access PFS such as burst buffers.

More aggressive communication latency hiding

In the context of stencil computations, there are methods to hide the latency of halo communication, such as temporal blocking. In the case of CNNs, if there are no layers that require process-wide communication (such as batch-normalization layers), it is expected that such techniques can reduce the halo communication latency by communicating extra halos as in stencil computations. To the extent that we have experimented in this thesis, most of the loss in computational efficiency is due to over-decomposition; instead, we demonstrated or predicted that the training scales to thousands of GPUs by increasing data parallelism. However, as we showed in Section 6.2.1, such a latency-hiding technique may become necessary if the mini-batch size falls below the parallelism available in the entire system and excessive model parallelism needs to be introduced. In such a case, it would be necessary to modify the framework at the framework-wide level since it is not possible to implement this method only by modifying each layer as we did.

More empirical and exhaustive parallel-strategy search

In Figure 4.25, we used the fixed mini-batch size of 64 and the fixed number of computational resources to evaluate the effect of increasing the input data size on the validation accuracy. As a result, for the same mini-batch size, we get better results by increasing the input size. Since this experiment cannot be performed in a conventional data-parallel framework, this is a finding that can only be obtained by our proposed hybrid-parallel implementation for 3D CNNs.

On the other hand, similar to other studies on optimizing deep learning models and optimizers, there is room for more exhaustive optimization using ample time and computational resources. For example, since SGD is an iterative method, the step size (i.e., mini-batch size) and the number of steps to convergence (i.e., the number of epochs) are often set experimentally. While this study demonstrated that increasing the input size is clearly beneficial for the same mini-batch size, it is also possible to improve the apparent convergence speed by increasing the mini-batch size beyond that. For example, there are studies that dynamically change the mini-batch size during an experiment to reduce the training time without loss of accuracy [145, 146]. Although it is difficult to know the inference accuracy of trained models for given learning configurations such as mini-batch size, there is room for more computationally-efficient scheduling by combining experimental prediction methods [147, 148] with our framework and performance model.

Chapter 7

Conclusion

As we explained throughout this thesis, emerging demands for training huge and diverse deep learning models on state-of-the-art accelerators expose various performance issues. In order to keep up with such rapid trend changes, it is required to build a fast training infrastructure that takes advantage of the parallelism of huge models at multiple levels. In this thesis, we have proposed two methods to optimize for multiple levels of parallelism. Overall, these methods are suitable for accelerating high-resolution and high-dimensional CNNs, which are assumed to get much more attention in the future in both the machine learning and application domains. Thus, our proposed methods are expected to make a significant contribution to speeding up such training tasks.

But still, we need to adapt our new methods to emerging hardware and models. For example, NVIDIA recently announced Tesla A100 GPUs with 80 GB high-bandwidth memory, which is theoretically able to train the CosmoFlow network without batch-normalization layers with 512^3 cubes only with data-parallelism (see Figure 4.6). This does not immediately mean that hybrid-parallel training is not needed anymore because domain scientists still intend to increase the model size (such as the number of layers) to improve the accuracy of such networks, but we must carefully investigate whether new bottlenecks do not appear with such a rapid expansion of the model size, like what we have observed in this thesis (note that, as we mentioned and demonstrated in Chapter 4, 1D spatial partitioning is believed to scale when the input size is increased unless per-layer memory requirements exceed a single GPU's memory). Besides, it is also possible that the trend in hardware performance requirements may change with the evolution of deep learning models; for example, a replacement to convolutional layers that is much faster and works better in terms of prediction accuracy has been proposed [149]. Therefore, it is important to use a method that automatically predicts the optimal parallelization strategy for a given model, and our proposed loop optimization method attributed to a mathematical problem and the optimal inter-node parallel strategy selection method using our performance model contribute to such an optimization.

Appendix A

cuDNN performance on DeepBench convolutional layers

We show cuDNN performance on DeepBench convolutional layers on three different GPUs on Figure A.1, Figure A.2, and Figure A.3.

