Flexible and Efficient Systems for Training Emerging Deep Neural Networks

by

Minjie Wang

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

Courant Institute of Mathematical Sciences

New York University

January, 2020

Professor Jinyang Li

Dedication

To my beloved Naonao

Acknowledgements

I am very thankful to work with my advisor Jinyang in this thrilling decade of deep learning explosion. As a quite demanding student, I constantly have a lot of questions or doubts, about research or others, and the most often response I hear from Jinyang is *"Sure, let's chat."*. Her sharpness and enthusiasm in her work is truly inspiring and is something I hope I could replicate.

I am indebted to ZZ, who has always put his trust in me via numerous projects, granting me the chances to freely explore the research wild, leading me out of swamps and pushing me to enjoy most of the applause. As a system research veteran, ZZ makes his decisive shift to embrace machine learning, an intrepid move that teaches me an ever-lasting lesson – it is always day one. ZZ's insightful perspective often amazes me in one and yet another projects; some failed but none of them missed.

I am also thankful to Chien-chin Huang, with whom I co-authored the Tofu system (Chapter 3) in this dissertation, for his dedication to this work at the time he just became a father. I will always remember the image that he skillfully launched a hundred instances with a baby in his arm. My committee members, Kyunghyung cho, Joan Bruna and He He, offered me precious advises and supports on my projects and thesis, which I really appreciate. I thank Yann Lecun, who helped my fellowship recommendation, and Sebastian Angel, who polished my application and also my coffee taste.

I feel grateful to Tianqi Chen and Mu Li, who brought me into open source community, and who have served as fantastic mentors, for their open attitude to collaboration and genuine open source spirit that inspired the DGL project (Chapter 4). I thank Tianjun Xiao, Yutian Li, Haoran Wang, Larry Tang, Yu Gai, Ziheng Jiang, Sean Welleck, with whom I co-worked on the Minerva and MinPy projects. My co-authors and co-workers, Lingfan Yu, Quan Qan, Jake Zhao, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Da Zheng, Haibin Lin, Chao Ma, Damon Deng, Qipeng Guo, Hao Zhang, Ziyue Huang, made significant contributions to the design and implementation of DGL and I am grateful for their help.

I thank Lakshmi Subramanian and Mike Walfish for the advises and encouragement on my work and presentations. To all my friends at NYU, Cheng Tan, Shuai Mu, Zhaoguo Wang, Changgeng Zhao, Shiva Iyer, Christopher Michell, Ye Ji, Yan Shvartzshnaider, thank you for making the lab feel like a home. I thank Andy and Pat Wang for providing a wonderful homestay during my dissertation writing. Finally, I thank my family for their love and support.

Abstract

The success of deep neural networks (DNNs) is due to its strong capability to learn from data. To continue such success, DNNs must handle the ever-increasing size and complexity of data. Two concrete challenges stand out. One, to leverage more data, one needs to train a large DNN model, the size of which becomes limited by the memory capacity of a single GPU. The other, to leverage graph structured data, one needs to use DNN models that perform sparse numerical computation. Unfortunately, current deep learning systems do not provide adequate support for very large or sparse models. This thesis develops two systems, Tofu and DGL, to enable efficient training of these emerging DNNs while minimizing user programming efforts.

Tofu supports very large DNNs by partitioning the computation across multiple GPUs to reduce per-GPU memory footprint. To automatically partition each operator, we propose a description language for annotating the semantics of an operator. To optimally partition the whole training, Tofu proposes an algorithm that minimizes the total communication cost. We evaluate and assess the capability of Tofu to train very models demonstrating the substantial gains by applying the design.

DGL is a new framework for training DNNs for graph structured data. DGL provides an intuitive and expressive message-passing interface that can cover a wide range of graph DNN models. We introduce batching and kernel fusion techniques that enable training GNNs on large graphs and achieve significant improvements in performance relative to existing systems.

Table of contents

List of Figures

List of Tables

Chapter 1

Introduction

Recent years have seen tremendous success of deep learning. The resurgence of *Deep Neural Networks* (DNNs) have caused major paradigm shifts with substantial breakthroughs [1, 2, 3, 4, 5, 6]. The success of *Deep Learning* (DL) can be attributed to four factors: the abundance of data accessible as training sets, the innovation in DNN model architecture, the commoditization of general-purpose massively parallel computing devices such as GPUs and TPUs to accelerate training, and the development of deep learning frameworks such as TensorFlow/PyTorch/MXNet to simplify the programming of sophisticated models. The thesis focuses on the aspect of deep learning frameworks.

Current DL systems [7, 8, 9, 10, 11] typically structure the design into two parts. A *frontend* lets users write programs using tensor operators. The dependencies among the operators are then recorded, declaratively or imperatively, in a *tensor dataflow graph* (or *tensor computation graph*), which is evaluated concretely in a *backend*. The frontend determines the programmability of the DL framework. By supporting many tensor operations or new programming primitives, the frontend could cover a wide range of DNNs. The backend determines the performance of DL framework. Most existing backends run on a single GPU, while more performance can be achieved if the backend can utilize multiple GPUs. However, deep learning research keeps pushing the limits of DL systems as newly

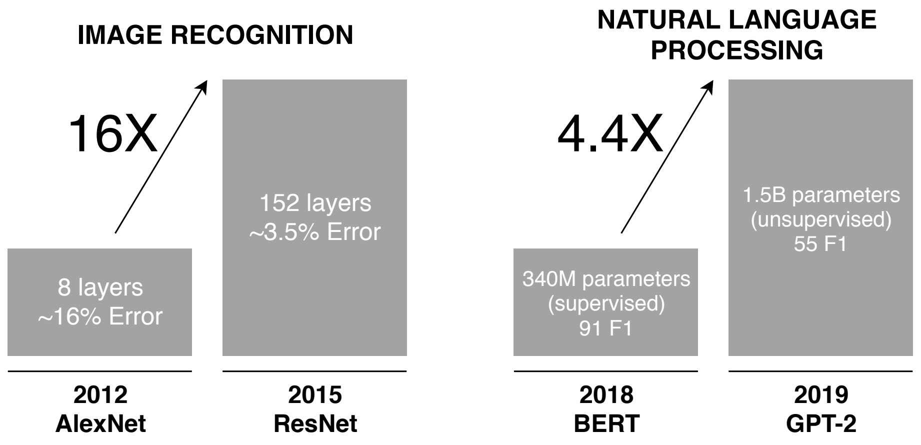


Figure 1.1: The growth of DNN model size.

developed DNN models become more complex in terms of both capacity and architecture.

One major trend of DNN development is to increase the model size. As shown in figure 1.1, the AlexNet [1] model that achieved 16% prediction error in the ImageNet image classification challenge in 2012 has only 8 layers, while in 2015, the ResNet [2] model has 152 layers and achieved 3.5% prediction error. The BERT [4] model proposed in 2018 that topped several natural language processing tasks has 340 million parameters, and was surpassed a year later by the GPT-2 [12] model that has 1.5 billion parameters. Empirical evidence shows that, since the 80s, the number of parameters in the state-of-the-art neural network has doubled roughly every 2.4 years [13]. As DNNs keep achieving impressive results with more data and parameters, training very large models beyond the capacity of a simple computing device (e.g., one GPU) becomes an immediate challenge to resolve.

Besides the scaling challenge, current DL systems are inadequate when dealing with the DNNs for structural data. These DNNs tend to define a computation over a sparse graph while current DL systems are highly optimized for dense tensor computation. As a result, users have to ponder implementation difficulty into model design. For example, the author of Graph Attention Networks [14] posted the following reply on OpenReview when was asked about the lack of result on Pubmed [15] dataset: *"Unfortunately, even though*

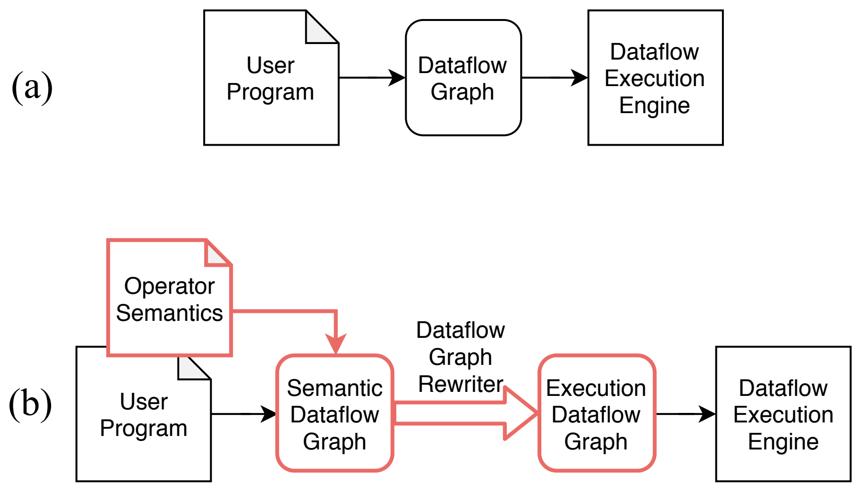


Figure 1.2: (a) Current DL system design. (b) SMEX system design.

the softmax computation on every node should be trivially parallelisable, we were unable to make advantage of our tensor manipulation framework to achieve this parallelisation, while retaining a favourable storage complexity ... and caused OOM errors on our GPU when the Pubmed dataset was provided."[1]. In fact, the Pubmed graph only has 20K nodes and 45K edges, which is several orders of magnitudes behind the scale of real-world graphs. Hence, an efficient and flexible system is highly desirable for training the DNNs for graph data.

The dissertation introduces two systems, Tofu and DGL, to address the two problems respectively. At a higher level, they share the same design principle we summarized as **SMEX**. Compared with the design of current DL systems, SMEX separates the semantic and runtime contexts to two stages and bridges them with a graph rewriter (figure 1.2). More precisely, the major components in the design are

- a **semantic dataflow graph** that includes the **semantic specification** of the involved tensor operations,
- a **dataflow graph rewriter** that analyzes the semantic dataflow graph and generates a new graph for execution,

1. `https://openreview.net/forum?id=rJXMpikCZ`

- and an **execution dataflow graph** that can efficiently run on one or multiple devices.

Thesis Statement. *The SMEX design can effectively expand the capacity and the range of DNN models that a deep learning system can support.*

We evaluate the statement by applying it to two systems (Tofu and DGL). Tofu targets the problem of training very large DNNs by partitioning and distributing the training to multiple computing devices. We design a domain-specific language which describes the fine-grained operator semantics. This allows the system to understand how to partition and parallelize each operator. We further develop an algorithm that can efficiently find the parallel strategy of an entire dataflow graph that is both load-balanced and communication-saving. The result is that Tofu can scale to DNNs that are much larger and achieve better speedup compared to previous work.

The second system DGL focuses on the emerging field of geometric deep learning. Since the models are meant for sparse and irregular graph data, it is challenging to train them efficiently in the DL systems optimized for dense tensor computation. DGL proposes a general programming interface specifying how the models compute on one node or one edge, and then generates efficient tensor dataflow graph for execution. The evaluation shows that DGL outperforms other specialized GNN frameworks by a large margin and can scale to graph of millions of nodes.

In the remainder of the chapter, we discuss more motivations behind the SMEX design and then highlight some key contributions.

1.1 Challenges with existing DL systems

Challenges in supporting very large DNNs The size of a DNN model that can be explored today is constrained by the limited GPU device memory. There have been many efforts to tackle the problem. Some proposals try to fit larger models into a single GPU, e.g. by using the much larger CPU memory as a swap area for the GPU [16] or by discarding intermediate results to save memory at the cost of re-computation [17, 18, 19]. Another promising solution is to partition a DNN model across multiple GPU devices. Doing so reduces per-GPU memory footprint and comes with the additional benefit of parallel speedup. This is commonly referred to as "model parallelism" in the literature.

A DNN model consists of a large number of layers, each parameterized by its own weights. There are two approaches to realize model parallelism. One approach is to assign the computation of different layers to different devices. The second approach is to partition the tensors to parallelize each layer across devices. For very large DNN models, tensor partitioning is the better approach; not only it results in balanced per-GPU memory usage but also it is necessary for speeding up popular models such as CNNs.

Early work on tensor partitioning [20, 21, 22] require users to manually partition tensors, which demands non-trivial analysis of the DNN architecture and implementation. Recent approaches [23, 24, 25] formulate it as an optimization problem and propose automatic solutions. However, they still rely on the knowledge on how to partition a few common layers, and thus support a limited type of DNN models. As a result, how to fully automate model partitioning for training very large DNNs remains a challenge.

Challenges in supporting DNNs for graph data A broad range of models can be unified as either learning from explicit or inferring latent structures. Examples include TreeLSTM [26] that works on sentence parsing trees, and the recent Graph Neural Networks (GNNs) family that aim to model a set of node entities together with their relationships (edges).

Unfortunately, existing tensor-based frameworks lack intuitive support for this trend of deep graph learning. Specifically, GNNs are defined using the message passing paradigm [6]. However, tensor-based frameworks do not support the message-passing interface. As such, researchers need to manually emulate graph computation using tensor operations, which poses an implementation challenge. Existing specialized tools [27] cannot fully utilize the sparse tensor operations involved in GNN training and hence cannot scale to graphs of moderate sizes. Moreover, none of the existing tools have support for GNNs over heterogeneous graphs.

1.2 Contributions

At a higher level, both challenges expose the same set of issues in current DL system design:

- **Entangled semantic and system contexts**: The dataflow graph in current DL systems expresses not only the actual training logic but also many system decisions such as device placement and communication pattern. Such practice is burdensome to the end user and also brittle to any changes of hardware environment and model architecture.

