AUTOMATED DISCOVERY OF MACHINE LEARNING OPTIMIZATIONS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Zhihao Jia
August 2020

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Alex Aiken) Principal Co-Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Matei Zaharia) Principal Co-Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Kunle Olukuton)

Approved for the Stanford University Committee on Graduate Studies

Abstract

The increasing complexity of machine learning (ML) models and ML-specific hardware architectures makes it increasingly challenging to build efficient and scalable ML systems. Today's ML systems heavily rely on human effort to optimize the deployment of ML models on modern hardware platforms, which requires a tremendous amount of engineering effort but only provides suboptimal runtime performance. Moreover, the rapid evolution of ML models and ML-specific hardware makes it infeasible to manually optimize performance for all model and hardware combinations.

In this dissertation, we propose a search-based methodology to build performant ML systems by automatically discovering performance optimizations for ML computations. Instead of only considering the limited set of manually designed performance optimizations in current ML systems, our approach introduces a significantly more comprehensive search space of possible strategies to optimize the deployment of an ML model on a hardware platform. In addition, we design efficient search algorithms to explore the search space and discover highly-optimized strategies. The search is guided by a cost model for evaluating the performance of different strategies. We also propose a number of techniques to accelerate the search procedure by leveraging the topology of the search space.

This dissertation presents three ML systems that apply this methodology to optimize different tasks in ML deployment. Compared to current ML systems relying on manually designed optimizations, our ML systems enable better runtime performance by automatically discovering novel performance optimizations that are missing in current ML systems. Moreover, the performance improvement is achieved with less engineering effort, since the code needed for discovering these optimizations is much less than manual implementation of these optimizations.

First, we developed TASO, the first ML graph optimizer that automatically generates graph optimizations. TASO formally verifies the correctness of the generated graph optimizations using an automated theorem prover, and uses cost-based backtracking search to discover how to apply the verified optimizations. In addition to improving runtime performance and reducing engineering effort, TASO also provides correctness guarantees using formal methods.

Second, to generalize and go beyond today's manually designed parallelization strategies for distributed ML computations, we introduce the SOAP search space, which contains a comprehensive set of possible strategies to parallelize ML computations by identifying parallelization opportunities across different Samples, Operators, Attributes, and Parameters. We developed FlexFlow, a deep learning engine that automatically searches over strategies in the SOAP search space. FlexFlow includes a novel execution simulator to evaluate the runtime performance of different strategies, and uses a Markov Chain Monte Carlo (MCMC) search algorithm to find performant strategies. FlexFlow discovers strategies that significantly outperform existing strategies, while requiring no manual effort during the search procedure.

Finally, we developed Roc, which automates data placement optimizations and minimizes data transfers in the memory hierarchy for large-scale graph neural network (GNN) computations. Roc formulates the task of optimizing data placement as a cost minimization problem and uses a dynamic programming algorithm to discover a globally optimal data management plan that minimizes data transfers between memories.

Acknowledgments

It is hard to believe that it finally comes to the end of my PhD study in the Computer Science Department at Stanford University. I would like to take this opportunity to acknowledge the many people whose help and support makes both this moment possible and this period of my life transformative.

First and foremost, I would like to thank my PhD advisors, Alex Aiken and Matei Zaharia, for their support and guidance over the years and for granting me the flexibility and space to work on the problems that seem most interesting and attractive to me. I started my PhD journey working with Alex on high-performance parallel computing and later on decided to switch my research direction to building efficient and scalable systems for machine learning at the end of my third year. Designing ML systems was a completely new research domain to both Alex and me at that time, but Alex had been extremely encouraging and supportive along the way. Because of the support and guidance from Alex, I experienced an incredibly smooth transition and quickly came up with new ideas that constitute this dissertation.

I also owe a big thanks to Matei for guiding me on researching complex real-world systems. It is always rewarding and a great pleasure to have individual meetings with Matei. By observing how he thought about and approached research problems, it helped me to learn how to extract fundamental research problems from sophisticated real-world systems.

Kunle Olukuton deserves mention too, but not just for reading the drafts of this dissertation and providing thoughtful and invaluable feedback. Kunle encouraged me to think creatively and actively look for ways to extend and generalize the techniques we invented in this dissertation to other domains.

This dissertation would have been impossible without the Legion project, which sets the foundation for the work presented in this dissertation. The fantastic Legion team members, Mike Bauer, Sean Treichler, Elliott Slaughter, Wonchan Lee, Manolis Papadakis, Karthik Murthy, Todd Warszawski, and the others, have always been available and willing to provide help when I need any assistance. Without them crafting and perfecting the Legion programming system, I would have to take much longer time to finish this dissertation and can only achieve much less. I am fortunate to be a part of the Legion team and am glad that I was able to make contributions to Legion.

During my undergraduate and graduate studies, I have done several unforgettable internships at Microsoft Research Asia (MSRA), Microsoft Research Silicon Valley (MSRSCV), and Los Alamos National Lab (LANL). First, I want to acknowledge Lidong Zhou, who hosted my undergraduate internship at MSRA and guided my first research project on computer systems. Thank you to Lidong for leading me into the palace of computer science research. Second, I want to thank Yuan Yu, who mentored my first graduate internship at MSRSCV and provided me valuable guidance on discovering research projects that solve real-world problems and bring real-world impacts. Finally, I would like to thank Galen Shipman and Pat McCormick, who jointly mentored my summer internship at LANL in the summer of 2016. It was an absolutely memorable experience and was also the start of my journey on researching ML systems. A significant portion of this dissertation is motivated and based on the work I did during this internship.

A special thanks to my undergraduate advisor, Andrew Chi-Chih Yao, for founding the Special Pilot Computer Science Class at Tsinghua University, which hosted my undergraduate study. Thank you to Andrew for sparking and sustaining my interests in computer science. I am fortunate to have received tremendous help and guidance from other Yao class members in my undergraduate and graduate studies. I hope that I will have the privilege to be helpful to future generations of the Yao class students.

Saving the best for last, I would like to thank my family for keeping supporting and believing in me throughout my PhD. Thank you to my parents, Shuze Geng and Leiming Jia, for installing in me a love of math and science and for no setting any expectations on what I should be when I grow up. Thank you for teaching me the art of cooking. Cooking good meals guided me through the ups and downs in my graduate schools.

Finally, I would like to acknowledge my wife, Sina Lin, for her endless love and support. Thank you for being my best friend, my partner, and my compass when I got lost along my PhD journey. Thank you for laughing with me, cooking with me, staying with me, and even discussing new research ideas with me. I am so lucky to live a life with you, and am looking forward to our future adventures together.

To all my family, friends, mentors, colleagues, and collaborators: thank you for being a part of my life!

Publications

Portions of this thesis appeared in the following publications, in collaboration with Alex Aiken, Mattan Erez, Mingyu Gao, Yongkee Kwon, Sina Lin, Pat McCormick, Oded Padon, Charles R. Qi, Galen Shipman, James Thomas, Todd Warszawski, and Matei Zaharia.

1. Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with Roc. In *Proceedings of the Conference on Machine Learning and Systems (MLSys)*, Austin, TX, March 2020.

2. Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. TASO: Optimizing Deep Learning Computation with Automated Generation of Graph Substitutions. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, October 2019.

3. Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond Data and Model Parallelism for Deep Neural Networks. In *Proceedings of the Conference on Systems and Machine Learning (SysML)*, Palo Alto, CA, April 2019.

4. Zhihao Jia, James Thomas, Todd Warszawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. Optimizing DNN Computation with Relaxed Graph Substitutions. In *Proceedings of the Conference on Systems and Machine Learning (SysML)*, Palo Alto, CA, April 2019.

5. Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. A Distributed Multi-GPU System for Fast Graph Processing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Rio, Brazil, August 2018.

6. Zhihao Jia, Sina Lin, Charles R. Qi, and Alex Aiken. Exploring Hidden Dimensions in Parallelizing Convolutional Neural Networks. In *Proceedings of the International Conference on Machine Learning (ICML)*, Stockholm, Sweden, July 2018.

Contents

Abstract **iv**

Acknowledgments **vi**

Publications **viii**

1 Introduction **1**

1.1 Machine Learning Systems . . . 2

1.2 Challenges of Building ML Systems . . . 3

1.3 Approach: Automated Discovery of ML Optimizations . . . 4

1.4 Discussion . . . 6

1.5 Roadmap . . . 6

2 Automated Discovery of Graph Optimizations **7**

2.1 Motivation . . . 7

2.2 The Approach . . . 9

2.3 Graph Substitution Generator . . . 10

2.3.1 Graph Substitution Definition . . . 11

2.3.2 Generation Algorithm . . . 12

2.4 Graph Substitution Verifier . . . 14

2.5 Pruning Redundant Substitutions . . . 18

2.6 Cost Model . . . 20

2.7 Joint Optimizer . . . 20

2.8 Implementation . . . 22

2.9 Evaluation . . . 23

2.9.1 Experimental Setup . . . 23

2.9.2 End-to-End Evaluation . . . 24

2.9.3 Substitution Case Study . . . 25

2.9.4 Analysis of Used Substitutions . . . 29

2.9.5 Joint Optimization of Graph Substitutions and Data Layout 29
2.9.6 Graph Substitution Verifier 30
2.10 Related Work 31
2.11 Limitations 32
2.12 Conclusion 32

3 Automated Discovery of Parallelization Optimizations 33
3.1 Motivation 33
3.2 The Approach 35
3.3 The SOAP Search Space 36
3.4 Execution Simulator 38
3.4.1 Task Graph 38
3.4.2 Full Simulation Algorithm 41
3.4.3 Delta Simulation Algorithm 41
3.5 Execution Optimizer 41
3.5.1 MCMC Sampling 42
3.5.2 Search Algorithm 42
3.6 FlexFlow Runtime 42
3.7 Evaluation 43
3.7.1 Experimental Setup 43
3.7.2 Parallelization Performance 45
3.7.3 DLRM and Candle Uno 49
3.7.4 Execution Simulator 51
3.7.5 Search Algorithm 53
3.7.6 Case Studies 53
3.8 Related Work 54
3.9 Limitations 56
3.10 Conclusion 56

4 Automated Discovery of Data Placement Optimizations 57
4.1 Motivation 57
4.2 Background and Related Work 60
4.2.1 Graph Neural Networks 60
4.2.2 Related Work 60
4.3 The approach 62
4.4 Graph Partitioner 63
4.4.1 Cost Model 63
4.4.2 Partitioning Algorithm 65

4.5 Memory Manager 66
4.6 Implementation 68
4.7 Evaluation 68
4.7.1 Experimental Setup 68
4.7.2 Single-GPU Results 70
4.7.3 Multi-GPU Results 70
4.7.4 Comparison with Graph Sampling 71
4.7.5 Deeper and Larger GNN Architectures 72
4.7.6 Graph Partitioning 73
4.7.7 Memory Management 75
4.8 Conclusion 75

5 Conclusions **76**

List of Tables

1.1 A search-based methodology for automatically discovering ML optimizations. 4

2.1 Tensor operators and constant tensors included in TASO. Similar to existing DNN frameworks [11, 70], pooling and convolution operators support different strides and padding modes (i.e., $\mathtt{P}_{\mathtt{same}}$ and $\mathtt{P}_{\mathtt{valid}}$); convolution supports different activation functions (i.e., $\mathtt{A}_{\mathtt{none}}$ and $\mathtt{A}_{\mathtt{relu}}$). Section 2.8 provides more details on the usage of the constants. 15

2.2 Operator properties used for verification. The operators are defined in Section 2.4, and the properties are grouped by the operators they involve. Logical variables w, x, y, and z are of type tensor, and variables a, c, k, p, and s are of type parameter. The variable a is used for the axis of concatenation and split, c for the activation mode of convolution, k for the kernel shape of pooling, p for the padding mode of convolution and pooling, and s for the strides of convolution and pooling. 16

2.3 The number of remaining graph substitutions after applying the pruning techniques in order. 20

3.1 Parallelizable dimensions for different operators. The *sample* and *channel* dimension index different samples and neurons, respectively. For images, the *length* and the combination of *height* and *width* dimensions specify a position in an image. 36

3.2 Properties for each task in the task graph. 40

3.3 Details of the DNNs and datasets used in evaluation. 43

3.4 The end-to-end search time with different simulation algorithms (seconds). 47

3.5 The parallelism dimensions used by different approaches. S, O, A, and P indicate parallelism in the Sample, Operator, Attribute, and Parameter dimensions. Hybrid parallelism indicates an approach supports parallelizing an operator in a combination of the sample, attribute, and parameter dimensions (see Figure 3.2). 54

4.1 The graph partitioning strategies used by different frameworks. Balanced training/inference indicates whether an approach can achieve balanced partitioning for GNN training/inference. 61
4.2 The vertex features used in the current cost model. The semantics of the features are described in Section 4.4.1. `WS` is the number of GPU threads in a warp, which is 32 for the V100 GPUs used in the experiments. 64
4.3 All the valid states and their activation tensors for the GNN architecture in Figure 4.3. 66
4.4 Graph datasets used in our evaluation. 69

List of Figures

1.1 An overview of ML system architecture. Existing ML systems rely on manually designed optimizations, including rule-based graph rewrites for graph optimizations, data and model parallelism for parallelization optimizations, and requiring users to manually specify the data placement for intermediate tensors. This dissertation introduces a search-based methodology to automatically discover performance optimizations for ML computation. 2

1.2 Memory hierarchy of a typical GPU cluster. 3

2.1 Comparing computation graph optimization in existing DNN frameworks with TASO. 8

2.2 Graph substitution examples. 11

2.3 A graph substitution for fusing matrix multiplications with a shared input. The target graph has a `concat` and a `split` operator, both of which are performed along the row dimension of a matrix. The split tree of the row dimension for each tensor is shown in a gray box. 12

2.4 Example redundant substitutions pruned by TASO. `Matmul` and `Add` refer to matrix multiplication and element-wise addition, respectively. For each subgraph, A, B, and C refer to its input tensors, while X refers to the output tensor. 19

2.5 A graph substitution using the transpose of matrix multiplication. `matmul` and `transpose` indicate matrix multiplication and transpose, respectively. The parentheses show the potential layouts for each tensor in the substitution, where `C` and `R` indicate the column-major and row-major layouts of a tensor. 21

2.6 A graph substitution example that introduces a cycle into a computation graph, where A, B are the inputs, and X is the output. The original graph computes $A \times \texttt{relu}(A \times B)$, and the new graph is the result of applying the substitution shown in Figure 2.2b that fuses the two matrix multiplications using concatenation and split. The two graphs of the substitution are shown in the dotted boxes. Both the original graph and the substitution are acyclic. However, the resulting graph contains a cycle (highlighted in red). 23

2.7 End-to-end inference performance comparison among existing DNN frameworks and TASO. The experiments were performed using a single inference sample, and all numbers were measured by averaging 1,000 runs on a NVIDIA V100 GPU. We evaluated the TASO's performance with both the cuDNN and TVM backends. For each DNN architecture, the numbers above the TASO bars show the speedup over the best existing approach with the same backend. 24

2.8 The NasNet-A architecture [95] and substitutions discovered by TASO to optimize NasNet-A. Figure 2.8a shows the architecture, where `avg`, `conv`, and `DWC` refer to average pooling, convolution, and depth-wise convolution, respectively. The weight tensors are eliminated for simplicity. Figures 2.8b and 2.8c shows two sequences of substitutions discovered by TASO that are used to optimize subgraphs marked in the black and red boxes in Figure 2.8a. In Figures 2.8b and 2.8c, each arrow refers to a substitution, and the subgraphs in the same color are the graph pair of the substitution. $\texttt{C}_{\texttt{pool}}(3 \times 3)$ in Figure 2.8b is a constant matrix whose entries are 1/9, as defined in Section 2.4. The `enlarge` operator in Figure 2.8c increases a convolution's kernel size by padding the weight (i.e., W_1) with extra 0's. For inference, operators in the gray areas in Figures 2.8b and 2.8c only depend on pre-trained weights (i.e., W_i), and therefore can be pre-computed. 26

2.9 Different approaches to perform multi-batch convolutions in ResNeXt-50 and their performance comparison. TensorFlow and TensorFlow XLA launch the 32 convolutions separately (Figure 2.9a). TensorRT and MetaFlow launch a single grouped convolution kernel that computes all 32 convolutions in parallel (Figure 2.9b). The best graph discovered by TASO uses 4 grouped convolutions, each of which computes 8 convolutions (Figure 2.9c). 27

2.10 A heat map of how often the verified substitutions are used to optimize the five DNN architectures. Only substitutions used in at least one DNN are listed. For each architecture, the number indicates how many times a substitution is used by TASO to obtain the optimized graph. 27

2.11 Performance comparison by using graph substitutions with different size limits. The y-axis shows the relative speedups over the input computation graphs. 28

2.12 End-to-end inference performance comparison on BERT using different strategies to optimize graph substitution and data layout. 30

3.1 FlexFlow overview. 35

3.2 Example parallelization configurations for 1D convolution. Dashed lines show partitioning the tensor. 37

3.3 An example parallelization configuration for a matrix multiplication operator. 37

3.4 Simulating an example parallelization strategy. The tasks' exeTime and device are shown on the top of each column. In Figure 3.4c and 3.4d, the letters "r" and "s" indicate the readyTime and startTime of each task, respectively, and the dashed edges indicate the nextTask. 39

3.5 Architectures of the GPU clusters used in the experiments. An arrow line indicates a NVLink connection. A solid line is a PCI-e connection. Dashed lines are Infiniband connections across different nodes. 44

3.6 Per-iteration training performance on six DNNs. Numbers in parendissertation are the number of compute nodes used in the experiments. The dash lines show the ideal training throughput. 45

3.7 Parallelization performance for NMT on 64 K80 GPUs (16 nodes). FlexFlow reduces per-iteration execution time by 1.7-2.4× and data transfers by 2-5.5× compared to other approaches. FlexFlow achieves similar overall task computation time as expert-designed strategy, which is 20% fewer than data parallelism. 46

3.8 Training curves of Inception-v3 in different systems. The model is trained on 16 P100 GPUs (4 nodes). 47

3.9 Comparison among the parallelization strategies found by different automated frameworks. 48

3.10 An overview of DLRM. 49

3.11 Training performance for DLRM and Candle Uno on the Summit supercomputer [82]. Each compute node is equipped with two IBM POWER9 CPUs and six V100 GPUs. GPUs on the same node are connected by NVLink, and nodes are connected over 100Gb/s EDR Infiniband. 50

3.12 Comparison between the simulated and actual execution time for different DNNs and device topologies. 52

3.13 Search performance with the full and delta simulation algorithms for the NMT model on 16 P100 GPUs (4 nodes). 52

3.14 The best discovered strategy for parallelizing Inception-v3 on 4 P100 GPUs. For each operator, the vertical and horizontal dimensions indicate parallelism in the sample and parameter dimension, respectively. Each GPU is denoted by a color. 53

3.15 The best discovered strategy for parallelizing NMT on 4 P100 GPUs. For each operator, the vertical and horizontal dimensions indicate parallelism in the sample and parameter dimension, respectively. Each grey box denotes a layer, whose operators share the same network parameters. Each GPU is denoted by a color. 53

4.1 Computation of one vertex (in red) in a GNN layer by first aggregating its neighbors' activations (in blue), and then applying DNN operations. 58

4.2 Roc system overview. DPMM represents dynamic-programming-based memory manager. 62

4.3 The computation graph of a toy 1-layer GIN architecture [89]. A box represents an operation, and a circle represents a tensor. Arrows indicate dependencies between tensors and operations. The `gather` operation performs neighborhood aggregation. The `linear` and the following `ReLU` are fused into a single operation as a common optimization in existing frameworks. h^0 and g denote the input features and neighbors of all vertices, respectively. w_1 and w_2 are the weights of the two linear layers. . . . 65

4.4 End-to-end training throughput comparison between existing GNN frameworks and Roc on a single P100 GPU (higher is better). 69

4.5 Training throughput comparison between NeuGraph and Roc using different numbers of GPUs (higher is better). Numbers in parenthesis are the number of compute nodes used in the experiments. 70

4.6 Time-to-accuracy comparison between state-of-the-art sampling techniques and Roc on the Reddit dataset [37]. All experiments used the same GCN model. Roc performed full-batch training on the entire graph, while GraphSAGE and FastGCN performed mini-batch sampling. Each dot indicates one training epoch for GraphSAGE and FastGCN, and five epochs for Roc. 71

4.7 Test accuracy on the Reddit dataset using deeper and larger GNN architectures. The dotted lines show the best test accuracy achieved by GraphSAGE (95.4%), FastGCN (93.7%), and the original GCN architecture (94.7%), respectively. 72

4.8 Training throughput comparison among different graph partitioning strategies on the Reddit dataset (higher is better). Numbers in parentheses are the number of compute nodes used. 73

4.9 End-to-end inference time for the test graphs in the PPI dataset (lower is better). The numbers were measured by averaging the inference time of the four test graphs. . . 74

4.10 Performance comparison among different memory management strategies (lower is better). All numbers are measured by training GCN on the Reddit dataset on a single GPU. 74

Chapter 1

Introduction

Machine learning (ML) techniques, especially recent advances in deep neural network (DNN) architectures, have surpassed human-level predictive performance in a variety of real-world tasks, including image classification [51, 39], face recognition [53], language modeling [28, 87], and game playing [74]. This success is enabled by the recent development of ML systems that provide high-level programming interfaces for programmers to easily prototype different ML models on modern hardware platforms. However, the rapid advance in ML models and ML-specific hardware architectures makes it increasingly challenging to build efficient and scalable ML systems.

Today's ML systems, especially deep learning frameworks such as TensorFlow [11], PyTorch [70], MXNet [18], and Caffe2 [16], heavily rely on human effort to manually optimize the deployment of ML models on specific hardware platforms, a common problem in building performant domain-specific systems. However, unlike conventional application domains, ML deployment requires addressing significantly higher complexity and diversity in both models and hardware architectures, making it infeasible to manually optimize performance for all model and hardware combinations. Moreover, these manually designed performance optimizations are very time-consuming to conceive, design, and implement, but the rapid evolution of ML algorithms requires quick deployment of emerging ML models.

Because of the increasing complexity of today's ML models and hardware architectures, there are many possible strategies to optimize the deployment of an ML model on a hardware platform, and optimized strategies depend on both the models and the hardware. Instead of only considering the limited set of manually designed performance optimizations in existing ML systems, in this dissertation, we propose a search-based methodology that explores a significantly larger and more comprehensive search space of possible strategies and automatically discovers highly optimized solutions.

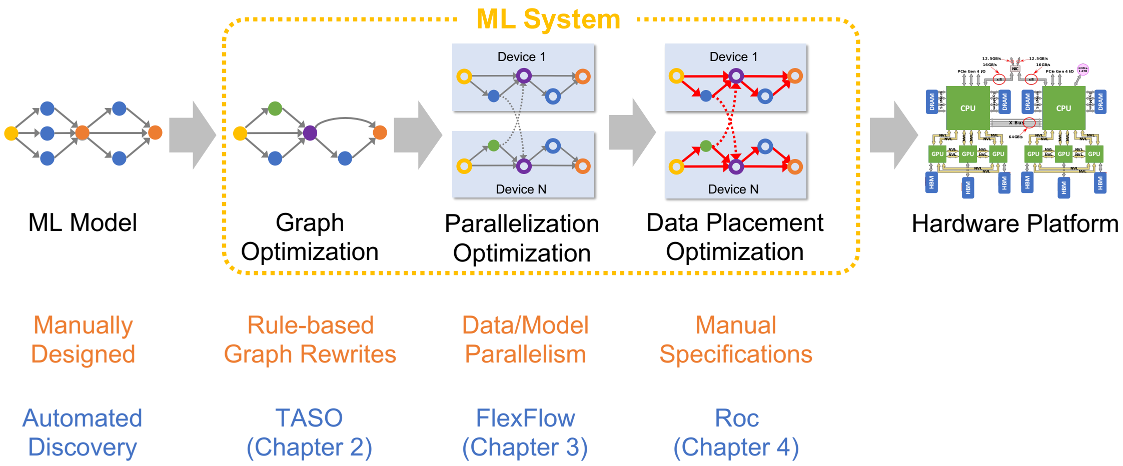


Figure 1.1: An overview of ML system architecture. Existing ML systems rely on manually designed optimizations, including rule-based graph rewrites for graph optimizations, data and model parallelism for parallelization optimizations, and requiring users to manually specify the data placement for intermediate tensors. This dissertation introduces a search-based methodology to automatically discover performance optimizations for ML computation.

1.1 Machine Learning Systems

An ML system deploys the mathematical computation defined in an ML model on a given hardware platform, as shown in Figure 1.1. An ML model is defined as a *computation graph*, where a node is a tensor algebra operator (e.g., matrix multiplication, convolution, etc.), and an edge is a tensor (i.e., an n-dimensional array).

To optimize ML computations, today's ML systems generally perform a sequence of performance optimizations shown in the yellow dotted box in Figure 1.1. First, for an input ML model described as a computation graph, an ML system performs graph-level transformations to produce a mathematically equivalent computation graph with improved runtime performance. Second, parallelization optimizations improve the parallelization performance of an ML model on a distributed heterogeneous hardware platform. Finally, for ML computations on each individual device, an ML system performs data placement optimizations on intermediate tensors (shown as the red directed edges in Figure 1.1) to improve data locality while minimizing data transfers in the memory hierarchy.

Today's ML systems (e.g., TensorFlow and PyTorch) heavily rely on human effort to manually design and implement performance optimizations for different tasks in ML deployment. For example, current systems optimize a computation graph by applying graph transformations that are manually designed by human experts. As another example, to parallelize ML training across multiple devices, existing systems use manually designed strategies, such as *data* and *model* parallelism [51, 27]. To optimize data placement on individual devices, these systems rely on programmers to manually

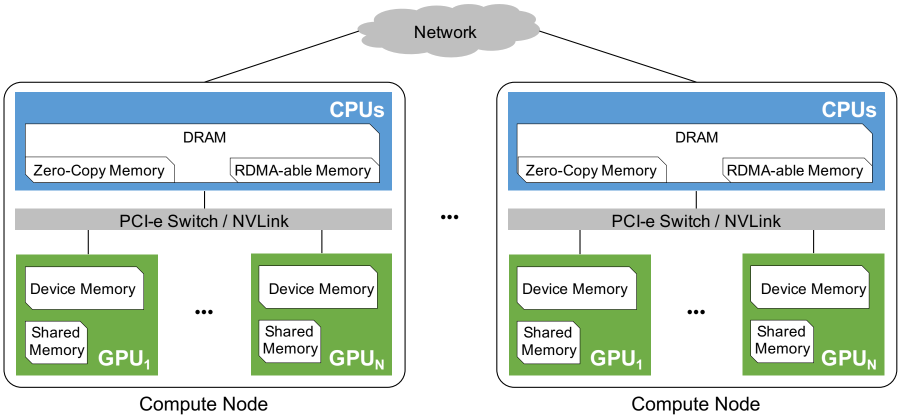


Figure 1.2: Memory hierarchy of a typical GPU cluster.

specify the data placement for the input and output tensors of each operator. Section 1.2 lists the main limitations of current ML system design and the key challenges of building efficient and scalable ML systems.

1.2 Challenges of Building ML Systems

Massively parallelizable computation. Tensor algebra operators can be parallelized in many dimensions. For example, Equation (1.1) shows the mathematical definition of convolution, where *input*, *output*, and *weight* denote the input, output, and weight tensors, respectively.

$$\forall_n \forall_c \forall_h \forall_w \, output(n,c,h,w) = \sum_k \sum_x \sum_y input(n,k,h+x,w+y) * weight(c,k,x,y) \tag{1.1}$$

The computation of a convolution can be parallelized in any combination of the four output dimensions (i.e., n, c, h, w), since the computation of $o(n, c, h, w)$ does not depend on any other elements in the output tensor. Meanwhile, previous work [44] shows that using different combinations of these dimensions to parallelize an operator may result in very different computation performance and communication patterns. This example demonstrates the parallelizable dimensions for a single ML operator. Moreover, the potential parallelization opportunities across multiple operators within an ML model adds another level of expressivity and complexity.