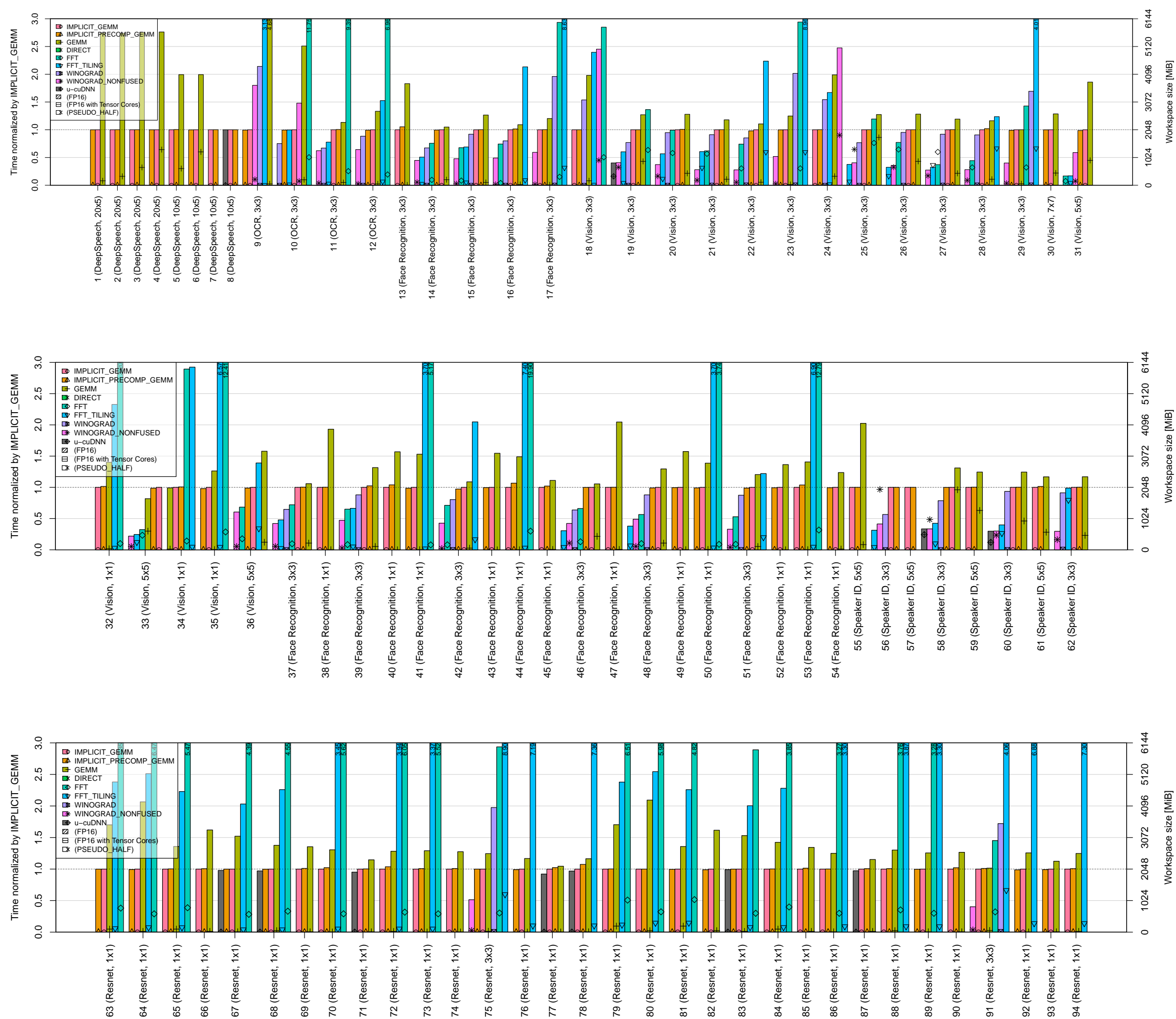


Figure A.1: **Time to compute DeepBench's convolution kernels on a K80 GPU**. We use a workspace size of 2 GiB. We use the same plot format of Figure 3.7a.

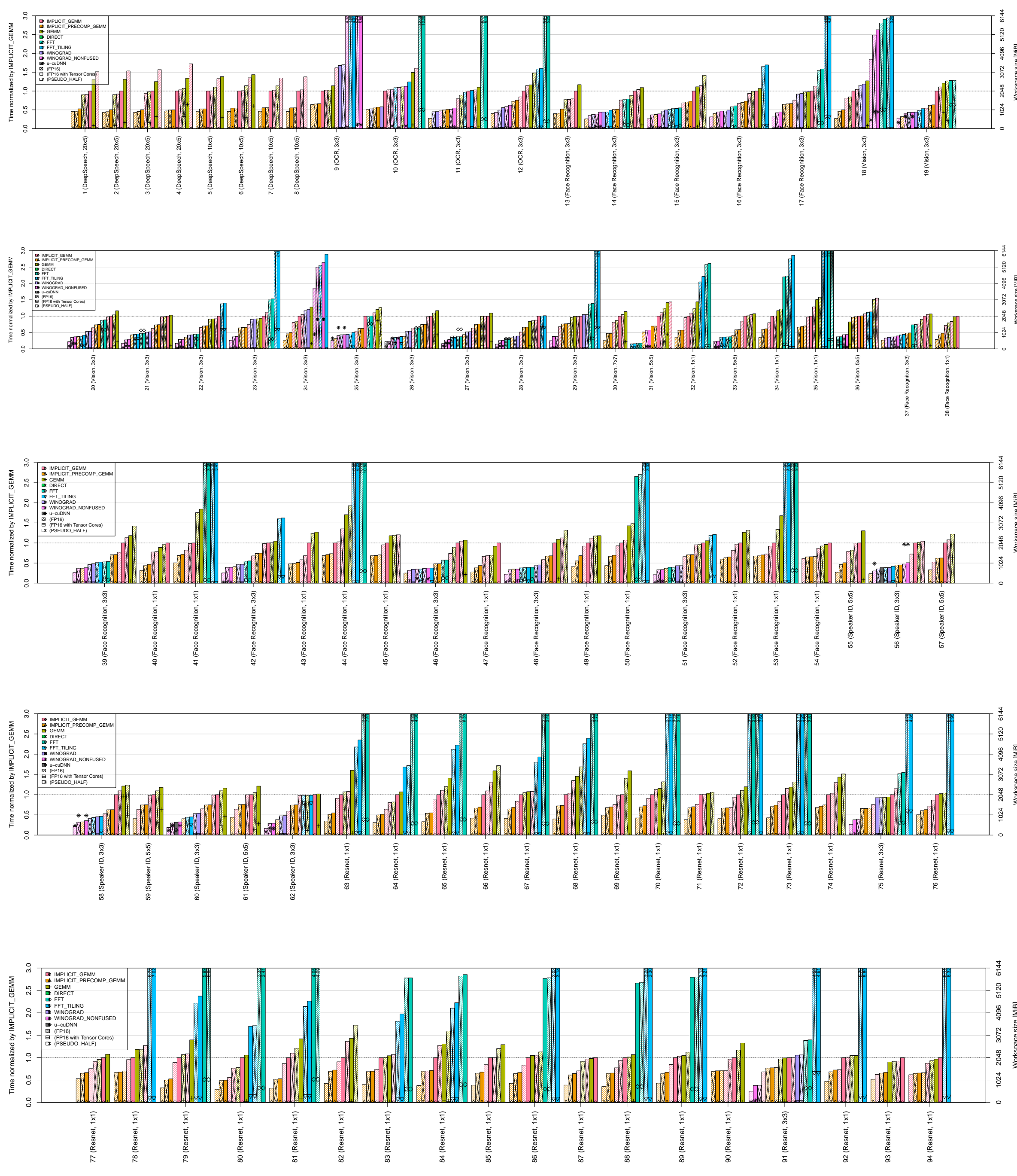


Figure A.2: **Time to compute DeepBench's convolution kernels on a P100-SXM2 GPU**. We use a workspace size of 2 GiB.