- **Lacking operator semantics**: The operators in existing DL systems are opaque and lack concrete semantics. In the case of automatic tensor partitioning, this prevents any solution to be applicable to a state-of-the-art DL system that can have hundreds of tensor operators. In terms of supporting DNNs for graphs, exposing operator semantics is vital to system optimizations such as automatic batching and fusion.

The SMEX system design tackles the problems in three steps. Firstly, a user program is directly represented by a *semantic dataflow graph* with few or no runtime contexts. Each operator has a specification describing how to compute each output element. The specification can be provided by an operator developer or naturally derived from a programming abstraction so that the effort is affordable. Secondly, a *dataflow graph rewriter* is in charge of transforming the semantic dataflow graph into an *execution dataflow graph*. The rewriter can implement algorithms to partition, fuse or batch operators for more parallelism as long as the two graphs produce the same result. The rewriter is also responsible of generating necessary device contexts and extra communication operators. Finally, the execution dataflow graph is evaluated by the existing DL system backend. The whole design is agnostic to the choice of DL platforms, making it widely applicable.

We then present two systems, Tofu and DGL, their technical contributions, and their connections to the SMEX design.

Tofu targets the scaling problem of training very large DNN models (§3). Given a DNN training program, Tofu aims to automatically partition every involved tensor and operator so that the partitioning is both memory-balanced and capable of accelerating the training. There are two technical challenges to resolve. (i) How to partition the input/output tensors and parallelize the execution an individual operator? What are the viable partition dimensions? (ii) how to optimize the partitioning of different operators for the overall graph? For (i), Tofu lets operator developers annotate operators in a lightweight *Tensor Description Language* (TDL). TDL describes tensor computation by specifying the output tensor value at each index with simple expressions over the input tensor elements. We then develop a technique based on symbolic execution to determine what input regions must be transferred across devices when a tensor are divided along a specific partition dimension (§3.4.2). For (ii), we formulate it as a combinatorial optimization problem that is proved to be NP-hard. We then propose several techniques to prune the search space including a recursive algorithm that can effectively reduce the search time from hours to seconds (§3.5.2). The results show that Tofu can train models that are 6× larger on 8 GPUs and achieve $60\% - 95\%$ of the ideal efficiency.

SMEX design in Tofu

- **Semantic dataflow graph:** A dataflow graph with TDL descriptions.
- **Dataflow graph rewriter:** Responsible for analyzing TDL annotations, discovering viable partition dimensions for individual operators, searching for global parallel strategy and generating the partitioned dataflow graph.
- **Execution dataflow graph:** A partitioned dataflow graph that can be executed on multiple devices.

DGL is a new framework for DNNs for structural data such as TreeLSTM and the models of the GNN family (§4). Following the message passing paradigm [6], DGL provides intuitive message passing programming interface, which describes the computation on a batch of nodes and edges (§4.5). Converting the program to an efficient tensor dataflow graph for execution requires batching. However, previous auto-batching approaches [28, 29] incur significant overhead from dataflow graph construction. By contrast, DGL forms batches by analyzing the input graph and generates the dataflow graph for a batch of nodes, giving an up to 10× speedup (§4.6.1). Furthermore, we develop a kernel fusion technique that avoids storing explicit messages, significantly improve both memory consumption and training speed (§4.6.2). As a result, DGL can scale to graphs with hundreds of millions of edges, and can train up to 7.5× faster than other specialized GNN frameworks.

SMEX design in DGL

- **Semantic dataflow graph:** A dataflow graph with operators described in message passing paradigm.
- **Dataflow graph rewriter:** Responsible for analyzing graph structure, auto-batching and kernel fusion.
- **Execution dataflow graph:** A dataflow graph with batched or fused operators.

1.3 Roadmap

The rest of the dissertation is organized as follows. Chapter 2 discusses more details about related backgrounds including the evolution of deep learning systems and the prior work for machine learning on graphs. Chapter 3 describes the overall design of Tofu, its technical solutions and evaluations. We also provide a theoretical analysis of the proposed recursive search algorithm in Appendix A. Chapter 4 discusses in detail the challenges of building a graph DNN system, and explains how the programming interface and optimization techniques of DGL address them. Finally, Chapter 5 summarizes this work, lists its limitations and provides an outlook of future directions.

Chapter 2

Related Work

2.1 Deep Learning Systems

The growing interest in deep learning leads to an explosion of specialized tools and systems over the decade. Early tools such as Cuda-Convnet [30], Caffe [31] and CXXNet [32] let users write a configuration script consisting of a stack of common neural network layers and their hyper-parameters. The models are trained by some built-in, opaque routines that perform forward and backward propagation. The design heavily targets the DNNs used in computer vision, by resembling the construction of feed-forward neural networks, at a cost of support of models like recurrent neural networks. Seeing the incompetence, frameworks such as Torch [33] and Theano [34] provide an *tensor programming interface*, which expresses neural networks by operators of multi-dimensional arrays (or tensors) and embeds them into popular programming languages such as Lua, Python and C++. Although these frameworks are capable of accelerating DNN training by utilizing specialized operator libraries such as cuDNN [35] and MKLDNN [36], training models in scale remains a challenge.

Inspired by the dataflow systems for cluster computing [37, 38, 39], modern deep learning systems such as MXNet [7], Tensorflow [8], Caffe2 [9], Chainer [10], PyTorch [11] and

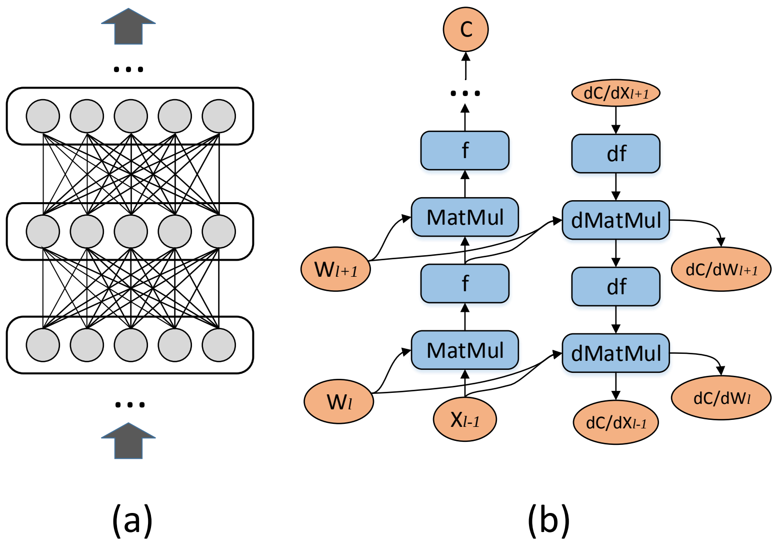


Figure 2.1: (a) A Multi-Layer Perceptron (MLP) model; (b) The dataflow graph for its forward and backward propagations.

Minerva [21] tackle the scalability challenge by translating the DNN training program in array-based interface into *tensor dataflow graph representation*. At its core, a tensor dataflow graph is a directed acyclic graph consisting of two types of nodes: tensor and operator. An edge connects either a tensor node to its producer or consumer operators or two operator nodes indicating the data dependency. Figure 2.1 shows an example dataflow graph for the forward and backward propagation of a Multi-Layer Perceptron (MLP) model. The weights of neuron connections between successive layers l and $l+1$ are represented by matrix W_l. The forward propagation calculates the training loss C by repeatedly computing the activation of a layer from its preceding layer. Specifically, $x_{l+1} = f(x_l \cdot W_l)$, where layer l's activation vector x_l is multiplied with the weight matrix W_l and then scaled using an element-wise non-linear function f. After that, the backward propagation calculates the gradient matrix $\frac{dC}{dW_l}$ of each weight matrix W_l from higher to lower layers. The dataflow graph thus consists of two towers for the two passes.

Using tensor dataflow graph to represent DNN training has many benefits. Firstly, the backward computation following the chain rule can be derived automatically from the dataflow graph for the forward computation. In the simple example, we could roughly observe the correspondence between the two towers; for each matrix multiplication `MatMul` or non-linear function f in the forward pass, there is the derivative function `dMatMul` or df in the backward pass. Auto-differentiation has been the de-facto equipment in modern DL systems, an essential enabler for training deep and complex neural networks.

Secondly, dataflow graph representation allows modern DL systems to train DNNs more efficiently because operators with no data dependency can be executed in parallel, which suits well with multi-threading execution on CPU and multi-stream model on GPU. Another benefit from an articulate expression of parallelism is that it eases the system design for distributed training by naturally overlapping independent network communication with computation. Built upon it, many research work have explored different approaches for parallelizing DNN training, which we elaborate in §2.2.

Another question is how to extract dataflow graph representation from user programs, and perhaps more importantly, how to design the programming interface? There are two considerations. On one hand, there has been a long history of programming machine learning algorithms in a dynamic scripting language such as Python, which proven to be very productive. On the other hand, efficient execution and many program analysis and optimization techniques require a strongly-typed static language. To strike a balance, DL systems such as Tensorflow and MXNet adopt a declarative programming interface. Users express training logic such as loss and gradient calculation symbolically in a domain-specific language (DSL) embedded in a dynamic language (e.g. Python), and perform concrete evaluation afterwards. This design leans towards execution efficiency at a cost of debuggability and usability because it artificially segments a user program into declaration and execution parts. With the explosion of DL research, DL systems shift more and more toward the productivity end. PyTorch, MXNet Gluon [40] and Tensorflow v2.0 all focus on imperative programming paradigm, where each tensor operator not only records data dependency but also concretely calculates the result. The insight is that although such dynamic execution mode (or eager mode) misses certain optimization chances when the

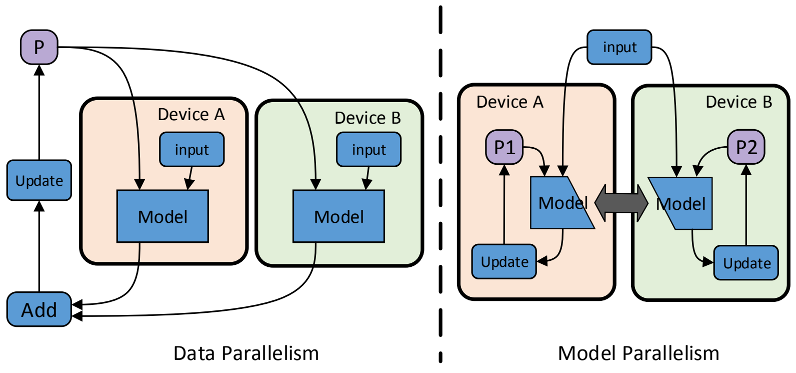


Figure 2.2: Data parallelism and model parallelism

whole program is available, the strong coherence between two training iterations fuels many runtime optimization techniques such as cached memory allocator, leading to competitive performance compared to symbolic mode [11].

A dataflow graph, no matter how it is extracted, is a form of intermediate representation (IR) of a user program, which by design is conductive for further optimization and translation. Many previous work have proposed techniques based on dataflow graph for memory optimization [19, 17], auto-batching [29, 28] and smart device placement [41]. NNVM [42] and ONNX [43] further define a unified and canonical dataflow format to enable the interoperability between different frameworks and streamline the path from research to production. However, dataflow representation has the fundamental defect for expressing control flows which limits its applicable scope. The rise of *differentiable programming paradigm*[1] pushes new IRs being developed [44, 45] that are both universal (i.e. Turing complete) and strongly-typed array language, which opens up new research directions such as auto-differentiation techniques [46, 47] and optimizations.

2.2 Parallel DNN Training

The most widely used method for scaling DNN training today is **data parallelism**. Traditional DNN training is based on batched stochastic gradient descent where the batch

1. `https://www.facebook.com/yann.lecun/posts/10155003011462143`

size is kept deliberately small. Within a batch, computation on each sample can be carried out independently and aggregated at the end of the batch. Data parallelism divides a batch among several GPU devices and incurs cross-device communication to aggregate and synchronize model parameters at the end of each batch using a parameter service [48, 49].

Data parallelism has achieved good speedup for some DNN models (e.g. Inception network [50]). However, since the communication overhead of data parallelism increases as the model grows bigger, one must train using a very large batch size to amortize the communication cost across many devices. In fact, for any DNN model, one can always scale the "training throughput" by ever increasing the batch size. Unfortunately, large batch training is known to be problematic such as longer convergence time or decreased model accuracy [51, 22].

Model parallelism partitions the model parameters of each layer among devices, so that the update of parameters can be performed locally (Figure 2.2). Each device can only calculate part of a layer's activation using its parameter partition, so all devices need to synchronize their activations and activation gradients for each layer during both the forward and backward propagations. Since model parallelism exchanges activations instead of the model parameters, it works well for models with small activation size such as DNN models with large fully-connected layers.

The trade-off between data and model parallelism leads to the development of more complex strategies. For examples, **mixed parallelism** [22] distributes some DNN layers using data parallelism while using model parallelism for others. **Combined parallelism** divides workers into groups and uses different strategies for inter-group and intra-group communication. Combined parallelism was proposed in the earlier generation of specialized training systems [48, 52], but is not thoroughly explored due to its programming complexity.

Chapter 3

Tofu: Supporting Very Large Models using Automatic Dataflow Graph Partitioning

In this chapter, we applied the SMEX design to the context of partitioning dataflow graph for training very large models. We will show that by separating the design into semantic and execution dataflow graphs, Tofu can efficiently utilize the aggregated memory capacity of multiple devices for training very large models with very little user effort.

3.1 Introduction

The deep learning community has been using larger deep neural network (DNN) models to achieve higher accuracy on more complex tasks over the past few years [53, 54]. Empirical evidence shows that, since the 80s, the number of parameters in the state-of-the-art neural network has doubled roughly every 2.4 years [13], enabled by hardware improvements and the availability of large datasets. As deployed DNN models remain many orders of magnitude smaller than that of a mammalian brain, there remains much room for growth. However, the size of a DNN model that can be explored today is constrained by the limited GPU device memory.