Heterogeneous hardware. The emergence of heterogeneous hardware architectures also significantly increases the complexity and difficulty of building ML systems. ML hardware is becoming more diverse and specialized, ranging from traditional CPUs and GPUs to newly introduced AI-specific hardware, such as TPUs [47]. These diverse processors are often combined into even more complex parallel machines to provide additional performance speedups to meet the increasing computational

System	Task	Phase 1: Search Space	Phase 2: Search Algorithm
TASO	Graph-level optimizations	Automated generation of graph substitutions	Cost-based backtracking search
FlexFlow	Parallelization optimizations	The SOAP search space	Markov Chain Monte Carlo (MCMC) search
Roc	Data placement optimizations	Valid data placement plans	Dynamic programming

Table 1.1: A search-based methodology for automatically discovering ML optimizations.

requirements of today's ML algorithms.

Hardware accelerators introduce heterogeneity not only in kinds of processors but also in kinds of memory. Figure 1.2 shows the memory hierarchy of typical multi-GPU nodes, each of which consists of a host (CPUs) and several GPU devices connected by PCI-e or NVLink [10]. In addition to the device memories attached to each processor (i.e., DRAM and GPU device/shared memory), today's heterogeneous machines also have a block of memory to facilitate direct access by hardware for optimizing data movement. For example, zero-copy and RDMA-able memory is a part of DRAM with direct GPU and network interface card (NIC) access, respectively.

Current ML systems require users to explicitly manage the assignment of operators to processors and the placement of intermediate tensors in memory. It generally requires a tremendous amount of time and effort to manually design performance optimizations for a given heterogeneous machine. However, these optimizations cannot easily generalize to other machines, which requires additional human effort to manually design machine-specific optimizations.

New ML operators. Today's manually designed ML optimizations require significant engineering effort. The engineering problem is aggravated by the fact that new operators are continuously introduced; for example, recent work has proposed depth-wise, grouped, and transposed convolutions for different image classification tasks [41, 88, 29]. While existing ML systems include a number of optimizations for ordinary convolution (e.g., fusing a convolution with different types of operators), supporting each new convolution variant still requires considerable human effort, as each has slightly different semantics, which prevents current ML systems from directly applying optimizations designed for ordinary convolution.

Moreover, manually designing and implementing performance optimizations for new operators is error-prone, and a bug in an optimization can lead to reduced performance [2] or incorrect results [6].

1.3 Approach: Automated Discovery of ML Optimizations

Instead of only considering the limited set of manually designed performance optimizations in existing ML systems, we propose a search-based methodology that explores a significantly larger and more comprehensive search space of possible strategies and automatically discovers highly optimized solutions. The search-based methodology includes two phases as shown in Table 1.1. First, we define a *search space* of possible strategies for a given ML deployment task. The search space can be manually defined by leveraging domain-specific knowledge (as we will show in FlexFlow and Roc) or automatically generated (as we will show in TASO). Second, we design a *search algorithm* to efficiently explore the search space and discover highly-optimized strategies. The search is guided by a *cost model* for evaluating the runtime performance of different strategies in the search space. We use different algorithms to accelerate the search procedure by leveraging the topology of the search space.

This dissertation presents three ML systems that apply this methodology to optimize different tasks in ML deployment.

TASO. Instead of relying on manually designed graph transformations for optimizing ML computation graphs, we propose TASO, the first ML graph optimizer that automatically generates graph optimizations. For each hardware backend, TASO generates potential graph optimizations using the supported operators as basic building blocks, and formally verifies the correctness of these graph optimizations using an automated theorem prover. TASO uses a cost-based backtracking search algorithm to discover how to apply the verified optimizations on an input computation graph to obtain an optimized graph, resulting in less engineering effort, better runtime performance, and stronger correctness guarantees.

FlexFlow. To generalize and go beyond today's manually designed parallelization strategies (e.g., data and model parallelism) for distributed ML training, we introduce the SOAP search space, which contains a comprehensive set of possible parallelization strategies. The SOAP search space identifies four parallelizable dimensions and captures potential parallelization opportunities across different Samples, Operators, Attributes, and Parameters. A key property of the SOAP search space is that all strategies perform the same computation defined by the ML model and therefore maintain the same model accuracy by design. FlexFlow is a deep learning engine that automatically searches over parallelization strategies in the SOAP search space. FlexFlow includes a novel execution simulator to evaluate the runtime performance of different strategies, and uses a Markov Chain Monte Carlo (MCMC) search algorithm to discover high performance strategies Compared to existing manually designed parallelization strategies, FlexFlow largely improves training throughput and scalability, while requiring no manual effort during the search procedure.

Roc. To automate data placement optimizations and minimize data transfers in the memory hierarchy, Roc formulates the task of optimizing data placement as a cost minimization problem and discovers data placement plans that minimize data transfer cost. To optimize data placement for a given hardware device (e.g., a GPU), Roc considers the search space of all valid data placement plans that satisfy memory capacity requirements, and uses a *dynamic programming* algorithm to explore the search space and discover a *globally optimal* data management plan that minimizes data transfers in the memory hierarchy.

1.4 Discussion

This dissertation is partially motivated by a long-term trend in modern hardware architectures — hardware is becoming more diverse and heterogeneous. This increasing diversity and heterogeneity of hardware is not temporary but an essential shift driven by the need to address the increasing computational requirement within a fixed power budget. The traditional approach to harness a new hardware platform is to manually design monolithic optimizing compilers that perform end-to-end optimizations from high-level programming languages to instruction set supported by the hardware platform. This approach requires a tremendous amount of time and engineering effort to manually design and implement performance optimizations, and cannot directly target heterogeneous parallel machines with a mixture of diverse hardware devices. Therefore, it is becoming a necessity to develop automated mechanisms to address the increasing diversity and heterogeneity of hardware architectures.

Although the search-based methodology proposed in this dissertation was designed to automate the discovery of ML optimizations, we believe many of these techniques can be generalized beyond ML applications. To apply the search-based methodology to other application domains, one key challenge is designing the right programming abstractions that allow domain-specific compilers to generate a comprehensive search space of optimizations and quickly search over the space of potential optimizations to discover performant programs. We believe our automated approach provides a promising alternative for designing future compilers targeting new hardware backend and heterogeneous machines.

1.5 Roadmap

The rest of the dissertation is organized as follows. Chapter 2 presents TASO for automatically discovering computation graph optimizations. Chapter 3 describes FlexFlow, which automatically searches for fast parallelization strategies for distributed ML training. Chapter 4 introduces Roc, which automates data placement optimizations. Finally, we conclude in Chapter 5.

Chapter 2

Automated Discovery of Graph Optimizations

Existing ML systems optimize the computation graph of an ML model by applying graph transformations manually designed by human experts. This approach misses possible graph optimizations and is difficult to scale, as new ML operators are introduced on a regular basis.

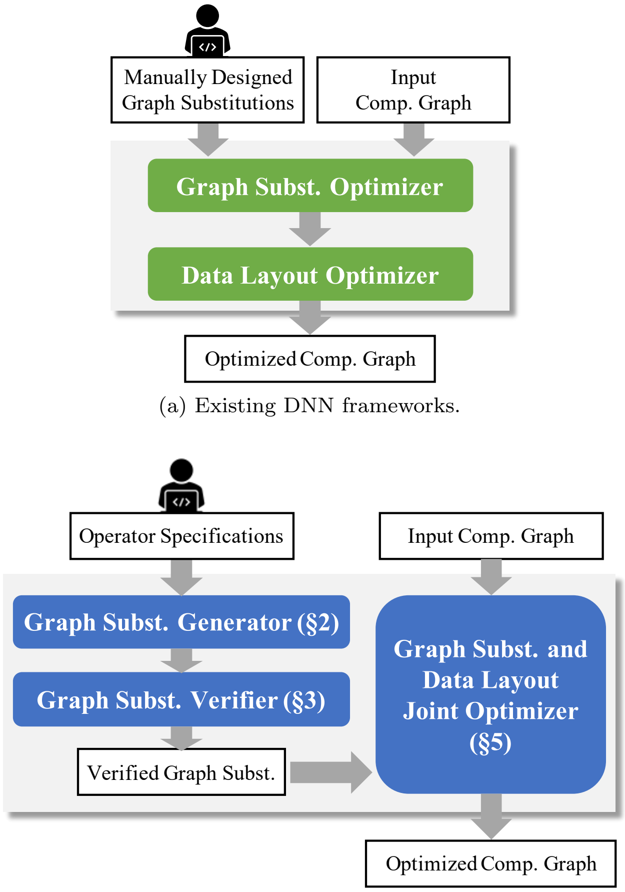
In this chapter, we present TASO, the first ML computation graph optimizer that automatically generates graph substitutions. TASO takes as input a list of operator specifications and generates candidate substitutions using the given operators as basic building blocks. All generated substitutions are formally verified against the operator specifications using an automated theorem prover. To optimize a given computation graph, TASO performs a cost-based backtracking search, applying the substitutions to find an optimized graph, which can be directly used by existing ML systems.

Our evaluation on real-world ML architectures shows that TASO significantly outperforms existing computation graph optimizers, while requiring much less human effort.

2.1 Motivation

Current ML systems optimize a computation graph by applying graph substitutions that are manually designed by domain experts, as depicted in Figure 2.1a. For example, TensorFlow, PyTorch, TensorRT, and TVM use a greedy rule-based optimization strategy and directly perform all applicable substitutions (i.e., rules) on an input computation graph [11, 70, 80, 19]. Although manually designed substitutions improve the performance of ML computations, they fall short in several respects.

Maintainability. Hand-written graph substitutions require significant engineering effort. For example, TensorFlow r1.14 includes 155 substitutions implemented in approximately 53K lines of



(a) Existing DNN frameworks.

(b) TASO.

Figure 2.1: Comparing computation graph optimization in existing DNN frameworks with TASO.

C++ code. The maintenance problem is aggravated by the fact that new operators are continuously introduced; for example, recent work has proposed depth-wise [41], grouped [88], and transposed convolutions [29] for different image classification tasks. TensorFlow r1.14 currently includes 17 graph substitutions (written in 4K lines of code) to optimize ordinary convolution (e.g., fusing it with different types of operators). With the existing approach, supporting each new convolution variant would require a similar implementation effort, as each has slightly different semantics and cannot be directly optimized using existing substitutions.

Data layout. Tensor data can be stored in memory in various layouts, and this choice has a high impact on runtime performance. The best layout depends on both the operator and the hardware. For example, on a P100 GPU, convolution performs best with row-major layout (i.e., the inner-most dimension is contiguously stored), while matrix multiplication performs best with column-major layout (i.e., the outer-most dimension is contiguously stored). On a Tesla V100 GPU with tensor cores [7] supporting 4×4 matrix operations, optimal performance may require tiling tensors into 4×4 chunks. However, considering layout transformations together with graph substitutions adds another level of complexity. For example, a graph substitution may only improve performance if it is combined with a particular layout transformation (see Section 2.9.5). Current frameworks avoid this complexity by treating data layout and graph substitution as separate optimization problems and solve them sequentially [19, 62], as shown in Figure 2.1a, but this separation misses many possible optimization opportunities.

Correctness. Hand-written graph substitutions are error-prone, and a bug in graph substitutions can lead to incorrect computation graphs [6, 2]. The same issue arises in compiler optimization, where an incorrect optimization leads to incorrect programs. In the compiler literature, significant effort has been devoted to formally verifying optimizations [14, 69, 67, 54, 79, 73, 22, 25]. However, to the best of our knowledge, such techniques have not been applied to graph substitution optimizations performed by DNN frameworks.

2.2 The Approach

Figure 2.1b shows an overview of TASO (Tensor Algebra SuperOptimizer), which differs from existing frameworks in three aspects. First, TASO only requires operator definitions and specifications, and automatically generates graph substitutions, reducing manual effort. Second, TASO employs formal verification to ensure correctness of the generated graph substitutions. Finally, TASO jointly optimizes graph substitutions and data layout transformations, achieving significantly better runtime performance.

Generating substitutions. TASO's *graph substitution generator* enumerates all possible computation graphs over a given set of DNN operators (e.g., the cuDNN kernels [21]) up to a fixed size k (i.e., all computation graphs with at most k nodes), and executes them on a set of random input tensors. Any pair of computation graphs that have identical results on the random inputs are considered as a candidate substitution. To efficiently find all such pairs, TASO constructs a hash table where computation graphs are stored based on the hash of their outputs for the random inputs.

Formal verification. TASO's *graph substitution verifier* is used to ensure correctness of the generated graph substitutions, relying on user provided *operator properties*. Operator properties capture mathematical properties of operators, e.g., linearity of convolution. The full list of 43 operator properties we used appears in Table 2.2. As our evaluation shows, a small set of properties for each operator suffices to prove the correctness of complex substitutions.

Formally, we model tensor operators using a symbolic representation based on first-order logic that is agnostic to the size of the underlying tensors, and can succinctly express operator properties. The verifier uses the specified properties to check the correctness of all generated graph substitutions using an automated theorem prover.

We also present a methodology for developing operator properties, which assists the developer in two ways: (1) discovery of required properties is guided by the graph substitution generator, and (2) operator properties are subject to further validation using symbolic execution on tensors of small sizes. During the development process, we found that the our verification methodology uncovered several bugs, both in the operator specifications and in the implementation of the graph substitution generator.

Joint optimization. TASO jointly optimizes graph substitutions and data layout transformations by integrating them into a common representation. TASO explores the search space of potential graph substitutions and layout transformations, and uses a *cost-based backtracking search* algorithm to discover high-performance computation graphs. During the search, TASO measures the performance of a proposed DNN operator with a specific proposed data layout on the hardware. These individual measurements are used to predict the performance of an entire computation graph with specific data layouts.

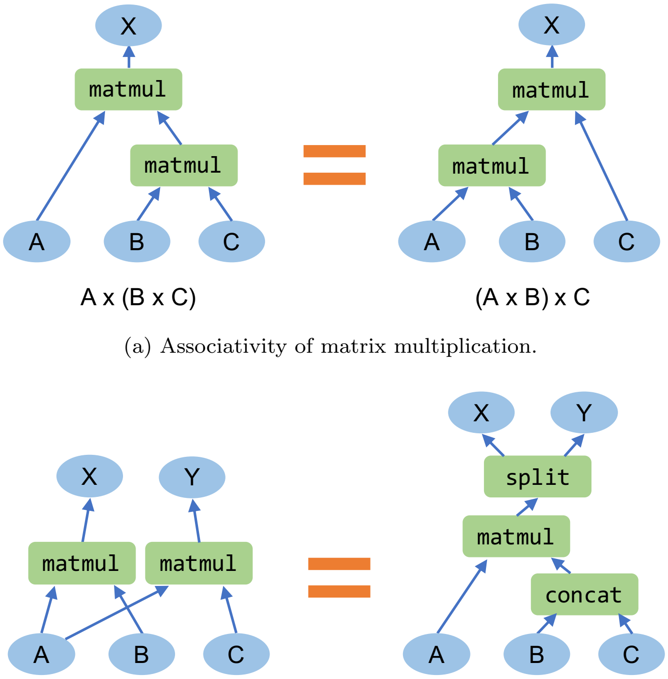
Evaluation. We evaluate TASO on five real-world DNN architectures. For widely used DNNs optimized by existing frameworks, such as ResNet-50 [39], TASO matches the performance of these frameworks with hand-written rules by using operator definitions and a total of 1,400 lines of operator specifications.

For ResNeXt-50 [88], NasRNN [94], NasNet-A [95], and BERT [28], TASO produces computation graphs up to 2.8× faster than state-of-the-art frameworks by automatically discovering novel graph substitutions to optimize these architectures. Compared to separately optimizing graph substitutions and data layout, we show that the joint optimization can further improve performance by 1.2×. In all experiments, TASO discovered an optimized graph in less than ten minutes, making it feasible to use when optimizing a DNN architecture before large-scale deployment.

2.3 Graph Substitution Generator

This section describes the TASO substitution generator that automatically generates potential substitutions given a list of primitive operators. The generation algorithm finds all valid substitutions up to a certain size.

To find all potential substitutions, a straightforward approach is to test all pairs of graphs for equivalence, which requires a quadratic number of tests between graphs. We adopt an idea from compiler superoptimization [14] and compute a *fingerprint* for each graph, which is a hash of the



(a) Associativity of matrix multiplication.

(b) Fusing two matrix multiplications using concatenation and split.

Figure 2.2: Graph substitution examples.

graph outputs on some specific inputs. Two graphs are certainly not equivalent if they have different fingerprints, and so by only comparing graphs with the same fingerprint, TASO significantly reduces the number of equivalence tests. In the experiments, we observe that all graphs with the same fingerprint are verified equivalent by TASO.

2.3.1 Graph Substitution Definition

A *graph substitution* consists of three components: (1) a *source graph* that is matched to subgraphs in a computation graph; (2) a *target graph*[1] that defines a functionally equivalent new subgraph to replace the matched subgraph; and (3) a *mapping* relation between input/output tensors in the source and target graphs. Figure 2.2a shows an example graph substitution using the associativity of matrix multiplication. Figure 2.2b fuses two matrix multiplications into one using concatenation and split along the row dimension. A, B, C, X, and Y identify the mapping between input and output tensors in the substitution.

A graph substitution is specified independently of the concrete tensor shapes. For example, the substitutions of Figure 2.2 can be applied to tensors A, B, and C of any concrete shape. Some

[1] In some of the superoptimization literature, what we call the *source* is called the *target*, and what we call the *target* is called the *rewrite*.

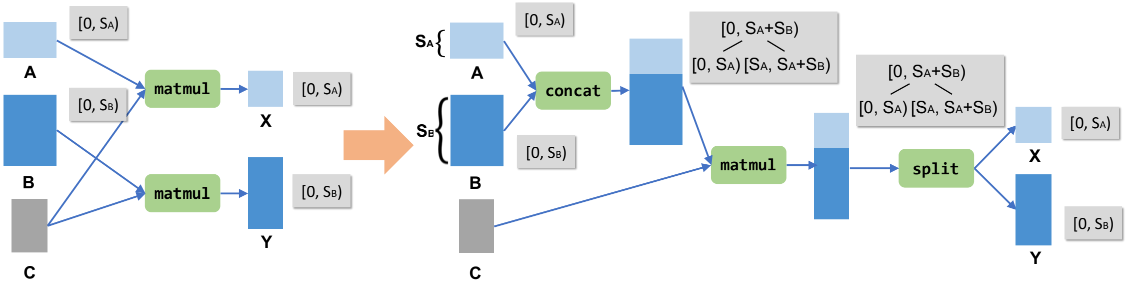


Figure 2.3: A graph substitution for fusing matrix multiplications with a shared input. The target graph has a `concat` and a `split` operator, both of which are performed along the row dimension of a matrix. The split tree of the row dimension for each tensor is shown in a gray box.

operators also depend on *configuration parameters* to determine the behavior of the operator. For example, the parameters of convolution determine the strides, padding, and activation (e.g., applying the `relu` function [64] as part of convolution); and the parameters of split or concatenation determine the axis along which to apply the operator.

Concatenation and split operators. Concatenation and split operators are commonly used in fusing operators with shared inputs, as illustrated in Figure 2.2b. A split operator partitions a tensor into two disjoint sub-tensors along a dimension determined by its parameter. This presents a complication, as the split point cannot be inferred from the input tensors or the parameter. To solve this problem, we observe that a split operator always partitions a tensor at previous concatenation points to "undo" the most recent concatenation operator. We use this fact to define a suitable semantics for the split operator.

Formally, we maintain a *split tree* for each dimension of a tensor to track the concatenation history. Figure 2.3 shows the split trees of the row dimension for all tensors in Figure 2.2b. The split trees allow the substitution to recover the split point without introducing any additional parameters. Our approach also supports multi-way concatenation and split by nesting of concatenation and split operators.

2.3.2 Generation Algorithm

For a given set of operator specifications, TASO generates potential graph substitutions in two steps, as shown in Algorithm 1.

Step 1: Enumerating potential graphs and collecting their fingerprints. TASO first enumerates all potential graphs up to a certain size for a given set of operators. To construct a graph, TASO iteratively adds an operator in the current graph by enumerating the type of the operator and the input tensors to the operator. The input tensors can be initial input tensors to the graph

Algorithm 1 Graph substitution generation algorithm.

```
 1: Input: A set of operators $\mathcal{P}$, and a set of input tensors $\mathcal{I}$.
 2: Output: Candidate graph substitutions $\mathcal{S}$.
 3:
 4: // Step 1: enumerating potential graphs.
 5: $\mathcal{D} = \{\}$  // $\mathcal{D}$ is a graph hash table indexed by their fingerprints.
 6: Build(1, $\emptyset$, $\mathcal{I}$)
 7: function Build($n$, $\mathcal{G}$, $\mathcal{I}$)
 8:     if $\mathcal{G}$ contains duplicated computation then
 9:         return
10:     end if
11:     $\mathcal{D} = \mathcal{D} + (\text{FingerPrint}(\mathcal{G}), \mathcal{G})$
12:     if $n < threshold$ then
13:         for $op \in \mathcal{P}$ do
14:             for $i \in \mathcal{I}$ and $i$ is a valid input to $op$ do
15:                 Add operator $op$ into graph $\mathcal{G}$.
16:                 Add the output tensors of $op$ into $\mathcal{I}$.
17:                 Build($n + 1$, $\mathcal{G}$, $\mathcal{I}$)
18:                 Remove operator $op$ from $\mathcal{G}$.
19:                 Remove the output tensors of $op$ from $\mathcal{I}$.
20:             end for
21:         end for
22:     end if
23: end function
24:
25: // Step 2: testing graphs with identical fingerprint.
26: $\mathcal{S} = \{\}$
27: for $\mathcal{G}_1, \mathcal{G}_2 \in \mathcal{D}$ with the same FingerPrint($\cdot$) do
28:     if $\mathcal{G}_1$ and $\mathcal{G}_2$ are equivalent for all test cases then
29:         $\mathcal{S} = \mathcal{S} + (\mathcal{G}_1, \mathcal{G}_2)$
30:     end if
31: end for
32: return $\mathcal{S}$
```

(e.g., A, B, and C in Figure 2.2) or the output tensors of previous operators (e.g., the output of the `matmul` and `concat` operators in Figure 2.2).

Algorithm 1 (line 7-23) shows a *depth-first search* algorithm for constructing all acyclic computation graphs that do not contain duplicated computation. We say a graph contains duplicated computation if it has two operators performing the same computation on the same input tensors. The generator ignores such graphs as substitutions containing duplicated computation can be subsumed by more general substitutions (see Section 2.5).

For each graph, we collect its fingerprint, which is a hash of the output tensors obtained by evaluating the graph on some input tensors. TASO uses both randomly initialized tensors and a number of constants as inputs to allow finding substitutions involving constant tensors, such as the identity matrix (see examples in Section 2.9.3). To avoid floating-point errors in computing a fingerprint, all tensors are represented with integers, following the method introduced in [86].

Since a graph can have an arbitrary number of output tensors, the hash function must ensure the fingerprint is independent of any permutation of the output tensors. To guarantee this property, TASO employs a two-step hash function to compute fingerprints as follows.

$$\textsc{FingerPrint}(\mathcal{G}) = hash_2\left(\{hash_1(t_i) \mid i \in \textsc{Outputs}(\mathcal{G})\}\right) \tag{2.1}$$

where t_i are the output tensors of graph $\mathcal{G}$; $hash_1$ takes as arguments the states and content of an output tensor, including the size, shape, and content of the tensor, and $hash_2$ is a symmetric hash function applied to an unordered set of hash values.

Step 2: Testing graphs with identical fingerprints. For graphs with the same fingerprint, TASO further examines their equivalence on a set of test cases. Similar to collecting fingerprints, each test case contains a set of randomized input tensors, and two graphs pass if they produce equivalent output tensors for all test cases. Unlike the fingerprints, these tests use floating point numbers ranging between -1 and 1, and classify two output tensors as equivalent if their outputs differ by no more than a small threshold value, which is 10^{-5} in the evaluation. For this threshold, we observed no differences from the integer tests. However, it is possible to use a smaller threshold to filter out substitutions that are valid for real numbers but result in floating point errors.

Each pair of graphs passing the random testing becomes a candidate graph substitution, and the mapping relation between the input/output tensors in the graph pair can be automatically inferred from the test cases. All candidate graph substitutions are then sent to the substitution verifier to check their correctness (Section 2.4), and later pruned to eliminate redundant substitutions (Section 2.5).

It is worth noting that prior work [14] reported false positives in using random testing to examine code transformations in compiler superoptimization. They observed that a number of incorrect code transformations passed a set of test cases. We have not observed any false positive cases in all the experiments. We use a single test case to examine all graph pairs with the same fingerprint, and all substitutions passing the test case are correct and verified by the substitution verifier. This is likely due to the high arithmetic density of DNN operators and the lack of branching (if statements) in computation graphs. As a reference, [36] shows that for programs with only linear operators, the probability that two nonequivalent programs produce identical output on a random input is at most $\frac{1}{d}$, where d is the number of possible values for a variable (i.e., $d = 2^{32}$ in TASO).

2.4 Graph Substitution Verifier

The key idea behind our approach to formally verifying substitutions is to use a small set of *operator properties* expressed in first-order logic. These properties are manually written and reviewed, and are further validated by symbolically executing operators on tensors of small sizes and confirming that the operator properties are satisfied for these tensor sizes. In practice, the discovery of operator

Table 2.1: Tensor operators and constant tensors included in TASO. Similar to existing DNN frameworks [11, 70], pooling and convolution operators support different strides and padding modes (i.e., $\texttt{P}_{\texttt{same}}$ and $\texttt{P}_{\texttt{valid}}$); convolution supports different activation functions (i.e., $\texttt{A}_{\texttt{none}}$ and $\texttt{A}_{\texttt{relu}}$). Section 2.8 provides more details on the usage of the constants.

Name	**Description**	**Parameters**
Tensor Operators		
`ewadd`	Element-wise addition	
`ewmul`	Element-wise multiplication	
`smul`	Scalar multiplication	
`transpose`	Transpose	
`matmul`	Batch matrix multiplication[#]	
`conv`	Grouped convolution[%]	stride, padding, activation
`enlarge`	Pad conv. kernel with zeros[†]	kernel size
`relu`	Relu operator	
$\texttt{pool}_{\texttt{avg}}$	Average pooling	kernel size, stride, padding
$\texttt{pool}_{\texttt{max}}$	Max pooling	kernel size, stride, padding
`concat`	Concatenation of two tensors	concatenation axis
$\texttt{split}_{\{0,1\}}$	Split into two tensors	split axis
Constant Tensors		
$\texttt{C}_{\texttt{pool}}$	Average pooling constant	kernel size
$\texttt{I}_{\texttt{conv}}$	Convolution id. kernel	kernel size
$\texttt{I}_{\texttt{matmul}}$	Matrix multiplication id.	
$\texttt{I}_{\texttt{ewmul}}$	Tensor with 1 entries	

[#] Normal matrix multiplication is considered as batch size equals 1.

[%] Normal and depth-wise conv. are special cases of grouped conv.

[†] Increase the size of a conv. kernel, restricted to operate on input tensors.

Table 2.2: Operator properties used for verification. The operators are defined in Section 2.4, and the properties are grouped by the operators they involve. Logical variables w, x, y, and z are of type tensor, and variables a, c, k, p, and s are of type parameter. The variable a is used for the axis of concatenation and split, c for the activation mode of convolution, k for the kernel shape of pooling, p for the padding mode of convolution and pooling, and s for the strides of convolution and pooling.