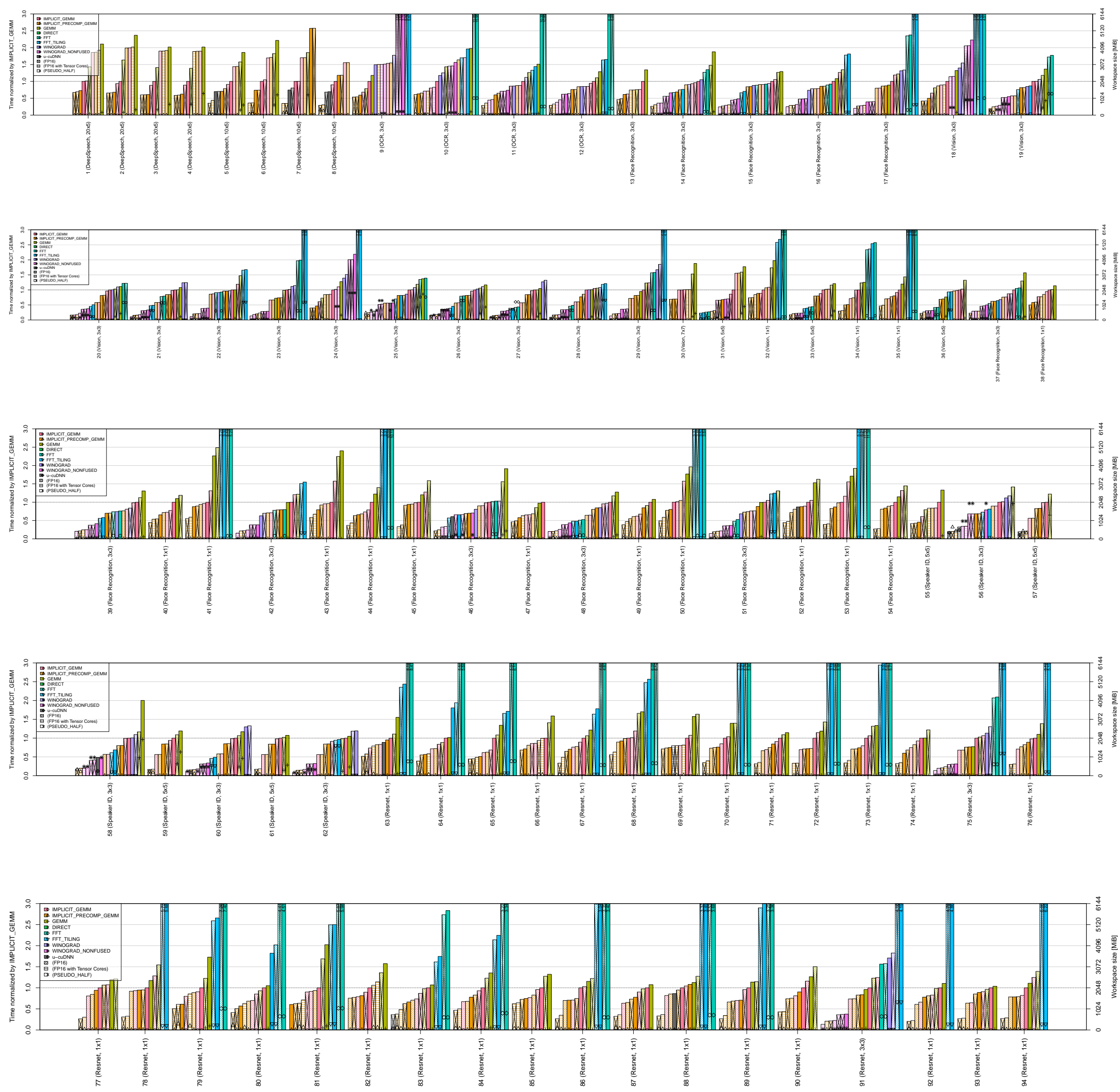


Figure A.3: **Time to compute DeepBench's convolution kernels on a V100-SXM2 GPU**. We use a workspace size of 2 GiB.

Bibliography

[1] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[2] Alex Krizhevsky. Learning Multiple Layers of Features from Tiny Images. `https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf`, 2009. Last visit: Jan 6, 2021.

[3] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[4] Google. Open Images V6. `https://storage.googleapis.com/openimages/web/index.html`, 2020. Last visit: Jan 6, 2021.

[5] Ignacio Arganda Carreras, Srinivas C. Turaga, Daniel R. Berger, Dan Cire San, Alessandro Giusti, Luca M. Gambardella, Jürgen Schmidhuber, Dmitry Laptev, Sarvesh Dwivedi, Joachim M. Buhmann, Ting Liu, Mojtaba Seyedhosseini, Tolga Tasdizen, Lee Kamentsky, Radim Burget, Vaclav Uher, Xiao Tan, Changming Sun, Tuan D. Pham, Erhan Bas, Mustafa G. Uzunbas, Albert Cardona, Johannes Schindelin, and H. Sebastian Seung. Crowdsourcing the creation of image segmentation algorithms for connectomics. *Frontiers in Neuroanatomy*, 9:1–13, 2015.

[6] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation. In *Proceedings of the Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, pages 234–241, 2015.

[7] Fisher Yu, Vladlen Koltun, and Thomas Funkhouser. Dilated residual networks. In *Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 636–644, 2017.