There have been many efforts to tackle the problem of limited GPU device memory. Some proposals try to fit larger models into a single GPU, e.g. by using the much larger CPU memory as a swap area for the GPU [16] or by discarding intermediate results to save memory at the cost of re-computation [17, 18, 19]. Another promising solution is to partition a DNN model across multiple GPU devices. Doing so reduces per-GPU memory footprint and comes with the additional benefit of parallel speedup. This is commonly referred to as "model parallelism" in the literature.

A DNN model consists of a large number of layers, each parameterized by its own weights. There are two approaches to realize model parallelism. One approach is to assign the computation of different layers to different devices. The second approach is to partition the tensors to parallelize each layer across devices. For very large DNN models, tensor partitioning is the better approach; not only it results in balanced per-GPU memory usage but also it necessary for speeding up popular models such as CNNs.

Tensor partitioning has been explored by existing work as a means for achieving parallel speedup [48, 20, 52] or saving memory access energy [55, 56]. Recent proposals [23, 24, 25] support partitioning a tensor along multiple dimensions and can automatically search for the best partition dimensions. The major limitation is that these proposals partition at the coarse granularity of individual DNN layers, such as fully-connected and 2D convolution layers. As such, they either develop specialized implementation for specific models [23, 20] or allow only a composition of common DNN layers [24, 25, 48, 52].

However, the vast majority of DNN development and deployment today occur on general-purpose deep learning platforms such as TensorFlow [8], MXNet [7], PyTorch [11]. These platforms represent computation as a dataflow graph of fine-grained tensor operators, such as matrix multiplication, various types of convolution and element-wise operations etc. Can we support tensor partitioning on one of these general-purpose platforms? To do so, we have built the Tofu system to automatically partition the input/output tensors of each operator in the MXNet dataflow system. This approach, which we call *operator* *partitioning*, is more fine-grained than layer partitioning. While we have built Tofu's prototype to work with MXNet, Tofu's solution is general and could potentially be applied to other dataflow systems such as TensorFlow.

In order to partition a dataflow graph of operators, Tofu must address two challenges. 1) How to partition the input/output tensors and parallelize the execution an individual operator? What are the viable partition dimensions? 2) how to optimize the partitioning of different operators for the overall graph? Both challenges are made difficult by the fine-grained approach of partitioning operators instead of layers. For the first challenge, existing work [23, 24, 25] manually discover how to partition a few common layers. However, a dataflow framework supports a large and growing collection of operators (e.g. 139 in MXNet), intensifying the manual efforts. Manual discovery is also error-prone, and can miss certain partition strategies. For example, [24] misses a crucial partition strategy that can significantly reduce per-worker memory footprint (§3.7.3). For the second challenge, existing proposals use greedy or dynamic-programming based algorithms [23, 24] or stochastic searches [25]. As the graph of operators is more complex and an order of magnitude larger than the graph of layers (e.g. the graph for training a 152-layer ResNet has >1500 operators in MXNet), these algorithms become inapplicable or run too slowly (§3.5, Table 3.1).

Tofu introduces novel solutions to address the above mentioned challenges. To enable the automatic discovery of an operator's partition dimensions, Tofu requires developers to specify what the operator computes using a lightweight description language called TDL. Inspired by Halide [57], TDL describes tensor computation by specifying the output tensor value at each index with simple expressions on the input tensors. The Halide-style description is useful because it makes explicit which input tensor regions are needed in order to compute a specific output tensor region. Thus, Tofu can statically analyze an operator's TDL description using symbolic execution to determine what input regions must be transferred among GPUs when tensors are divided along a specific partition dimension.

To partition each tensor in the overall dataflow graph, we propose several techniques to shrink the search space. These include a recursive search algorithm which partitions the graph among only two workers at each recursive step, and graph coarsening by grouping related operators.

We have implemented a prototype of Tofu in MXNet and evaluated its performance on a single machine with eight GPUs. Our experiments use large DNN models including Wide ResNet [53] and Multi-layer Recurrent Neural Networks [58], most of which do not fit in a single GPU's memory. Compared with other approaches to train large models, Tofu's training throughput is 25% - 400% higher.

To the best of our knowledge, Tofu is the first system to automatically partition a dataflow graph of fine-grained tensor operators. Though promising, Tofu has several limitations. Some operators (e.g. Cholesky) cannot be expressed in TDL and thus cannot be automatically partitioned. The automatically discovered partition strategies do not exploit the underlying communication topology. Tofu is also designed for very large DNN models. For moderately sized models that do fit in the memory of a single GPU, Tofu's approach of operator partitioning are likely no better than the much simpler approach of data parallelism. Removing these limitations requires further research.

3.2 Problem settings

The problem. Training very large DNN models is limited by the size of GPU device memory today. Compared with CPU memory, GPU memory has much higher bandwidth but also smaller capacity, ranging from 12GB (NVIDIA K80) to 16GB (NVIDIA Tesla V100). Google's TPU hardware has similar limitations, with 8GB attached to each TPU core [59].

Partitioning each tensor in the DNN computation across multiple devices can lower per-GPU memory footprint, thereby allowing very large models to be trained. When par-

```python
def conv1d(data, filters):
 for b in range(output.shape[0]): #b is batch dimension
   for co in range(output.shape[1]): #co is output channel
    for x in range(output.shape[2]): #x is output pixel
     for ci in range(filters.shape[0]): #di is input channel
      for dx in range(filters.shape[2]): #dx is filter window
       output[b, co, x] += data[b, ci, x+dx]
                         * filters[ci, co, dx]
```

Figure 3.1: The naive implementation of `conv1d` in Python.

titioning across k devices, each device roughly consumes $\frac{1}{k}$ times the total memory required to run the computation on one device. Furthermore, partitioning also has the important benefit of performance speedup via parallel execution. As most DNN development today is done on dataflow platforms such as TensorFlow and MXNet, our goal is to automatically partition the tensors and parallelize the operators in a dataflow graph to enable the training of very large DNN models. The partitioning should be completely transparent to the user: the same program written for a single device can also be run across devices without changes.

System setting. When tensors are partitioned, workers must communicate with each other to fetch the data needed for computation. The amount of bytes transferred divided by the computation time forms a lower bound of the communication bandwidth required to achieve competitive performance. For training very large DNNs on fast GPUs, the aggregate bandwidth required far exceeds the network bandwidth in deployed GPU clusters (e.g. Amazon's EC2 GPU instances have only 25Gbps aggregate bandwidth). Thus, for our implementation and evaluation, we target a single machine with multiple GPU devices.

3.3 Challenges and our approach

In order to partition a dataflow graph of operators, we must tackle the two challenges mentioned in §3.1. We discuss these two challenges in details and explain at a high level how Tofu solves them.

3.3.1 How to partition a single operator?

To make the problem of automatic partitioning tractable, we consider only a restricted parallelization pattern, which we call "**partition-n-reduce**". Suppose operator c computes output tensor O. Under partition-n-reduce, c can be parallelized across two workers by executing the *same* operator on each worker using smaller inputs. The final output tensor O can be obtained from the output tensors of both workers (O_1, and O_2) in one of the two ways. 1) O is the concatenation of O_1 and O_2 along some dimension. 2) O is the element-wise reduction of O_1 and O_2. Partition-n-reduce is crucial for automatic parallelization because it allows an operator's existing single-GPU implementation to be re-used for parallel execution. Such implementation often belongs to a highly optimized closed-source library (e.g. cuBLAS, cuDNN).

Partition-n-reduce is not universally applicable, e.g. Cholesky [60] cannot be parallelized this way. Nor is partition-n-reduce optimal. One can achieve more efficient communication with specialized parallel algorithms (e.g. Cannon's algorithm [61] for matrix multiplication) than with partition-n-reduce. Nevertheless, the vast majority of operators can be parallelized using partition-n-reduce (§3.4.1) and have good performance.

Tensors used in DNNs have many dimensions so there are potentially many different ways to parallelize an operator. Figure 3.1 shows an example operator, `conv1d`, which computes 1-D convolution over `data` using `filters`. The 3-D `data` tensor contains a batch (`b`) of 1-D pixels with `ci` input channels. The 3-D `filters` tensor contains a convolution window for each pair of `ci` input and `co` output channel. The 3-D `output` tensor contains

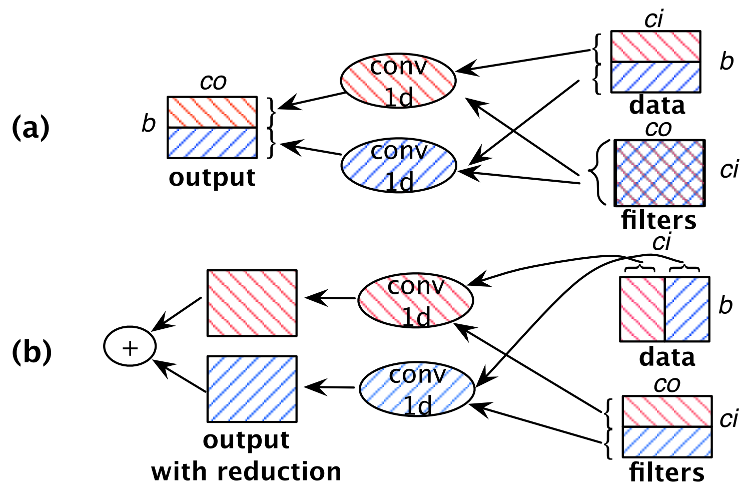


Figure 3.2: Two of several ways to parallelize `conv1d` according to partition-n-reduce. Each 3D tensor is represented as a 2D matrix of vectors. Different stripe patterns show the input tensor regions required by different workers.

the convolved pixels for the batch of data on all output channels.

There are many ways to parallelize `conv1d` using partition-n-reduce; Figure 3.2 shows two of them. In Figure 3.2(a), the final output is a concatenation (along the `b` dimension) of output tensors computed by each worker. Each worker reads the entire `filters` tensor and half of the `data` tensor. In Figure 3.2(b), the final output is a reduction (sum) of each worker's output. Figure 3.1 shows what input tensor region each work reads from. If tensors are partitioned, workers must perform remote data fetch.

Prior work [23, 24, 25] manually discovers the partition strategies for a few common DNN layers. Some [24, 25] have ignored the strategy that uses output reduction (i.e. Figure 3.2(b)), which we show to have performance benefits later (§3.7.3). Manual discovery is tedious for a dataflow system with a large number of operators (341 and 139 in TensorFlow and MXNet respectively). Can one support automatic discovery instead?

Our approach. Tofu analyzes the access pattern of an operator to determine all viable partition strategies. As such, we require the developer of operators to provide a succinct description of what each operator computes in a light-weight language called TDL (short for Tensor Description Language). An operator's TDL description is separate from its implementation. The description specifies at a high-level how the output tensor is derived from its inputs, without any concern for algorithmic or architectural optimization, which are handled by the operator's implementation. We can statically analyze an operator's TDL description to determine how to partition it along different dimensions. §3.4 describes this part of Tofu's design in details.

3.3.2 How to optimize partitioning for a graph?

As each operator has several partition strategies, there are combinatorially many choices to partition each tensor in the dataflow graph, each of which has different execution time and per-GPU memory consumption.

It is a NP-hard problem to partition a general dataflow graph for optimal performance [62, 63, 64, 65]. Existing proposals use greedy or dynamic-programming algorithm to optimize a mostly linear graph of layers [23, 24], or perform stochastic searches [25, 41, 66] for general graphs. The former approach is faster, but still impractical when applied on fine-grained dataflow graphs. In particular, its running time is proportional to the number of ways an operator can be partitioned. When there are 2^m GPUs, each input/output tensor of an operator can be partitioned along a combination of any 1, 2, ..., or m dimensions, thereby dramatically increasing the number of partition strategies and exploding the search time.

Our approach. We use an existing dynamic programming (DP) algorithm [24] in our search and propose several key techniques to make it practical. First, we leverage the unique characteristics of DNN computation to "coarsen" the dataflow graph and shrink the search space. These include grouping the forward and backward operations, and coalescing element-wise or unrolled operators. Second, to avoid blowing up the search space in the face of many GPUs, we apply the basic search algorithm recursively. In each recursive step, the DP algorithm only needs to partition each tensor in the coarsened graph among

```
@tofu.op
def conv1d(data, filters):
  return lambda b, co, x:
    Sum(lambda ci, dx: data[b, ci, x+dx]*filters[ci, co, dx])

@tofu.op
def batch_cholesky(batch_mat):
  Cholesky = tofu.Opaque()
  return lambda b, i, j: Cholesky(batch_mat[b, :, :])[i,j]
```

Figure 3.3: Example TDL descriptions.

two "groups" (of GPUs). §3.5 describes this part of Tofu's design in details.

3.4 Partitioning a single operator

This section describes TDL (§3.4.1) and its analysis (§3.4.2).

3.4.1 Describing an operator

Our Tensor Description Language (TDL) is inspired by Halide[57]. The core idea is "*tensor-as-a-lambda*", i.e. we represent tensors as lambda functions that map from coordinates (aka index variables) to values, expressed as a TDL expression. TDL expressions are side-effect free and include the following:

- Index variables (i.e. arguments of the lambda function).
- Tensor elements (e.g. `filters[ci, co, dx]`).
- Arithmetic operations involving constants, index variables, tensor elements or TDL expressions.
- Reduction over a tensor along one or more dimensions.

Reducers are commutative and associative functions that aggregate elements of a tensor along one or more dimensions. Tofu supports `Sum`, `Max`, `Min` and `Prod` as built-in reducers.

It is possible to let programmers define custom reducers, but we have not encountered the need to do so.