Operator Property	**Comment**
$\forall x, y, z.\ \texttt{ewadd}(x, \texttt{ewadd}(y, z)) = \texttt{ewadd}(\texttt{ewadd}(x, y), z)$	`ewadd` is associative
$\forall x, y.\ \texttt{ewadd}(x, y) = \texttt{ewadd}(y, x)$	`ewadd` is commutative
$\forall x, y, z.\ \texttt{ewmul}(x, \texttt{ewmul}(y, z)) = \texttt{ewmul}(\texttt{ewmul}(x, y), z)$	`ewmul` is associative
$\forall x, y.\ \texttt{ewmul}(x, y) = \texttt{ewmul}(y, x)$	`ewmul` is commutative
$\forall x, y, z.\ \texttt{ewmul}(\texttt{ewadd}(x, y), z) = \texttt{ewadd}(\texttt{ewmul}(x, z), \texttt{ewmul}(y, z))$	distributivity
$\forall x, y, w.\ \texttt{smul}(\texttt{smul}(x, y), w) = \texttt{smul}(x, \texttt{smul}(y, w))$	`smul` is associative
$\forall x, y, w.\ \texttt{smul}(\texttt{ewadd}(x, y), w) = \texttt{ewadd}(\texttt{smul}(x, w), \texttt{smul}(y, w))$	distributivity
$\forall x, y, w.\ \texttt{smul}(\texttt{ewmul}(x, y), w) = \texttt{ewmul}(x, \texttt{smul}(y, w))$	operator commutativity
$\forall x.\ \texttt{transpose}(\texttt{transpose}(x)) = x$	`transpose` is its inverse
$\forall x, y.\ \texttt{transpose}(\texttt{ewadd}(x, y)) = \texttt{ewadd}(\texttt{transpose}(x), \texttt{transpose}(y))$	operator commutativity
$\forall x, y.\ \texttt{transpose}(\texttt{ewmul}(x, y)) = \texttt{ewmul}(\texttt{transpose}(x), \texttt{transpose}(y))$	operator commutativity
$\forall x, w.\ \texttt{smul}(\texttt{transpose}(x), w) = \texttt{transpose}(\texttt{smul}(x, w))$	operator commutativity
$\forall x, y, z.\ \texttt{matmul}(x, \texttt{matmul}(y, z)) = \texttt{matmul}(\texttt{matmul}(x, y), z)$	`matmul` is associative
$\forall x, y, w.\ \texttt{smul}(\texttt{matmul}(x, y), w) = \texttt{matmul}(x, \texttt{smul}(y, w))$	`matmul` is linear
$\forall x, y, z.\ \texttt{matmul}(x, \texttt{ewadd}(y, z)) = \texttt{ewadd}(\texttt{matmul}(x, y), \texttt{matmul}(x, z))$	`matmul` is linear
$\forall x, y.\ \texttt{transpose}(\texttt{matmul}(x, y)) = \texttt{matmul}(\texttt{transpose}(y), \texttt{transpose}(x))$	`matmul` and `transpose`
$\forall s, p, c, x, y, w.\ \texttt{conv}(s, p, c, \texttt{smul}(x, w), y) = \texttt{conv}(s, p, c, x, \texttt{smul}(y, w))$	`conv` is bilinear
$\forall s, p, x, y, w.\ \texttt{smul}(\texttt{conv}(s, p, \texttt{A}_{\texttt{none}}, x, y), w) = \texttt{conv}(s, p, \texttt{A}_{\texttt{none}}, \texttt{smul}(x, w), y)$	`conv` is bilinear
$\forall s, p, x, y, z.\ \texttt{conv}(s, p, \texttt{A}_{\texttt{none}}, x, \texttt{ewadd}(y, z)) = \texttt{ewadd}(\texttt{conv}(s, p, \texttt{A}_{\texttt{none}}, x, y), \texttt{conv}(s, p, \texttt{A}_{\texttt{none}}, x, z))$	`conv` is bilinear
$\forall s, p, x, y, z.\ \texttt{conv}(s, p, \texttt{A}_{\texttt{none}}, \texttt{ewadd}(x, y), z) = \texttt{ewadd}(\texttt{conv}(s, p, \texttt{A}_{\texttt{none}}, x, z), \texttt{conv}(s, p, \texttt{A}_{\texttt{none}}, y, z))$	`conv` is bilinear
$\forall s, c, k, x, y.\ \texttt{conv}(s, \texttt{P}_{\texttt{same}}, c, x, y) = \texttt{conv}(s, \texttt{P}_{\texttt{same}}, c, x, \texttt{enlarge}(k, y)),$	enlarge convolution kernel
$\forall s, p, x, y.\ \texttt{conv}(s, p, \texttt{A}_{\texttt{relu}}, x, y) = \texttt{relu}(\texttt{conv}(s, p, \texttt{A}_{\texttt{none}}, x, y))$	`conv` w/ $\texttt{A}_{\texttt{relu}}$ applies `relu`
$\forall x.\ \texttt{relu}(\texttt{transpose}(x)) = \texttt{transpose}(\texttt{relu}(x))$	operator commutativity
$\forall s, p, x, k.\ \texttt{conv}(s, p, \texttt{A}_{\texttt{none}}, x, \texttt{C}_{\texttt{pool}}(k)) = \texttt{pool}_{\texttt{avg}}(k, s, p, x)$	pooling by conv. w/ $\texttt{C}_{\texttt{pool}}$
$\forall k, x.\ \texttt{conv}(1, \texttt{P}_{\texttt{same}}, \texttt{A}_{\texttt{none}}, x, \texttt{I}_{\texttt{conv}}(k)) = x$	identity kernel
$\forall x.\ \texttt{matmul}(x, \texttt{I}_{\texttt{matmul}}) = x$	identity matrix
$\forall x.\ \texttt{ewmul}(x, \texttt{I}_{\texttt{ewmul}}) = x$	`ewmul` identity
$\forall a, x, y.\ \texttt{split}_0(a, \texttt{concat}(a, x, y)) = x$	split definition
$\forall a, x, y.\ \texttt{split}_1(a, \texttt{concat}(a, x, y)) = y$	split definition
$\forall x, y, z, w.\ \texttt{concat}(0, \texttt{concat}(1, x, y), \texttt{concat}(1, z, w)) = \texttt{concat}(1, \texttt{concat}(0, x, z), \texttt{concat}(0, y, w))$	geometry of concat
$\forall a, x, y, w.\ \texttt{concat}(a, \texttt{smul}(x, w), \texttt{smul}(y, w)) = \texttt{smul}(\texttt{concat}(a, x, y), w)$	operator commutativity
$\forall a, x, y, z, w.\ \texttt{concat}(a, \texttt{ewadd}(x, y), \texttt{ewadd}(z, w)) = \texttt{ewadd}(\texttt{concat}(a, x, z), \texttt{concat}(a, y, w))$	operator commutativity
$\forall a, x, y, z, w.\ \texttt{concat}(a, \texttt{ewmul}(x, y), \texttt{ewmul}(z, w)) = \texttt{ewmul}(\texttt{concat}(a, x, z), \texttt{concat}(a, y, w))$	operator commutativity
$\forall a, x, y.\ \texttt{concat}(a, \texttt{relu}(x), \texttt{relu}(y)) = \texttt{relu}(\texttt{concat}(a, x, y))$	operator commutativity
$\forall x, y.\ \texttt{concat}(1, \texttt{transpose}(x), \texttt{transpose}(y)) = \texttt{transpose}(\texttt{concat}(0, x, y))$	concat and transpose
$\forall x, y, z.\ \texttt{concat}(1, \texttt{matmul}(x, y), \texttt{matmul}(x, z)) = \texttt{matmul}(x, \texttt{concat}(1, y, z))$	concat and matrix mul.
$\forall x, y, z, w.\ \texttt{matmul}(\texttt{concat}(1, x, z), \texttt{concat}(0, y, w)) = \texttt{ewadd}(\texttt{matmul}(x, y), \texttt{matmul}(z, w))$	concat and matrix mul.
$\forall s, p, c, x, y, z.\ \texttt{concat}(0, \texttt{conv}(s, p, c, x, z), \texttt{conv}(s, p, c, y, z)) = \texttt{conv}(s, p, c, \texttt{concat}(0, x, y), z)$	concat and conv.
$\forall s, p, c, x, y, z.\ \texttt{concat}(1, \texttt{conv}(s, p, c, x, y), \texttt{conv}(s, p, c, x, z)) = \texttt{conv}(s, p, c, x, \texttt{concat}(0, y, z))$	concat and conv.
$\forall s, p, x, y, z, w.\ \texttt{conv}(s, p, \texttt{A}_{\texttt{none}}, \texttt{concat}(1, x, z), \texttt{concat}(1, y, w)) = \texttt{ewadd}(\texttt{conv}(s, p, \texttt{A}_{\texttt{none}}, x, y), \texttt{conv}(s, p, \texttt{A}_{\texttt{none}}, z, w))$	concat and conv.
$\forall k, s, p, x, y.\ \texttt{concat}(1, \texttt{pool}_{\texttt{avg}}(k, s, p, x), \texttt{pool}_{\texttt{avg}}(k, s, p, y)) = \texttt{pool}_{\texttt{avg}}(k, s, p, \texttt{concat}(1, x, y))$	concat and pooling
$\forall k, s, p, x, y.\ \texttt{concat}(0, \texttt{pool}_{\texttt{max}}(k, s, p, x), \texttt{pool}_{\texttt{max}}(k, s, p, y)) = \texttt{pool}_{\texttt{max}}(k, s, p, \texttt{concat}(0, x, y))$	concat and pooling
$\forall k, s, p, x, y.\ \texttt{concat}(1, \texttt{pool}_{\texttt{max}}(k, s, p, x), \texttt{pool}_{\texttt{max}}(k, s, p, y)) = \texttt{pool}_{\texttt{max}}(k, s, p, \texttt{concat}(1, x, y))$	concat and pooling

properties is guided by the substitutions discovered by the substitution generator.

For purposes of verification, we model tensor operators using first-order logic, where operators are represented using functions of both their parameters and their input tensors. For example $\texttt{conv}(s, p, c, x, y)$ represents the convolution operator applied to tensors x and y, where the parameter s determines the stride, p determines padding mode, and c determines the activation mode, e.g., applying a `relu` activation function as part of the convolution operator kernel. For example, the fact that convolution without activation (denoted by $\texttt{A}_{\texttt{none}}$) is linear in its first argument is captured by the following operator property (where `ewadd` represents element-wise tensor addition):

$$\forall s, p, x, y, z.\texttt{conv}(s, p, \texttt{A}_{\texttt{none}}, \texttt{ewadd}(x, y), z) = \texttt{ewadd}(\texttt{conv}(s, p, \texttt{A}_{\texttt{none}}, x, z), \texttt{conv}(s, p, \texttt{A}_{\texttt{none}}, y, z)) \tag{2.2}$$

Section 2.4 lists all operators and tensor constants used in our evaluation, and Table 2.2 shows the full list of operator properties used in our evaluation to verify graph substitutions.

Given the operator properties, we use a first-order theorem prover—our implementation uses Z3 [26]—to verify all generated substitutions. This verification amounts to entailment checking in first-order logic, checking that the operator properties entail functional equivalence of the two graphs of each generated substitution.

Modeling the operators using first-order logic involves a degree of abstraction (e.g., the shapes of tensors are not modeled). We found this level of abstraction to be suitable for verifying graph substitutions. We also note that the data layout is abstracted for verification purposes—layout does not affect operator semantics, and the optimizer (Section 2.7) ensures that layouts are used consistently.

Methodology for developing operator properties. We developed operator properties as needed to determine the correctness of generated graph substitutions using an iterative process. During the development process, we ran the substitution generator and tried to verify all discovered substitutions. If a substitution could not be verified and appeared correct, we added an appropriate property (or properties). To safeguard against mistakes in operator properties, we used further *validation* steps.

To validate operator properties, TASO verifies the operator properties themselves for all combinations of parameter values and tensor sizes up to a small bound—in our evaluation the bound was 4×4×4×4. To implement this checking, TASO requires a basic symbolic implementation of each tensor operator in Python. TASO symbolically executes this implementation for tensors of small size, effectively elaborating the tensor operations into symbolic real arithmetic expressions, where activation functions (e.g., `relu`) are modeled using uninterpreted functions. TASO then uses Z3, here as an SMT solver for the theory of real arithmetic, to verify the operator properties. For example, if a user would try to add the (wrong) property stating the convolution operator is linear for all activation modes (including `relu` activation), then this check would show that this property is not satisfied by the actual operators.

As an additional validation step that assists the development process, TASO checks that the set of operator properties is consistent and does not contain redundancies (i.e., a property entailed by other properties), which amounts to first-order entailment checks. These checks are also useful for discovering erroneous properties, and are cheaper to perform than the verification for small tensor sizes.

During our development process, the verification methodology revealed several subtle bugs. Some bugs in the graph substitution generator were found when it generated substitutions that could not be verified, and the validation steps described above revealed several bugs in candidate operator properties. In our experience, a new operator can be supported with a small amount of effort, usually a few hours of work by an expert. Typically a few properties must be written for each operator. In our evaluation, we were able to verify all 743 generated graph substitutions using 43 operator properties (see Table 2.2).

2.5 Pruning Redundant Substitutions

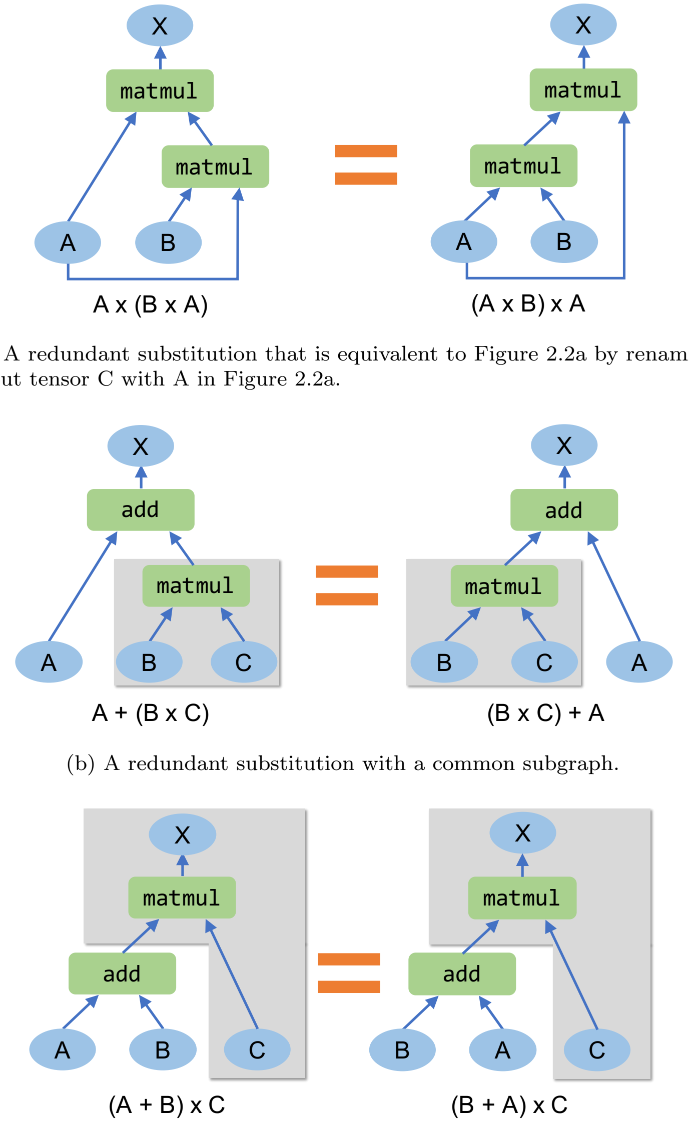
A graph substitution is redundant if it is subsumed by a more general valid substitution. This section describes the pruning techniques used by TASO to eliminate redundant graph substitutions. All pruning steps preserve *all* optimization opportunities: if graph $\mathcal{G}$ can be transformed into graph $\mathcal{G}'$ using a sequence of substitutions, then $\mathcal{G}$ can always be transformed into $\mathcal{G}'$ after pruning (possibly using a different set of transformations).

Input tensor renaming. TASO eliminates graph substitutions identical to other substitutions modulo input tensor renaming. For example, Figure 2.4a shows a redundant substitution equivalent to Figure 2.2a by renaming input tensor C with A. For substitutions that are equivalent through input tensor renaming, TASO prunes all but a single most general substitution.

Common subgraph. TASO also tries to eliminate substitutions whose graphs have a common subgraph. TASO identifies two forms of common subgraphs that can lead to pruning.

The first form of common subgraph is illustrated in Figure 2.4b. Here, the two graphs of a substitution both contain a common operator with the same input tensors (highlighted in gray boxes). The common subgraph represents an input to other operators in both graphs of a substitution. Therefore, we can obtain a more general substitution by replacing the common subgraph with a fresh input tensor. If this more general substitution is indeed valid, then TASO prunes the less general substitution.

The second form of common subgraph is demonstrated in Figure 2.4c. Here, the common subgraph (highlighted in gray boxes) includes all the outputs in both graphs of a the substitution. In this case, a more general substitution can be obtained by completely removing the common subgraph, making



(a) A redundant substitution that is equivalent to Figure 2.2a by renaming input tensor C with A in Figure 2.2a.

(b) A redundant substitution with a common subgraph.

(c) A redundant substitution with a common subgraph.

Figure 2.4: Example redundant substitutions pruned by TASO. `Matmul` and `Add` refer to matrix multiplication and element-wise addition, respectively. For each subgraph, A, B, and C refer to its input tensors, while X refers to the output tensor.

its inputs new outputs of the two graphs. TASO prunes the less general substitution if the more general one is valid.

Table 2.3 shows the effect of the TASO pruning techniques on the number of substitutions. We observe that both pruning techniques play an important role in eliminating redundant substitutions and their combination reduces the number of substitutions TASO must consider by 39×.

Table 2.3: The number of remaining graph substitutions after applying the pruning techniques in order.

Pruning Techniques	**Remaining Substitutions**	**Reduction v.s. Initial**
Initial	28744	1×
Input tensor renaming	17346	1.7×
Common subgraph	43	39×

2.6 Cost Model

We introduce a cost model that incorporates multiple dimensions to evaluate the runtime performance of a computation graph. The cost model computes metrics for each operator in a graph and combines them appropriately to obtain a total cost. This includes both metrics that can be computed statically (e.g., FLOPs, memory usage, and number of kernel launches) as well as dynamic metrics that usually require measurements on specific hardware (e.g., execution time on a particular GPU or CPU). The design of the cost model is motivated by the fact that most DNN operators perform dense linear algebra with no branches, and therefore their performance on hardware is highly consistent and predictable given the same data layouts and configuration parameters (e.g., the strides and padding of a convolution). TASO measures the execution time of a DNN operator once for each configuration and data layout, and estimates the performance of a graph by summing up the measured execution time of its operators.

Our cost model can optimize a single cost dimension (e.g., minimizing overall FLOPs) as well as incorporate multiple cost dimensions, such as minimizing execution time while maintaining a memory usage limit (by returning an infinite cost if the memory usage limit is exceeded). We observe that many graph substitutions result in a tradeoff among several cost dimensions instead of improving all of them. Users can directly encode a desired tradeoff in the cost model, and TASO's search algorithm can automatically discovers optimized computation graphs that best satisfy these requirements.

2.7 Joint Optimizer

We now describe the TASO optimizer for jointly optimizing data layout and graph substitution. The optimizer uses a *cost-based backtracking search* algorithm to search for an optimized computation graph by applying verified substitutions. The search algorithm also considers possible layout optimization opportunities when performing substitutions.

When applying a substitution on a matched subgraph, based on the data layouts of tensors in the original graph and the layouts supported by the operators, TASO enumerates possible layouts for tensors in the target graph. As a result, applying a substitution on a matched computation graph may result in multiple graphs with identical graph structure but different data layouts.

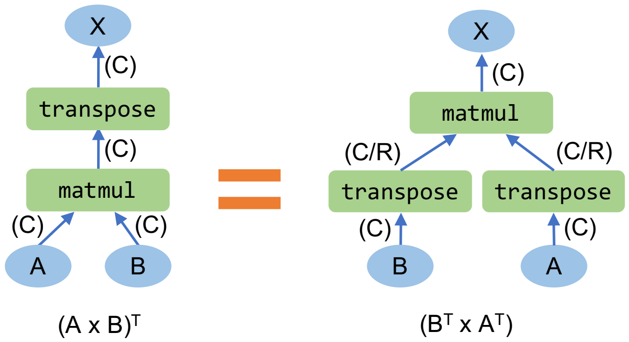


Figure 2.5: A graph substitution using the transpose of matrix multiplication. `matmul` and `transpose` indicate matrix multiplication and transpose, respectively. The parentheses show the potential layouts for each tensor in the substitution, where `C` and `R` indicate the column-major and row-major layouts of a tensor.

For example, Figure 2.5 shows the potential computation graphs that can be derived by applying the transpose of matrix multiplication on an input graph with a default column-major layout (shown as `C`). Both the matrix multiplication and transpose operators also support an alternative row-major layout (shown as `R`). The data layouts for all mapped tensors in the new graph (i.e., A, B, and X) must match the layouts in the original graph. The two intermediate tensors in the target graph can have either a row-major or a column-major layout, therefore TASO considers four different computation graphs (i.e., `CC`, `CR`, `RC`, and `RR` for the two intermediate tensors) when applying this substitution. This allows TASO to capture potential layout transformation opportunities when performing graph substitutions.

Algorithm 2 shows our *cost-based backtracking search* algorithm for jointly optimizing substitution and data layout. To search for an optimized graph, all candidate graphs are maintained in a priority queue $\mathcal{P}$ and are dequeued in increasing order of cost. For each dequeued graph $\mathcal{G}$, TASO considers each verified substitution and possible layouts applicable to the substitution, and applies them to obtain functionally equivalent new graphs $\mathcal{G}'_1, ..., \mathcal{G}'_n$ that are added to $\mathcal{P}$.

A non-obvious property of graph substitutions is that applying them can introduce cycles into a graph. Figure 2.6 shows one example where applying a valid substitution results in a cyclic graph. Since computation graphs must be acyclic, TASO checks the acyclicity of $\mathcal{G}'$ (line 12 of Algorithm 2) before enqueuing it in $\mathcal{P}$.

Finally, the best discovered graph $\mathcal{G}_{opt}$ is returned by the search algorithm. The search space is pruned by a hyper parameter α, which eliminates all graphs whose cost is α times worse than the best discovered graph. The parameter α trades off between the search time and the best discovered graph. Setting $\alpha = 1$ reduces the search to a simple greedy algorithm without backtracking, and a high value for α makes the search explore more possible candidates and causes more backtracking. We observe that $\alpha = 1.05$ achieves good performance in our evaluation.

Algorithm 2 Cost-Based Backtracking Search

```
Input: an input graph $\mathcal{G}_{in}$, verified substitutions $\mathcal{S}$, a cost model $Cost(\cdot)$, and a hyper parameter $\alpha$.
Output: an optimized graph.

$\mathcal{P} = \{\mathcal{G}_{in}\}$ // $\mathcal{P}$ is a priority queue sorted by Cost.
while $\mathcal{P} \neq \{\}$ do
    $\mathcal{G} = \mathcal{P}$.dequeue()
    for substitution $s \in \mathcal{S}$ do
        // Layout($\mathcal{G}, s$) returns possible layouts applying $s$ on $\mathcal{G}$.
        for layout $l \in$ Layout($\mathcal{G}, s$) do
            // Apply($\mathcal{G}, s, l$) applies $s$ on $\mathcal{G}$ with layout $l$.
            $\mathcal{G}' =$ Apply($\mathcal{G}, s, l$)
            if $\mathcal{G}'$ is valid then
                if $Cost(\mathcal{G}') < Cost(\mathcal{G}_{opt})$ then
                    $\mathcal{G}_{opt} = \mathcal{G}'$
                end if
                if $Cost(\mathcal{G}') < \alpha \times Cost(\mathcal{G}_{opt})$ then
                    $\mathcal{P}$.enqueue($\mathcal{G}'$)
                end if
            end if
        end for
    end for
end while
return $\mathcal{G}_{opt}$
```

2.8 Implementation

TASO is designed and implemented as a generic and extensible computation graph optimizer for tensor computations, such that new tensor operators can be easily added. Section 2.4 lists the tensor operators included in the current implementation of TASO. Some operators also depend on additional parameters to determine the behavior of the operator, such as the strides, padding, and activation of a convolution. In addition to operators, TASO also includes four types of constant tensors that are useful in substitutions. In particular, $\texttt{I}_{\texttt{ewmul}}$, $\texttt{I}_{\texttt{matmul}}$, and $\texttt{I}_{\texttt{conv}}$ are identity tensors for element-wise multiplication, matrix multiplication, and convolution, respectively. $\texttt{C}_{\texttt{pool}}$ is used to convert an average pooling operator to a depth-wise convolution (see examples in Section 2.9.3).

As explained in Section 2.4, TASO uses operator properties specified by the user to verify the generated graph substitutions. Table 2.2 lists the 43 properties used to verify all substitutions in our evaluation.

TASO can easily be extended to include new tensor operators. For each operator, TASO requires two forms of input: (1) reference implementations for the operator, and (2) specifications of operator properties. (1) consists of a concrete implementation (in C++) used by the substitution generator and a symbolic implementation (in Python) used to validate the operator specifications. In our experience, adding a new operator requires a few hours of work by an expert.

For a new operator whose specifications are currently missing, TASO treats it as an opaque

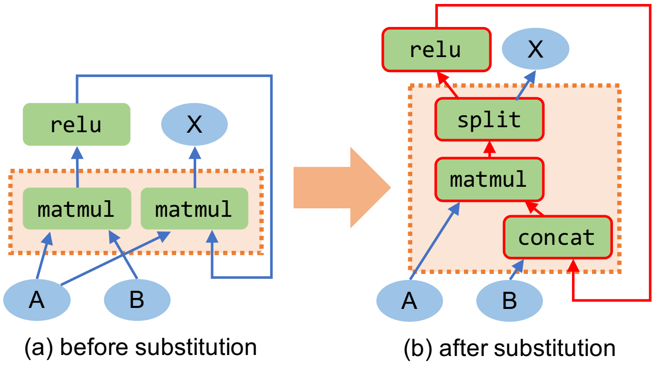


Figure 2.6: A graph substitution example that introduces a cycle into a computation graph, where A, B are the inputs, and X is the output. The original graph computes $A \times \texttt{relu}(A \times B)$, and the new graph is the result of applying the substitution shown in Figure 2.2b that fuses the two matrix multiplications using concatenation and split. The two graphs of the substitution are shown in the dotted boxes. Both the original graph and the substitution are acyclic. However, the resulting graph contains a cycle (highlighted in red).

operator and can still optimize the rest of the graph using verified substitutions.

Overall, our implementation of TASO contains around 8,000 lines of code for the core components (i.e., the substitution generator, verifier, and optimizer), and 1,400 lines of code for the operator reference implementations, including the 43 operator properties.

TASO is framework-agnostic and can be plugged in to existing DNN frameworks such as TensorRT and TVM by simply emitting the optimized graph in the target framework's input format. In the evaluation, we demonstrate this portability on TensorRT and TVM, and show that they can directly use TASO's optimizations to improve performance.

2.9 Evaluation

In this section we aim to evaluate the following points:

- Can TASO automatically generate and verify graph substitutions in acceptable run time?
- Can TASO improve the end-to-end performance of real-world DNN architectures, especially for emerging architectures with recently introduced operators?
- Can TASO's joint optimization of computation graphs and data layouts achieve better performance than separate optimizations?

2.9.1 Experimental Setup

DNNs. We use five real-world DNN architectures to evaluate TASO. ResNet-50 [39] is a widely used

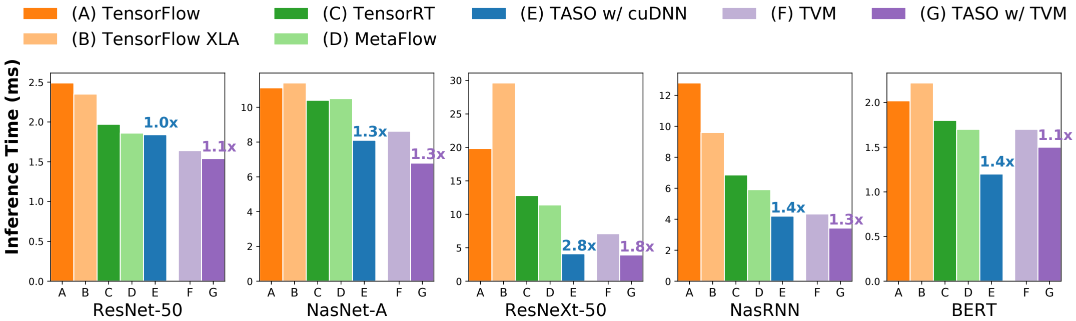


Figure 2.7: End-to-end inference performance comparison among existing DNN frameworks and TASO. The experiments were performed using a single inference sample, and all numbers were measured by averaging 1,000 runs on a NVIDIA V100 GPU. We evaluated the TASO's performance with both the cuDNN and TVM backends. For each DNN architecture, the numbers above the TASO bars show the speedup over the best existing approach with the same backend.

convolutional neural network for image classification and achieved the best classification performance in the ILSVRC [71] competition. ResNeXt-50 [88] improves the model accuracy and runtime efficiency of ResNet-50 by introducing a new grouped convolution operator. NasNet-A [95] and NasRNN [94] are two DNN architectures automatically discovered by machines through neural architecture search. NasNet-A and NasRNN exceed the best human-designed DNN architectures for image classification and language modeling tasks, respectively. Finally, BERT [28] is a language representation architecture that obtained the state-of-the-art model accuracy on a spectrum of language tasks.