[8] Liang Chieh Chen, Yukun Zhu, George Papandreou, Florian Schroff, and Hartwig Adam. Encoder-decoder with atrous separable convolution for semantic image segmentation. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 833–851, 2018.

[9] Pierre Sermanet, David Eigen, Xiang Zhang, Michael Mathieu, Rob Fergus, and Yann LeCun. OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks. *arXiv e-prints*, 2013.

[10] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 580–587, 2014.

[11] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng Yang Fu, and Alexander C. Berg. SSD: Single shot multibox detector. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 21–37, 2016.

[12] Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. SuperGLUE: A stickier benchmark for general-purpose language understanding systems. In *Proceedings of the 33rd Conference on Neural Information Processing Systems (NeurIPS)*, pages 1–29, 2019.

[13] ACL 2019 Fourth Conference on Machine Translation (WMT19). Translation Task - ACL 2019 fourth Conference on Machine Translation. `http://www.statmt.org/wmt19/translation-task.html`, 2019. Last visit: Jan 6, 2021.

[14] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Y. Ng. Deep Speech: Scaling up end-to-end speech recognition. *arXiv e-prints*, pages 1–12, 2014.

[15] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, Jie Chen, Jingdong Chen, Zhijie Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Ke Ding, Niandong Du, Erich Elsen, Jesse Engel, Weiwei Fang, Linxi Fan, Christopher Fougner, Liang Gao, Caixia Gong, Awni Hannun, Tony Han, Lappi Vaino Johannes, Bing Jiang, Cai Ju, Billy Jun, Patrick LeGresley, Libby Lin, Junjie Liu, Yang Liu, Weigao Li, Xiangang Li, Dongpeng Ma, Sharan Narang, Andrew Ng, Sherjil Ozair, Yiping Peng, Ryan Prenger, Sheng Qian, Zongfeng Quan, Jonathan Raiman, Vinay Rao, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Kavya Srinet, Anuroop Sriram, Haiyuan Tang, Liliang Tang, Chong Wang, Jidong Wang, Kaifu Wang, Yi Wang, Zhijian Wang, Zhiqian Wang, Shuang Wu, Likai Wei, Bo Xiao, Wen Xie, Yan Xie, Dani Yogatama, Bin Yuan, Jun Zhan, and Zhenyao Zhu. Deep speech 2: End-to-end speech recognition in english and mandarin. In *Proceedings of the 33rd International Conference on Machine Learning (ICML)*, page 173–182, 2016.

[16] D Palaz, M Magimai.-Doss, and R Collobert. Convolutional Neural Networks-based continuous speech recognition using raw speech signal. In *Proceedings of the 2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4295–4299, 2015.

[17] Fisher Yu, Haofeng Chen, Xin Wang, Wenqi Xian, Yingying Chen, Fangchen Liu, Vashisht Madhavan, and Trevor Darrell. BDD100K: A Diverse Driving Dataset for Heterogeneous Multitask Learning. In *Proceedings of the 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2633–2642, 2020.

[18] Özgün Çiçek, Ahmed Abdulkadir, Soeren S Lienkamp, and Olaf Ronneberger. 3D U-Net: Learning Dense Volumetric Segmentation from Sparse Annotation. *arXiv e-prints*, 2016.

[19] Guodong Zeng, Xin Yang, Jing Li, Lequan Yu, Pheng-Ann Heng, and Guoyan Zheng. 3D U-net with Multi-level Deep Supervision: Fully Automatic Segmentation of Proximal Femur in 3D MR Images. In *Proceedings of the International Workshop on Machine Learning in Medical Imaging (MLMI)*, pages 274–282, 2017.

[20] Paul Blanc-Durand, Axel Van Der Gucht, Niklaus Schaefer, Emmanuel Itti, and John O Prior. Automatic lesion detection and segmentation of 18F-FET PET in gliomas: A full 3D U-Net convolutional neural network study. *PLOS ONE*, 13(4):1–11, 2018.

[21] Wei Chen, Boqiang Liu, Suting Peng, Jiawei Sun, and Xu Qiao. S3D-UNet: Separable 3D U-Net for Brain Tumor Segmentation. In *Proceedings of the International MICCAI Brainlesion Workshop (BrainLes)*, pages 358–368, 2019.

[22] Patrick Bilic, Patrick Ferdinand Christ, Eugene Vorontsov, Grzegorz Chlebus, Hao Chen, Qi Dou, Chi-Wing Fu, Xiao Han, Pheng-Ann Heng, Jürgen Hesser, Samuel Kadoury, Tomasz Konopczynski, Miao Le, Chunming Li, Xiaomeng Li, Jana Lipkovà, John Lowengrub, Hans Meine, Jan Hendrik Moltz, Chris Pal, Marie Piraud, Xiaojuan Qi, Jin Qi, Markus Rempfler, Karsten Roth, Andrea Schenk, Anjany Sekuboyina, Eugene Vorontsov, Ping Zhou, Christian Hülsemeyer, Marcel Beetz, Florian Ettlinger, Felix Gruen, Georgios Kaissis, Fabian Lohöfer, Rickmer Braren, Julian Holch, Felix Hofmann, Wieland Sommer, Volker Heinemann, Colin Jacobs, Gabriel Efrain Humpire Mamani, Bram van Ginneken, Gabriel Chartrand, An Tang, Michal Drozdzal, Avi Ben-Cohen, Eyal Klang, Marianne M. Amitai, Eli Konen, Hayit Greenspan, Johan Moreau, Alexandre Hostettler, Luc Soler, Refael Vivanti, Adi Szeskin, Naama Lev-Cohain, Jacob Sosna, Leo Joskowicz, and Bjoern H. Menze. The Liver Tumor Segmentation Benchmark (LiTS). *arXiv e-prints*, pages 1–43, 2019.