We implemented TDL as a DSL using Python. As an example, Figure 3.3 shows the description of `conv1d`, whose output is a 3D tensor defined by `lambda b, co, x: ...` Each element of the output tensor is the result of reduction (`Sum`) over an internal 2D tensor (`lambda ci, dx: ...`) over both `ci` and `dx` dimensions.

Opaque function. We have deliberately designed TDL to be simple and not Turing-complete. For example, TDL does not support loops or recursion, and thus cannot express sophisticated computation such as Cholesky decomposition. In such cases, we represent the computation as an *opaque function*. Sometimes, such an operator has a batched-version that can be partitioned along the batch dimension. Figure 3.3 shows the TDL description of the operator `batch_cholesky`. The output is a 3-D tensor (`lambda b,i,j:...`) where the element at (b, i, j) is defined to be the (i, j) element of the matrix obtained from performing Cholesky on the b-th slice of the input tensor. Note that, `batch_mat[b, :, :]` represents the bth slice of the `batch_mat` tensor. It is syntactic sugar for the lambda expression `lambda r, c: batch_mat[b, r, c]`.

Describing MXNet operators in TDL. Ideally, operator developers should write TDL descriptions. As Tofu is meant to work with an existing dataflow system (MXNet), we have written the descriptions ourselves as a way to bootstrap. We found that TDL can describe 134 out of 139 MXNet v0.11 operators. Out of these, 77 are simple element-wise operators; 2 use the opaque function primitive, and 11 have output reductions. It takes one of the authors one day to write all these descriptions; most of them have fewer than three LoC. Although we did not build Tofu's prototype for TensorFlow, we did investigate how well TDL can express TensorFlow operators. We found that TDL can describe 257 out of 341 TensorFlow operators. Out of these, 140 are element-wise operators; 22 use the opaque function. For those operators that cannot be described by TDL, they belong to three categories: sparse tensor manipulations, operators with dynamic output shapes and operators requiring data-dependent indexing. MXNet has no operators in the latter two categories.

TDL vs. other Halide-inspired language. Concurrent with our work, TVM [67] and TC [68] are two other Halide-inspired DSLs. Compared to these DSLs, TDL is designed for a different purpose. Specifically, we use TDL to analyze an operator's partition strategies while TVM and TC are designed for code generation to different hardware platforms. The different usage scenarios lead to two design differences. First, TDL does not require users to write intricate execution schedules – code for describing how to perform loop transformation, caching, and mapping to hardwares, etc. Second, TDL supports opaque functions that let users elide certain details of the computation that are not crucial for analyzing how the operator can be partitioned.

3.4.2 Analyzing TDL Descriptions

Tofu analyzes the TDL description of an operator to discover its basic partition strategies. A basic partition strategy parallelizes an operator for 2 workers only. Our search algorithm uses basic strategies recursively to optimize partitioning for more than two workers (§3.5.2).

A partition strategy can be specified by describing the input tensor regions required by each worker to perform its "share" of the computation. This information is used later by our search algorithm to optimize partitioning for the dataflow graph and to generate the partitioned graph in which required data is fetched from different workers.

Obtaining input regions from a TDL description is straightforward if tensor shapes are known. For example, consider the following simple description:

```
def shift_two(A): B = lambda i : A[i+2]; return B
```

Suppose we want to partition along output dimension `i`. Given `i`'s concrete range, say $[0, 9]$, we can compute that the worker needs `A`'s data over range $[2, 6]$ (or $[7, 11]$) in order to compute `B` over range $[0, 4]$ (or $[5, 9]$).

Analyzing with concrete ranges is hugely inefficient as a dataflow graph can contain thousands of operators, many of which are identical except for their tensor shapes (aka index ranges). Therefore, we perform TDL analysis in the abstract domain using *symbolic interval analysis*, a technique previously used for program variable analysis[69], boundary checking[70], parameter validation[71].

Symbolic interval analysis. Suppose the output tensor of an operator has n dimensions and is of the form `lambda x1, ..., xn : ...`. We consider the range of index variable `xi` to be $[0,\ \mathcal{X}_i]$, where $\mathcal{X}_i$ is a symbolic upper bound. We then symbolically execute the lambda function to calculate the symbolic intervals indicating the range of access on the operator's input tensors.

Symbolic execution should keep the range as precise as possible. To do so, we represent symbolic interval ($\mathcal{I}$) as an affine transformation of all symbolic upper bounds,

$$\mathcal{I} \triangleq [\Sigma_i l_i \mathcal{X}_i + c,\ \Sigma_i u_i \mathcal{X}_i + c], l_i, u_i, c \in \mathbb{R} \tag{3.1}$$

In equation 3.1, l_i, u_i and c are some constants. Thus, we can represent $\mathcal{I}$ as a vector of $2 * n + 1$ real values $\langle l_1,\ ...,\ l_n, u_1, ..., u_n,\ c\rangle$. Let `ZV`$[u_i = a]$ denote a vector of all 0s except for the position corresponding to u_i which has value a. By default, lambda variable `xi` for dimension i is initialized to `ZV`$[u_i = 1]$.

Our representation can support affine transformation on the intervals, as shown by the allowed interval arithmetic in Figure 3.4. Product or comparison between two intervals are not supported and will raise an error. We did not encounter any such non-affine operations among MXNet operators.

TDL description: `lambda x1, ..., xi, ..., xn: ...`

$$\begin{aligned}
\mathcal{I} &\triangleq \langle l_1,\ ...,\ l_n,\ u_1,\ ...,\ u_n,\ c\rangle \\
\mathcal{I} \pm k, k \in \mathbb{R} &= \langle l_1,\ ...,\ l_n,\ u_1,\ ...,\ u_n,\ c \pm k\rangle \\
\mathcal{I} \times k, k \in \mathbb{R} &= \langle l_1 k,\ ...,\ l_n k,\ u_1 k,\ ...,\ u_n k,\ c * k\rangle \\
\mathcal{I}/k, k \in \mathbb{R} &= \langle l_1/k,\ ...,\ l_n/k,\ u_1/k,\ ...,\ u_n/k,\ c/k\rangle \\
\mathcal{I} \pm \mathcal{I}' &= \langle l_1 \pm l_1',\ ...,\ u_1 \pm u_1',\ ...,\ c \pm c'\rangle
\end{aligned}$$

Figure 3.4: Tofu's symbolic interval arithmetic.

Discover operator partition strategies. Using the symbolic interval analysis, we infer the input regions required by each of the 2 workers for every partitionable dimension. There are two cases.

Case-1 corresponds to doing partition-n-reduce without the reduction step. In this case, each partition strategy corresponds to some output dimension. Suppose we are to partition `conv1d`'s output tensor along dimension `b`. We use two different initial intervals for lambda variable `b`, $\texttt{ZV}[u_{\texttt{b}} = \frac{1}{2}]$ and $\texttt{ZV}[l_{\texttt{b}} = \frac{1}{2}, u_{\texttt{b}} = 1]$, in two separate analysis runs. Each run calculates the input regions needed to compute half of the output tensor. The result shows that that each worker reads half of the `data` tensor partitioned on the `b` dimension and all of the `filter` tensor, as illustrated in Figure 3.2(a). Similarly, the analysis shows how to partition the other output dimensions, `co` and `x`. Partitioning along dimension `x` is commonly referred to as parallel convolution with "halo exchange" [20, 23, 56].

Case-2 corresponds to doing partition-n-reduce with the reduction step. In this case, we partition along a reduction dimension. In the example of Figure 3.3, the reduction dimensions corresponding to `ci` and `dx` in `Sum(lambda ci, dx: ...)`. The analysis will determine that, when partitioning along `ci`, each partially reduced tensor will require half of the `data` tensor partitioned on the second dimension and half of the `filter` tensor partitioned on the first dimension, as shown in Figure 3.2(b). Similar analysis is also done for dimension `dx`. Out of 47 non-element-wise MXNet operators describable by TDL, 11 have at least one reduction dimension.

3.5 Partitioning the dataflow graph

To partition a dataflow graph, one needs to specify which partition strategy to use for each operator. This section describes how Tofu finds the best partition plan for a dataflow graph.

Different plans result in different running time and per-worker memory consumption, due to factors including communication, GPU kernel efficiency and synchronization. Finding the best plan is NP-hard for an arbitrary dataflow graph [72]. Recent work has proposed an algorithm based on dynamic programming (DP) for partitioning a certain type of graphs. §3.5.1 presents techniques to make a dataflow graph applicable to DP, and §3.5.2 improves search time via recursion.

Optimization goal. Ideally, our optimization goal should consider both the end-to-end execution time of the partitioned dataflow graph and the per-worker memory consumption. Unfortunately, neither metric can be optimized perfectly. Prior work [25] optimizes the approximate end-to-end execution time by minimizing the sum of total GPU kernel execution time and total data transfer time.

In Tofu, we choose to minimize the total communication cost based on two observations. First, the GPU kernels for very large DNN models process large tensors and thus have similar execution time no matter which dimension its input/output tensors are partitioned on. Consequently, a partition plan with lower communication cost tends to result in lower end-to-end execution time. Second, the memory consumed at each GPU worker is used in two areas: (1) for storing a worker's share of tensor data, (2) for buffering data for communication between GPUs. The memory consumed for (1) is the same for every partition plan: for k GPUs, it is always $1/k$ times the memory required to run the dataflow graph on one GPU. The memory consumed for (2) is proportional to the amount of communication. Therefore, a partition plan with lower communication cost results in a

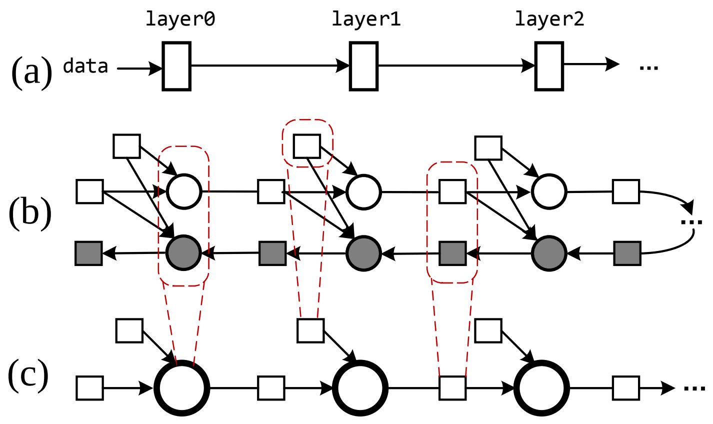


Figure 3.5: (a) Layer graph of a MLP model. (b) Its dataflow graph including forward and backward computation (in grey). (c) Coarsened graph. For cleanness, we only illustrate one operator group, one group for activation tensors and one group for weight tensor (dashed lines).

smaller per-worker memory footprint.

3.5.1 Graph coarsening

The algorithm in [24] is only applicable for linear graphs[1], such as the graph of DNN layers shown in Figure 3.5(a). Dataflow graphs of fine-grained operators are usually non-linear. For example, Figure 3.5(b) is the non-linear dataflow graph of the same DNN represented by Figure 3.5(a). Here, we propose to "coarsen" a dataflow graph into a linear one by grouping or coalescing multiple operators or tensors.

Grouping forward and backward operations. Almost all DNN models are trained using gradient-based optimization method. The training includes a user-written forward propagation phase to compute the loss function and a system-generated backward propagation phase to compute the gradients using the chain rule. Thus, we coarsen as follows:

- Each forward operator (introduced by the user) and its auto-generated backward operators (could be more than one) to form a group.

1. We say a graph G is linear if it is homeomorphic to a chain graph G', meaning there exists a graph isomorphism from some subdivision of G to some subdivision of G' [73]. Note that a "fork-join" style graph is linear by this definition.

- Each forward tensor (e.g. weight or intermediate tensors) and its gradient tensor form a group. If a (weight) tensor is used by multiple operators during forward propagation and thus has multiple gradients computed during backward propagation, the chain rule requires them to be summed up and the summation operator is added to the group as well.

Figure 3.5(c) shows the coarsened dataflow graph for a MLP model. As forward and backward operators for the same layer are grouped together, the resulting graph becomes isomorphic to the forward dataflow graph. For MLPs and CNNs, their coarsened graphs become linear. We perform the DP-based algorithm [24] on the coarsened graph. When the algorithm adds a group in its next DP step, we perform a brute-force combinatorial search among all member operators/tensors within the group to find the minimal cost for adding the group. This allows tensors involved in the forward and backward operators to be partitioned differently, while [24] forces them to share the same partition configurations. As there are only a few operators (typically 2) in each group, the cost of combinatorial search is very low.

Coalescing operators. In DNN training, it makes sense for some operators to share the same partition strategy. These operators can be merged into one in the coarsened dataflow graph. There are two cases:

- *Merging consecutive element-wise operators*, because the input and output tensors of an element-wise operator should always be partitioned identically. We analyze the TDL description to determine if an operator is element-wise. Consecutive element-wise operators are very common in DNN training. For instance, almost all gradient-based optimizers (e.g. SGD, Adam, etc.) are composed of only element-wise operators.