All experiments were performed on an Amazon p3.2xlarge instance [1] with an 8-core Intel E5-2600 CPU, 64 GB DRAM, and one NVIDIA Tesla V100 GPU.

To generate candidate graph substitutions, TASO enumerates all potential graphs with up to four operators by using all DNN operators listed in Section 2.4. TASO generated 43 candidate substitutions in around 5 minutes.

In the cost-based backtracking search for optimized DNN graphs, we set the hyperparameter α to be 1.05. In all experiments, the end-to-end search time to discover an optimized computation graph is less than ten minutes.

2.9.2 End-to-End Evaluation

We first compare the end-to-end inference performance between TASO and existing DNN graph optimizers, including TensorFlow [11], TensorFlow XLA [3], TensorRT [80], TVM [19], and MetaFlow [45], which is an earlier version of TASO (see Section 2.10). Figure 2.7 shows the results. TensorFlow, TensorFlow XLA, TensorRT, and MetaFlow use the highly-engineered cuDNN and cuBLAS libraries [21, 24] to perform DNN operators on GPUs, while TVM generates customized GPU kernels for the DNN operators. To eliminate the impact of different operator libraries, we evaluate the performance of TASO on both backends.

To generate GPU kernels in TVM, we allow the auto tuner [20] to run 2000 trials and use the best discovered configuration for each DNN operator. It takes 2 hours on average to tune a GPU kernel for each DNN operator. The TASO graph optimizer needs to query the execution time of hundreds of DNN operators for its cost model, therefore, for the TVM backend, we reuse the best discovered computation graph for the cuDNN backend, assuming the cost of an operator in cuDNN is a reasonable estimate for its cost in TVM.

Among the five DNN architectures, ResNet-50 has been commonly used and heavily optimized by existing DNN frameworks. TASO achieves on-par performance for ResNet-50 with existing frameworks, showing that TASO is able to automatically discover graph substitutions manually designed by domain experts. For the remaining four DNN architectures with new operators and graph structures, TASO outperforms existing DNN frameworks with speedups ranging from 1.3× to 2.8× on the cuDNN backend and 1.1× to 1.8× on the TVM backend. The speedup is achieved by (1) automatically discovering optimizing substitutions for the new operators and (2) jointly optimizing graph substitution and data layout. We analyze the substitutions discovered by TASO in Sections 2.9.3 and 2.9.4, and the joint optimization of substitution and data layout in Section 2.9.5.

2.9.3 Substitution Case Study

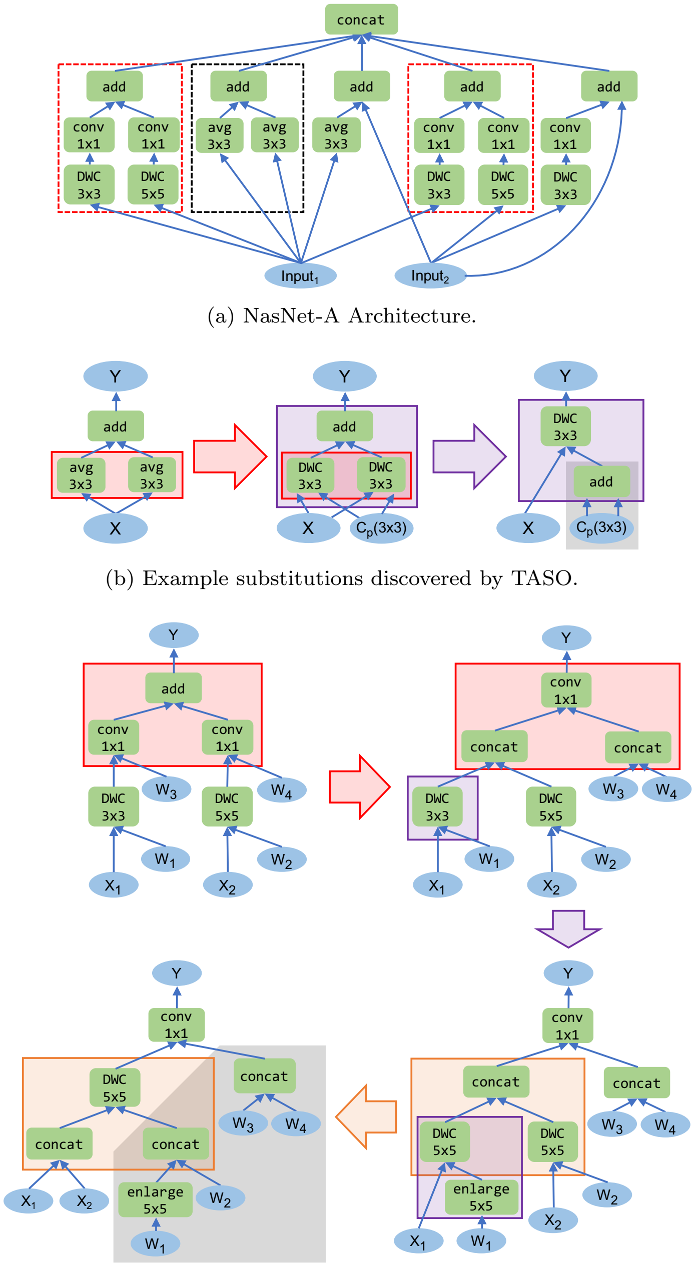
To understand how the substitutions generated and verified by TASO improve runtime performance, we study a few graph substitution examples in detail.

NasNet-A is the best discovered CNN architecture for the CIFAR-10 dataset, obtained by neural architecture search. Figure 2.8a shows a convolutional cell in NasNet-A. Unlike human-designed architectures, NasNet-A contains unconventional graph structures, making it hard to optimize with manual substitutions designed for more standard DNN architectures. To illustrate how TASO optimizes this architecture, we show two example substitutions discovered by TASO; neither is present in any existing DNN framework.

Figure 2.8b shows graph substitutions that transform two average pooling operators followed by element-wise addition to a single depth-wise convolution, by using a constant tensor $\texttt{C}_{\texttt{pool}}$ defined in Section 2.4. The mathematical definition of average pooling is:

$$output(n, c, x, y) = \frac{1}{K_X \times K_Y} \sum_{k_x} \sum_{k_y} input(n, c, x + k_x, y + k_y)$$

where *input* and *output* denote the input and output tensors of an average pooling, respectively, and K_X and K_Y are the height and width of the pooling filter. Similarly, the formula for depth-wise

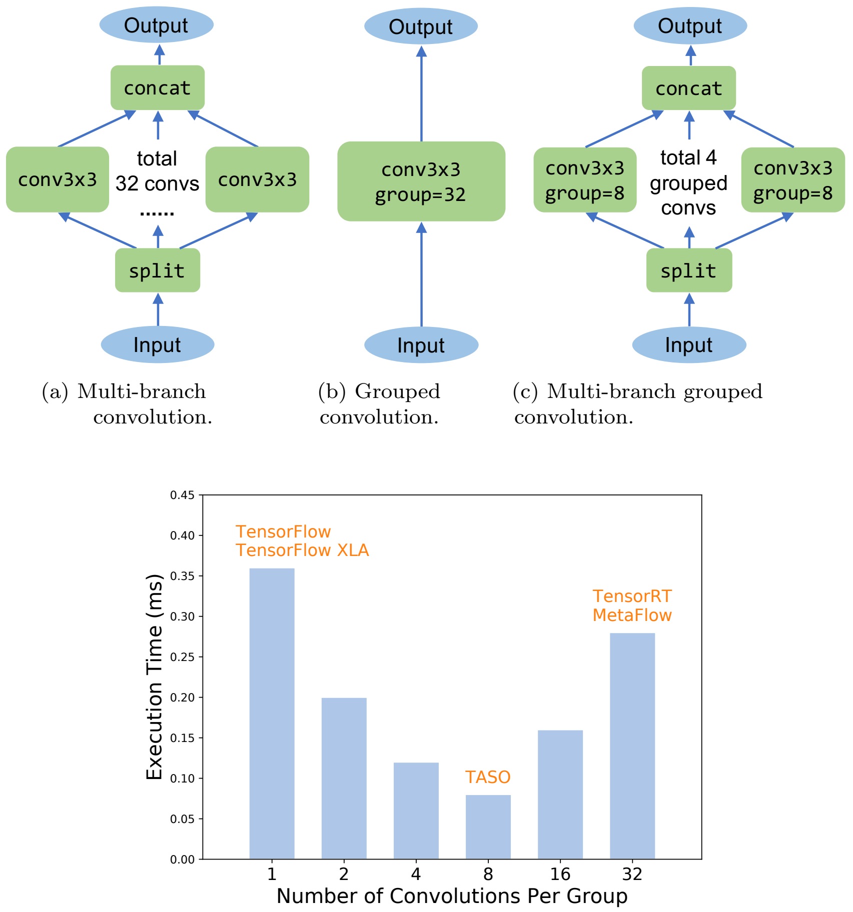


(a) NasNet-A Architecture.

(b) Example substitutions discovered by TASO.

(c) A sequence of substitutions discovered by TASO.

Figure 2.8: The NasNet-A architecture [95] and substitutions discovered by TASO to optimize NasNet-A. Figure 2.8a shows the architecture, where `avg`, `conv`, and `DWC` refer to average pooling, convolution, and depth-wise convolution, respectively. The weight tensors are eliminated for simplicity. Figures 2.8b and 2.8c shows two sequences of substitutions discovered by TASO that are used to optimize subgraphs marked in the black and red boxes in Figure 2.8a. In Figures 2.8b and 2.8c, each arrow refers to a substitution, and the subgraphs in the same color are the graph pair of the substitution. $\mathtt{C}_{\mathtt{pool}}(3 \times 3)$ in Figure 2.8b is a constant matrix whose entries are 1/9, as defined in Section 2.4. The `enlarge` operator in Figure 2.8c increases a convolution's kernel size by padding the weight (i.e., W_1) with extra 0's. For inference, operators in the gray areas in Figures 2.8b and 2.8c only depend on pre-trained weights (i.e., W_i), and therefore can be pre-computed.



(a) Multi-branch convolution.

(b) Grouped convolution.

(c) Multi-branch grouped convolution.

(d) Performance comparison.

Figure 2.9: Different approaches to perform multi-batch convolutions in ResNeXt-50 and their performance comparison. TensorFlow and TensorFlow XLA launch the 32 convolutions separately (Figure 2.9a). TensorRT and MetaFlow launch a single grouped convolution kernel that computes all 32 convolutions in parallel (Figure 2.9b). The best graph discovered by TASO uses 4 grouped convolutions, each of which computes 8 convolutions (Figure 2.9c).

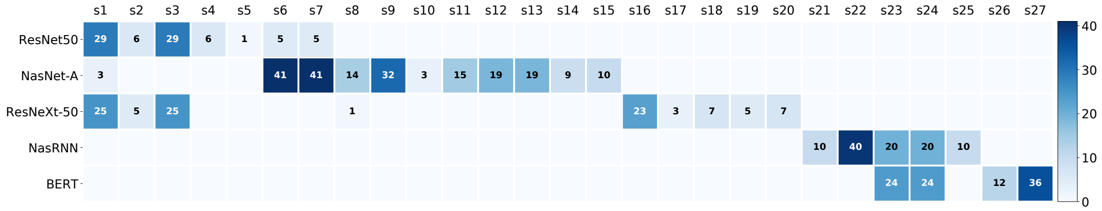


Figure 2.10: A heat map of how often the verified substitutions are used to optimize the five DNN architectures. Only substitutions used in at least one DNN are listed. For each architecture, the number indicates how many times a substitution is used by TASO to obtain the optimized graph.

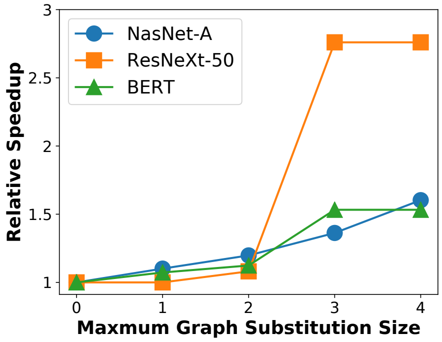


Figure 2.11: Performance comparison by using graph substitutions with different size limits. The y-axis shows the relative speedups over the input computation graphs.

convolution is:

$$output(n, c, x, y) = \sum_{k_x} \sum_{k_y} input(n, c, x + k_x, y + k_y) \times weight(c, k_x, k_y)$$

which produces a mathematically equivalent result as an average pooling if we have $weight(c, k_x, k_y) = 1/(K_X \times K_Y)$. In addition, TASO also fuses the two depth-wise convolutions into one using its linearity.

A second new sequence of substitutions for NasNet-A is shown in Figure 2.8c, which fuses two depth-wise convolutions and two convolutions followed by addition to a depth-wise convolution followed by a standard convolution. This substitution increases the operator granularity and reduces the operator launch overhead by using larger operators.

For inference workloads, the weights in DNN architectures (e.g., W_i and $\mathtt{C_{pool}}$ in Figure 2.8) are fixed and independent of the inputs. TASO preprocesses operators whose inputs are all pre-trained weights (e.g., the gray areas in Figure 2.8) to further reduce the inference time.

ResNeXt-50 replaces large convolutions in ResNet-50 with multiple branches of much smaller convolutions to improve both model accuracy and runtime efficiency, as shown in Figure 2.9a. However, directly launching these small convolutions incurs high kernel launch overhead. The cuDNN library has recently introduced grouped convolution kernels that perform multiple convolutions in parallel using a single CUDA kernel [21]. TensorFlow and TensorFlow XLA (r1.14 as of August 2019) currently do not support grouped convolution, so the fastest available ResNeXt-50 implementation in TensorFlow launches convolutions in multiple branches separately with the resulting high kernel launch overhead. TensorRT and MetaFlow use a single grouped convolution kernel that computes a group of 32 convolutions in parallel. While grouped convolution enables additional parallelism and reduces kernel launch overhead, it also requires a larger cache to save intermediate states for all convolutions, which results in decreased runtime performance when too many convolutions are aggregated in a single kernel. Figure 2.9d gives the time to run all 32 convolutions using different group sizes (i.e., the number of convolutions in a group), showing that neither launching individual convolutions nor grouping all 32 convolutions is the best option.

Existing frameworks either launch 32 individual convolutions or a single grouped convolution, both of which result in suboptimal performance. For ResNeXt-50, TASO uses a mixture of previous approaches and launches multiple grouped convolutions, as shown in Figure 2.9c. TASO discovered this mixture automatically, resulting in a speedup of 2.8× compared to the best existing approach.

2.9.4 Analysis of Used Substitutions

We now present a detailed analysis of how the graph substitutions discovered by TASO impact the performance of the optimized graphs. Figure 2.10 shows a heat map of the substitutions used to optimize each of the five DNN architectures. Each DNN uses 4-10 different substitutions to achieve optimized performance, and different DNNs require different sets of substitutions. This shows the difficulty of manually designing a few core substitutions to optimize today's DNN architectures with increasingly high diversity. TASO is better positioned for optimizing new DNNs by automatically discovering performance critical substitutions.

Additionally, we evaluate the scalability of TASO by considering substitutions with different size limitations, and measuring the runtime performance of the optimized graphs. Figure 2.11 shows the results. For all three DNN architectures, performance improvement is consistently achieved by using larger substitutions up to size 3. ResNeXt-50 and BERT do not obtain additional speedups by using substitutions with 4 operators, while NasNet-A achieves 1.2× by considering larger substitutions. Our current implementation of TASO does not scale to generate all substitutions with 5 or more operators, since the generator is limited by the memory needed to hold the fingerprints of all potential graphs, which scales exponentially with graph size. A distributed fingerprint generator could potentially handle graphs of size 5 and even more, which we leave as future work.

2.9.5 Joint Optimization of Graph Substitutions and Data Layout

To evaluate the performance of the joint optimization in TASO, we compare the joint optimization with three baseline strategies: (1) performing only graph substitution optimizations; (2) performing only data layout optimizations; and (3) performing the two optimizations sequentially.

Figure 2.12 shows the comparison results among the four strategies on BERT. TASO outperforms the three baseline strategies by 1.2-1.3×. We observe that the speedup is achieved by using graph substitutions that transform both graph structure and data layout. One example is depicted in Figure 2.5. The most time consuming operation in BERT is matrix multiplication $A \times B$, where A is 64 by 1024 and B is 1024 by 4096. In cuBLAS, the transposed version of this matrix multiplication (i.e., $(B^T \times A^T)^T$) achieves 1.5× speedup when B^T and A^T are in the column-major and row-major

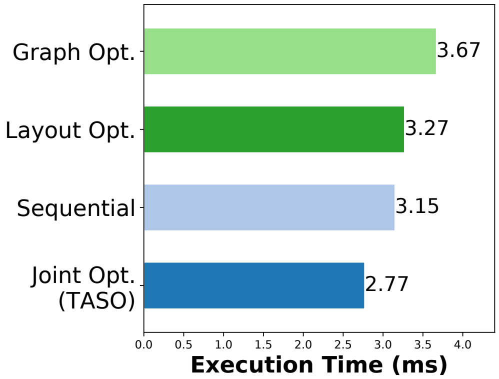


Figure 2.12: End-to-end inference performance comparison on BERT using different strategies to optimize graph substitution and data layout.

layout, respectively. This graph optimization can only be captured when graph substitution and data layout are jointly considered.

2.9.6 Graph Substitution Verifier

We evaluate the performance of the graph substitution verifier on its two key tasks: verifying generated substitutions against operator specifications and validating the operator specifications themselves to aid in the development process (Section 2.4). Our implementation uses Z3 [26] to automatically discharge all proof obligations, and our experiments were performed with Z3 version 4.8.5.

Generating the 743 graph substitutions takes around 5 minutes, and verifying them against the 43 specified operator properties takes less than 10 minutes. When checking the specification for redundancies we use Z3 to search for a proof of an invalid formula (stating that a specified property is entailed by the rest of the specification). This search can continue indefinitely, and in our evaluation we used a timeout of 10 seconds per query, resulting in a run time of less than 10 minutes (for 43 axioms). During the development process, when we had some redundant specifications they were discovered in a few seconds.

The validation check that verifies the operator specification for all combinations of parameter values and tensor sizes up to 4×4×4×4 is more computationally expensive, with roughly one million proof obligations. We parallelized it using 128 CPU cores, which resulted in a run time of roughly one hour. During the development process, we also found it useful to verify the operators for more restricted combinations. For example, verifying the specification for tensors of size *exactly* 4×4×4×4 (rather than *all* tensors up to that size) takes under 10 minutes using a single CPU core.

2.10 Related Work

Manually designed graph substitutions are used in existing DNN frameworks to optimize DNN architectures. For example, TensorFlow, TensorRT, and TVM use a greedy rule-based strategy and directly perform all applicable substitutions on an input graph [11, 80, 19].

MetaFlow [45] is an earlier version of this work. It differs from previous graph optimizers in that it allows substitutions that may either increase or decrease performance to enable a larger search space of equivalent computation graphs and uses back-tracking search to explore this space, but it still requires manually specified substitutions.

The key difference between TASO and these frameworks is that TASO can automatically generate candidate substitutions, and also provides semi-automatic support for verifying their correctness. In the evaluation, we also show that existing frameworks can directly use TASO's optimized graphs to improve performance.

Automated DNN code generation. Recent work has proposed various approaches to generate hardware-specific code for DNN operators. For example, TVM [19, 20] uses a learning-based approach and automatically generates low-level optimized code for a diverse set of hardware backends. Astra [75] optimizes DNN computation by exploring the optimization space of multi-version complication during training. Compared to these approaches, TASO aims at optimizing DNN computation at a higher graph level, and therefore TASO's optimizations are orthogonal and can be combined with code generation techniques. It still remains an open problem of how to jointly optimize DNN computation at both graph-level and operator-level.

Automated DNN parallelization. ColocRL [61] uses reinforcement learning to automatically discover an efficient device placement for parallelizing DNN training across multiple GPUs. FlexFlow [46, 44] introduces a comprehensive search space of parallelization strategies for DNN training, and uses a randomized search algorithm to find efficient strategies in the search space. These frameworks optimize distributed DNN training assuming a fixed computation graph. We believe it is possible to combine TASO's graph optimizations with training parallelization techniques.

Superoptimization is a compiler optimization technique that was originally designed to find the optimal code for a sequence of instructions [60]. TASO's approach to identifying potential substitutions via enumeration of graphs and fingerprinting is similar to work in automatically generating peephole optimizers using superoptimization techniques [14]. TASO's approach to verification, however, is significantly different. Verification in superoptimization typically relies on "bit blasting", that is, modeling every bit in a computation explicitly in a logical formula (e.g., as a boolean variable). This approach is possible only when all aspects of a program transformation, including the computation and the data, can be expressed using a known number of bits. For TASO, where the input tensor sizes for graph substitutions are unknown, we must take a different approach. While not fully automatic like verification via bit blasting, our methodology based on writing operator specifications is much more flexible in being able to model future operators with almost arbitrary semantics, in addition to smoothly handling the issue of unknown tensor dimensions and split points.

Data layout optimizations. Existing DNN frameworks that support data layout optimizations treat data layouts and graph transformations as separate optimization problems [19, 56, 62]. TASO formulates the problem of performing graph substitutions and deciding the data layout of each DNN operator as a joint optimization problem and considers layout conversions as a part of graph substitutions. As a result, TASO can automatically generate graph substitutions that optimize both graph structures and data layouts, and our evaluation shows that jointly optimizing the two tasks can significantly improve the end-to-end performance, compared to optimizing the them separately.

2.11 Limitations

One limitation of TASO is the reliance on user provided operator properties. While our experience has been that the required effort is manageable, it would be better to eliminate it altogether. One possible approach is to automatically verify substitutions directly against the implementations of the operators, e.g., cuDNN kernels.

Another limitation of TASO is the scalability of the generator, which requires saving the fingerprints of all computation graphs up to a fixed and is exponential in the size of the graphs. To scale beyond size 4, a distributed generator (or a very large memory machine) would be needed. A second possibility is to replace the brute-force enumeration with more memory-efficient algorithms.

An additional avenue for future research is combining graph-level and operator-level optimizations. This joint optimization is challenging as both problems involve large and complex search spaces, and optimizations at one level affect the search space of the other.

2.12 Conclusion

TASO is a ML computation graph optimizer that automatically generates graph substitutions. TASO formally verifies the substitutions, and considers graph substitutions and layout transformations together as a joint optimization problem, exploiting more optimization opportunities. TASO matches the performance of existing frameworks on DNNs for which these frameworks have been heavily optimized such as ResNet-50, and outperforms existing frameworks by up to 2.8× on other DNNs, finding novel optimizations not present in the hundreds of optimization rules in existing frameworks. TASO achieves these results with dramatically less human effort than existing frameworks, and provides a higher level of correctness guarantees.

Chapter 3

Automated Discovery of Parallelization Optimizations

Existing deep learning systems commonly parallelize deep neural network (DNN) training using data or model parallelism, but it turns out that these strategies do not capture all of the possible ways to parallelize a DNN. In this chapter, we introduce SOAP, a more comprehensive search space of parallelization strategies for distributed DNN training that includes strategies to parallelize a DNN in the Sample, Operator, Attribute, and Parameter dimensions. We present FlexFlow, a deep learning engine that uses guided randomized search of the SOAP space to find a fast parallelization strategy for a specific parallel machine. To accelerate this search, FlexFlow introduces a novel execution simulator that can accurately predict a parallelization strategy's performance and is three orders of magnitude faster than prior approaches that execute each strategy. Our evaluation on real-world DNN benchmarks shows that FlexFlow can automatically discover strategies that significantly outperform existing manually designed parallelization strategies and improve scalability.

3.1 Motivation

It is now standard practice to parallelize ML training across distributed heterogeneous clusters [27, 11]. Although ML models and the clusters used to parallelize them are increasingly complex, the strategies used by today's ML systems to parallelize training remain simple, and often suboptimal.

The most common parallelization strategy is *data parallelism* [51], which places a replica of the entire neural network on each device, so that each device processes a subset of the training data, and synchronizes network parameters across replicas at the end of an iteration. Data parallelism is efficient for compute-intensive operators with a few trainable parameters (e.g., convolution) but achieves suboptimal parallelization performance for operators with a large number of parameters (e.g., embedding layers).

Another common parallelization strategy is *model parallelism* [27], which assigns disjoint subsets of a neural network each to a dedicated device. This approach eliminates parameter synchronization between devices but requires data transfers between operators. ColocRL [61] uses reinforcement learning to learn efficient operator assignments for model parallelism but only explores parallelism in the operator dimension.

We introduce a more comprehensive *SOAP* (Sample-Operator-Attribute-Parameter) search space of parallelization strategies for DNNs that generalizes and goes beyond previous approaches. The *operator* dimension describes how different operators in a DNN are parallelized. For a single operator, the *sample* and *parameter* dimensions indicate how training samples and model parameters are distributed across devices. Finally, the *attribute* dimension defines how different attributes within a sample are partitioned (e.g., the height and width dimensions of an image).

We use SOAP in FlexFlow, a deep learning engine that automatically finds fast parallelization strategies in the SOAP search space for arbitrary DNNs. Existing approaches only consider one or a subset of SOAP dimensions. FlexFlow considers parallelizing any DNN (linear or non-linear) in all SOAP dimensions and explores a more comprehensive search space that includes existing approaches as special cases. As a result, FlexFlow is able to find parallelization strategies that significantly outperform existing approaches.

The key challenge FlexFlow must address is how to efficiently explore the SOAP search space, which is much larger than those considered in previous systems and includes more sophisticated parallelization strategies. To this end, FlexFlow uses two main components: a fast, incremental execution simulator to evaluate different parallelization strategies, and a Markov Chain Monte Carlo (MCMC) search algorithm that takes advantage of the incremental simulator to rapidly explore the large search space.

FlexFlow's *execution simulator* can accurately predict the performance of a parallelization strategy in the SOAP search space for arbitrary DNNs and is three orders of magnitude faster than profiling real executions. We borrow the idea from Chapter 2 of measuring the performance of an operator once for each configuration and feed these measurements into a *task graph* that models both the architecture of a DNN model and the network topology of a cluster. The execution simulator estimates the performance of a parallelization strategy by simulating the execution on the task graph. In addition, we introduce a *delta simulation algorithm* that simulates a new strategy using incremental updates to previous simulations and further improves performance over naive simulations by up to 6.9×.

The execution simulator achieves high accuracy for predicting parallelization performance. We evaluate the simulator with six real-world DNNs on two different GPU clusters and show that, for all the measured executions, the relative difference between the real and simulated execution time is less than 30%. Most importantly for the search, we test different strategies for a given DNN and show

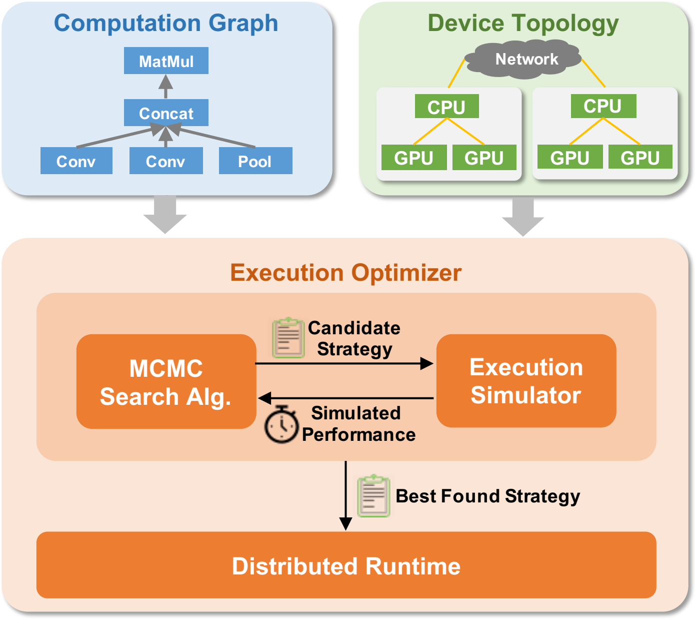


Figure 3.1: FlexFlow overview.

that their simulated execution time preserves real execution time ordering.