[23] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. *arXiv e-prints*, 2020.

[24] Cornell University. arXiv. `https://arxiv.org/`. Last visit: Jan 6, 2021.

[25] Tal Ben-Nun and Torsten Hoefler. Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis. *ACM Computing Surveys*, 52(4), 2019.

[26] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[27] Takuya Akiba, Shuji Suzuki, and Keisuke Fukuda. Extremely Large Minibatch SGD: Training ResNet-50 on ImageNet in 15 Minutes. *arXiv e-prints*, pages 1–4, 2017.

[28] 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. *arXiv e-prints*, 2018.

[29] Masafumi Yamazaki, Akihiko Kasagi, Akihiro Tabuchi, Takumi Honda, Masahiro Miwa, Naoto Fukumoto, Tsuguchika Tabaru, Atsushi Ike, and Kohta Nakashima. Yet Another Accelerated SGD: ResNet-50 Training on ImageNet in 74.7 seconds. *arXiv e-prints*, 2019.

[30] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. ImageNet Training in Minutes. *arXiv e-prints*, pages 1–11, 2017.

[31] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *arXiv e-prints*, pages 1–12, 2017.

[32] PyTorch core team. PyTorch. `http://pytorch.org/`, 2017. Last visit: Jan 6, 2021.

[33] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. *arXiv e-prints*, pages 1–19, 2015.

[34] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a Next-Generation Open Source Framework for Deep Learning. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The 29th International Conference on Neural Information Processing Systems (NIPS)*, pages 1–6, 2015.

[35] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning. *arXiv e-prints*, pages 1–9, 2014.

[36] NVIDIA. NVIDIA Collective Communications Library (NCCL). `https://developer.nvidia.com/nccl`, 2017. Last visit: Jan 6, 2021.

[37] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.

[38] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv e-prints*, 2014.

[39] Facebook. Caffe2. `https://caffe2.ai/`, 2017. Last visit: Jan 6, 2021.

[40] NVIDIA. NVIDIA Caffe. `https://github.com/NVIDIA/caffe`, 2017. Last visit: Jan 6, 2021.

[41] Ronan Collobert, Samy Bengio, and Johnny Marithoz. Torch: A Modular Machine Learning Software Library, 2002.

[42] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *arXiv e-prints*, pages 1–6, 2015.

[43] Frank Seide and Amit Agarwal. CNTK: Microsoft's Open-Source Deep-Learning Toolkit. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, page 2135, 2016.

[44] Brian Van Essen, Hyojin Kim, Roger Pearce, Kofi Boakye, and Barry Chen. LBANN: Livermore Big Artificial Neural Network HPC Toolkit. In *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments (MLHPC)*, pages 1–6, 2015.

[45] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems (NIPS)*, page 1097–1105, 2012.

[46] Yosuke Oyama. *Automatic and Adaptive Optimization of Convolution Workspace and Communication Precision for Deep Learning.* Master's thesis, Tokyo Institute of Technology, 2018.

[47] 大山 洋介, 野村 哲弘, 佐藤 育郎, and 松岡 聡. ディープラーニングのデータ並列学習における少精度浮動小数点数を用いた通信量の削減. *IPSJ SIG Technical Report*, 2017-HPC-158(30):1–10, 2017.

[48] Yosuke Oyama, Tal Ben-Nun, Torsten Hoefler, and Satoshi Matsuoka. Less is More: Accelerating Deep Neural Networks with Micro-Batching. *IPSJ SIG Technical Report*, 2017-HPC-162(22):1–9, 2017.

[49] NVIDIA. NVIDIA TESLA V100 GPU ARCHITECTURE. `https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`. Last visit: Jan 6, 2021.

[50] Satoshi Matsuoka, Hideharu Amano, Kengo Nakajima, Koji Inoue, Tomohiro Kudoh, Naoya Maruyama, Kenjiro Taura, Takeshi Iwashita, Takahiro Katagiri, Toshihiro Hanawa, and Toshio Endo. From FLOPS to BYTES: Disruptive change in high-performance computing towards the post-moore era. In *Proceedings of the 2016 ACM International Conference on Computing Frontiers (CF)*, pages 274–281, 2016.

[51] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, and Andrew Y Ng. Large Scale Distributed Deep Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems (NIPS)*, pages 1223–1231, 2012.

[52] Adam Coates, Brody Huval, Tao Wang, David Wu, and Andrew Y Ng. Deep learning with COTS HPC systems. In *Proceedings of the 30th International Conference on Machine Learning (ICML)*, pages 1337–1345, 2013.

[53] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project Adam : Building an Efficient and Scalable Deep Learning Training System. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 571–582, 2014.

[54] NVIDIA. Tesla K20 GPU Active Accelerator. `https://www.nvidia.com/content/PDF/kepler/tesla-k20-active-bd-06499-001-v03.pdf`. Last visit: Jan 6, 2021.

[55] NVIDIA. Tesla K80 GPU Accelerator. `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/Tesla-K80-BoardSpec-07317-001-v05.pdf`. Last visit: Jan 6, 2021.

[56] NVIDIA. Tesla P100 Most Advanced Data Center Accelerator. `http://www.nvidia.com/object/tesla-p100.html`. Last visit: Jan 6, 2021.

[57] NVIDIA. NVIDIA A100 Tensor Core GPU Architecture. `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf`. Last visit: Jan 6, 2021.

[58] Baidu Research. DeepBench. `https://github.com/baidu-research/DeepBench`, 2016. Last visit: Jan 6, 2021.