- *Merging unrolled timesteps.* Recurrent neural networks (RNNs) process a variable sequence of token over multiple timesteps. RNN has the key property that different

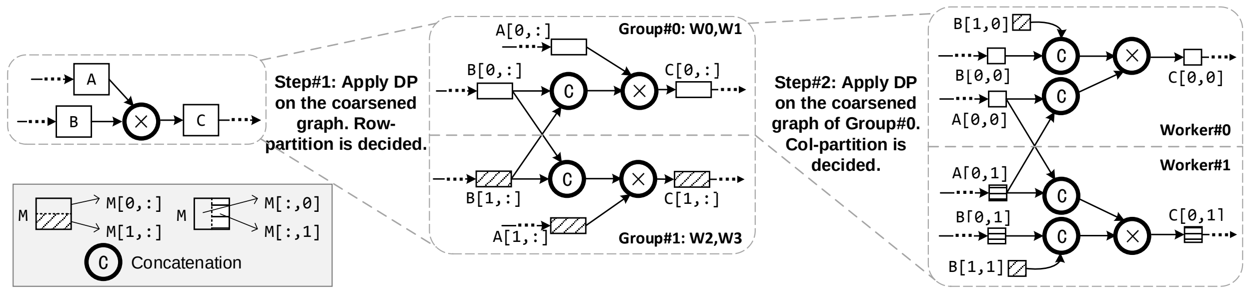


Figure 3.6: Recursively partition a dataflow graph to four workers. Only one matrix multiplication is drawn for cleanness. In step#1, every matrix is partitioned by row, and for group#0, B[1,:] is fetched from the other group. Because of this, B[1,:] becomes an extra input in step#2 when the graph is further partitioned to two workers. Because step#2 decides to partition every matrix by column, every matrix is partitioned into a 2x2 grid, with each worker computes one block.

	Search Time	
	WResNet-152	RNN-10
Original DP [24]	n/a	n/a
DP with coarsening	8 hours	>24 hours
Using recursion	8.3 seconds	66.6 seconds

Table 3.1: Time to search for the best partition for 8 workers. WRestNet-152 and RNN-10 are two large DNN models described in §3.7.

time steps share the same computation logic and weight tensors. Thus, they should be coalesced to share the same partition strategy. As a result, the dataflow graph of a multi-layer RNN becomes a chain of coalesced and grouped operators. To detect operators that belong to different timesteps of the same computation, we utilize how RNN is programmed in DNN frameworks. For example, systems like MXNet and PyTorch call a built-in function to unroll a basic unit of RNN computation into many timesteps, allowing Tofu to detect and merge timesteps.

3.5.2 Recursive partitioning

When there are more than two workers, each operator can be partitioned along multiple dimensions. This drastically increases the number of partition strategies available to each operator and explodes the running time of the DP-based search algorithm. To see this, consider the coarsened graph of Figure 3.5(b). Every operator group has two input tensor groups and one output tensor group. Each tensor group contains one forward tensor and one gradient tensor. At each step, the DP algorithm needs to consider all the possible configurations of an operator group including different ways to partition the six input/output tensors. For each 4D tensor used in 2D-convolution, there are in total 20 different ways to partition it evenly across 8 workers. Hence, the number of possible configurations of 2D-convolution's operator group is $20^6 = 6.4 \times 10^7$. Although not all the dimensions are available for partition in practice (e.g. the convolution kernel dimension is usually very small) , the massive search space still results in 8 hours of search time when partitioning the WResNet-152 model (Table 3.1).

Our insight is that the basic DP search algorithm can be *recursively* applied. For instance, a matrix, after being first partitioned by row, can be partitioned again. If the second partition is by column, the matrix is partitioned into a 2×2 grid; if the second partition is by row, the matrix is partitioned into four parts along the row dimension.

This observation inspires our recursive optimization algorithm to handle $k = 2^m$ GPUs:

1. Given a dataflow graph G, run the DP algorithm with coarsening to partition G for two worker groups, each consisting of 2^{m-1} workers. Note that each tensor is only partitioned along *one* dimension.

2. Consider the partitioned dataflow graph as consisting of two halves: G_0 for worker group#0 and G_1 for worker group#1. Each half also contains the data fetched from the other group as extra input tensors.

3. Repeat step 1 on G_0 and apply the partition result to G_1 until there is only one worker per group.

This recursive algorithm naturally supports partitioning along multiple dimensions. Figure 3.6 illustrates two recursive steps using an example dataflow graph (for brevity, we only show one matrix multiplication operator in the graph). Note the recursion must be done over the entire dataflow graph instead of a single operator, as the partition plan of the previous recursive step will influence the global decision of the current one.

While the recursive algorithm may seems straightforward, it is less obvious why the resulting partition plan has the optimal overall communication cost. In particular, the recursive algorithm chooses a sequence of basic partition plans $\{\mathcal{P}_1, \mathcal{P}_2, ...\mathcal{P}_m\}$ in m recursive steps, and we need to prove that no other sequence of choices leads to a better plan with a smaller communication cost. The main insight of our proof is that the partition plan decided in each recursive step is commutative (i.e, choosing partition plan $\mathcal{P}$ followed by $\mathcal{P}'$ results in the same total communication cost as choosing $\mathcal{P}'$ followed by $\mathcal{P}$.) Based on this insight, we derive the following property and use it to prove optimality.

Theorem ??. *Let the total communication cost incurred by all worker groups at step i be δ_i. Then $\delta_i \leq \delta_{i+1}$.*

Suppose $\{\mathcal{P}_1, \mathcal{P}_2, ...\mathcal{P}_m\}$ is the sequence of partition plans chosen and it is not optimal. Then there exists a different sequence $\{\mathcal{P}'_1, \mathcal{P}'_2, ...\mathcal{P}'_m\}$ with smaller total cost. Hence, there must be two consecutive steps $k-1$ and k, such that $\delta_{k-1} \leq \delta'_{k-1}$ and $\delta'_k < \delta_k$. We can show that, by choosing $\mathcal{P}'_k$ instead of $\mathcal{P}_k$ at step k, the search could have produced a better partition plan. This contradicts the optimality of the DP algorithm. The full proof is included in Appendix A.

If the number of GPUs k is not a power of two, we factorize it to $k = k_1 * k_2 * ... * k_m$, where $k_i \geq k_{i+1}$ for all i. At each step i in the recursive algorithm, we partition the dataflow graph into k_i workers in which each partition strategy still partitions a tensor along only one dimension but across k_i workers.

The benefits of recursion. Recursion dramatically cuts down the search time by partitioning along only one dimension at each step. For example, the number of configurations to be enumerated at each step for a 2D-convolution operator group is only $4^6 = 4096$. Therefore, the total number of partition strategies searched for the 2D-convolution opera-

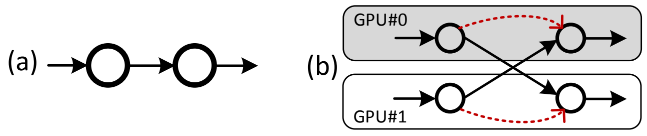


Figure 3.7: (a) Original dataflow graph; (b) Partitioned graph with extra control dependencies (dashed lines).

tor with 8 workers (3 recursive steps) is $3*4096$, which is far fewer than 20^6 when recursion is not used. Table 3.1 shows the search time for two common large DNN models when applying the original DP algorithm on coarsened graph without and with recursion.

As another important benefit, recursion finds partition plans that work well with common hierarchical physical interconnects which have less aggregate bandwidth near the top of the hierarchy. For example, many commercial servers group GPUs by faster PCI-e buses first and then connect the groups with slower QPI buses or Infinibands. As theorem **??** indicates, Tofu assigns worker groups with less communication near the top of the hierarchical interconnects in earlier steps of the recursion.

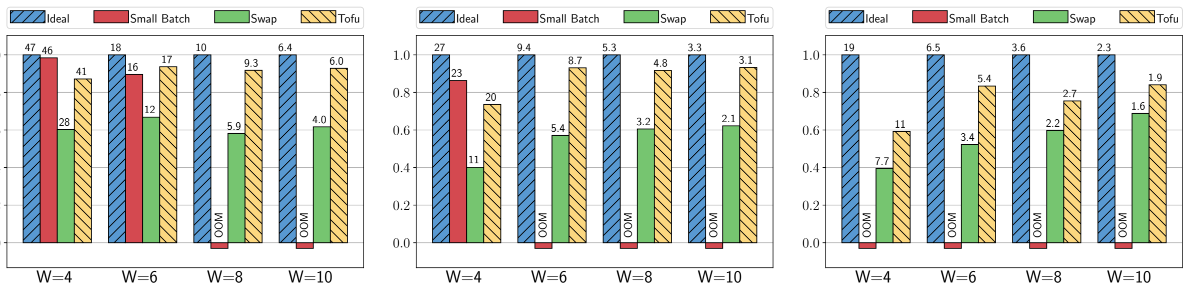
3.6 Optimizations in generating the partitioned graph

Once the search algorithm determines how to partition for every tensor and operator, Tofu generates a corresponding partitioned dataflow graph. The graph generation process is mostly straightforward save for two optimizations, which are crucial to keep the per-worker memory consumption low.

Leveraging the existing memory planner. Systems like MXNet and TensorFlow have their own memory planners to statically allocate and re-use memory buffers among operators according to their dependencies. Ideally, the per-worker memory consumption for k workers should be $1/k$ of the original memory consumption. In our initial implementation, per-worker memory consumption far exceeded the expected amount. We found that this is because the partitioning of a dataflow graph changes the dependencies between original operators. Figure 3.7 illustrates an example. In the original graph, the second operator can reuse the memory buffer of the first one (such as the workspace of a convolution operator) due to the dependency between the two. Naive graph generation may result in the graph with solid edges in Figure 3.7(b), in which the two operators executed by each worker no longer have a direct dependency between them and thus allows no immediate memory-reuse. To fix this, Tofu maintains the original operator dependencies on each worker by generating the extra control dependencies (dashed lines), so that the memory planner can immediately re-use buffers across dependent operators.

Fusing operators for remote data fetch. For each operator in the original graph, Tofu generates a copy for each GPU worker in the partitioned graph. Often, these operators need to fetch data from a different worker. MXNet already supports copy, split, concatenate operators, which can be used to support data movements. A naively generated graph would use split to extract the required input regions from the other workers, copy data to the local worker, and concatenate them together to assemble the input region needed by the operator's GPU kernel. Extra reduce operators can also be generated if the output tensors of different workers need to be aggregated according to the partition strategy used. Execution of such graphs results in many intermediate memory blocks, increasing the per-worker memory consumption. To mitigate this, we wrote a custom GPU kernel called MultiFetch to retrieve remote data and assemble the input region in-place using CUDA Unified Virtual Addressing (UVA). CUDA UVA allows a kernel running on one GPU to directly access the memory on another, which avoids explicit data copying before kernel execution. Our MultiFetch kernel takes multiple pointers to the memory blocks of the input regions from the other GPUs and assembles them in one kernel launch.

Beyond the two optimizations described above, we also spread out the reduction workload to all GPUs (all-reduce) when performing output reduction. This is important for avoiding any single aggregation bottleneck. We also find that the MXNet scheduler can execute the remote fetch operator much earlier than required, resulting in memory being occupied for longer than necessary. We adopt the same technique proposed by TensorFlow



(a) Wide ResNet-50 (b) Wide ResNet-101 (c) Wide ResNet-152

Figure 3.8: Normalized WResNet throughput relative to the ideal performance. The number on each bar shows the absolute throughput in samples/sec.

to delay the execution of the remote fetch operator.

3.7 Evaluation

This section evaluates Tofu and compares with various alternative approaches. The highlights of our results are the following:

- Tofu can train very large WResNet and RNN models across 8 GPUs with high throughput that is within 60%-98% of a hypothetical ideal baseline.
- Except for a few exceptions, Tofu outperforms existing alternative approaches including shrinking the mini-batch size used for training, swapping to CPU memory, and placing different operators on different GPUs.
- Tofu's recursive partition algorithm leads to better training throughput than existing partition algorithms [24, 72] and simple heuristics.
- The overall partition plan found by Tofu is highly non-trivial, even though the underlying DNN model has a regular structure.

RNN			
	L=6	L=8	L=10
H=4K	8.4	11.4	14.4
H=6K	18.6	28.5	32.1
H=8K	33.0	45.3	57.0

Wide ResNet			
	L=50	L=101	L=152
W=4	4.2	7.8	10.5
W=6	9.6	17.1	23.4
W=8	17.1	30.6	41.7
W=10	26.7	47.7	65.1

Table 3.2: Total weight tensor sizes (GB) of our benchmarks.

3.7.1 Experimental setup

Prototype Implementation. We implement Tofu based on MXNet 0.11. The TDL components (operator descriptions and the region analyzer) are written in Python (2K LoC). The recursive search algorithm is implemented as a graph transformation pass in NNVM (4K LoC in C++). As we need information from gradient calculation and shape inference, we also made slight modifications to the corresponding NNVM passes.

Testbed: The experiments run on an EC2 p2.8xlarge instance. The instance has 8 K80 GPUs with 12GB memory each. GPUs are connected by PCI-e bus with 21GB/s peer-to-peer bandwidth. It has 32 virtual CPU cores and 488GB CPU memory. The CPU-GPU bandwidth is 10GB/s.

DNN Benchmarks: We evaluate the WResNet [53] convolutional neural network and recurrent neural network (RNN). We choose these two benchmarks because they correspond to very large models. We do not evaluate those well-known DNNs that fit into a single GPU's memory, such as AlexNet, VGGNet and Inception.

WResNet [53] is a widened version of the original residual network model [2]. It has a widening scalar to increase the number of channels on each convolution layer. The model size grows quadratically as each weight tensor is widened on both the input and output channel. WResNet has been shown to achieve a better accuracy when the model is widened by 10×. Due to the memory limitation, such improvement is only demonstrated on CIFAR-10 dataset of small images (32x32) using a 50-layer model. We experiment with WResNet on ImageNet dataset with images of size (224x224). We also test different model variations: widening scalar from 4 to 10 on networks with 50, 101 and 152 layers. We use notations like WResNet-101-8 to denote the 101-layer ResNet model widened by 8 times.

For RNN, there are two ways to increase model capacity. The number of neurons in each hidden layers can be increased, and multiple RNN layers can be stacked to form a deeper model. Researchers have explored very large RNNs by increasing the number of RNN layers to 8 [41, 66], or by using a large hidden layer size such as 8192 [58]. We use the model described in [58], and test it with different configurations varying from 6 to 10 layers with 4K, 6K, and 8K hidden sizes. All RNN model variants use LSTM cell [74] and are unrolled for 20 steps as in [58]. We use the RNN-8-8K to denote the 8-layer RNN model with 8K hidden size.