Using the execution simulator as an oracle, the FlexFlow *execution optimizer* uses a MCMC search algorithm to explore the SOAP search space and iteratively propose candidate strategies based on the simulated performance of previous candidates. The execution simulator can also work with other search strategies, such as learning-based search algorithms. When the search procedure is finished, the execution optimizer returns the best strategy it has discovered.

We evaluate FlexFlow on six real-world DNN benchmarks including AlexNet [51], ResNet-50 [39], Inception-v3 [78], RNN Text Classification [48], RNN Language Modeling [92] and Neural Machine Translation [87]. Compared to data/model parallelism and strategies manually designed by domain experts [50, 87], FlexFlow increases training throughput by up to 3.3×, reduces communication costs by up to 5×, and achieves significantly better scaling. In addition, we also evaluate FlexFlow on two large DNN models (DLRM [66] and Candle Uno [9]) that cannot scale under existing manually designed parallelization strategies. By automatically discovering more efficient strategies, FlexFlow improves the overall training throughput of DLRM and Candle Uno by up to 10× and 7.8×, respectively.

3.2 The Approach

This section describes FlexFlow for automatically discovering fast parallelization strategies for distributed ML training.

Table 3.1: Parallelizable dimensions for different operators. The *sample* and *channel* dimension index different samples and neurons, respectively. For images, the *length* and the combination of *height* and *width* dimensions specify a position in an image.

Operator	**Parallelizable Dimensions**		
	(S)ample	**(A)ttribute**	**(P)arameter**
1D pooling	sample	length, channel	
1D convolution	sample	length	channel
2D convolution	sample	height, width	channel
Matrix multiplication	sample		channel

In addition to taking the computation graph of an ML model as an input, FlexFlow also takes a *device topology* graph $\mathcal{D} = (\mathcal{D}_N, \mathcal{D}_E)$ describing all available hardware devices and their interconnections, as shown in Figure 3.1. Each node $d_i \in \mathcal{D}_N$ represents a device (e.g., a CPU or a GPU), and each edge $(d_i, d_j) \in \mathcal{D}_E$ is a hardware connection (e.g., a NVLink, a PCI-e, or a network link) between device d_i and d_j. The edges are labeled with the bandwidth and latency of the connection.

The main components of FlexFlow are shown in Figure 3.1. The *execution optimizer* uses a *MCMC search algorithm* to explore the space of possible parallelization strategies and iteratively proposes candidate strategies that are evaluated by an *execution simulator*. The execution simulator uses a *delta simulation algorithm* that simulates a new strategy using incremental updates to previous simulations. The simulated execution time guides the search in generating future candidates. When the search time budget is exhausted, the execution optimizer sends the best discovered strategy to a *distributed runtime* for parallelizing the actual executions.

3.3 The SOAP Search Space

This section introduces the SOAP search space of parallelization strategies for DNNs. To parallelize a DNN operator across devices, we require each device to compute a disjoint subset of the operator's output tensors. Therefore, we model the parallelization of an operator o_i by defining how the output tensor of o_i is partitioned.

For an operator o_i, we define its *parallelizable dimensions* $\mathcal{P}_i$ as the set of all divisible dimensions in its output tensor. $\mathcal{P}_i$ always includes a *sample* dimension. For all other dimensions in $\mathcal{P}_i$, we call it a *parameter* dimension if partitioning over that dimension requires splitting the model parameters and call it an *attribute* dimension otherwise. Table 3.1 shows the parallelizable dimensions of some example operators. Finally, we also consider parallelism across different operators in the *operator* dimension.

A *parallelization configuration* c_i of an operator o_i defines how the operator is parallelized across multiple devices. Figure 3.2 shows some example configurations for parallelizing a 1D convolution

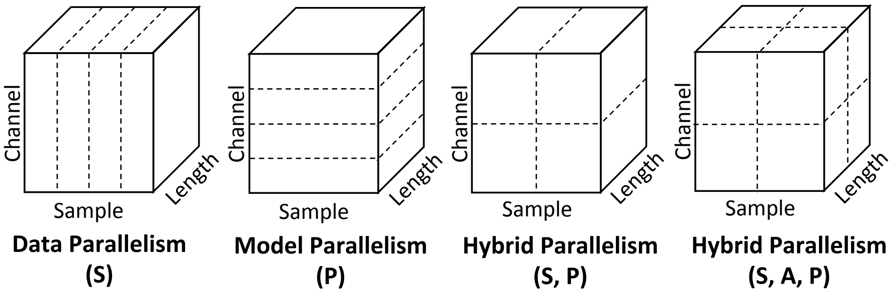


Figure 3.2: Example parallelization configurations for 1D convolution. Dashed lines show partitioning the tensor.

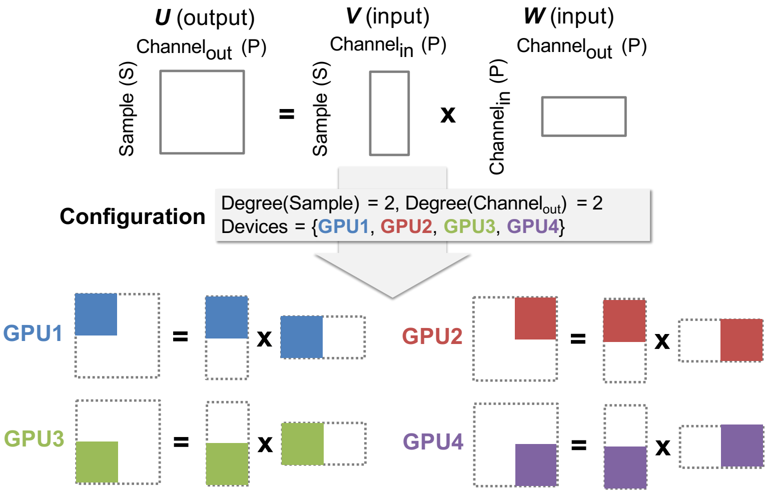


Figure 3.3: An example parallelization configuration for a matrix multiplication operator.

operator in a single dimension as well as combinations of multiple dimensions.

For each parallelizable dimension in $\mathcal{P}_i$, c_i includes a positive integer that is the degree of parallelism in that dimension. $|c_i|$ is the product of the parallelism degrees for all parallelizable dimensions of c_i. We use equal size partitions in each dimension to balance workload distributions. A parallelization configuration c_i partitions the operator o_i into $|c_i|$ independent *tasks*, denoted as $t_{i:1}, ..., t_{i:|c_i|}$, meanwhile c_i also includes the device assignment for each task $t_{i:k}$ $(1 \leq k \leq |c_i|)$. Given the output tensor of a task and its operator type, we can infer the necessary input tensors to execute each task.

Figure 3.3 shows an example parallelization configuration for a matrix multiplication operator (i.e., $U = VW$). The operator is partitioned into four independent tasks assigned to different GPU devices. The input and output tensors of the tasks are shown in the figure.

A *parallelization strategy* $\mathcal{S}$ describes one possible parallelization of an application. $\mathcal{S}$ includes a parallelization configuration c_i for each operator o_i, and each o_i's configuration can be chosen independently from among all possible configurations for o_i. Note that all strategies in the SOAP search space perform the same computation defined by the DNN and therefore maintain the same model accuracy by design.

3.4 Execution Simulator

In this section, we describe the execution simulator, which takes a computation graph $\mathcal{G}$, a device topology $\mathcal{D}$, and a parallelization strategy $\mathcal{S}$ as inputs and predicts the execution time to run $\mathcal{G}$ on $\mathcal{D}$ using strategy $\mathcal{S}$.

The simulator depends on the following assumptions:

A1. The execution time of each task is predictable with low variance and is independent of the contents of input tensors.

A2. For each connection (d_i, d_j) between device d_i and d_j with bandwidth b, transferring a tensor of size s from d_i to d_j takes s/b time (i.e., the communication bandwidth can be fully utilized).

A3. Each device processes the assigned tasks with a FIFO (first-in-first-out) scheduling policy. This is the policy used by modern devices such as GPUs.

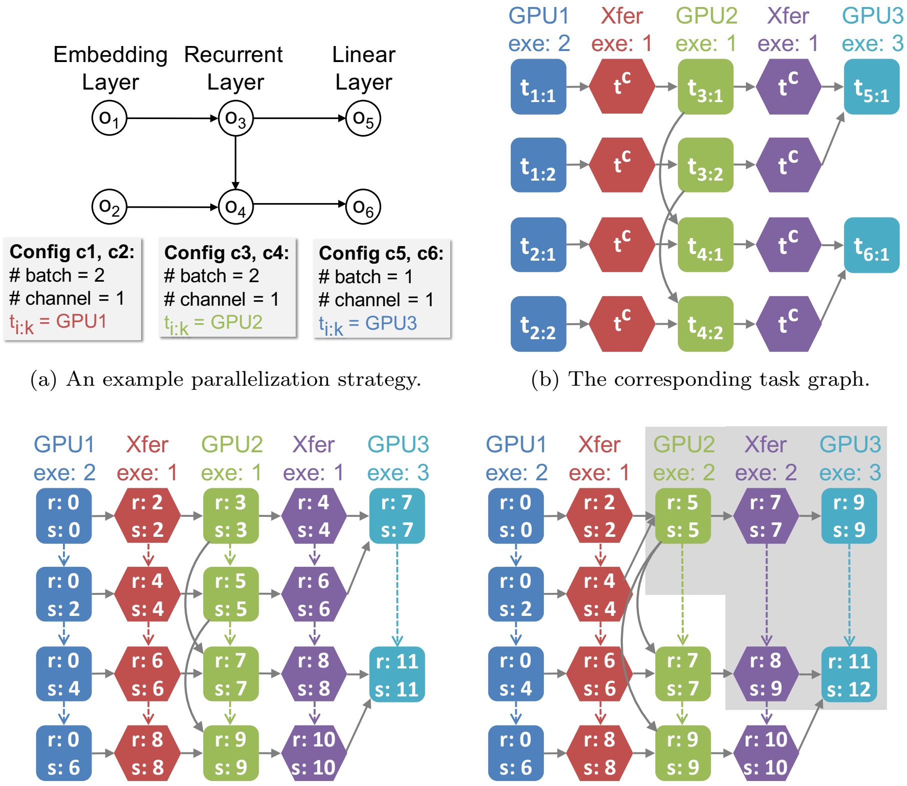
A4. The runtime has negligible overhead. A device begins processing a task as soon as its input tensors are available and the device has finished previous tasks.

To simulate an execution, we reuse the idea from Section 2.6 to measure the execution time of each distinct operator once for each configuration and include these measurements in a *task graph*, which includes all tasks derived from operators and dependencies between tasks. The simulator can generate an execution timeline by running a simulation algorithm on the task graph.

3.4.1 Task Graph

A *task graph* models dependencies between individual *tasks* derived from operators. To unify the abstraction, we model each hardware connection between devices as a *communication device* that can only perform *communication tasks* (i.e., data transfers). Note that devices and hardware connections are modeled as separate devices, which allows computation (i.e., normal tasks) and communication (i.e., communication tasks) to be overlapped if possible.

Given a computation graph $\mathcal{G}$, a device topology $\mathcal{D}$, and a parallelization strategy $\mathcal{S}$, we use the following steps to construct a task graph $\mathcal{T} = (\mathcal{T}_N, \mathcal{T}_E)$, where each node $t \in \mathcal{T}_N$ is a task (i.e., a normal task or a communication task) and each edge $(t_i, t_j) \in \mathcal{T}_E$ is a dependency that task t_j cannot start until task t_i is completed. Note that the edges in the task graph are simply ordering constraints—the edges do not indicate data flow, as all data flow is included in the task graph as communication tasks.



(a) An example parallelization strategy.

(b) The corresponding task graph.

(c) The task graph after full simulation.

(d) The task graph after delta simulation.

Figure 3.4: Simulating an example parallelization strategy. The tasks' `exeTime` and `device` are shown on the top of each column. In Figure 3.4c and 3.4d, the letters "r" and "s" indicate the `readyTime` and `startTime` of each task, respectively, and the dashed edges indicate the `nextTask`.

Table 3.2: Properties for each task in the task graph.

Property	**Description**
Properties set in graph construction	
`exeTime`	The elapsed time to execute the task.
`device`	The assigned device of the task.
$\mathcal{I}$(t)	$\{t_{in} \| (t_{in}, t) \in \mathcal{T}_E\}$
$\mathcal{O}$(t)	$\{t_{out} \| (t, t_{out}) \in \mathcal{T}_E\}$
Properties set in simulation	
`readyTime`	The time when the task is ready to run.
`startTime`	The time when the task starts to run.
`endTime`	The time when the task is completed.
`preTask`	The previous task performed on `device`.
`nextTask`	The next task performed on `device`.
Internal properties used by the full simulation algorithm	
`state`	Current state of the task, which is one of `NOTREADY`, `READY`, and `COMPLETE`.

1. For each operator $o_i \in \mathcal{G}$ with parallelization configuration c_i, we add tasks $t_{i:1}, ..., t_{i:|c_i|}$ to $\mathcal{T}_N$.

2. For each tensor $(o_i, o_j) \in \mathcal{G}$, which is an output of operator o_i and an input of o_j, we compute the output sub-tensors written by tasks $t_{i:k_i}$ $(1 \leq k_i \leq |c_i|)$ and the input sub-tensors read by tasks $t_{j:k_j}$ $(1 \leq k_j \leq |c_j|)$. For every task pair $t_{i:k_i}$ and $t_{j:k_j}$ with shared tensors, if two tasks are assigned to the same device, we add an edge $(t_{i:k_i}, t_{j:k_j})$ into $\mathcal{T}_E$, indicating a dependency between the two tasks, and no communication task is needed. If $t_{i:k_i}$ and $t_{j:k_j}$ with shared tensors are assigned to different devices, we add a communication task t^c to $\mathcal{T}_N$ and two edges $(t_{i:k_i}, t^c)$ and $(t^c, t_{j:k_j})$ to $\mathcal{T}_E$. The new task t^c is assigned to the communication device between the devices that perform $t_{i:k_i}$ and $t_{j:k_j}$.

Figure 3.4a shows an example parallelization strategy for a standard 3-layer RNN consisting of an embedding layer, a recurrent layer, and a linear layer. It represents commonly used model parallelism that assigns operators in each layer to a dedicated GPU. Figure 3.4b shows the corresponding task graph. Each square and hexagon indicate a normal and a communication task, respectively, and each directed edge represents a dependency between tasks.

Table 3.2 lists the properties for each task in the task graph. For a normal task derived from an operator, its `exeTime` is the time to execute the task on the given device and is estimated by running the task multiple times on the device and measuring the average execution time (assumption A1). A task's `exeTime` is cached, and all future tasks with the same operator type and input/output tensor shapes will use the cached value without rerunning the task. For a communication task, its `exeTime` is the time to transfer a tensor (of size s) between devices with bandwidth b and is estimated as s/b (assumption A2).

In addition to `exeTime`, FlexFlow also sets `device`, $\mathcal{I}(t)$, and $\mathcal{O}(t)$ (defined in Table 3.2) during graph construction. Other properties in Table 3.2 remain unset and must be filled in by the simulation.

3.4.2 Full Simulation Algorithm

We now describe a full simulation algorithm that we use as a baseline for comparisons with our delta simulation algorithm. The full simulation algorithm first builds a task graph using the method described in Section 3.4.1 and then sets the properties for each task using a variant of Dijkstra's shortest-path algorithm [23]. Tasks are enqueued into a global priority queue when ready (i.e., all predecessor tasks are completed) and are dequeued in increasing order by their `readyTime`. Therefore, when a task t is dequeued, all tasks with an earlier `readyTime` have been scheduled, and we can set the properties for task t while maintaining the FIFO scheduling order (assumption A3). Figure 3.4c shows the execution timeline of the example parallelization strategy.

3.4.3 Delta Simulation Algorithm

FlexFlow uses a MCMC search algorithm (see Section 3.5) that proposes a new parallelization strategy by changing the parallelization configuration of a single operator in the previous strategy (see Section 3.5.2). As a result, in the common case, most of the execution timeline does not change from one simulated strategy to the next. Based on this observation, we introduce a *delta simulation algorithm* that starts from a previous task graph and only re-simulates tasks involved in the portion of the execution timeline that changes, an optimization that dramatically speeds up the simulator, especially for strategies for large distributed machines.

To simulate a new strategy, the delta simulation algorithm first updates tasks and dependencies from an existing task graph and enqueues all modified tasks into a global priority queue. Similar to the Bellman-Ford shortest-path algorithm [23], the delta simulation algorithm iteratively dequeues updated tasks and propagates the updates to subsequent tasks.

For the example in Figure 3.4, consider a new parallelization strategy derived from the original strategy (Figure 3.4a) by only reducing the parallelism of operator o_3 to 1 (i.e., $|c_3| = 1$). Figure 3.4d shows the task graph for the new parallelization strategy, which can be generated from the original task graph (in Figure 3.4c) by updating the simulation properties of tasks in the grey area.

3.5 Execution Optimizer

The *execution optimizer* takes a computation graph and a device topology as inputs and automatically finds an efficient parallelization strategy. Using the simulator as an oracle, FlexFlow transforms the parallelization optimization problem into a cost minimization problem, namely minimizing the predicted execution time.

Finding the optimal parallelization strategy is NP-hard, by an easy reduction from *minimum* *makespan* [52]. In addition, the number of possible strategies is exponential in the number of operators of a computation graph (see Section 3.3), which makes it intractable to exhaustively enumerate the search space. To find a low-cost strategy, FlexFlow uses a cost minimization search to heuristically explore the space and returns the best strategy discovered.

3.5.1 MCMC Sampling

This section briefly introduces the Metropolis-Hastings algorithm [38] we use for MCMC sampling in the execution optimizer. The algorithm maintains a current strategy $\mathcal{S}$ and randomly proposes a new strategy $\mathcal{S}^*$. $\mathcal{S}^*$ is accepted and becomes the new current strategy with the following probability:

$$\alpha(\mathcal{S}^*|\mathcal{S}) = \min\Big(1, \exp\big(\beta \cdot (\mathit{cost}(\mathcal{S}) - \mathit{cost}(\mathcal{S}^*))\big)\Big) \tag{3.1}$$

This acceptance criteria has several important properties. If $\mathcal{S}^*$ has a lower cost than $\mathcal{S}$, then $\mathcal{S}^*$ is always accepted. If $\mathcal{S}^*$ has a higher cost than $\mathcal{S}$, then $\mathcal{S}^*$ may still be accepted with a probability that decreases as a function of the difference between $\mathit{cost}(\mathcal{S})$ and $\mathit{cost}(\mathcal{S}^*)$. Intuitively, MCMC tends to behave as a greedy search algorithm, preferring to move towards lower cost whenever that is readily available, but can also escape local minima.

3.5.2 Search Algorithm

Our method for generating proposals is simple: an operator in the current parallelization strategy is selected at random, and its parallelization configuration is replaced by a random configuration. We use the predicted execution time from the simulator as the cost function in Equation 3.1 and use existing strategies (e.g., data parallelism, expert-designed strategies) as well as randomly generated strategies as the initial candidates for the search algorithm. For each initial strategy, the search algorithm iteratively proposes new candidates until one of the following two criteria is satisfied: (1) the search time budget for current initial strategy is exhausted; or (2) the search procedure cannot further improve the best discovered strategy for half of the search time.

3.6 FlexFlow Runtime

We found that existing deep learning systems (e.g., TensorFlow, PyTorch, Caffe2, and MXNet) only support parallelizing an operator in the sample dimension through data parallelism, and it is non-trivial to parallelize an operator in other dimensions or combinations of several SOAP dimensions in these systems.

To support parallelizing DNN models using any strategy in the SOAP search space, we implemented the FlexFlow distributed runtime in Legion [15], a high-performance parallel runtime for distributed heterogeneous architectures, and use cuDNN [21] and cuBLAS [24] as the underlying libraries for

Table 3.3: Details of the DNNs and datasets used in evaluation.

DNN	**Description**	**Dataset**	**Report Acc.**	**Our Acc.**
Convolutional Neural Networks (CNNs)				
AlexNet	A 12-layer CNN	Synthetic data	-	-
Inception-v3	A 102-layer CNN with Inception modules [77]	ImageNet	78.0%[a]	78.0%[a]
ResNet-101	A 101-layer residual CNN with shortcuts	ImageNet	76.4%[a]	76.5%[a]
Recurrent Neural Networks (RNNs)				
RNNTC	4 recurrent layers followed by a softmax layer	Movie Reviews [63]	79.8%	80.3%
RNNLM	2 recurrent layers followed by a softmax layer	Penn Treebank [59]	78.4[b]	76.1[b]
NMT	4 recurrent layers followed by an attention	WMT [85]	19.67[c]	19.85[c]

[a] top-1 accuracy for single crop on the validation dataset (higher is better).
[b] word-level test perplexities on the Peen Treebank dataset (lower is better).
[c] BLEU scores [68] on the test dataset (higher is better).

processing DNN operators. We use the Legion high-dimensional partitioning interface [81] to support parallelizing an operator in any combination of the parallelizable dimensions.

3.7 Evaluation

In this section, we aim to evaluate the following points.

1. Can FlexFlow improve the end-to-end training performance of real-world DNN benchmarks, especially for DNNs that do not scale under existing manually designed parallelization strategies?

2. Can FlexFlow's execution simulator predict the execution times of different parallelization strategies with reasonable accuracy?

3. Can FlexFlow's execution optimizer discover strategies outperforming the strategies manually designed by domain experts?

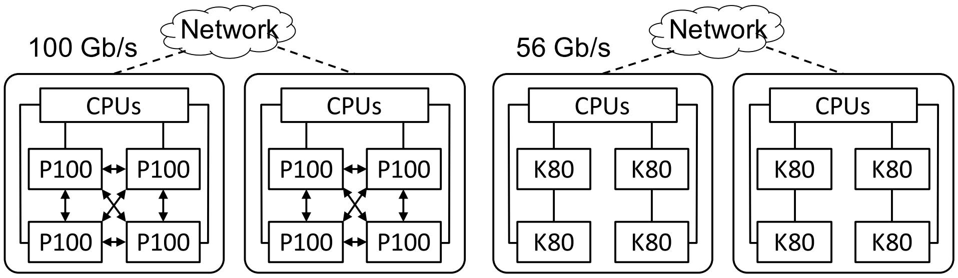
3.7.1 Experimental Setup

Table 3.3 summarizes the DNN benchmarks used in our experiments.

AlexNet, Inception-v3, and ResNet-101 are three CNNs that achieved the best accuracy in the ILSVRC competitions [71]. For AlexNet, the per-iteration training time is smaller than the time to load training data from disk. We follow the suggestions in TensorFlow Benchmarks [1] and use synthetic data to benchmark the performance of AlexNet. For all other experiments, the training data is loaded from disk in the training procedure.

RNNTC, RNNLM and NMT are sequence-to-sequence RNN models for text classification, language modeling, and neural machine translation, respectively. RNNTC uses four LSTM layers with a hidden size of 1024. RNNLM uses two LSTM layers with a hidden size of 2048. Both RNN models

[1] https://www.tensorflow.org/performance/benchmarks



(a) The P100 Cluster (4 nodes).

(b) The K80 Cluster (16 nodes).

Figure 3.5: Architectures of the GPU clusters used in the experiments. An arrow line indicates a NVLink connection. A solid line is a PCI-e connection. Dashed lines are Infiniband connections across different nodes.

include a softmax linear after the last LSTM layer. NMT includes an encoder and a decoder, both of which consist of 2 LSTM layers with a hidden size of 1024. To improve model accuracy, we also use an attention layer on top of the last decoder LSTM layer [13]. Figure 3.15 illustrates the structure of the NMT model. For all three RNN models, we set the number of unrolling steps for each recurrent layer to 40.

We follow prior work [51, 78, 39, 48, 92, 87] to construct computation graphs and set hyperparameters (e.g., learning rates, weight decays). We use synchronous training and a per-GPU batch size of 64 for all DNN benchmarks, except for AlexNet, which has a much smaller model and uses a per-GPU batch size of 256.

To evaluate the performance of FlexFlow with different device topologies, we performed the experiments on two GPU clusters, as shown in Figure 3.5. The first cluster contains 4 compute nodes, each of which is equipped with two Intel 10-core E5-2600 CPUs, 256GB main memory, and four NVIDIA Tesla P100 GPUs. GPUs on the same node are connected by NVLink, and nodes are connected over 100Gb/s EDR Infiniband. The second cluster consists of 16 nodes, each of which is equipped with two Intel 10-core E5-2680 GPUs, 256GB main memory, and four NVIDIA Tesla K80 GPUs. Adjacent GPUs are connected by a separate PCI-e switch, and all GPUs are connected to CPUs through a shared PCI-e switch. Compute nodes in the cluster are connected over 56 Gb/s EDR Infiniband.

Unless otherwise stated, we set 30 minutes as the time budget for the execution optimizer and use data parallelism and a randomly generated strategy as the initial candidates for the search. As shown in Table 3.4, the search terminates in a few minutes in most cases.

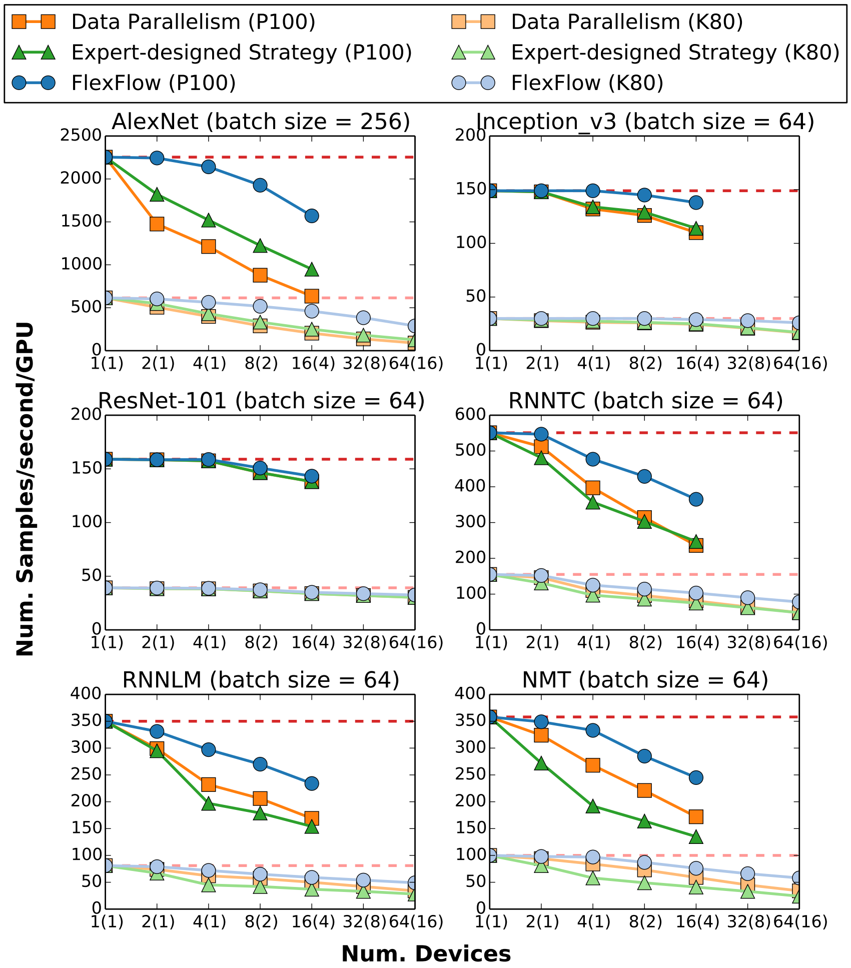


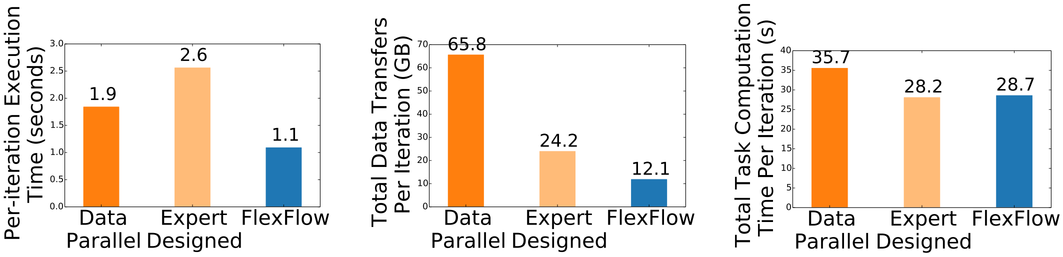
Figure 3.6: Per-iteration training performance on six DNNs. Numbers in parendissertation are the number of compute nodes used in the experiments. The dash lines show the ideal training throughput.