[59] Amrita Mathuriya, Deborah Bard, Peter Mendygral, Lawrence Meadows, James Arnemann, Lei Shao, Siyu He, Tuomas Karna, Daina Moise, Simon J. Pennycook, Kristyn Maschoff, Jason Sewall, Nalini Kumar, Shirley Ho, Mike Ringenburg, Prabhat, and Victor Lee. CosmoFlow: Using Deep Learning to Learn the Universe at Scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC18)*, 2018.

[60] Vinod Nair and Geoffrey E Hinton. Rectified Linear Units Improve Restricted Boltzmann Machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML)*, pages 807–814, 2010.

[61] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1026–1034, 2015.

[62] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier Nonlinearities Improve Neural Network Acoustic Models. In *Proceedings of the ICML Workshop on Deep Learning for Audio, Speech and Language Processing*, 2013.

[63] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv e-prints*, pages 1–14, 2014.

[64] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015.

[65] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 171–180, 2016.

[66] Gao Huang, Zhuang Liu, Kilian Q. Weinberger, and Laurens van der Maaten. Densely Connected Convolutional Networks. *arXiv e-prints*, pages 1–12, 2016.

[67] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning Long-Term Dependencies with Gradient Descent is Difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

[68] Min Lin, Qiang Chen, and Shuicheng Yan. Network In Network. In *Proceedings of the International Conference on Learning Representations (ICLR)*, pages 1–10, 2014.

[69] Netlib. BLAS (Basic Linear Algebra Subprograms). `http://www.netlib.org/blas/`, 2017. Last visit: Jan 6, 2021.

[70] Shmuel Winograd. Arithmetic Complexity of Computations, 1980.

[71] Andrew Lavin and Scott Gray. Fast Algorithms for Convolutional Neural Networks. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4013–4021, 2016.

[72] Michaël Mathieu, Mikael Henaff, and Yann Lecun. Fast Training of Convolutional Networks through FFTs. In *Proceedings of the International Conference on Learning Representations (ICLR)*, pages 1–9, 2014.

[73] Nicolas Vasilache, Jeff Johnson, Michael Mathieu, Soumith Chintala, Serkan Piantino, and Yann LeCun. Fast Convolutional Nets With fbfft: A GPU Performance Evaluation. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.

[74] Aleksandar Zlateski, Kisuk Lee, and H. Sebastian Seung. ZNN - A Fast and Scalable Algorithm for Training 3D Convolutional Networks on Multi-core and Many-Core Shared Memory Machines. In *Proceedings of the 2016 IEEE 30th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 801–811, 2016.

[75] Aleksandar Zlateski, Kisuk Lee, and H. Sebastian Seung. ZNNi: Maximizing the Inference Throughput of 3D Convolutional Networks on CPUs and GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC16)*, pages 73:1–73:12, 2016.

[76] Aleksandar Zlateski, Zhen Jia, Kai Li, and Fredo Durand. FFT Convolutions are Faster than Winograd on Modern CPUs, Here is Why. *arXiv e-prints*, 2018.

[77] John Duchi, Elad Hazan, and Yoram Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2011.

[78] Matthew D. Zeiler. ADADELTA: An Adaptive Learning Rate Method. *arXiv e-prints*, 2012.

[79] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *Proceedings of the 3rd International Conference for Learning Representations (ICLR)*, pages 1–15, 2015.

[80] NVIDIA. CUDA Tooklit. `https://developer.nvidia.com/cuda-toolkit`. Last visit: Jan 6, 2021.

[81] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. VDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, 2016.

[82] Yuki Ito, Ryo Matsumiya, and Toshio Endo. ooc-cuDNN: Accommodating convolutional neural networks over GPU memory capacity. In *Proceedings of the 2017 IEEE International Conference on Big Data (Big Data)*, pages 183–192, 2017.

[83] Mohamed Wahib, Haoyu Zhang, Truong Thao Nguyen, Aleksandr Drozd, Jens Domke, Lingqi Zhang, Ryousei Takano, and Satoshi Matsuoka. Scaling Distributed Deep Learning Workloads beyond the Memory Capacity with KARMA. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC20)*, 2020.

[84] Audrūnas Gruslys, Remi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. Memory-Efficient Backpropagation Through Time. In *Proceedings of the 29th International Conference on Neural Information Processing Systems (NIPS)*, pages 4125–4133, 2016.

[85] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. SuperNeurons: Dynamic GPU Memory Management for Training Deep Neural Networks. *ACM SIGPLAN Notices*, 53(1):41–53, 2018.

[86] Koichi Shirahata, Yasumoto Tomita, and Atsushi Ike. Memory Reduction Method for Deep Neural Network Training. In *Proceedings of the 2016 IEEE 26th International Workshop on Machine Learning for Signal Processing (MLSP)*, 2016.

[87] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. In *Proceedings of the International Conference on Learning Representations (ICLR)*, pages 1–16, 2016.

[88] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv e-prints*, pages 1–7, 2014.

[89] Yasunorikudo. chainer-ResNet. `https://github.com/yasunorikudo/chainer-ResNet`, 2017. Last visit: Jan 6, 2021.

[90] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[91] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS)*, pages 5999–6009, 2017.

[92] Barret Zoph and Quoc V. Le. Neural Architecture Search with Reinforcement Learning. *arXiv e-prints*, pages 1–16, 2016.

[93] NVIDIA. cuBLAS. `https://developer.nvidia.com/cublas`. Last visit: Jan 6, 2021.

[94] Zhang Xianyi. OpenBLAS. `https://www.openblas.net/`. Last visit: Jan 6, 2021.

[95] Intel. Intel®Math Kernel Library. `https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html`. Last visit: Jan 6, 2021.

[96] oneAPI-SRC. oneAPI Deep Neural Network Library (oneDNN). `https://github.com/oneapi-src/oneDNN`. Last visit: Jan 6, 2021.

[97] OpenMP. OpenMP. `https://www.openmp.org/`. Last visit: Jan 6, 2021.