All the benchmarks are tested by running a full training iteration including forward/backward propagation and weight update. State-of-the-art weight optimizers such as Adam [75] and Adagrad [76] must maintain an extra buffer for storing the gradient history. Therefore, a model of weight size W needs to consume at least $3W$ size of memory for storing the weight, gradient and the history tensors. Table 3.2 shows the total weight memory consumption for all the benchmarks.

Baseline and Alternatives for Comparison. We consider an ideal baseline and several alternative approaches for comparison.

Ideal is a hypothetical baseline that assumes each GPU has infinite memory. We simulate this by modifying the memory allocator of MXNet to always return the same memory block. We measure the single-GPU throughput number and multiply it by 8 as the performance of running on 8 GPUs.

SmallBatch is a baseline that tries to fit the model in a single GPU by reducing the mini-batch size. Like the *ideal* baseline, we scale the single-GPU throughput number by 8 for 8 GPUs. Thus, neither SmallBatch nor Ideal baseline consider the communication cost and represent performance upper-bounds.

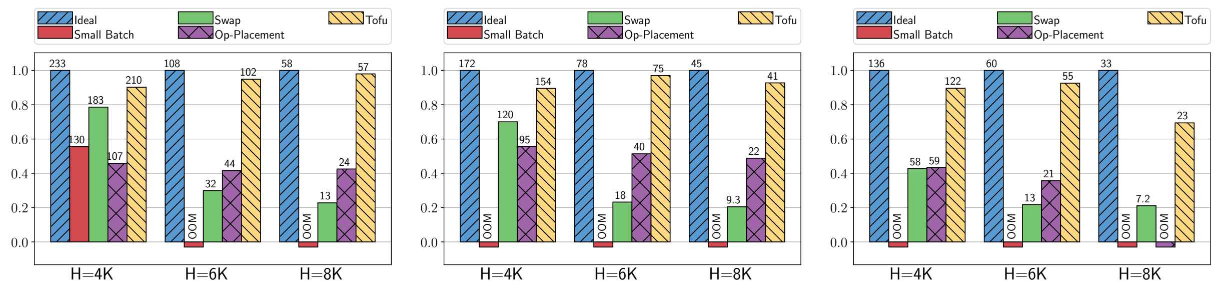
Swapping [16, 77, 78] is a baseline that swaps in/out GPU memory blocks to CPU. There are many ways to design the swapping policy. Our baseline combines many of these techniques in order for a fair comparison. First, our baseline follows the design of [78], which includes a least recently used algorithm to decide the tensor to be swapped out and a prefetching unit based on the execution. This supports swapping in/out any memory block instead of only activation tensors as in [16]. Second, read-only tensors are copied to CPU only once and simply dropped the next time they are to be swapped out. Third, we combine dataflow analysis similar to [16] to disable swapping out memory blocks that will soon be used.

Operator Placement [79, 41, 80, 54] assigns operators to different devices to spread out memory usage. For RNN, this baseline assigns the computation of different layers to different GPUs to leverage the pipelining effect, as it is originally proposed in [79]. If there are more layers than the number of GPUs, we balance the assignment in a round-robin manner. Operator placement does not perform well for CNNs due the mostly serial layer-by-layer execution. Therefore, we skip this baseline for all WResNet benchmarks.

In our experiments, the ideal baseline uses a batch size that can saturate the GPU for the best performance. SmallBatch, Swapping and Tofu all use the largest batch size that make the execution fit in the GPU memory.

3.7.2 Training Large and Deep Models

We show the performance of Tofu and compare it to the ideal baseline and alternatives. Since different systems use different batch sizes to achieve the best performance, we use throughput (samples/sec) instead of training time per iteration as the metric for comparison In Figures 3.8 and 3.9, each bar shows the throughput relative to the ideal baseline performance. The absolute throughput numbers are shown on top of each bar. *OOM* indicates out-of-memory error.



(a) 6 layers RNN (b) 8 layers RNN (c) 10 layers RNN

Figure 3.9: Normalized RNN throughput relative to the ideal performance. The number on each bar shows the absolute throughput in samples/sec.

WResNet Performance. Figure 3.8 shows the WResNet throughput achieved by different systems. The ideal baseline uses a global batch size of 128. Only 3 models, WResNet-50-4,6 and WResNet-101-4 can be fit in a single GPU memory by shrinking the batch size (aka SmallBatch).

Tofu can achieve 60%-95% of the ideal performance for all the models. The largest model, WResNet-152, has the biggest performance gap. This is because we configured the ideal baseline to use a much larger mini-batch size for peak throughput without any consideration for memory consumption. For example, the ideal baseline uses base size 128 for WResNet-152-4 while Tofu can fit at most 32. The batch sizes used by Tofu ranges from 8 (for WResNet-152-10) to 128 (for WResNet-50-4). Tofu performs better than alternatives in all scenarios except for WResNet-50-4 and WResNet-101-4, in which SmallBatch achieves 12% and 15% better throughput than Tofu. This is because convolution kernels have good GPU utilization even for small batch sizes. However, SmallBatch runs out of memory for most of the models in Figure 3.8.

As shown in Figure 3.8, swapping is 20%-63% slower than Tofu across all the models. This is due to swapping's much larger communication amount. Although we implemented prefetching to "hide" communication latency in swapping, the CPU-GPU communication is the bottleneck as all 8 GPUs share the same bandwidth to communicate with the CPU.

RNN Performance. Figure 3.9 shows the throughput for RNNs. The ideal baseline uses a (global) batch size of 512. Tofu performs better than the other baselines in all RNN configurations, achieving 70% - 98% of ideal throughput. Unlike the WResNet experiments, SmallBatch does not achieve better throughput than Tofu in any RNN configuration. This is because the main RNN computation is matrix multiplication, which has much less arithmetic density than convolution. Thus, performing matrix multiplication using small batch sizes results in decreased GPU utilization. The same reasoning explains why Tofu's relative performance with the largest model (RNN-10-8K) is worse than with other RNN models; Tofu uses a batch size of 128 in order to fit RNN-10-8K in memory while it uses larger batch sizes (256 or 512) with other RNN models. As is also the case with WResNet, SmallBatch results in OOM for larger RNN configurations.

Operator placement achieves 38%-61% of Tofu's throughput and cannot train RNN-10-8K (OOM). Two reasons contribute to the lower performance. First, layer-wise placement results in imbalanced load because the number of layers is not a multiple of the number of GPUs. Second, layer-wise placement relies on pipelined parallelism: GPU-1 executes the first operator in the first layer and forwards its result to GPU-2. GPU-2 can execute the first operator in the second layer while GPU-1 concurrently executes the second node in the first layer. Pipelined parallelism cannot fully saturate GPUs at all times: e.g. GPU-2 is idle while GPU-1 executes its first operator. By contrast, Tofu parallelizes the execution of each operator and keeps all GPUs busy at all times.

Swapping achieves 23% - 30% throughput of Tofu and 48% - 53% throughput of operator placement when the weight size is large. The main reason is that many tensors may be used simultaneously in RNN training. To fully saturate a GPU, most deep learning frameworks, including MXNet and Tensorflow, schedule operators immediately when they are ready. RNN's mesh-like dataflow graph results in more tensors to be used at the same time. When the weight size is large, the amount of swapping increases significantly. Coupled with the CPU-GPU communication bottleneck, swapping is unable to achieve good

	RNN-6	**RNN-8**	**RNN-10**
Tofu	210	154	122
MX-OpPlacement	107	95	59
TF-OpPlacement	50	36	30

Table 3.3: Comparison of throughput (samples/second) for RNN models. The hidden size is 4096.

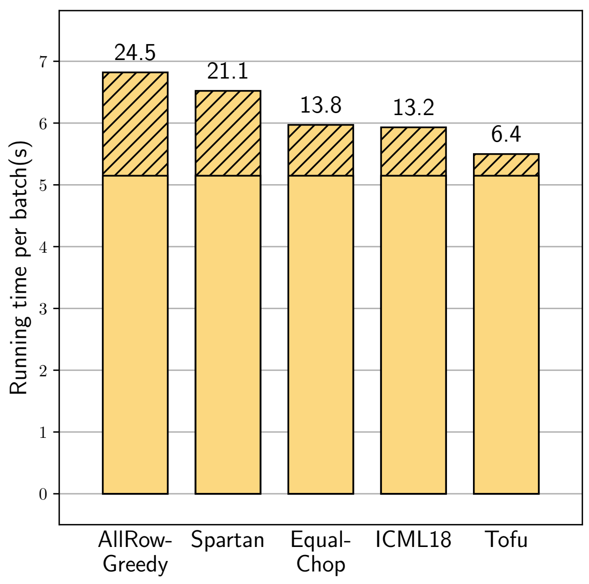
throughputs for RNNs.

Comparing with TensorFlow. We compare with Tensorflow v1.8 (using Op-Placement) for training RNNs. Table 3.3 shows the throughputs for running on RNN-6-4K, RNN-8-4K, and RNN-10-4K. For additional comparison points, we also include MXNet (using Op-Placement). Note that the throughputs of Tofu and MXNet are same as those in Figure 3.9. Tensorflow's throughput is roughly half of MXNet and about 23% of Tofu. As Tensorflow and MXNet use the same operator kernel implementations, we originally expected the two systems to have similar throughput. However, further investigation shows that TensorFlow does not support in-place gradient aggregation which may be crucial for the performance of large RNNs.

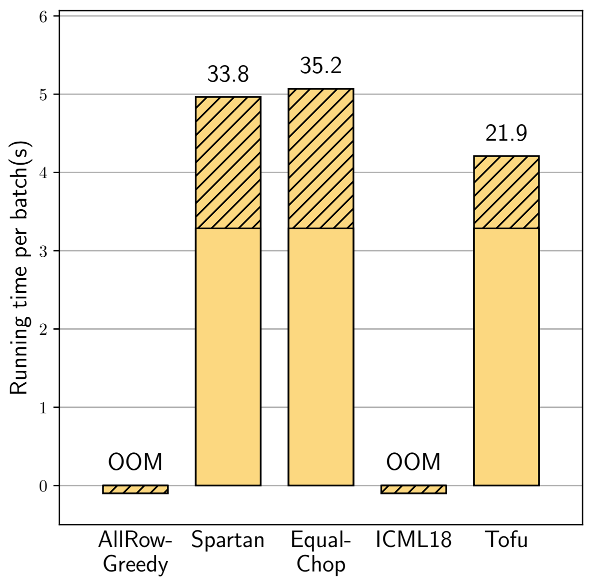
3.7.3 Comparing different partition algorithms

We have compared Tofu's search time with the original DP algorithm [24] in §3.5.2 (Table 3.1). We now compare the quality of partition plan found by Tofu vs. [24] and various other heuristics.

The simplest heuristic (`AllRow-Greedy`) partitions all tensors along the first dimension and partitions each operator using the best strategy given that its input/output tensors are partitioned on the first dimension. Note that, for the case of WResNet, this gives similar result as the *one-weird-trick* strategy proposed in [22], because all the convolution layers are partitioned by the batch dimension and the only fully-connected layer in WResNet occupies $<1\%$ of the total time. Our next heuristic is to greedily partition the largest



(a) RNN-4-8K



(b) WResNet-152-10

Figure 3.10: Comparison of different partition algorithms using RNN-4-8K and WResNet-152-10 on 8 GPUs. Striped parts show the overhead (percentage) due to communication.

tensor first (along any dimension), followed by its incident operators, followed by the second largest tensor and so on. This is equivalent to what is proposed by Spartan [72]. We also compare with Tofu's DP algorithm applied to chop each tensor equally along only one dimension (`EqualChop`). Finally, we compare with the algorithm in [24](`ICML18`) which does not consider the partition strategy of aggregating output tensors (aka output-reduction).

Figure 3.10 shows the execution time of training one batch on 8 GPUs for RNN-4-8K (batch size is 512) and WResNet-152-10 (batch size is 8). To see the impact of communication on the execution time, we modify the backend to skip memory copy among GPUs and measure the resulting pure computation time, which is shown as the lower light-colored portion of the bars in Figure 3.10.

`AllRow-Greedy` performs worse among all the algorithms and run out of memory for WResNet-152-10 because it needs to fetch too much data from the other GPUs. `Spartan` and `EualChop` reduce the communication overhead by 3%-10% but are still worse than Tofu.

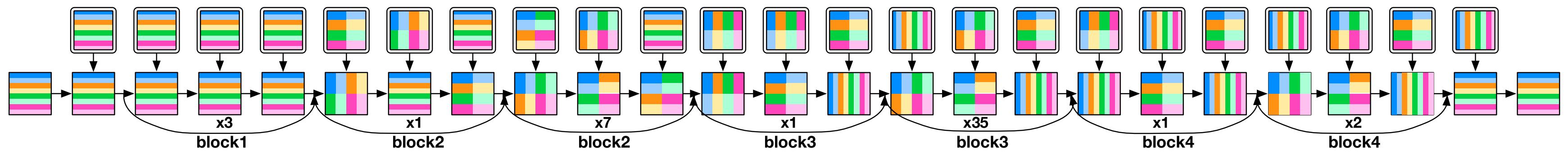


Figure 3.11: The partition found by Tofu for WResNet-152-10 on 8 GPUs. We draw the weight tensors (top row) and the activation/data tensors (bottom row) used by convolution operators. Partitioning is marked by the tiles and each color shows the tiles owned by the same GPU. The vertical and horizontal dimensions of an activation tensor indicate the batch and channel dimensions. 'xN' symbol means the corresponding block is repeated N times.