3.7.2 Parallelization Performance

Per-iteration Performance

We compare the per-iteration training performance of FlexFlow with the following baselines. Data parallelism is commonly used in existing deep learning systems. To control for implementation differences, we ran data parallelism experiments in TensorFlow r1.7, PyTorch v0.3, and our implementation and compared the performance numbers. Compared to TensorFlow and PyTorch, FlexFlow achieves the same or better performance numbers on all six DNN benchmarks, and therefore we report the data parallelism performance achieved by FlexFlow in the experiments.

Expert-designed strategies optimize parallelization based on domain experts' knowledge and



(a) Per-iteration run time. (b) Total transfers per iteration. (c) Total task run time per iteration.

Figure 3.7: Parallelization performance for NMT on 64 K80 GPUs (16 nodes). FlexFlow reduces per-iteration execution time by 1.7-2.4× and data transfers by 2-5.5× compared to other approaches. FlexFlow achieves similar overall task computation time as expert-designed strategy, which is 20% fewer than data parallelism.

experience. For CNNs, [50] uses data parallelism for parallelizing convolutional and pooling layers and switches to model parallelism for densely-connected layers. For RNNs, [87] uses data parallelism that replicates the entire computation graph on each compute node and uses model parallelism that assign operators with the same depth to the same GPU on each node. These expert-designed strategies are used as a baseline in our experiments. Model parallelism only exposes limited parallelism by itself, and we compare against model parallelism as a part of these expert-designed strategies.

Figure 3.6 shows the per-iteration training performance on all six DNN benchmarks. For ResNet-101, FlexFlow finds strategies similar to data parallelism (except using model parallelism on a single node for the last fully-connected layer) and therefore achieves similar parallelization performance. For other DNN benchmarks, FlexFlow finds more efficient strategies than the baselines and achieves 1.3-3.3× speedup. Note that FlexFlow performs the same operators as data parallelism and expert-designed strategies, and the performance improvement is achieved by using faster parallelization strategies. We found that the parallelization strategies discovered by FlexFlow have two advantages over data parallelism and expert-designed strategies.

Reducing overall communication costs. Similar to existing deep learning systems, the FlexFlow distributed runtime supports overlapping data transfers with computation to hide communication overheads. However, as we scale the number of devices, the communication overheads increase, but the computation time used to hide communication remains constant. Therefore, reducing overall communication costs is beneficial for large-scale distributed training. Figure 3.7b shows that, to parallelize the NMT model on 64 K80 GPUs (16 nodes), FlexFlow reduces the per-iteration data transfers by 2-5.5× compared to other parallelization approaches.

Reducing overall task computation time. Data parallelism always parallelizes an operator in the sample dimension. However, as reported in [44], parallelizing an operator through different

Table 3.4: The end-to-end search time with different simulation algorithms (seconds).

Num.	**AlexNet**			**ResNet**			**Inception**		
GPUs	Full	Delta	Speedup	Full	Delta	Speedup	Full	Delta	Speedup
4	0.11	0.04	**2.9**×	1.4	0.4	**3.2**×	14	4.1	**3.4**×
8	0.40	0.13	**3.0**×	4.5	1.4	**3.2**×	66	17	**3.9**×
16	1.4	0.48	**2.9**×	22	7.3	**3.1**×	388	77	**5.0**×
32	5.3	1.8	**3.0**×	107	33	**3.2**×	1746	298	**5.9**×
64	18	5.9	**3.0**×	515	158	**3.3**×	8817	1278	**6.9**×
Num.	**RNNTC**			**RNNLM**			**NMT**		
GPUs	Full	Delta	Speedup	Full	Delta	Speedup	Full	Delta	Speedup
4	16	7.5	**2.2**×	21	9.2	**2.3**×	40	16	**2.5**×
8	91	39	**2.3**×	76	31	**2.5**×	178	65	**2.7**×
16	404	170	**2.4**×	327	121	**2.7**×	998	328	**3.0**×
32	1358	516	**2.6**×	1102	342	**3.2**×	2698	701	**3.8**×
64	4404	1489	**3.0**×	3406	969	**3.6**×	8982	2190	**4.1**×

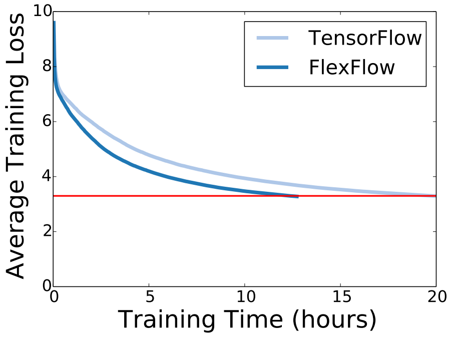


Figure 3.8: Training curves of Inception-v3 in different systems. The model is trained on 16 P100 GPUs (4 nodes).

dimensions can result in different task computation time. For the matrix multiplication operator in the NMT model, parallelizing it in the channel dimension reduces the operator's overall computation time by 38% compared to parallelizing the operator in the sample dimension. Figure 3.7c shows that FlexFlow reduces the overall task computation time by 20% compared to data parallelism for the NMT model. The expert-designed strategy achieves slightly better total task computation time than FlexFlow. However, this is achieved by using model parallelism on each node, which disables any parallelism within each operator and results in imbalanced workloads. FlexFlow reduces the task computation time while enabling parallelism within an operator and maintaining load balance.

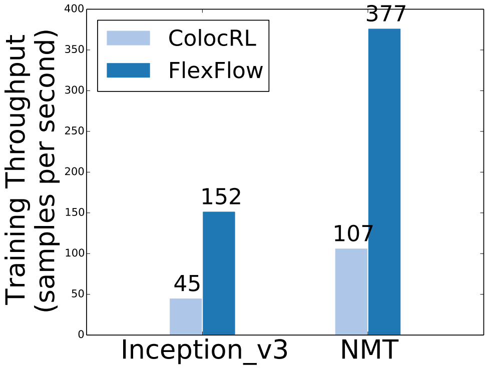


Figure 3.9: Comparison among the parallelization strategies found by different automated frameworks.

End-to-end Performance

FlexFlow performs the same computation as other deep learning systems for a DNN model and therefore achieves the same model accuracy. Table 3.3 verifies that FlexFlow achieves the state-of-the-art accuracies on the DNN benchmarks used in the experiments.

In this experiment, we compare the end-to-end training performance between FlexFlow and TensorFlow on Inception-v3. We train Inception-v3 on the ImageNet dataset until the model reaches the single-crop top-1 accuracy of 72% on the validation set. The training processes in both frameworks use stochastic gradient decent (SGD) with a learning rate of 0.045 and a weight decay of 0.0001. Figure 3.8 illustrates the training curves of the two systems and show that FlexFlow reduces the training time by 38% compared to TensorFlow.

Automated Frameworks

We compare against ColorRL [61], an automated framework that find parallelization strategies in a limited search space. ColocRL uses reinforcement learning to learn device placement for model parallelism. We are not aware of any publicly available implementation of ColocRL, so we compare against the learned device placement for Inception-v3 and NMT, as reported in the ColocRL paper, and performed the experiments on the same machine.

Figure 3.9 compares the training throughput of the strategies found by FlexFlow and ColocRL for four K80 GPUs on a single node. The parallelization strategies found by FlexFlow achieve 3.4 - 3.8× speedup compared to ColocRL. We attribute the performance improvement to the larger search space explored by FlexFlow.

Besides improving training performance, FlexFlow has two additional advantages over ColocRL. First, ColocRL requires executing each strategy in the hardware environment to get reward signals and takes 12-27 hours to find the best placement, while FlexFlow finds efficient parallelization strategies for these executions in 14-40 seconds. Second, ColocRL uses up to 160 compute nodes

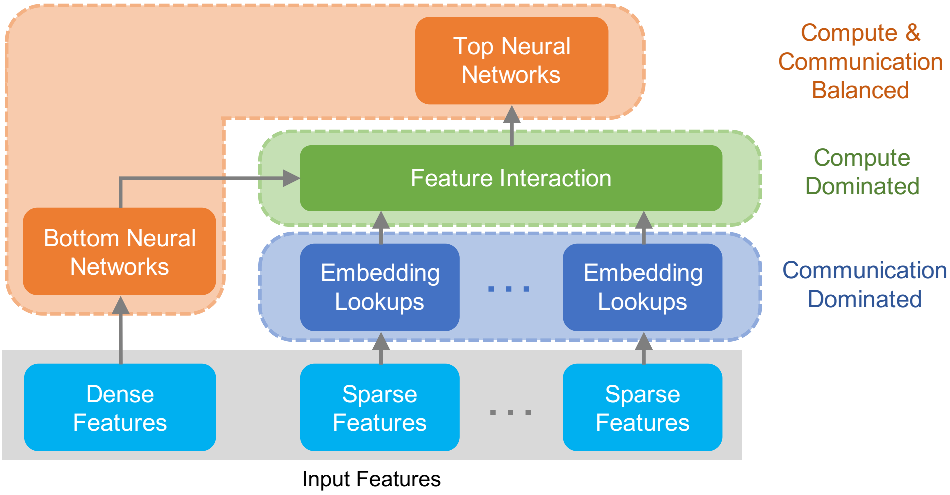


Figure 3.10: An overview of DLRM.

(with 4 GPUs on each node) to find the placement in time, while FlexFlow uses a single compute node to run the execution optimizer.

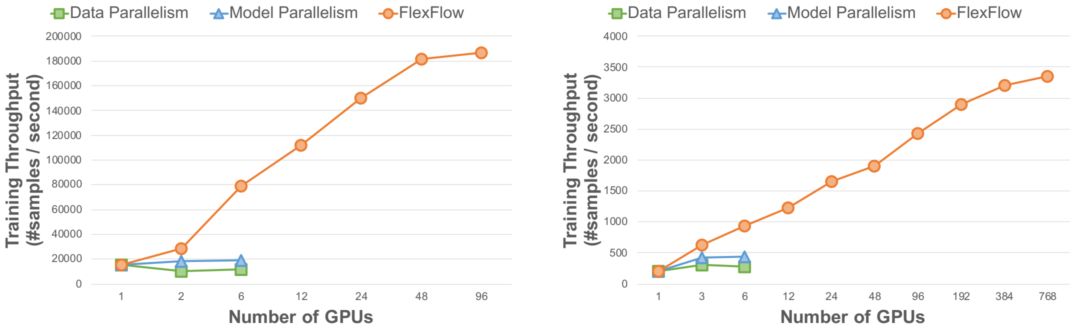
3.7.3 DLRM and Candle Uno

In addition to the six DNN benchmarks shown in Table 3.3, we also evaluate FlexFlow on two large DNN models that cannot scale to large numbers of GPUs using data or model parallelism.

DLRM

DLRM [66] is a deep learning recommendation model for personalization and ads recommendation, shown in Figure 3.10. The model uses embedding layers to process sparse input features representing categorical data, and uses the bottom neural network to process dense input features. DLRM has a feature interaction operator that combines the representations learnt from both the dense and sparse features (e.g., concatenating the representations). The output of the feature interaction operator is then sent to the top neural network for downstream prediction tasks.

It is challenging to parallelize the training of DLRM, as the model involves a mixture of compute and communication intensive workloads, as depicted in Figure 3.10. In particular, the embedding layers include larger numbers of trainable parameters, which introduce significant communication overhead when parallelizing DLRM using data parallelism. On the other hand, the top/bottom neural networks and the feature interaction layer have sequential dependencies, making model parallelism not scalable for DLRM.



(a) DLRM

(b) Candle Uno

Figure 3.11: Training performance for DLRM and Candle Uno on the Summit supercomputer [82]. Each compute node is equipped with two IBM POWER9 CPUs and six V100 GPUs. GPUs on the same node are connected by NVLink, and nodes are connected over 100Gb/s EDR Infiniband.

Model configuration. We follow prior work [66] to construct the computation graph of DLRM and set hyperparameters for training. The bottom neural network is a multi-layer perceptron (MLP) consisting of 3 linear layers with output dimensions of 512, 512, and 64, respectively. The top neural network contains 4 linear layers with output dimensions of 1024, 1024, 1024, and 1. Note that the output of the top neural network has a dimension of 1 (i.e., a scalar), which indicates the probability that a given user clicks the recommended ads in a training sample. We use an individual embedding table for each categorical sparse feature. All embedding tables have output dimensions of 64.

Dataset. We use the Criteo Ad Kaggle dataset [8] in our evaluation, which consists of click logs for ad CTR prediction and involves 26 categorical sparse features. The vocabulary sizes of the embedding tables range from 3 to 1.7 million. The dataset contains approximately 45 million samples over 7 days. We use the first 6 days as the training set and equally split the 7th day into the validation and test set.

Result. Figure 3.11a shows the performance of training DLRM on the Criteo Ad Kaggle dataset using different parallelization strategies on the Summit supercomputer [82]. Using data and model parallelism, DRLM simply fails to scale beyond a very small number GPUs. On the other hand, by automatically discovering more efficient parallelization strategies that combine parallelization opportunities across different dimensions, FlexFlow scales DLRM training to 96 GPUs and improves the overall training throughput by up to 10× compared to data and model parallelism.

Candle Uno

Candle [4] is a set of deep neural network architectures that address cancer-related learning tasks at different biological scales. We evaluate FlexFlow on the Candle Uno model [9], which takes as inputs the RNA sequence of human body cells and the dense features of drugs, and predicts their reactions on a scale between 0 and 1. The Candle Uno model uses multi-layer perceptrons (MLPs) to process both cell and drug features, which are concatenated as a single high-dimensional representation and sent to another MLP to predict an reaction score of the cell and drug pair.

Model configuration. We follow prior work [9] to construct the model and set hyperparameters for training. The bottom MLPs for processing cell and drug features each contains 3 linear layers with an output dimension of 1000 in each layer. The top MLP for predicting an reaction score includes 4 linear layers with out dimensions of 1000, 1000, 1000, and 1, respectively. Similar to the DLRM model, the output of the top MLP has a dimension of 1 indicating the reaction score between the input cell and drug pair.

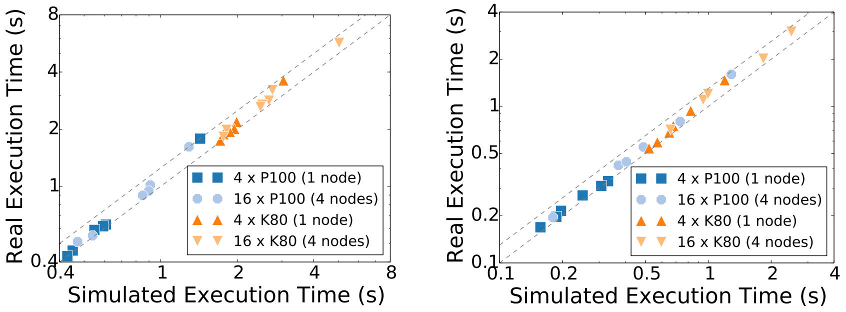
Dataset. We use a combination of six datasets (namely CCLE, CTRP, gCSI, GDSC, NCI60, and ALMANAC) in the Candle benchmark in our evaluation. The combined dataset contains 20 million samples for training and 5 million samples for validation and testing. Each sample includes two drug features with dimensions of 5270 and 2048, respectively. The cell feature describes the RNA sequence of a human body cell with a dimension of 942.

Result. Figure 3.11b shows the performance of training Candle Uno using different strategies on Summit. Both data and model parallelism cannot scale beyond a single compute node with 6 GPUs because the large numbers of trainable parameters in multiple MLPs involve significant communication overhead when using data parallelism, and the lack of inter-operator parallelization opportunities prevents model parallelism from scaling. For the Candle Uno model, FlexFlow automatically discovers faster parallelization strategies and improves the overall training throughput by up to 7.8× compared to data and model parallelism.

3.7.4 Execution Simulator

We evaluate the performance of the simulator using two metrics: simulator accuracy and simulator execution time.

Simulator accuracy. We first compare the estimated execution time predicted by the execution simulator with the real execution time measured by actual executions. Figure 3.12 shows the results for different DNNs and different available devices. The dashed lines indicate a relative difference of 0% and 30%, respectively, which encompasses the variance between actual and predicted execution



(a) Inception-v3

(b) NMT

Figure 3.12: Comparison between the simulated and actual execution time for different DNNs and device topologies.

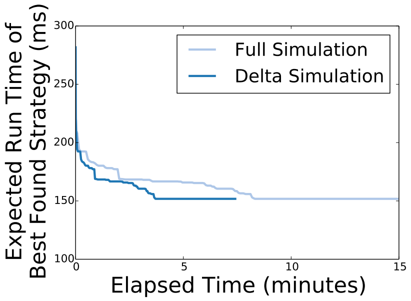


Figure 3.13: Search performance with the full and delta simulation algorithms for the NMT model on 16 P100 GPUs (4 nodes).

time. In addition, for different parallelization strategies with the same computation graph and device topology (i.e., points of the same shape in the figure), their simulated execution time preserves actual execution time ordering, which shows that simulated execution time is an appropriate metric to evaluate the performance of different strategies.

Simulator execution time. Figure 3.13 shows the search performance with different simulation algorithms for finding a strategy for the NMT model on 16 P100 GPUs on 4 nodes. The full and delta simulation algorithms terminate in 16 and 6 minutes, respectively. If the allowed time budget is less than 8 minutes, the full simulation algorithm will find a worse strategy than the delta simulation algorithm.

We compare the end-to-end search time of the execution optimizer with different simulation algorithms. For a given DNN model and device topology, we measure the average execution time of the optimizer using 10 random initial strategies. The results are shown in Table 3.4. The delta simulation algorithm is 2.2-6.9× faster than the full simulation algorithm. Moreover, the speedup over the full simulation algorithm increases as we scale the number of devices.

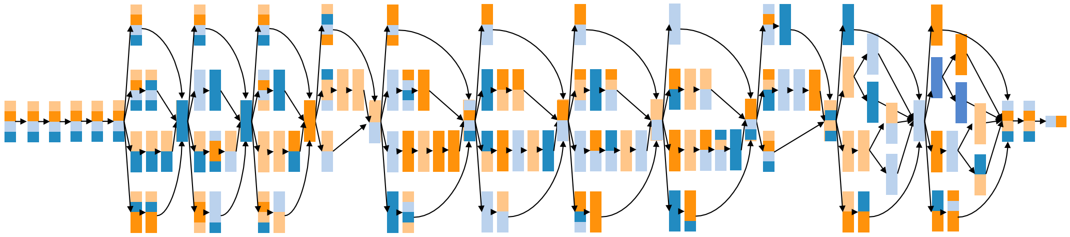


Figure 3.14: The best discovered strategy for parallelizing Inception-v3 on 4 P100 GPUs. For each operator, the vertical and horizontal dimensions indicate parallelism in the sample and parameter dimension, respectively. Each GPU is denoted by a color.

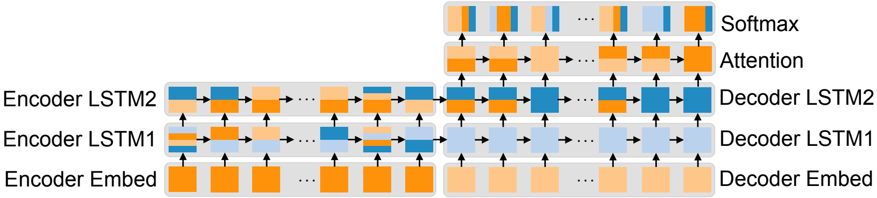


Figure 3.15: The best discovered strategy for parallelizing NMT on 4 P100 GPUs. For each operator, the vertical and horizontal dimensions indicate parallelism in the sample and parameter dimension, respectively. Each grey box denotes a layer, whose operators share the same network parameters. Each GPU is denoted by a color.

3.7.5 Search Algorithm

We compare the best discovered strategies with the global optimal strategies for small executions. To obtain a search space of reasonable size, we limit the number of devices to 4 and consider the following two DNNs. LeNet [55] is a 6-layer CNN. The second DNN is a variant of RNNLM where the number of unrolling steps for each recurrent layer is restricted to 2. We use depth-first search to explore the space and use A* [23] to prune the search. Finding the optimal strategies for LeNet and RNNLM took 0.8 and 18 hours, respectively. For both DNNs, FlexFlow finds the same global optimal strategy in less than 1 second.

3.7.6 Case Studies

We discuss the best strategies discovered by FlexFlow and how they improve parallelization performance.

Inception-v3. Figure 3.14 shows the best discovered strategy for parallelizing Inception-v3 on four P100 GPUs, which exploits intra-operator parallelism for operators on the critical path and uses a combination of intra- and inter-operator parallelism for operators on different branches. This results in a well-balanced workload and reduces data transfers for parameter synchronization. Compared to data parallelism, this strategy reduces the parameter synchronization costs by 75% and the

Table 3.5: The parallelism dimensions used by different approaches. S, O, A, and P indicate parallelism in the Sample, Operator, Attribute, and Parameter dimensions. Hybrid parallelism indicates an approach supports parallelizing an operator in a combination of the sample, attribute, and parameter dimensions (see Figure 3.2).

Parallelization Approach	**Parallelism Dimensions**	**Hybrid Parallelism**	**Supported DNNs**
Data Parallelism	S		partial**
Model Parallelism	O, P		all
[50]	S, P		CNNs**
[87]	S, O		RNNs#
ColocRL	O		partial#
OptCNN*	S, A, P	✓	linear%
FlexFlow	S, O, A, P	✓	all

* OptCNN is an earlier version of FlexFlow.
** Does not work for DNNs whose entire model cannot fit on a single device.
# Does not work for DNNs with large operators that cannot fit on a single device.
% Only works for DNNs with linear computation graphs.

per-iteration execution time by 12%.

For parallelizing the same Inception-v3 model on four K80 GPUs with asymmetric connections between GPUs (see Figure 3.5b), we observe that the best discovered strategy tends to parallelize operators on adjacent GPUs with a direct connection to reduce the communication costs.

NMT. Figure 3.15 shows the best discovered strategy for parallelizing NMT on four P100 GPUs. First, for a layer with a large number of network parameters and little computation (e.g., embed layers), it performs the computation on a single GPU to eliminate parameter synchronization. Second, for a layer with a large number of parameters and heavy computation (e.g., softmax layers), FlexFlow uses parallelism in the parameter dimension and assigns the computation for a subset of parameters to each task. This reduces parameter synchronization costs while maintaining load balance. Third, for multiple recurrent layers (e.g., LSTM and attention layers), FlexFlow uses concurrency among different layers as well as parallelism within each operator to reduce parameter synchronization costs while balancing load.

3.8 Related Work

Data and model parallelism. Existing ML systems use data and model parallelism as the default strategies for distributed ML training. Data parallelism [51] is inefficient for operators with a large number of parameters (e.g., densely-connected layers) and becomes a scalability bottleneck in large scale distributed training. Model parallelism [27] splits a DNN into disjoint subsets and trains each subset on a dedicated device, which reduces communication costs for synchronizing network parameters but exposes limited parallelism.

Expert-designed parallelization strategies. Recent work optimizes parallelization for specific DNNs by using experts' domain knowledge and experience. For example, [50] introduces "one weird trick" that uses data parallelism for convolutional and pooling layers and switches to model parallelism for densely-connected layers to accelerate CNNs. To parallelize RNNs, [87] uses data parallelism that replicates the entire RNN on each node and switches to model parallelism for intra-node parallelization. Although these expert-designed strategies improve performance over data and model parallelism, they are suboptimal. We use these expert-designed strategies as baselines in our experiments and show that FlexFlow can further improve training throughput by up to 2.3×.

Automated frameworks have been proposed for finding efficient parallelization strategies in a limited search space. ColocRL [61] uses reinforcement learning to find efficient device placement for model parallelism. [31, 32] exploited hybrid parallelization on tiled domain-specific hardware and proposed various dataflow optimizations for both intra-layer and inter-layer data communication.

In an earlier approach, we proposed OptCNN [44], which also uses layer-wise parallelism for parallelizing CNNs with linear computation graphs. OptCNN uses dynamic programming instead of MCMC to jointly optimize how to parallelize each operator. The major difference with the results reported in this chapter is that OptCNN's dynamic programming approach does not apply to many DNNs used for language modeling, machine translation, and recommendations, which tend to be RNNs or other non-linear networks.

Table 3.5 summarizes the parallelism dimensions explored by existing approaches. Data parallelism uses the sample dimension to parallelize training, while model parallelism exploits the parameter and operator dimensions. Expert-designed strategies exploit parallelism in the sample or parameter dimension to parallelize an operator but do not support hybrid parallelism that uses a combination of the sample, attribute, and parameter dimensions to parallelize an operator (see Figure 3.2). Compared to these manually designed strategies, FlexFlow considers more sophisticated, and often more efficient, strategies to parallelize a single operator. In addition, compared to existing automated frameworks (e.g., ColocRL and OptCNN), FlexFlow supports more generic DNNs and finds strategies that are up to 3.8× faster by exploring a significantly larger search space.

Graph-based cluster schedulers. Previous work has proposed cluster schedulers that schedule cluster-wide tasks by using graph-based algorithms. For example, Quincy [42] maps task scheduling to a flow network and uses a min-cost max-flow (MCMF) algorithm to find efficient task placement. Firmament [33] generalizes Quincy by employing multiple MCMF optimization algorithms to reduce task placement latencies. Existing graph-based schedulers optimize task placement by assuming a fixed task graph. However, FlexFlow solves a different problem that requires jointly optimizing how to partition an operator into tasks by exploiting parallelism in the SOAP dimensions and how to assign tasks to devices.

3.9 Limitations

One limitation of FlexFlow is that the SOAP search space it explores only considers parallelization opportunities within a single training batch but does not support *pipeline parallelism* [65], which explores inter-batch parallelization opportunities by pipelining the forward processing and back propagation across different training iterations. In pipeline parallelism each training iteration uses a staled version of the model parameters in order to enable inter-batch pipelines, which changes the training semantics and may affect the model's final accuracy. Combining pipeline parallelism and the SOAP dimensions will result in a search space orders of magnitude larger than the original SOAP search space, and it remains an open research problem how to effectively explore the new search space and discover efficient solutions.

3.10 Conclusion

FlexFlow is a deep learning engine that automatically finds efficient parallelization strategies in the SOAP search space for distributed DNN training. FlexFlow uses a guided randomized search procedure to explore the space and includes an execution simulator that is an efficient and accurate predictor of DNN performance. We evaluate FlexFlow with six real-world DNN benchmarks on two GPU clusters and show FlexFlow significantly outperforms state-of-the-art parallelization approaches.

Chapter 4

Automated Discovery of Data Placement Optimizations

4.1 Motivation

Graphs provide a natural way to represent real-world data with relational structures, such as social networks, molecular networks, and webpage graphs. Recent work has extended deep neural networks to extract high-level features from data sets structured as graphs, and the resulting architectures, known as *graph neural networks* (GNNs), have recently achieved state-of-the-art prediction performance across a number of graph-related tasks, including vertex classification, graph classification, and link prediction [49, 37, 89].