[98] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of Collective Communication Operations in MPICH. *International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[99] Forrest N. Iandola, Khalid Ashraf, Matthew W. Moskewicz, and Kurt Keutzer. FireCaffe: near-linear acceleration of deep neural network training on compute clusters. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–13, 2015.

[100] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv e-prints*, 2018.

[101] Naoya Maruyama, Tatsuo Nomura, Kento Sato, and Satoshi Matsuoka. Physis: An implicitly parallel programming model for stencil computations on large-scale gpu-accelerated supercomputers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC11)*, 2011.

[102] Feng Yan, Olatunji Ruwase, Yuxiong He, and Trishul Chilimbi. Performance Modeling and Scalability Optimization of Distributed Deep Learning Systems. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 1355–1364, 2015.

[103] Amir Gholami, Ariful Azad, Peter Jin, Kurt Keutzer, and Aydın Buluç. Integrated Model, Batch, and Domain Parallelism in Training Neural Networks. In *Proceedings of the 30th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 77–86, 2018.

[104] Nikoli Dryden, Naoya Maruyama, Tom Benson, Tim Moon, Marc Snir, and Brian Van Essen. Improving strong-scaling of CNN training by exploiting finer-grained parallelism. In *Proceedings of the 2019 IEEE 33rd International Parallel and Distributed Processing Symposium (IPDPS)*, pages 210–220, 2019.

[105] NVIDIA. NVIDIA CUDNN DOCUMENTATION. `https://docs.nvidia.com/deeplearning/cudnn/api/index.html`, 2020. Last visit: Jan 6, 2021.

[106] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debojyoti Dutta, Udit Gupta, Kim Hazelwood, Andrew Hock, Xinyuan Huang, Atsushi Ike, Bill Jia, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Guokai Ma, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St. John, Tsuguchika Tabaru, Carole-Jean Wu, Lingjie Xu, Masafumi Yamazaki, Cliff Young, and Matei Zaharia. MLPerf Training Benchmark. *arXiv e-prints*, 2019.

[107] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. MLPerf Inference Benchmark. In *Proceedings of the 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 446–459, 2020.

[108] Tal Ben-Nun, Maciej Besta, Simon Huber, Alexandros Nikolaos Ziogas, Daniel Peter, and Torsten Hoefler. A modular benchmarking infrastructure for high-performance and reproducible deep learning. In *Proceedings of the 2019 IEEE 33rd International Parallel and Distributed Processing Symposium (IPDPS)*, pages 66–77, 2019.

[109] The TensorFlow Authors. tensorflow/models. `https://github.com/tensorflow/models`. Last visit: Jan 6, 2021.

[110] The TensorFlow Authors. tensorflow/benchmarks. `https://github.com/tensorflow/benchmarks`. Last visit: Jan 6, 2021.

[111] Zhuang Liu. Densely Connected Convolutional Network (DenseNet). `https://github.com/liuzhuang13/DenseNetCaffe`, 2016. Last visit: Jan 6, 2021.

[112] Andrew Makhorin. GLPK (GNU Linear Programming Kit). `https://www.gnu.org/software/glpk/`. Last visit: Jan 6, 2021.

[113] Yosuke Oyama, Akihiro Nomura, Ikuro Sato, Hiroki Nishimura, Yukimasa Tamatsu, and Satoshi Matsuoka. Predicting Statistics of Asynchronous SGD Parameters for a Large-Scale Distributed

Deep Learning System on GPU Supercomputers. In *Proceedings of the 2016 IEEE International Conference on Big Data (Big Data)*, pages 66–75, 2016.

[114] Dan C. Cireşan, Alessandro Giusti, Luca M. Gambardella, and Jürgen Schmidhuber. Deep Neural Networks Segment Neuronal Membranes in Electron Microscopy Images. In *Proceedings of the 25th International Conference on Neural Information Processing Systems (NIPS)*, pages 2843–2851, 2012.

[115] Thorsten Kurth, Jian Zhang, Nadathur Satish, Ioannis Mitliagkas, Evan Racah, Mostofa Ali Patwary, Tareq Malas, Narayanan Sundaram, Wahid Bhimji, Mikhail Smorkalov, Jack Deslippe, Mikhail Shiryaev, Srinivas Sridharan, Prabhat, and Pradeep Dubey. Deep Learning at 15PF: Supervised and Semi-Supervised Classification for Scientific Data. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC17)*, 2017.

[116] Thorsten Kurth, Sean Treichler, Joshua Romero, Mayur Mudigonda, Nathan Luehr, Everett Phillips, Ankur Mahesh, Michael Matheson, Jack Deslippe, Massimiliano Fatica, Prabhat, and Michael Houston. Exascale Deep Learning for Climate Analytics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2018.

[117] Robert M. Patton, J. Travis Johnston, Steven R. Young, Catherine D. Schuman, Don D. March, Thomas E. Potok, Derek C. Rose, Seung Hwan Lim, Thomas P. Karnowski, Maxim A. Ziatdinov, and Sergei V. Kalinin. 167-PFlops deep learning for electron microscopy: From learning physics to atomic manipulation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC18)*, pages 638–648, 2018.

[118] Siamak Ravanbakhsh, Junier Oliva, Sebastien Fromenteau, Layne C. Price, Shirley Ho, Jeff Schneider, and Barnabas Poczos. Estimating Cosmological Parameters from the Dark Matter Distribution. In *Proceedings of the 33rd International Conference on Machine Learning (ICML)*, pages 2407–2416, 2016.

[119] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv e-prints*, pages 1–11, 2015.

[120] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[121] National Energy Research Scientific Computing Center. CosmoFlow datasets. `https://portal.nersc.gov/project/m3363`, 2019. Last visit: Jan 6, 2021.