This result shows the benefit of partitioning a tensor along multiple dimensions. `ICML18` is 7% slower than Tofu for RNN-4-8K and results in OOM for WResNet-152-10 due to the lack of output-reduction. After adding output-reduction, `ICML18` can find the same strategy as Tofu, albeit with a much longer search time (see Table 3.1).

3.7.4 Partition Results

Figure 3.11 shows the partition found by Tofu for WResNet-152-10. ResNet-152 contains 4 groups of residual blocks: each block includes 3 convolutions and is repeated 3, 8, 36, and 3 times for each group respectively. The lower residual blocks (those close to the input layer) have larger feature map but smaller weight tensors while the higher ones are the opposite.

We make the following observations:

- Tofu partitions both the batch and channel dimensions and the resulting partition plan is a complicated combination of different partition strategies.
- Tofu chooses different partition plans for different convolution layers within one residual block. Repeated residual blocks are partitioned in the same way except for the first block in the group which has a different configuration to shrink the initial input feature map size by half.
- As the activation tensors in lower layers are larger and the weight tensor smaller, Tofu chooses to fetch weight tensors from remote GPUs to save communication. As the weight tensors are larger in the higher layers, Tofu switches to partition strategies that fetch the relatively smaller activation tensors.

3.8 Additional Related Work

Parallel DNN training. Many parallel strategies have been developed to speedup DNN training. Some strategies such as the popular data parallelism [49, 81, 82, 83] cannot be used for training very large models because the parameters are replicated to each device. Model parallelism spreads out the model parameters to multiple GPUs, thus is suitable for training very large models. Early work[20, 22, 48] parallelizes specific classes of DNN models, and is limited in flexibility and generality. Minerva[21] and Strads[84] require users to implement extra interfaces to partition model parameters while Tofu requires no change to the user program. Another approach is to assign different layers/operators to different devices via heuristics [80] or stochastic search [79, 41]. However, operator placement only works well only when there are sufficiently many concurrent operators, and thus is not suitable for DNN models with a deep stack of layers.

Out-of-core DNN training. This includes recomputation on demand [17, 18, 19] , swapping and prefetching from host memory [16, 77, 78]. Recomputation is not viable for large weight tensors. Swapping with host memory reduces the opportunity of co-locating computation and data, and scales poorly when there are multiple GPUs. None of them can efficiently utilize the aggregated memory capacity of multiple cards as Tofu does. Moreover, Tofu can also be combined with these techniques.

Model compression. This includes network pruning [85, 86] (which removes small weight values), quantization[87] and reduced precision[88]. The compressed model can then be deployed on mobile or edge devices or to speed up the inference. However, these approaches affect model accuracy while Tofu allows exploring very large models without changing the model behavior.

Parallel tensor computing. There is a long history in developing efficient parallel systems for tensor computing. The very first effort starts from developing low-level, optimized, parallel matrix/tensor libraries [89, 90, 91, 92, 93]. These libraries implement efficient parallel matrix algorithms [61, 94] and tensor operations [95]. However, they have very limited programmability support and adding new operators requires tremendous manual efforts.

Many frameworks or tools have been built to ease the programming of parallel tensor computation. In the low-level, ZPL [96], Chapel [97] and Unified Parallel C [98] are parallel language supports. In the higher-level, systems such as [99, 100, 101, 102, 103, 72] let users write programs in high-level primitives like map and reduce. MadLinq [60] and Presto [104] let user describe operators using parallel loop primitives. Users need to express parallelism using the proper combination of these primitives. For example, implementing a parallel matrix multiplication needs to call the `shuffle` primitive in Spartan [72] or the `Collect` primitive in [103]. However, these primitives are limited (e.g. it is hard to express halo-exchange in convolution). Distributed Halide [105] lets user describe the algorithm in their DSL and specifies how it is paralleled. As there are usually multiple ways of partitioning data and computation, the efficiency varies with different implementations. Spartan [72] and Kasen [102] propose algorithm to automatically optimize array/matrix partitioning to reduce communication. [103] further improves this by also considering different parallel patterns via transformations of nested high-level primitives.

More recent proposals aim to fully automate the whole stack – user programs are written in array language and the system can distribute the data and computation automatically. There are several approaches. Cylops Tensor Framework [106] and Tensor Contraction Engine [107] are specialized systems for automatically parallelizing tensor contraction. Spartan tries to map Numpy operators to high-level map and reduce primitives and then partitions them accordingly. Others tried to leverage the parallelism among array operators. For example, Pydron [108] translates Python program into an internal dataflow graph to parallelize independent loops. [79, 41] tries to dispatch array operators to different devices automatically based on the dataflow graph. However, they are not suitable for DNN computation that is mostly sequential. Compared with previous systems, Tofu automatically discovers the partition-n-reduce parallel patterns of operators using TDL description and optimizes partitioning for the entire dataflow graph.

Data layout optimization. There have been extensive work on optimizing communication (aka remote memory access) on the multiprocessor architecture (e.g. [109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119]) or the new hardware [56, 23, 55]. Since searching the optimal solution is NP-Complete [62, 63, 64, 65], heuristics are used in practice [65, 111]. By contrast, Tofu analyzes the relatively simpler operator description language instead of the source code, and exploits the DNN computation structure for its optimization.

3.9 Conclusion

We present the Tofu system, which enables the training of very large DNN models by partitioning a dataflow graph of tensors across multiple GPU devices. To automate this process, Tofu infers each operator's valid partition strategies by analyzing its semantics written in a simple description language (TDL). Tofu uses a recursive search algorithm based on dynamic programming and DNN-specific heuristics to find the best partition plan that minimizes communication for the entire dataflow graph.

Chapter 4

Deep Graph Library: An Efficient and Scalable Framework for Deep Learning on Graph

4.1 Introduction

Learning from structured data like graphs is widely regarded as an important problem [120], because the graph is inherently a more general form of data structure than tensors. A broad range of models can be unified as either learning from explicit or inferring latent structures. Examples include TreeLSTM [26] that works on sentence parsing trees, Capsule Network [121] and Transformer [122] that learns soft connections among entities (e.g. capsules, words). Recently, Graph neural networks (GNNs) is a rising family of models that aim to model a set of node entities together with their relationships (edges). The application regime of the GNN framework is broad, such as chemical molecules, social networks, knowledge graphs and recommender systems [123, 124, 125, 5].

Unfortunately, existing tensor-based frameworks (e.g. Tensorflow, Pytorch, MXNet) lack intuitive support for this trend of deep graph learning. Specifically, GNNs are defined using the message passing paradigm [6], which can be seen as a model inductive bias to facilitate information flow across the graph. However, tensor-based frameworks do not support the message-passing interface. As such, researchers need to manually emulate graph computation using tensor operations, which poses an implementation challenge.

During the past year we have seen the release of several graph training systems [126, 127, 128, 27]. However, most if not all of these libraries compromise programming flexibility to boost the performance of a narrow range of GNNs, as we briefly summarize in the Table 4.1. As research on deep graph learning is going to evolve and iterate quickly, we dive in to the source of graph learning problems to provide a more comprehensive graph-oriented solution, the **Deep Graph Library (DGL)**[1].

Currently, with DGL, we provide 1) graph as the central abstraction for users; 2) flexible APIs allowing arbitrary message-passing computation over a graph; 3) support for gigantic and dynamic graphs; 4) efficient memory usage and high training speed.

DGL is **platform-agnostic** so that it can easily be integrated with tensor-oriented frameworks like PyTorch and MXNet. It is an open-source project under active development. In this paper, we compare DGL against the state-of-the-art library on multiple standard GNN setups and show the improvement of training speed and memory efficiency.

4.2 Background

Graph has been an intriguing subject of study for a long history. Frequently applied algorithms include shortest-path, different kinds of traversal algorithms, belief propagation [129] for probabilistic reasoning and variations on the page rank theme [130]. These *graph analytic algorithms* have played a central role during the boom of internet and social network assisted by specialized systems [131, 132, 133, 134, 135].

In the deep learning world, there has been continuous interest in developing DNNs for graph data. In natural language processing, TreeLSTM [26] generalizes LSTM [74] model to sentence syntactic parse trees. SPINN [136] develops a tree-sequence hybrid model for

1. Project Website: `http://dgl.ai`

more efficient training. Recently, an emerging family of neural networks, Graph Neural Networks, have achieved significant breakthroughs in modeling general graphs such as citation networks [137], knowledge graphs [124] and molecular structures [138]. The ability of GNNs to combine structural signals with side channel features, potentially extracted by other DNN models, makes them promising end-to-end approaches.

At a higher level, all these DNN models for graph data fit into the *message passing paradigm* [6, 126]. Formally, we define a graph $G(V, E)$. V is the set of nodes with $\mathbf{v}_i$ being the feature vector associated with each node. E is the set of the edge tuples $(\mathbf{e}_k, r_k, s_k)$, where $s_k \rightarrow r_k$ represents the edge from node s_k to r_k, and $\mathbf{e}_k$ is feature vector associated with the edge. The message passing paradigm contains the following edge-wise and node-wise computation:

$$\text{Edge-wise: } \mathbf{m}_k^{(t)} = \phi^e(\mathbf{e}_k^{(t-1)}, \mathbf{v}_{r_k}^{(t-1)}, \mathbf{v}_{s_k}^{(t-1)}), \tag{4.1}$$

$$\text{Node-wise: } \mathbf{v}_i^{(t)} = \phi^v(\mathbf{v}_i^{(t-1)}, \bigoplus_{\substack{k \\ \text{s.t. } r_k = i}} \mathbf{m}_k^{(t)}) \tag{4.2}$$

The equation updates a node representation $\mathbf{v}_i^{(t)}$ by collecting messages $\mathbf{m}_k^{(t)}$ sent from neighbors, which contain information of the neighbors and edges. ϕ^e and ϕ_v are *message function* and *update function* shared by all the nodes and edges, commonly parameterized by a neural network. So is the *reduce function* $\bigoplus$ for aggregating messages. A GNN model can then apply such computation iteratively each with different functions (or different set of neural network parameters) so that a node can gather information from further neighbors. Each iteration is also known as one GNN layer. The final node representations are the inputs to other classifier or decoder modules depending on the tasks.

Despite the similarity to the vertex-centric programming model [131, 132] core to the systems for graph analytics, the message passing paradigm in graph DNNs has disparate characteristics. First, each layer needs to book-keep its representations for gradient computation, which means that directly mutating node-wise states is not allowed. By contrast, graph analytic systems provide a shared-memory abstraction and state updates are usu-

		GNet	NGra	Euler	PyG	DGL
Message Passing	arbitrary ϕ^e	✓	✓	✓	✓	✓
	arbitrary ϕ^v	✓	✓	✓	✓	✓
	arbitrary $\bigoplus$	✓	✗	✓	✗	✓
Propa-gation Order	full	✓	✓	✗	✓	✓
	partial	✗	✗	✗	✓	✓
	random walk	✗	✗	✗	✓	✓
	sampling	✗	✗	✓	✓	✓
Graph Type	many & small	✓	✗	✗	✓	✓
	single & giant	✗	✓	✓	✗	✓
	dynamic	✗	✗	✗	✗	✓
	heterogeneous	✗	✗	✗	✗	✓
System	multi-platform	✗	✗	✗	✗	✓

Table 4.1: DGL vs. GraphNet (GNet), NGra, Euler and Pytorch Geometric (PyG)

ally inplace. Second, the node and edge states in DNNs are tensors while graph analytic algorithms typically have scalar states. Therefore, a graph DNN system should utilize the parallelism in node-wise and edge-wise computation which is not available in graph analytic algorithms. Third, graph analytic systems heavily rely on the assumption that the reducer function is commutative, which does not hold for graph DNNs. Finally, the node-wise and edge-wise computations reuse standard DNN modules. However, none of the graph analytic systems is compatible with existing DL frameworks.

4.3 System Challenges for DNNs on Graphs

In the recent years, there are many efforts in developing specialized systems for graph DNNs (e.g., Pytorch Geometric [27], NeuGraph [127], Euler [128], Aligraph [139], Graph-Net [126]). We elaborate the main challenges in designing such a system, what are the issues with existing GNN tools and DGL's contributions. Table 4.1 summarizes the differences between these GNN systems and DGL from multiple aspects.

Support flexible message passing paradigm. To strictly follow the message passing paradigm defined in equation 4.1, the system should let users express arbitrary message function ϕ^e, update function ϕ^v and reduce function $\oplus$. Specifically, $\oplus$ is not necessarily a commutative sum, mean, or max/min, but could be an LSTM network [125], since simple reduce operations limit the theoretical capability of GNNs [140]. Supporting arbitrary reduce function efficiently is challenging on GPU because each node receives different number of messages. As such, PyG and NeuGraph do not support arbitrary reduce function while GraphNet and Euler let users implement the non-trivial batching strategy.

Enable flexible propagation order. The message passing paradigm defines the computation on each node/edge while the propagation order determines how messages are propagated. One can trigger the computation synchronously [137, 14, 141] (*full propagation*), or following a certain order like TreeLSTM [26] and Belief Propagation [142] (*partial propagation*), or on some random walks [143, 144] and sampled subgraphs [125, 145, 146]. All said, the propagation aspect is another crucial aspect to consider.

Scale to giant graphs. Real-world graphs such as social networks, knowledge graphs and the user-item interaction graphs for recommendation can have millions of nodes. However, many graph DNN systems are incapable of training GNNs on even moderately large graphs. This is because systems like GraphNet and PyG generate explicit edge-wise messages during message passing. Storing messages consumes the size of memory proportional to the number of edges in a graph, which could be several orders of magnitude larger than the memory required for storing node features.

Support dynamic and heterogeneous graphs. Besides giant graphs, there are other dimensions to categorize graph data. The first dimension is whether they are static. Dynamic graphs are particularly important in generative models, for example to infer latent graph structure which is crucial in relational reasoning AI [147]. Graph mutation occurs from time to time, e.g. by adding/removing nodes/edges [142, 148] or pooling [141]. Finally, no graphs are born giant, they grow to be (e.g. a knowledge graph); it is important to keep this *evolution* viewpoint in mind.