GNNs combine DNN operations (e.g., convolution and matrix multiplication) with iterative graph propagation: In each GNN layer, the activations of each vertex are computed with a set of DNN operations, using the activations of its neighbors from the previous GNN layer as inputs. Figure 4.1 illustrates the computation of one vertex (in red) in a GNN layer, which aggregates the activations from its neighbors (in blue), and then applies DNN operations to compute new activations of the vertex.

Existing deep learning frameworks do not easily support GNN training and inference at scale. TensorFlow, PyTorch, and Caffe2 were originally designed to handle situations where the model and data collection can be large, but each sample of the collection is relatively small (e.g., a single image). These systems typically leverage data and/or model parallelism by partitioning the batch of input samples or the DNN models across multiple devices, such as GPUs, while each input sample is still stored on a single GPU and not partitioned. However, GNNs typically use small DNN models (a couple of layers) on very large and irregular input samples — graphs. These large graphs do not fit in a single device and so must be partitioned and processed in a distributed manner. Recent

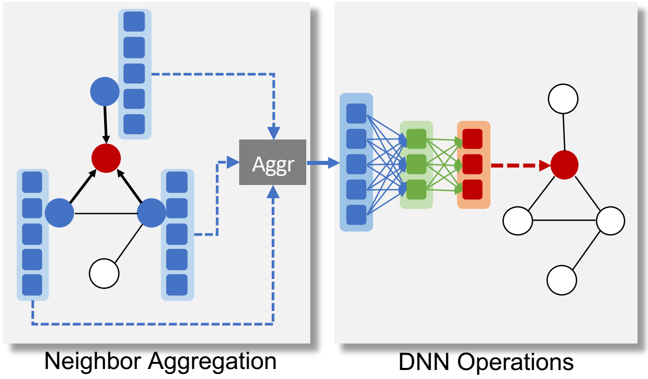


Figure 4.1: Computation of one vertex (in red) in a GNN layer by first aggregating its neighbors' activations (in blue), and then applying DNN operations.

GNN frameworks such as DGL [5] and PyG [30] are implemented on top of PyTorch, and have the same scalability limitation. NeuGraph [57] stores intermediate GNN data in the host CPU DRAM to support multi-GPU training, but it is still limited to the compute resources of a single machine. AliGraph [90] is a distributed GNN framework on CPU platforms, which does not exploit GPUs for performance acceleration.

The current lack of system support has limited the potential application of GNN algorithms on large-scale graphs, and has also prevented the exploration of larger and more sophisticated GNN architectures. To alleviate these limitations, various *sampling* techniques [37, 91] were introduced to first down-sample the original graphs before applying the GNN models, so that the data fit in a single device. Sampling allows existing frameworks to train larger graphs at the cost of potential model accuracy loss [37].

We propose ROC, a distributed multi-GPU framework for fast GNN training and inference on large-scale graphs. ROC leverages the compute resources of multiple GPUs on multiple compute nodes to train large GNN models on the *full* real-world graphs, achieving up to 4× performance over existing GNN frameworks. Despite its use of full graphs, ROC also achieves better time-to-accuracy performance compared to existing sampling techniques. Moreover, the better scalability allows ROC to easily support larger and more sophisticated GNNs than those possible in existing frameworks. To demonstrate ROC's scalability and improved accuracy, we design a class of deep GNN architectures by stacking multiple GCN layers [49]. By using significantly larger and deeper GNN architectures, we improve the classification accuracy over state-of-the-art sampling techniques by 1.5% on the widely used Reddit dataset [37].

To achieve these results, ROC tackles two significant system challenges for distributed GNN computation.

Graph partitioning. Real-world graphs can have arbitrary sizes and variable per-vertex computation loads, which are challenging to partition in a balanced way [35, 93]. GNNs mix compute-intensive DNN operations with data-intensive graph propagation, making it hard to statically compute a good load-balancing partitioning. Furthermore, GNN inference requires partitioning new input graphs that only run for a few iterations, such as predicting the properties of newly discovered proteins [37], in which case existing dynamic repartitioning approaches do not work well [84]. Roc uses an *online linear regression model* to optimize graph partitioning. During the training phase of a GNN architecture, Roc learns a cost model for predicting the execution time of performing a GNN operation on an input (sub)graph. To capture the runtime performance of a GNN operation, the cost model includes both graph-related features such as the number of vertices and edges in the graph, and hardware-related features such as the number of GPU memory accesses to perform the operation. During each training iteration of a GNN architecture, Roc computes a graph partitioning using the run time predictions from the cost model, and uses the graph partitioning to parallelize training. At the end of each training iteration, the actual run time of the subgraphs is sent back to the Roc graph partitioner, which updates the cost model by minimizing the difference between the actual and predicted run times. We show that this linear regression-based graph partitioner outperforms existing static and dynamic graph partitioning strategies by up to 1.4×.

Memory management. In GNNs, the computation for even a single vertex requires accessing a potentially large number of neighbor vertices that may span multiple GPUs and compute nodes. These data transfers can have a significant impact on overall performance. The framework thus must carefully decide in which device memory (CPU or GPU) to store each intermediate tensor, in order to minimize data transfer costs. The memory management is hard to optimize manually as the optimal strategy depends on the input graph size and topology as well as the device constraints such as memory capacity and communication bandwidth. We formulate the task of optimizing data transfers as a cost minimization problem, and introduce a dynamic programming algorithm to quickly find a globally optimal strategy that minimizes data transfers between CPU and GPU memories. We compare the Roc memory management algorithm with existing heuristic approaches [57], and show that Roc reduces data transfer costs between CPU and GPU by 2×.

Overall, compared to NeuGraph, Roc improves the runtime performance by up to 4× for multi-GPU training on a single compute node. Beyond improved partitioning and memory management, Roc sees other smaller performance improvements from a more efficient distributed runtime [46] and the highly optimized kernels adopted from Lux for fast graph propagation on GPUs [43].

Besides performance acceleration, Roc also enables exact GNN computation on *full* original graphs without using sampling techniques, as well as the exploration of more sophisticated GNN architectures beyond the commonly used two-layer models. For large real-world graphs, we show that performing exact GNN computation on the original graphs and using larger and deeper GNN architectures can increase the model accuracy by up to 1.5% on the widely used Reddit dataset compared to existing sampling techniques.

This chapter describes the contributions of Roc in both systems and machine learning:

- On the systems side, Roc enables fast GNN training and inference on large-scale graphs. Roc uses a novel online linear regression model to achieve efficient graph partitioning, and introduces a dynamic programming algorithm to minimize data transfer cost.
- On the machine learning side, Roc removes the necessity of using sampling techniques for GNN training on large graphs, and also enables the exploration of more sophisticated GNN architectures. We demonstrate this potential by achieving new state-of-the-art classification accuracy on the Reddit dataset.

4.2 Background and Related Work

4.2.1 Graph Neural Networks

A GNN takes graph-structured data as input, and learns a representation vector for each vertex in the graph. The learned representation can be used for down-stream tasks such as vertex classification, graph classification, and link prediction [49, 37, 89].

As shown in Figure 4.1, each GNN layer gathers the activations of the neighbor vertices from the previous GNN layer, and then updates the activations of the vertex, using DNN operations such as convolution or matrix multiplication. Formally, the computation in a GNN layer is:

$$a_v^{(k)} = \textsc{Aggregate}^{(k)}\big(\{h_u^{(k-1)} | u \in \mathcal{N}(v)\}\big) \tag{4.1}$$

$$h_v^{(k)} = \textsc{Update}^{(k)}(a_v^{(k)}, h_v^{(k-1)}) \tag{4.2}$$

where $h_v^{(k)}$ is the learned activation of vertex v at the k-th layer, $h_v^{(0)}$ is the input features of v. $\mathcal{N}(v)$ denotes v's neighbors in the graph. For each vertex, Aggregate gathers the activations of its neighbors using an accumulation function such as average or summation. For each vertex v, Update computes its new activations $h_v^{(k)}$ by combining its previous activations $h_v^{(k-1)}$ and the neighborhood aggregation $a_v^{(k)}$. The activations of the last layer $h_v^{(K)}$ capture the structural information for all neighbors within K hops of v, and can be used as the input for down-stream prediction tasks.

4.2.2 Related Work

Distributed DNN training. One of the key differences between GNNs and conventional DNNs is that partitioning in the attribute dimension (recall the SOAP search space in Chapter 3) is necessary for supporting GNN training on large samples (e.g., graphs). The lack of system support for parallelizing in the attribute dimension prevents most existing DNN frameworks from training GNNs on large graphs.

Table 4.1: The graph partitioning strategies used by different frameworks. Balanced training/inference indicates whether an approach can achieve balanced partitioning for GNN training/inference.

Frameworks	**Partitioning Strategies**	**Balanced Training**	**Balanced Inference**
TensorFlow, NeuGraph	Equal		
GraphX, Gemini	Static		
Presto, Lux	Dynamic	✓	
Roc	Online learning	✓	✓

GNN frameworks. Most of the existing GNN frameworks (e.g., DGL [5] and PyG [30]) do not support graphs where the data cannot fit in a single device. NeuGraph [57] supports GNN computation on multiple GPUs in a single machine. AliGraph [90] is a distributed GNN framework but only uses CPUs rather than GPUs.

Sampling in GNNs. As discussed in Section 4.2.1, due to the highly connected nature of real-world graphs, computing $h_v^{(k)}$ may require accessing more data than the GPU memory capacity. A number of sampling techniques have been proposed to support GNN training on large graphs, by down-sampling the neighbors of each vertex [37, 91, 17]. The sampling techniques can be formalized as follows.

$$a_v^{(k)} = \textsc{Aggregate}^{(k)}\big(\{h_u^{(k-1)} | u \in \widehat{\mathcal{N}}(v)\}\big) \tag{4.3}$$

where $\widehat{\mathcal{N}}(v)$ is the sampled subset of $\mathcal{N}(v)$ with a size limit. For example, GraphSAGE [37] samples at most 25 neighbors for each vertex (i.e., $|\widehat{\mathcal{N}}(v)| \leq 25$), while a vertex may actually have thousands of neighbors.

Our evaluation shows that existing sampling techniques come with potential model accuracy loss for large real-world graphs. This observation is consistent with previous work [37]. Roc provides an orthogonal approach to support GNN training on large graphs. Any existing sampling technique can be additionally applied in Roc to further accelerate large-scale GNN training.

Graph frameworks and graph partitioning. A number of distributed graph processing frameworks [58, 35, 43] have been proposed to accelerate data-intensive graph applications. These systems generally adopt the Gather-Apply-Scatter (GAS) [34] vertex-centric programming model. GAS can naturally express the data propagation in GNNs, but cannot support many neural network operations. For example, computing the attention scores [83] between vertices not directly connected cannot be easily expressed in the GAS model.

Table 4.1 summarizes the graph partitioning strategies used in existing deep learning and graph processing frameworks. Deep learning frameworks [11, 57] typically partition data (e.g., tensors) *equally* across GPUs. On the other hand, graph processing frameworks use more complicated strategies to achieve load balance. For example, GraphX [35] and Gemini [93] *statically* partition

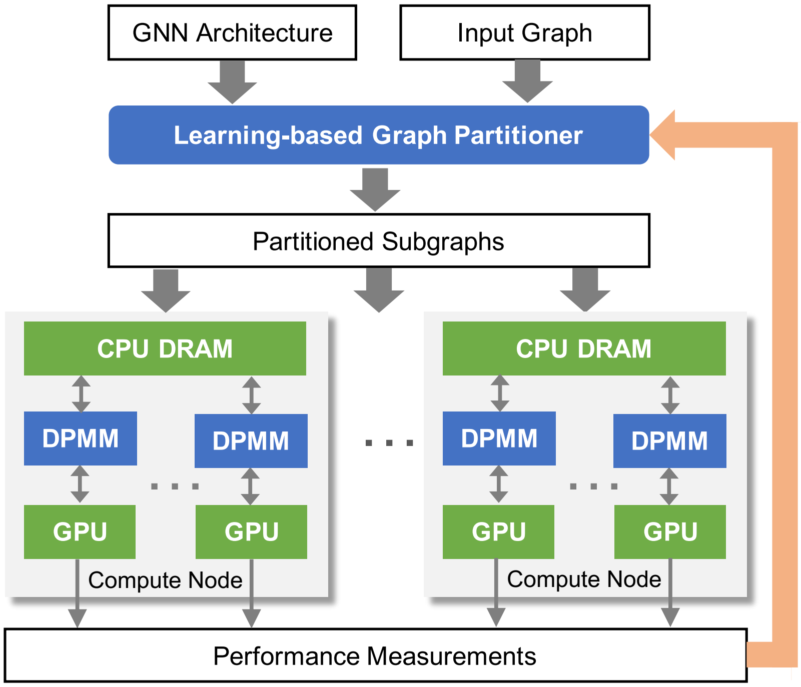


Figure 4.2: Roc system overview. DPMM represents dynamic-programming-based memory manager.

input graphs by minimizing a heuristic objective function, such as the number of edges spanning different partitions. These simple objective functions can achieve good performance for data-intensive graph processing, but they do not work well for compute-intensive GNNs due to the highly varying per-vertex computation loads. *Dynamic* repartitioning [84, 43] exploits the iterative nature of many graph applications and rebalances the workload in each iteration based on the measured performance of previous iterations. This approach converges to a balanced workload distribution for GNN training, but is much less effective for inference which computes the GNN model only once for each new graph. Roc uses an online-linear-regression-based algorithm to achieve balanced partitioning for both GNN training and inference, through jointly learning a cost model to predict the execution time of the GNN model on arbitrary graphs.

4.3 The approach

Figure 4.2 shows an overview of Roc, which takes a GNN model and a graph as inputs, and distributes the GNN computations across multiple GPUs (potentially on different compute nodes) by partitioning the input graph into multiple subgraphs. Each GPU worker performs the computation of the GNN model on a subgraph, and communicates with CPU DRAM to obtain input tensors and save intermediate results. The communication is optimized by a per-GPU dynamic-programming-based memory manager (DPMM) to minimize data transfers between CPU and GPU memories.

Roc uses an online-linear-regression-based graph partitioner to address the unique load imbalance challenge of distributed GNN inference, where a trained GNN model is used to perform inference on previously unseen graphs (Section 4.4). This problem exists today in real-world GNN inference services [37], and our partitioning technique improves the inference performance by up to 1.4× compared to existing graph partitioning strategies. The graph partitioner is trained jointly with the training phase of the GNN architecture, and is also used to partition inference workloads on new input graphs that are not in the training dataset.

After graph partitioning, all subgraphs are sent to different GPUs to perform GNN computations in parallel. Instead of requiring all the intermediate results related to each subgraph to fit in GPU device memory, Roc uses the much larger CPU DRAM on the host machines to hold all the data, and treats the GPU memories as caches. Such a design allows us to support much larger GNN architectures and input graphs. However, transferring tensors between a GPU and the host DRAM has a major impact on runtime performance. Roc introduces a dynamic programming algorithm to discover a memory management plan that minimizes these data transfers (see Section 4.5).

4.4 Graph Partitioner

The goal of the Roc graph partitioner is discovering balanced partitioning for GNN training and inference on arbitrary input graphs, which is especially challenging for distributed inference on new graphs where no existing performance measurements are available. We introduce an *online-linear-regression-based graph partitioner* that takes the runtime performance measurements of previously processed graphs as training samples for a cost model, which is then used to predict performance on arbitrary new graphs and enable efficient partitioning.

We formulate graph partitioning for GNNs as an *online learning* task. The performance measurements on partitioned graphs are training samples. Each training iteration produces new data points, and the graph partitioner computes a balanced graph partitioning based on all existing data points.

4.4.1 Cost Model

The key component of the Roc graph partitioner is a cost model that predicts the execution time of computing a GNN layer on an arbitrary graph, which could be the whole or any subset of an input graph. Note that the cost model learns to predict the execution time of a GNN layer instead of an entire GNN architecture for two reasons. First, Roc exploits the composability of neural network architectures and the learned cost model can be directly applied to a variety of GNN architectures. Second, this approach allows Roc to gather much more training data in each training iteration. For a GNN architecture with N layers and P partitions, Roc collects $(N \times P)$ training data points, while modeling the entire GNN architecture only provides P data points.

Table 4.2: The vertex features used in the current cost model. The semantics of the features are described in Section 4.4.1. `WS` is the number of GPU threads in a warp, which is 32 for the V100 GPUs used in the experiments.

	Definition	**Description**
x_1	1	the vertex itself
x_2	$\lvert\mathcal{N}(v)\rvert$	number of neighbors
x_3	$\lvert\mathcal{C}(v)\rvert$	continuity of neighbors
x_4	$\sum_i \lceil \frac{c_i(v)}{\texttt{WS}} \rceil$	# mem. accesses to load neighbors
x_5	$\sum_i \lceil \frac{c_i(v) \times d_{in}}{\texttt{WS}} \rceil$	# mem. accesses to load the activations of all neighbors

As collecting new training data points is expensive, requiring measuring GNN computations on GPU devices, we employ a simple linear regression model to minimize the number of trainable parameters. Our model assumes that the cost to perform a DNN operation on a vertex is linear in a collection of vertex features, such as number of neighbors, and the cost to run an arbitrary graph is the summation of the cost of all its vertices.

We formalize the cost for running a GNN layer l on an input graph $\mathcal{G}$ as follows.

$$\begin{aligned} t(l, v) &= \sum_i w_i(l) x_i(v) && (4.4) \\ t(l, \mathcal{G}) &= \sum_{v \in \mathcal{G}} t(l, v) = \sum_{v \in \mathcal{G}} \sum_i w_i(l) x_i(v) && (4.5) \\ &= \sum_i w_i(l) \sum_{v \in \mathcal{G}} x_i(v) = \sum_i w_i(l) x_i(\mathcal{G}) && (4.6) \end{aligned}$$

where v denotes a vertex in the input graph $\mathcal{G}$, $w_i(l)$ is a trainable parameter for layer l, $x_i(v)$ is the i-th feature of v, and $x_i(\mathcal{G})$ sums up the i-th feature of all vertices in $\mathcal{G}$.

Our model minimizes the mean square error over all available data points.

$$Loss(l) = \frac{1}{N} \sum_{i=1}^{N} \big(t(l, \mathcal{G}_i) - y(l, \mathcal{G}_i) \big)^2 \qquad (4.7)$$

where N is the total number of available data points for the GNN layer l, and $y(l, \mathcal{G}_i)$ is the performance measurement for the i-th data point.

Table 4.2 lists the vertex features used in the cost model; $x_1(v)$ and $x_2(v)$ capture the computation workload associated with vertex v and its edges, respectively. The remaining features estimate the required memory accesses to GPU device memory. Recall that when multiple threads in a GPU warp issue memory references to consecutive memory addresses, the GPU automatically *coalesces* these references to a single memory access that is handled more efficiently. To describe continuity of a vertex's neighbors, we partition all neighbors of v as $\mathcal{C}(v) = \{c_1(v), ..., c_{|C|}(v)\}$, where each $c_i(v)$ is a range of consecutively numbered vertices. For example, for vertex v_1 with neighbors $\{v_3, v_4, v_6, v_8\}$,

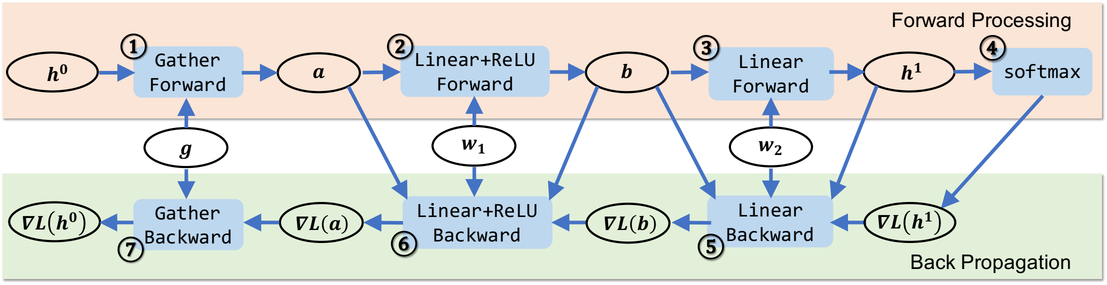


Figure 4.3: The computation graph of a toy 1-layer GIN architecture [89]. A box represents an operation, and a circle represents a tensor. Arrows indicate dependencies between tensors and operations. The `gather` operation performs neighborhood aggregation. The `linear` and the following `ReLU` are fused into a single operation as a common optimization in existing frameworks. h^0 and g denote the input features and neighbors of all vertices, respectively. w_1 and w_2 are the weights of the two linear layers.

we have $c_1(v_1) = \{v_3, v_4\}$, $c_2(v) = \{v_6\}$, and $c_3(v) = \{v_8\}$. The feature $x_3(v)$ is the number of consecutive blocks in v's neighbors, which is 3 in the example. In addition, $x_4(v)$ and $x_5(v)$ estimate the number of GPU memory accesses to load all neighbors and their input activations.

The cost model can be easily extended to include new features to capture additional model- and hardware-specific information if needed.

4.4.2 Partitioning Algorithm

Using the learned cost model, the Roc graph partitioner computes a graph partitioning that achieves balanced workload distribution under the cost model.

Roc uses the graph partitioning strategy we developed in Lux [43] to maximize coalesced accesses to GPU device memory, which is critical to achieve optimized GPU performance. Each vertex in a graph is assigned a unique number between 0 and $V-1$, where V is the number of vertices in the graph. In Roc, each partition holds consecutively numbered vertices, which allows us to use $N-1$ numbers $\{p_0, p_1, ..., p_{N-1}\}$ to partition the graph into N subgraphs where the i-th subgraph contains all vertices ranging from p_{i-1} to $p_i - 1$ and their in-edges.

Roc preprocesses an input graph by computing the partial sums of each vertex feature, which allows Roc to estimate the runtime performance of a subgraph in $O(1)$ time. In addition, Roc uses *binary search* to find a splitting point p_i in $O(\log V)$, and therefore computing balanced partitioning only takes $O(N \log V)$ time, where N and V are the number of partitions and input vertices, respectively.

Table 4.3: All the valid states and their activation tensors for the GNN architecture in Figure 4.3.

Valid State $\mathcal{S}$	**Activation Tensors $\mathcal{A}(\mathcal{S})$**
{①}	$\{g, a\}$
{①, ②}	$\{g, a, b, w_1\}$
{①, ②, ③}	$\{g, a, b, h^1, w_1, w_2\}$
{①, ②, ③, ④}	$\{g, a, b, w_1, w_2, \nabla L(h^1)\}$
{①, ②, ③, ④, ⑤}	$\{g, a, b, w_1, \nabla L(b)\}$
{①, ②, ③, ④, ⑤, ⑥}	$\{g, a, \nabla L(a)\}$
{①, ②, ③, ④, ⑤, ⑥, ⑦}	$\{\}$

4.5 Memory Manager

As discussed in Section 4.3, Roc performs all GNN computations on GPUs to optimize runtime performance, but only requires all the GNN data to fit in the host CPU DRAM to support large GNN architectures and input graphs. The device memory of each GPU therefore only needs to cache a subset of intermediate tensors, whose corresponding data transfers between CPU and GPU memories can be saved to reduce communication cost. How to select this subset of tensors to minimize the data transfers within the limited GPU memory is a critical memory management problem. The optimal strategy depends not only on the GPU device memory capacity and the sizes of the input graph and GNN tensors, but also on the topology of the GNN architecture, which determines the reuse distance for each tensor.

The page replacement algorithms for memory management in operating systems [12] assume pages are all the same size and that pages are accessed sequentially. Neither assumption holds for GNN computations since tensors generally have different sizes, and an operator may access multiple tensors simultaneously. More fundamentally, page replacement algorithms are history-based and assume no knowledge of future accesses, whereas in the case of a GNN we have access to the program and can statically determine at any point which tensors will be accessed in the future.

Roc formulates GPU memory management as a cost minimization problem: given an input graph, a GNN architecture, and a GPU device, find the subset of tensors to cache in the GPU memory that minimizes data transfers between the CPU and GPU. Roc introduces a dynamic programming algorithm to quickly find a globally optimal solution.

The key insight of the dynamic programming algorithm is that, at each stage of the computation, we only need to consider caching tensors that will be reused by future operations. For a GNN architecture $\mathcal{G}$, we define a *state* $\mathcal{S}$ to be the set of operations that have already been performed in $\mathcal{G}$. A state is valid only if the operations it contains preserve all the data dependencies in $\mathcal{G}$, i.e., for any operation in $\mathcal{S}$, all its predecessor operations in $\mathcal{G}$ must be also in $\mathcal{S}$. Such a definition allows the valid states to capture all possible execution orderings of the operators in $\mathcal{G}$. For each state $\mathcal{S}$, we define its *live tensors* $\mathcal{A}(\mathcal{S})$ to be the set of tensors that were produced by the operations in $\mathcal{S}$ and will be consumed as inputs by the operations outside of $\mathcal{S}$. Intuitively, $\mathcal{A}(\mathcal{S})$ captures all the tensors we can cache in the GPU to eliminate future data transfers at the stage $\mathcal{S}$.

Figure 4.3 shows the computation graph of a toy 1-layer Graph Isomorphism Network [89], whose computation can be formalized as following.

$$h_v^{(1)} = W_2 \times \textsc{ReLU}(W_1 \times \sum_{u \in \mathcal{N}(v)} h_u^{(0)}) \tag{4.8}$$

For this GNN architecture, all the valid states and their live tensors are listed in Table 4.3.

Algorithm 3 A recursive dynamic programming algorithm for computing minimum data transfers. $\textsc{In}(o_i)$ and $\textsc{Out}(o_i)$ return the input and output tensors of the operation o_i, respectively, and size($\mathcal{T}$) returns the memory space required to save all tensors in $\mathcal{T}$.

```
 1: Input: An input graph $g$, a GNN architecture $\mathcal{G}$, and the GPU device memory capacity $cap$.
 2: Output: Minimum data transfers required to compute $\mathcal{G}$ on $g$ within capacity $cap$.
 3: ▷ $\mathcal{D}$ is a database storing all computed Cost functions.
 4:
 5: function Cost($\mathcal{S}$, $\mathcal{T}$)
 6:     if $(\mathcal{S}, \mathcal{T}) \in \mathcal{D}$ then
 7:         return $\mathcal{D}(\mathcal{S}, \mathcal{T})$
 8:     end if
 9:     if $\mathcal{S}$ is $\varnothing$ then
10:         return size($\mathcal{T}$)
11:     end if
12:     $cost \leftarrow \infty$
13:     for $o_i \in \mathcal{S}$ do
14:         if $(\mathcal{S} \setminus o_i)$ is a valid state then
15:             $\mathcal{S}' \leftarrow \mathcal{S} \setminus o_i$
16:             $\mathcal{T}' \leftarrow \big(\mathcal{T} \setminus \textsc{Out}(o_i)\big) \cap \mathcal{A}(\mathcal{S}')$
17:             $xfer \leftarrow \text{size}\big(\textsc{In}(o_i) \setminus \mathcal{T}'\big)$
18:             if $\text{size}\big(\mathcal{T} \cup \textsc{In}(o_i) \cup \textsc{Out}(o_i)\big) \leq cap$ then
19:                 $cost = \min\{cost, \textsc{Cost}(\mathcal{S}', \mathcal{T}') + xfer\}$
20:             end if
21:         end if
22:     end for
23:     $\mathcal{D}(\mathcal{S}, \mathcal{T}) \leftarrow cost$
24:     return $\mathcal{D}(\mathcal{S}, \mathcal{T})$
25: end function
```

Since the valid states represent all the possible execution orderings of the GNN, we can use dynamic programming to compute the optimal memory management strategy associated with each execution state. Algorithm 3 shows the pseudocode. $\textsc{Cost}(\mathcal{S}, \mathcal{T})$ computes the minimum data transfers required to compute all the operations in a state $\mathcal{S}$, with $\mathcal{T}$ being the set of tensors cached in the GPU memory; $\mathcal{T}$ should be a subset of $\mathcal{A}(\mathcal{S})$. We reduce the task of computing $\textsc{Cost}(\mathcal{S}, \mathcal{T})$ to smaller tasks by enumerating the last operation to perform in $\mathcal{S}$ (line 13). The cost is the specific data transfers to perform this last operation (*xfer* in line 17) adding the cost of the corresponding previous state $(\mathcal{S}', \mathcal{T}')$. To improve performance, we leverage memoization to only evaluate $\textsc{Cost}(\mathcal{S}, \mathcal{T})$ once for each $(\mathcal{S}, \mathcal{T})$ pair.