[122] Nikoli Dryden, Tim Moon, Tom Benson, and Marc Snir. Channel and Filter Parallelism for Large-Scale CNN Training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC19)*, 2019.

[123] Lawrence Livermore National Laboratory. Conduit. https://github.com/LLNL/conduit, 2014. Last visit: Jan 6, 2021.

[124] Jiri Kraus. An Introduction to CUDA-Aware MPI. https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/, 2013. Last visit: Jan 6, 2021.

[125] Nikoli Dryden, Naoya Maruyama, Tim Moon, Tom Benson, Andy Yoo, Marc Snir, and Brian Van Essen. Aluminum : An Asynchronous , GPU-Aware Communication Library Optimized for Large-Scale Training of Deep Neural Networks on HPC Systems. In *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments (MLHPC)*, 2018.

[126] Lawrence Livermore National Laboratory. Hydrogen. https://github.com/llnl/Elemental, 2010. Last visit: Jan 6, 2021.

[127] The HDF5 Group. The hdf5®library & file format. https://www.hdfgroup.org/solutions/hdf5/, 2006. Last visit: Jan 6, 2021.

[128] Fahim Chowdhury, Yue Zhu, Todd Heer, Saul Paredes, Adam Moody, Robin Goldstone, Kathryn Mohror, and Weikuan Yu. I/O characterization and performance evaluation of BeeGFS for deep learning. *ACM International Conference Proceeding Series*, 2019.

[129] Lawrence Livermore National Laboratory. Using LC's Sierra Systems. https://hpc.llnl.gov/training/tutorials/using-lcs-sierra-system, 2020. Last visit: Jan 6, 2021.

[130] NVIDIA. Profiler user's guide. https://docs.nvidia.com/cuda/profiler-users-guide/, 2020. Last visit: Jan 6, 2021.

[131] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond Data and Model Parallelism for Deep Neural Networks. *arXiv e-prints*, 2018.

[132] Yanping Huang, Yonglong Cheng, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, and Zhifeng Chen. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. *arXiv e-prints*, 2018.

[133] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. Mesh-TensorFlow: Deep Learning for Supercomputers. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS)*, pages 10414–10423, 2018.

[134] Norm Jouppi. Google supercharges machine learning tasks with TPU custom chip. https://cloud.google.com/blog/products/gcp/google-supercharges-machine-learning-tasks-with-custom-chip, 2016. Last visit: Jan 6, 2021.

[135] TensorFlow. Tfrecord and tf.train.example. https://www.tensorflow.org/tutorials/load_data/tfrecord, 2020. Last visit: Jan 6, 2021.

[136] Chao Li, Yi Yang, Min Feng, Srimat Chakradhar, and Huiyang Zhou. Optimizing Memory Efficiency for Deep Convolutional Neural Networks on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC16)*, pages 633–644, 2016.

[137] Le Hou, Youlong Cheng, Noam Shazeer, Niki Parmar, Yeqing Li, Panagiotis Korfiatis, Travis M. Drucker, Daniel J. Blezek, and Xiaodan Song. High Resolution Medical Image Analysis with Spatial Partitioning. *arXiv e-prints*, pages 15–19, 2019.

[138] Mark Harris. Unified memory for cuda beginners. https://developer.nvidia.com/blog/unified-memory-cuda-beginners/, 2017. Last visit: Jan 6, 2021.

[139] Loyd Case. Volta Tensor Core GPU Achieves New AI Performance Milestones. https://developer.nvidia.com/blog/tensor-core-ai-performance-milestones/, 2018. Last visit: Jan 6, 2021.

[140] The Linux Foundation. Open Neural Network Exchange. https://onnx.ai/, 2019. Last visit: Jan 6, 2021.

[141] NVIDIA. cuBLAS CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/cublas/index.html, 2020. Last visit: Jan 6, 2021.

[142] Lawrence Livermore National Laboratory. Sierra. https://computing.llnl.gov/computers/sierra, 2020. Last visit: Jan 6, 2021.

[143] FUJITSU. Specifications - Supercomputer Fugaku. https://www.fujitsu.com/global/about/innovation/fugaku/specifications/, 2020. Last visit: Jan 6, 2021.

[144] M D Zakir Hossain, Ferdous Sohel, Mohd Fairuz Shiratuddin, and Hamid Laga. A Comprehensive Survey of Deep Learning for Image Captioning. *ACM Computing Surveys*, 51(6), 2019.

[145] Samuel L. Smith, Pieter-Jan Jan Kindermans, Chris Ying, and Quoc V. Le. Don't Decay the Learning Rate, Increase the Batch Size. In *Proceedings of the 3rd International Conference for Learning Representations (ICLR)*, 2018.

[146] Hiroaki Mikami, Hisahiro Suganuma, Pongsakorn U-chupala, Yoshiki Tanaka, and Yuichi Kageyama. ImageNet/ResNet-50 Training in 224 Seconds. *arXiv e-prints*, 2018.

[147] Aaron Klein, Stefan Falkner, Jost Tobias Springenberg, and Frank Hutter. Learning Curve Prediction With Bayesian Neural Networks. In *Proceedings of the 3rd International Conference for Learning Representations (ICLR)*, 2017.

[148] Bowen Baker, Otkrist Gupta, Ramesh Raskar, and Nikhil Naik. Accelerating neural architecture search using performance prediction. *arXiv e-prints*, 2, 2017.

[149] Yunpeng Chen, Haoqi Fan, Bing Xu, Zhicheng Yan, Yannis Kalantidis, Marcus Rohrbach, Yan Shuicheng, and Jiashi Feng. Drop an octave: Reducing spatial redundancy in convolutional neural networks with octave convolution. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pages 3434–3443, 2019.

Created at: 2021/02/27 20:51:21
ID: bd715c24112f0f0d6c4fa2d145eb46e4b34280e1 (tag: t2r2)