Time and space complexity. Overall, the time and space complexity of Algorithm 3 are $O(S^2T)$ and $O(ST)$, respectively, where S is the number of possible execution states for a GNN architecture, and T is the maximum number of available tensor sets for a state. We observed that S and T are at most 16 and 4096 for all GNN architectures in our experiments, making it practical to use the dynamic programming algorithm to minimize data transfer cost.

4.6 Implementation

Roc is implemented on top of FlexFlow (see Chapter 3), and extends FlexFlow in the following aspects to support efficient GNN computations. First, we have replaced the equal partitioning strategy in FlexFlow with a fine-grained partitioning interface that supports splitting tensors at arbitrary points. This extension is critical to efficient partitioning for GNN computations. Second, we have added a graph propagation engine to support neighborhood aggregation operations in GNNs, such as the `gather` operation in Figure 4.3. To achieve performant graph propagation on GPUs, we have reused the highly optimized CUDA kernels in Lux [43], which is a multi-GPU system we developed for graph analytics. This adoption allows Roc to directly benefit from all kernel-level optimizations in Lux.

4.7 Evaluation

In this section, we aim to evaluate the following points:

- Can Roc achieve comparable runtime performance compared to state-of-the-art GNN frameworks on a single GPU?
- Can Roc improve the end-to-end performance of distributed GNN training and inference?
- Can we improve the model accuracy on existing datasets by using larger and more sophisticated GNNs?

4.7.1 Experimental Setup

GNN architectures. We use three real-world GNN architectures to evaluate Roc. GCN is a widely used graph convolutional network for semi-supervised learning on graph-structured data [49]. GIN is provably the most expressive GNN architecture for the Weisfeiler-Lehman graph isomorphism test [89]. CommNet consists of multiple cooperating agents that learn to communicate amongst themselves before taking actions [76].

Datasets. We use four real-world graph datasets in our evaluation, listed in Table 4.4. Pubmed is a citation network dataset [72], containing sparse bag-of-words feature vectors for each document (i.e.,

Table 4.4: Graph datasets used in our evaluation.

Dataset	Vertex	Edge	Feature	Label
Pubmed	19,717	108,365	500	3
PPI	56,944	1,612,348	700	121
Reddit	232,965	114,848,857	602	41
Amazon	9,430,088	231,594,310	300	24

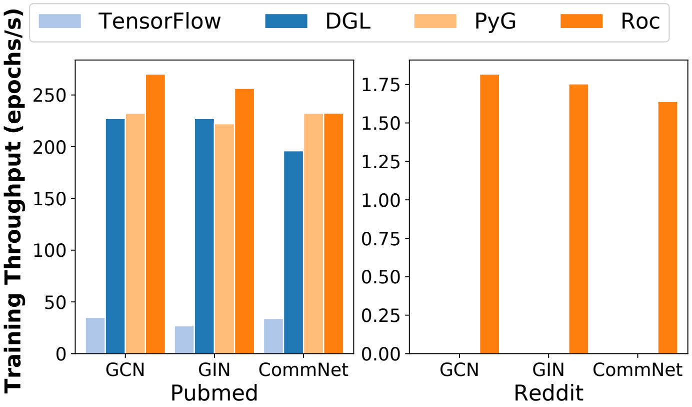


Figure 4.4: End-to-end training throughput comparison between existing GNN frameworks and Roc on a single P100 GPU (higher is better).

vertex), and citation links between documents (i.e., edges). PPI contains a number of protein-protein interaction graphs, each of which represents a human tissue [37]. Reddit is a dataset for online discussion forum, with each node being a post, and each edge being a comment between posts [37]. Amazon is the product dataset from Amazon [40]. Each node is a product, and each edge represents also-viewed information between products. The task is to categorize a product using its description and also-viewed relations.

All experiments were performed on a GPU cluster with 4 compute nodes, each of which contains two Intel 10-core E5-2600 CPUs, 256GB DRAM, and four NVIDIA Tesla P100 GPUs. GPUs on the same node are connected with NVLink, and nodes are connected with 100Gb/s EDR Infiniband.

For each training experiment, the Roc graph partitioner learned a new cost model by only using performance measurements obtained during the single experiment. For each inference experiment, the graph partitioner used the learned cost model from the training phase on the same dataset.

Unless otherwise stated, all experiments use the same training/validation/test splits as prior work [37, 49, 40]. All training throughput and inference latency were measured by averaging 1,000 iterations.

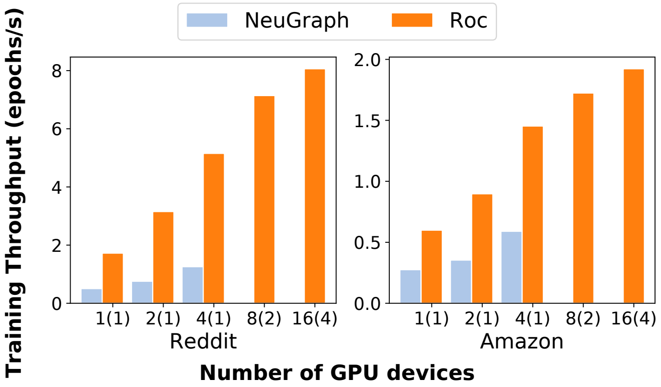


Figure 4.5: Training throughput comparison between NeuGraph and Roc using different numbers of GPUs (higher is better). Numbers in parenthesis are the number of compute nodes used in the experiments.

4.7.2 Single-GPU Results

First, we compare the end-to-end training performance of Roc with existing GNN frameworks on a single GPU. Due to the small device memory on a single GPU, we limited these experiments to graphs that can fit in a single GPU.

Figure 4.4 shows the results among TensorFlow [11], DGL [5], PyG [30], and Roc. We expected that Roc would be slightly slower than the other frameworks on a single GPU, since it writes the output tensors of each operator back to CPU DRAM for distributed computation, while other frameworks keep all tensors in a single GPU, and do not involve such data transfers. However, for these graphs, Roc reuses cached tensors on the GPU to minimize data transfers from DRAM to GPU, and overlaps the data transfers back to DRAM with subsequent GNN computations. As a result, Roc achieves on-par or even better performance compared to the other GNN frameworks on a single GPU. The performance improvement over existing frameworks is likely due to the highly-optimized CUDA kernels adopted from Lux for fast graph propagation on GPUs, while the other frameworks such as TensorFlow and DGL perform sparse matrix multiplications on GPUs for graph propagation.

TensorFlow, DGL, and PyG were not able to run the Reddit dataset due to out-of-device-memory errors. Roc can still train Reddit on a single GPU, by using DRAM to save some of the intermediate tensors.

4.7.3 Multi-GPU Results

Second, we compare the end-to-end training performance of Roc with NeuGraph. NeuGraph supports GNN training across multiple GPUs on a single compute node.

A NeuGraph implementation is not yet available publicly, so we run Roc using the same GPU version and software library versions cited in [57] and directly compare with the performance numbers

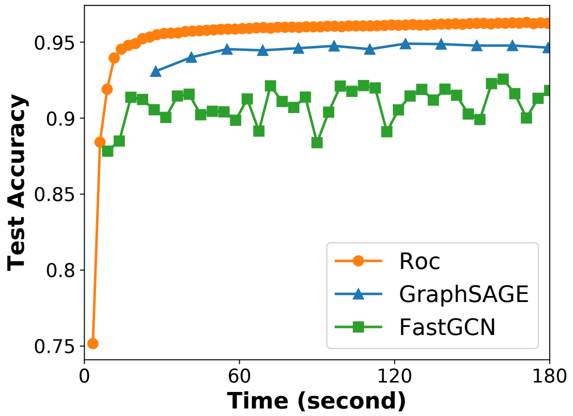


Figure 4.6: Time-to-accuracy comparison between state-of-the-art sampling techniques and Roc on the Reddit dataset [37]. All experiments used the same GCN model. Roc performed full-batch training on the entire graph, while GraphSAGE and FastGCN performed mini-batch sampling. Each dot indicates one training epoch for GraphSAGE and FastGCN, and five epochs for Roc.

reported in the NeuGraph paper. We also disabled NVLink for this experiment to rule out the effect of NVLink, which was not used in [57]. We do not claim that these comparisons control for all possible differences as well as directly executing both systems on the same machine, but that preferred approach was simply not available to us.

Figure 4.5 shows the results. For experiments on a single compute node, Roc outperforms NeuGraph by up to 4×. The speedup is mainly because of the graph partitioning and memory management optimizations that are not available in NeuGraph. First, NeuGraph uses the *equal vertex partitioning* strategy that equally distributes the vertices across multiple GPUs. Section 4.7.6 shows that the linear regression-based graph partitioner in Roc improves training throughput by up to 1.4× compared to the equal vertex partitioning strategy. Second, NeuGraph uses a stream processing approach that partitions each GNN operation into multiple chunks, and sequentially streams each chunk along with its input data to GPUs. Therefore, it does not consider the memory management optimization used in Roc, and Section 4.7.7 shows that the Roc memory manager improves training throughput by up to 2×.

The remaining performance improvement is likely due to other aspects of Roc, such as the use of the highly optimized CUDA kernels in Lux for fast graph propagation, and the performance of the underlying Legion runtime [15]. However, we were not able to further investigate the performance difference due the absence of a publicly available implementation of NeuGraph.

4.7.4 Comparison with Graph Sampling

We compare the training performance of Roc with state-of-the-art graph sampling approaches on the Reddit dataset. All frameworks use the same GCN model [49]. Roc performs full-batch training

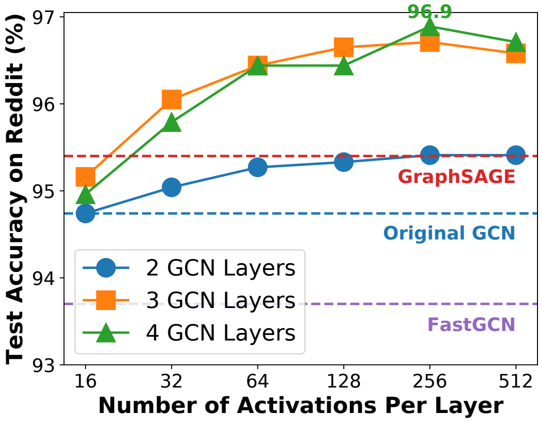


Figure 4.7: Test accuracy on the Reddit dataset using deeper and larger GNN architectures. The dotted lines show the best test accuracy achieved by GraphSAGE (95.4%), FastGCN (93.7%), and the original GCN architecture (94.7%), respectively.

on the entire graph as in [49], while GraphSAGE and FastGCN uses mini-batch sampling with a batch-size of 512.

Figure 4.6 shows the time-to-accuracy comparison on a single P100 GPU, where the x-axis shows the end-to-end training time for each epoch, and the y-axis shows the test accuracy of the current model at the end of each epoch. For GraphSAGE and FastGCN, each dot indicates one training epoch, while for Roc each dot represents five training epochs for simplicity. Note that GraphSAGE and FastGCN can achieve relatively high accuracy within a few training epochs. For example, GraphSAGE achieves 93.4% test accuracy in two epochs. However, Roc requires around 20 epochs to achieve the same test accuracy because Roc uses full-batch training (following [49]), and only updates parameters once per epoch, while existing sampling approaches generally perform mini-batch training and have more frequent parameter updates. Even though Roc uses more epochs, it is still as fast or faster than GraphSAGE and FastGCN to any given level of accuracy.

4.7.5 Deeper and Larger GNN Architectures

Roc enables the exploration of larger and more sophisticated GNN architectures than those possible in existing frameworks. As a demonstration, we consider a class of deep GNN architectures formed by stacking multiple GCN layers [49]. We add residual connections [39] between subsequent GCN layers to facilitate training of deeper GNN architectures by allowing preservation of information learned from previous layers.

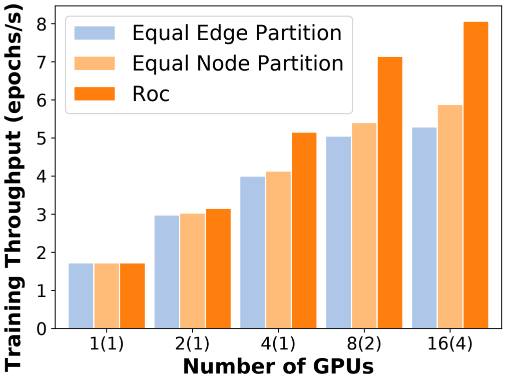


Figure 4.8: Training throughput comparison among different graph partitioning strategies on the Reddit dataset (higher is better). Numbers in parentheses are the number of compute nodes used.

Formally, each layer of our GNN is defined as follows.

$$H^{(k+1)} = \begin{cases} GCN(H^{(k)}) + H^{(k)} & d(H^{(k+1)}) = d(H^{(k)}) \\ GCN(H^{(k)}) + WH^{(k)} & d(H^{(k+1)}) \neq d(H^{(k)}) \end{cases}$$

where GCN is the original GCN layer [49], and $d(\cdot)$ is the number of activations in the input tensor. When $H^{(k)}$ and $H^{(k+1)}$ have the same number of activations, we directly insert a residual connection between the two layers. When $H^{(k)}$ and $H^{(k+1)}$ have different numbers of activations, we use a linear layer to transform $H^{(k)}$ to the desired shape. This design allows us to add residual connections for all GCN layers.

We increase the *depth* (i.e., number of GCN layers) and *width* (i.e., number of activations per layer) to obtain larger and deeper GNN architectures beyond the commonly used 2-layer GNNs. Figure 4.7 shows the accuracy achieved by our GNN architectures on the Reddit dataset. The figure shows that improved accuracy can be obtained by increasing the depth and width of a GNN architecture. As a result, our GNN architectures achieve up to 96.9% test accuracy on the Reddit dataset, outperforming state-of-the-art sampling techniques by 1.5%.

4.7.6 Graph Partitioning

To evaluate the linear regression-based graph partitioner in Roc, we compare the performance of the graph partitioning achieved by Roc with (1) *equal vertex partitioning* and (2) *equal edge partitioning*; (1) is used in NeuGraph to parallelize GNN training, and (2) has been widely used in previous graph processing systems. Figure 4.8 shows the training throughput comparison on different sets of GPUs. Neither of these baseline strategies perform as well as the Roc linear regression-based partitioner.

To evaluate the distributed inference performance on new graphs not used during training, we

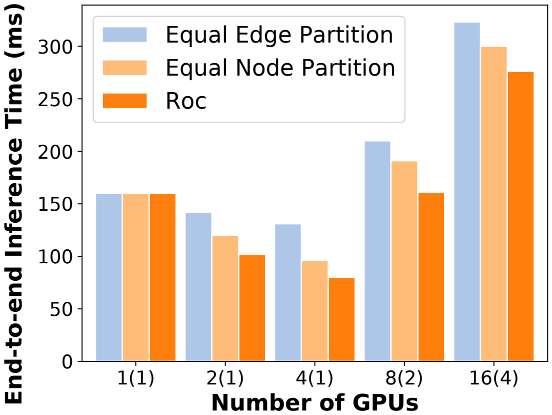
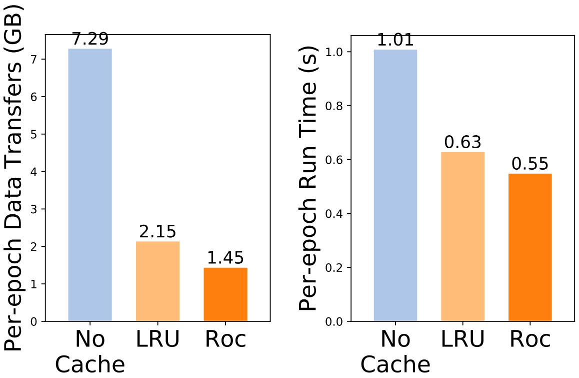


Figure 4.9: End-to-end inference time for the test graphs in the PPI dataset (lower is better). The numbers were measured by averaging the inference time of the four test graphs.



(a) Data transfers.

(b) Training time.

Figure 4.10: Performance comparison among different memory management strategies (lower is better). All numbers are measured by training GCN on the Reddit dataset on a single GPU.

used the PPI dataset containing 24 protein graphs. Following prior work [37], we trained the GIN architecture on 20 graphs and measured the inference latency on the remaining four graphs by using the graph partitioner learned during training. Figure 4.9 shows that the the learned cost model enables the graph partitioner to discover efficient partitioning on new graphs for inference services, by reducing the inference latency by up to 1.2×. For the PPI graphs, the distributed inference across multiple compute nodes achieves worse performance than the inference on a single node, which is due to the small sizes of the inference graphs.

4.7.7 Memory Management

We evaluate the performance of the Roc memory manager by comparing it with (1) the streaming processing approach in NeuGraph that streams input data along with computation (i.e., no caching optimization) and (2) the *least-recently-used* (LRU) cache replacement policy.

Figure 4.10 shows the comparison results for training GCN on the Reddit dataset on a single GPU. The dynamic programming-based memory manager reduces the data transfers between GPU and DRAM by 1.4–5× and reduces the per-epoch training time by 1.2–2× compared with the baseline memory management strategies.

4.8 Conclusion

To conclude, this chapter presents Roc, a distributed multi-GPU framework for high-performance and large-scale GNN training and inference. Roc partitions an input graph onto multiple GPUs on multiple compute nodes using an online-linear-regression-based strategy to achieve load balance, and coordinates optimized data transfers between GPU devices and host CPU memories with a dynamic programming algorithm. Roc increases the performance by up to 4× over existing GNN frameworks, and offers better scalability. The ability to process larger graphs and GNN architectures additionally enables model accuracy improvements. We achieve new state-of-the-art classification accuracy on the Reddit dataset by using significantly deeper and larger GNN architectures.

Chapter 5

Conclusions

We presented a search-based methodology to build efficient and scalable machine learning systems by automatically discovering potential ML optimizations. Compared to current ML systems relying on manually designed ML optimizations, our methodology provides better runtime performance, less engineering effort, and, for the work in Chapter 2, stronger correctness guarantees.

The search-based methodology proposed in this dissertation has two primary limitations.

First, our approach requires a search procedure to discover optimized solutions by exploring many (up to millions) of possible candidates. As a result, these automated techniques only apply to static ML models without data-dependent behaviors and optimize the models in an offline manner. One future research direction is designing automated techniques for optimizing dynamic ML models, which requires different methods to quickly discover performant strategies in a large and evolving search space. Reinforcement learning and other learning-based methods seem to provide a promising way to optimize dynamic ML models.

Second, the systems presented in this dissertation are designed for optimizing individual tasks of ML deployment (e.g., TASO for computation graphs, FlexFlow for parallelization, and Roc for data placement). However, joint optimization opportunities across multiple tasks of ML deployment are currently missing. Coordinating among multiple optimizations tasks requires combining the search spaces of individual tasks into a joint search space orders of magnitude larger than the individual ones. It remains an open research problem how to cooperatively design the search space, cost model, and search algorithm to discover joint and even end-to-end ML optimizations.

We have demonstrated the applicability of our approach using three ML systems that automatically discovers performance optimizations for different tasks in ML deployment.

All source code for the three systems is publicly available:

- TASO: `https://github.com/jiazhihao/TASO`
- FlexFlow: `https://github.com/FlexFlow/flexflow`

- Roc: `https://github.com/jiazhihao/Roc`

FlexFlow and Roc are both developed on top of the Legion distributed runtime, and Roc is also built on top of the Lux graph analytics system. Both Legion and Lux are also publicly available:

- Legion: `https://legion.stanford.edu`
- Lux: `https://github.com/LuxGraph/Lux`

Bibliography

[1] Amazon EC2 P3 Instances. https://aws.amazon.com/ec2/instance-types/p3/, 2017.

[2] Tensorflow graph transform creates corrupted graph. https://github.com/tensorflow/tensorflow/issues/7523, 2017.

[3] XLA: Optimizing Compiler for TensorFlow. https://www.tensorflow.org/xla, 2017.

[4] CANDLE Benchmarks. https://github.com/ECP-CANDLE/Benchmarks, 2018.

[5] Deep Graph Library: towards efficient and scalable deep learning on graphs. https://www.dgl.ai/, 2018.

[6] Graph transform: fold constant with invalid graph. https://github.com/tensorflow/tensorflow/issues/16545, 2018.

[7] Tensor Cores in NVIDIA Volta Architecture. https://www.nvidia.com/en-us/data-center/tensorcore/, 2018.

[8] The Criteo Ad Kaggle dataset. https://www.kaggle.com/c/criteo-display-ad-challenge, 2018.

[9] Uno: Predicting tumor dose response across multiple data sources. https://github.com/ECP-CANDLE/Benchmarks/tree/master/Pilot1/Uno, 2018.

[10] NVLink and NVSwitch: Advanced Multi-GPU Systems. https://www.nvidia.com/en-us/data-center/nvlink/, 2019.

[11] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI, 2016.

[12] Alfred V Aho, Peter J Denning, and Jeffrey D Ullman. Principles of optimal page replacement. *Journal of the ACM (JACM)*, 18(1):80–93, 1971.

[13] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014.

[14] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2006.

[15] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC, 2012.

[16] A New Lightweight, Modular, and Scalable Deep Learning Framework. `https://caffe2.ai`, 2016.

[17] Jie Chen, Tengfei Ma, and Cao Xiao. FastGCN: Fast learning with graph convolutional networks via importance sampling. In *Proceedings of the International Conference on Learning Representations*, ICLR, 2018.

[18] 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. *CoRR*, abs/1512.01274, 2015.

[19] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, Carlsbad, CA, 2018.

[20] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NeurIPS. 2018.

[21] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014.

[22] Berkeley R. Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. Semantic program alignment for equivalence checking. In *Proceedings of the 2019 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2019.

[23] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[24] Dense Linear Algebra on GPUs. `https://developer.nvidia.com/cublas`, 2016.

[25] Manjeet Dahiya and Sorav Bansal. Black-box equivalence checking across compiler optimizations. In *Asian Symposium on Programming Languages and Systems*, pages 127–147. Springer, 2017.

[26] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS/ETAPS, 2008.

[27] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems*, NIPS, 2012.

[28] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[29] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *CoRR*, abs/1603.07285, 2016.

[30] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[31] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. Tetris: Scalable and efficient neural network acceleration with 3d memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2017.

[32] Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis. Tangram: Optimized coarse-grained dataflow for scalable nn accelerators. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2019.

[33] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. Firmament: Fast, centralized cluster scheduling at scale. In *12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, Savannah, GA, 2016.

[34] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI, 2012.

[35] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. GraphX: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI, 2014.

[36] Sumit Gulwani and George C. Necula. Discovering affine equalities using random interpretation. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, 2003.

[37] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS. 2017.

[38] W. K. Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.

[39] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR, 2016.

[40] Ruining He and Julian McAuley. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In *Proceedings of the 25th International Conference on World Wide Web*, WWW, 2016.

[41] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[42] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP, 2009.

[43] Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. A Distributed multi-GPU System for Fast Graph Processing. *Proc. VLDB Endow.*, 11(3), November 2017.

[44] Zhihao Jia, Sina Lin, Charles R. Qi, and Alex Aiken. Exploring hidden dimensions in accelerating convolutional neural networks. In *Proceedings of the 35th International Conference on Machine Learning*, Proceedings of Machine Learning Research, 2018.

[45] Zhihao Jia, James Thomas, Todd Warzawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. Optimizing dnn computation with relaxed graph substitutions. In *Proceedings of the 2nd Conference on Systems and Machine Learning*, SysML, 2019.

[46] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In *Proceedings of the 2nd Conference on Systems and Machine Learning*, SysML, 2019.

[47] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA, 2017.

[48] Yoon Kim. Convolutional neural networks for sentence classification. *CoRR*, abs/1408.5882, 2014.

[49] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *Proceedings of the International Conference on Learning Representations*, ICLR, 2017.

[50] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997, 2014.

[51] 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, 2012.

[52] Shui Lam and Ravi Sethi. Worst case analysis of two scheduling algorithms. *SIAM Journal on Computing*, 6, 1977.

[53] Steve Lawrence, C Lee Giles, Ah Chung Tsoi, and Andrew D Back. Face recognition: A convolutional neural-network approach. *IEEE transactions on neural networks*, 1997.

[54] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2014.

[55] Yann LeCun. LeNet-5, convolutional neural networks. `http://yann.lecun.com/exdb/lenet`, 2015.

[56] 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*, SC, 2016.

[57] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. NeuGraph: Parallel Deep Neural Network Computation on Large Graphs. In *Proceedings of the 2019 USENIX Annual Technical Conference*, ATC, 2019.

[58] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD, 2010.

[59] Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of english: The penn treebank. *Comput. Linguist.*, 19.

[60] Henry Massalin. Superoptimizer: a look at the smallest program. In *ACM SIGARCH Computer Architecture News*, volume 15, 1987.

[61] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning*, ICML, 2017.

[62] Intel Math Kernel Library for Deep Neural Networks. `https://01.org/mkl-dnn`, 2016.

[63] Movie review data. `https://www.cs.cornell.edu/people/pabo/movie-review-data/`, 2005.

[64] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML, USA, 2010.

[65] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP, 2019.

[66] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.

[67] George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2000.

[68] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL, 2002.

[69] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, Held as Part of the European Joint Conferences on the Theory and Practice of Software*, TACAS, 1998.

[70] Tensors and Dynamic neural networks in Python with strong GPU acceleration. `https://pytorch.org`, 2017.

[71] 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 (IJCV)*, 115(3):211–252, 2015.

[72] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI magazine*, 29(3):93–93, 2008.

[73] Rahul Sharma, Eric Schkufza, Berkeley R. Churchill, and Alex Aiken. Data-driven equivalence checking. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 391–406, 2013.

[74] 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*, 529:484–489, 2016.

[75] Muthian Sivathanu, Tapan Chugh, Sanjay S. Singapuram, and Lidong Zhou. Astra: Exploiting predictability to optimize deep learning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, New York, NY, USA, 2019. ACM.

[76] Sainbayar Sukhbaatar, arthur szlam, and Rob Fergus. Learning multiagent communication with backpropagation. In *Proceedings of the 29th International Conference on Neural Information Processing Systems*, NIPS, 2016.

[77] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014.

[78] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[79] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: A new approach to optimization. *Logical Methods in Computer Science*, 7(1), 2011.

[80] NVIDIA TensorRT: Programmable inference accelerator. `https://developer.nvidia.com/tensorrt`, 2017.

[81] Sean Treichler, Michael Bauer, Rahul Sharma, Elliott Slaughter, and Alex Aiken. Dependent partitioning. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA. ACM, 2016.

[82] Sudharshan S Vazhkudai, Bronis R de Supinski, Arthur S Bland, Al Geist, James Sexton, Jim Kahle, Christopher J Zimmer, Scott Atchley, Sarp Oral, Don E Maxwell, et al. The design, deployment, and evaluation of the coral pre-exascale systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC. IEEE, 2018.

[83] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. *Proceedings of the International Conference on Learning Representations*, 2018.

[84] Shivaram Venkataraman, Erik Bodzsar, Indrajit Roy, Alvin AuYoung, and Robert S. Schreiber. Presto: Distributed machine learning and graph processing with sparse matrices. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys, 2013.

[85] Acl 2016 first conference on machine translation. `http://www.statmt.org/wmt16`, 2016.

[86] Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. Training and inference with integers in deep neural networks. In *Proceedings of the International Conference on Learning Representations*, ICLR, 2018.

[87] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.

[88] Saining Xie, Ross B. Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. *CoRR*, abs/1611.05431, 2016.

[89] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *Proceedings of the International Conference on Learning Representations*, ICLR, 2019.

[90] Hongxia Yang. Aligraph: A comprehensive graph neural network platform. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD. ACM, 2019.

[91] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD, New York, NY, USA, 2018. ACM.

[92] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *CoRR*, abs/1409.2329, 2014.

[93] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI. USENIX Association, 2016.

[94] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. *CoRR*, abs/1611.01578, 2016.

[95] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, CVPR, 2018.