The second dimension is whether they are heterogeneous. *Heterogeneous graphs* are graphs that contain different types of nodes and edges. The different types of nodes and edges tend to have different types of attributes that are designed to capture the characteristics of each node and edge type. Moreoever, within the context of graph neural networks, depending on their complexity, certain node and edge types may need to be modeled with representations that have different number of dimensions. Unfortunately, none of the existing systems supports GNNs on heterogeneous graphs.

Obviously, the above categorization is not mutually exclusive and a complete model can embrace multiple aspects, depending on model dynamics, dataset type or optimization algorithm design.

To address these challenges, we design DGL with following technical contributions.

- DGL provides graph as the central abstraction for users with efficient graph and feature storage. The graph data structure allows mutation and heterogeneous attributes.

- DGL allows the most general form of the message passing paradigm. To support arbitrary reduce function, DGL proposes an auto-batching technique that efficiently utilizes GPU and introduces little overhead. Moreover, DGL allows performing message passing on a subset of nodes and edges. Building upon this, DGL enables flexible propagation order naturally.

- DGL fuses the message and reduce functions and avoids storing explicit messages, enabling training on graphs up to 512K nodes and 209M edges on one GPU.

- The design of DGL is platform-agnostic and is compatible with multiple popular DL frameworks.

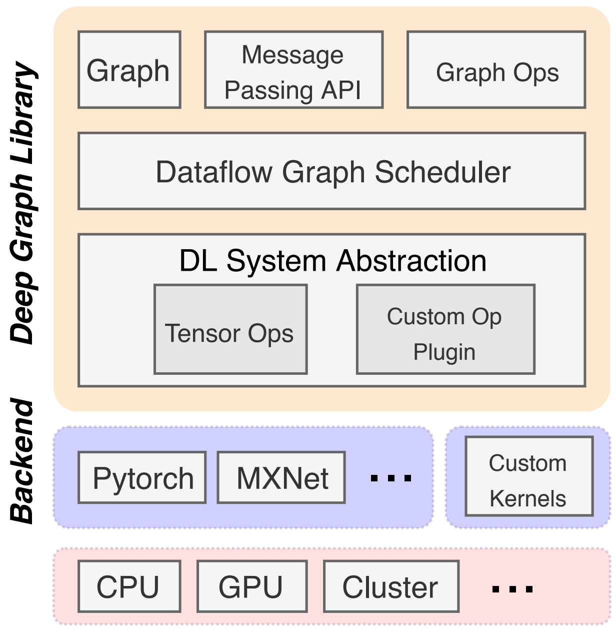


Figure 4.1: DGL system stack.

4.4 Design Overview

Figure 4.1 depicts an overview of DGL's system stack. At the top level, DGL provides programming interfaces that are intuitive to express graph DNNs including graph, message passing interfaces and common utilities on graphs such as traversal, random walks, etc. In the middle there is the *dataflow graph scheduler* in charge of translating the message passing computation into a dataflow graph for execution. The scheduler also performs batching and fusion automatically. To support multiple DL frameworks, DGL defines a *DL system abstraction* level including the tensor operators and a custom operator interface required by DGL's execution. Currently, DGL realizes the abstraction for MXNet and Pytorch, while Tensorflow support is on the way.

In the remainder of this chapter, we first highlight two key user-facing APIs: the graph (§4.5.1) and the message passing interface (§4.5.2). We then explain DGL's auto-batching (§4.6.1) and kernel fusion (§4.6.2) strategies.

```
def mfunc(edges):
    # message UDF
    return {'msg' : edges.src['h']}

def rfunc(nodes):
    # reduce UDF
    return {'newh' : nodes.mailbox['msg'].sum(dim=1)}

class GraphConv(torch.nn.Module):
    def __init__(self, in_size, out_size):
        self.linear = torch.nn.Linear(in_size, out_size)

    def forward(g, feat):
        # g : DGLGraph object, feat : node feature tensor
        g.ndata['h'] = feat / g.out_degrees().sqrt()
        g.update_all(mfunc, rfunc)
        newh = g.ndata['newh'] / g.in_degrees().sqrt()
        return self.linear(newh)
```

Figure 4.2: Example codes of Graph Convolution layer in DGL.

4.5 Programming Interface

Figure 4.2 shows the example codes in DGL for implementing a Graph Convolutional [137] layer defined as follows:

$$\mathbf{v}_i^{(t)} = \sum_{j \in \mathcal{N}(i)} \frac{1}{\sqrt{|\mathcal{N}(i)||\mathcal{N}(j)|}} \mathbf{v}_j^{(t-1)} W$$

Here, $\mathcal{N}(i)$ is the set of neighbors of node i and W is the trainable weight matrix. We use this example to lead the discussion in this section.

4.5.1 Graph and feature storage

Prior libraries including GraphNet and PyG require users to maintain graphs as sparse matrices and features separately as dense tensors. Although this is convenient for quick prototyping, many low-level system design choices are exposed to users: which sparse format (CSR or COO) to use, what is the order of stacking node/edge features?

DGL's central abstraction for graph data is `DGLGraph`. It is inspired by NetworkX [149] – a popular package for graph analytic, to which we maintain maximal similarity (e.g., `g.out_degrees()` at L13). Nodes and edges are assigned integer IDs starting from zero as indices to their features. The graph structure is stored in adjacency matrix. Different sparse formats have their own strength depending on the operations performed. For example, CSR is good at finding the successors of a node while CSC is more efficient in finding the predecessors. In DGL, we store copies of a graph in all these formats and use the suitable one depending on the invoked operation. Since the memory consumption of storing the graph structure is much smaller than storing the features, such redundancy is affordable. These copies are also created on demand. In most cases, only CSC is materialized due to the common operations to retrieve in-coming messages.

`DGLGraph` stores node/edge features so that it can access or modify them at any time, even when the graph is mutated. Users query or modify these features with a dictionary-style interface. For example `g.ndata['newh']` returns the *newh* features of all nodes (L17). The node/edge features of a certain key are packed in one dense tensor, where each row slice corresponds to the feature of a node/edge. In order to support gradient computation, feature update cannot be inplace. Therefore, DGL simulates the inplace write with an outplace tensor operator. The operator creates a *new* tensor that has the same values as the old one except the rows being updated, which is expensive when there are many writes. We address this issue in §4.6.

4.5.2 Message passing interface

DGL provides the following two basic primitives to perform computation on graphs:

$$\texttt{send}(\mathcal{E}, \phi^e), \qquad \texttt{recv}(\mathcal{V}, \bigoplus, \phi^v) \tag{4.3}$$

Here, `send` and `recv` are message passing triggers on edge and node respectively. $\mathcal{E}$ and $\mathcal{V}$ define the set of edges and nodes that are triggered for message passing. Collectively, we call them *active set*. Computation can be repeatedly triggered on different active sets to propagate messages and update features along a walk on the graph or a preset path. DGL supports a variety of propagation orders using the active set design:

- full propagation involving all nodes/edges by `update_all`($\phi^e, \bigoplus, \phi^v$) [137, 14];
- propagation from/to neighbors by `pull`($\mathcal{V}, \phi^e, \bigoplus, \phi^v$) and `push`($\mathcal{V}, \phi^e, \bigoplus, \phi^v$) [145];
- subset node/edge propagation under certain traversal order [26, 142];
- random walk [5, 125];
- active sets that are generated dynamically, including exploration algorithms on graph navigation [150, 151] and graph generative models [148].

ϕ^e, $\bigoplus$, ϕ^v are the message, reduce and update functions. More precisely, they can be specified as follows:

- **User-defined function (UDF).** A UDF can be any function or class that acts like function. For example, they could be parameterized by neural network modules (e.g., `torch.nn.module`). The message function has the access to the feature data of an edge and its source and destination nodes and returns the messages (L1-3). The messages are then delivered to the `mailbox` of the destination nodes, accessible by the reduce function for aggregation (L5-7). The results are stored as node features. To harness the parallelism of GPU devices, we require a UDF to perform computation in a batch which we will explain later in §4.6.1.
- **Built-in function.** DGL provides symbolic functions for common message and reduce functions. For example, users can replace `rfunc` at L16 with `dgl.function.sum('msg', 'newh')` for message summation, saving programming efforts. Built-in functions

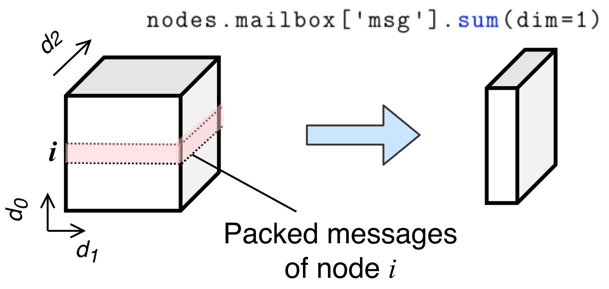


Figure 4.3: The semantics of the message tensor dimensions in a reduce UDF.

makes it easier to analyze the program and enables the kernel fusion technique in §4.6.2.

Besides the graph and message passing interfaces, DGL exposes many common routines for training various graph DNNs. We provide a batching API to collect a set of small graphs of different sizes into a larger graph by treating them as disjoint components. When faced with much larger graphs, many [125, 146] propose to sample the graph to avoid full-graph propagation. In DGL, we provide graph sampling API in the form of data loader rendering sampled subgraphs. Optionally, these subgraphs can then be batched for efficient processing.

4.6 Optimization and Implementation

This section explains the optimizations in our system for accelerating graph DNN training.

4.6.1 Auto-batching user-defined message passing

The node-wise and edge-wise computations in the message passing must be batched in order to run efficiently on GPUs. Batching message and update functions is straightforward because they operate on every node and edge independently. Batching reduce function is non-trivial because the number of messages to reduce is different for each node. Prior work such as DyNet [29] and Tensorflow Fold [28] perform auto-batching by analyzing dataflow graphs for batching opportunities, but bring significant overhead. Specifically, DyNet constructs one dataflow graph for every node which is not viable in a graph of hundreds of thousands of nodes.

DGL leverages two insights to efficiently batch reduce functions. First, most tensor operators in deep learning systems support batched computation by specifying the dimension the inputs are batched on. Second, nodes receiving the same number of messages can compute in a batch. Therefore, we let the UDFs operate directly on the features of a batch of nodes and edges. For example, the message function at L3 operates on `edges.src['h']`, which packs the source node features of all the edges. For reduce phase, we divide nodes into buckets based on the number of received messages and invoke reduce UDF to aggregate the messages of each bucket. The (i, j) slice of the tensor for the batched messages is the j^{th} message received by the i^{th} node in the current bucket (Figure 4.3). In such way, DGL replaces the expensive dataflow graph construction in previous approaches with a lightweight degree calculation, which is essential to scaling to large graphs.

4.6.2 Fusing message and reduce functions

The batching-by-degree strategy is applicable to arbitrary reduce function, but suffers from two problems. For graphs with high degree variance, it generates small batches with few nodes leading to under-utilization of GPUs. Moreover, storing messages consumes an excessive amount of memory especially on graphs of many edges.

In DGL, system detects and fuses message and reduce phases. Taking `update_all` $(\phi^e, \bigoplus, \phi^v)$, it assesses ϕ^e and $\bigoplus$. When ϕ^e is an element-wise operation like $+ - \times \div$ and $\bigoplus$ is a commutative reduction like sum/mean/max/min, it calls a fused GPU/CPU kernel. The kernel divides the workload into edge chunks for parallel processing, where a block of threads further process the feature dimension cooperatively. Each thread loads the required node or edge feature data, computes ϕ^e, and saves the message into its register.

Dataset		Model	Accuracy	Time		Memory	
|V|	|E|			PyG	DGL	PyG	DGL
Cora		GCN	81.31 ± 0.88	**0.478**	0.666	1.1	1.1
3K	11K	GAT	83.98 ± 0.52	1.608	**1.399**	1.2	**1.1**
CiteSeer		GCN	70.98 ± 0.68	**0.490**	0.674	1.1	1.1
3K	9K	GAT	69.96 ± 0.53	1.606	**1.399**	1.3	**1.2**
PubMed		GCN	79.00 ± 0.41	**0.491**	0.690	1.1	1.1
20K	889K	GAT	77.65 ± 0.32	1.946	**1.393**	1.6	**1.2**
Reddit 232K	114M	GCN	93.46 ± 0.06	*OOM*	**28.6**	*OOM*	**11.7**
Reddit-S 232K	23M	GCN	N/A	29.12	**9.44**	15.7	**3.6**

Table 4.2: Training time (in seconds) for 200 epochs and memory consumption (GB).

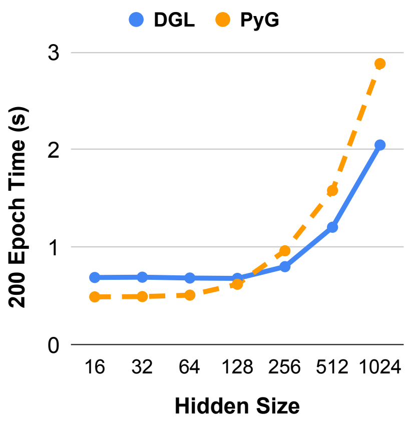


Figure 4.4: GCN training time on Pubmed with varying hidden size.

It then uses atomic instructions to accumulate the results. When fusion is possible it saves both time and memory, since messages typically do not need to be materialized. When computing the gradients, the backward kernel adopts the same thread scheduling but needs to re-compute the messages dropped in the forward pass. The re-computation does not involve message reduction so has negligible overhead.

4.7 Evaluation

We evaluate our system to demonstrate its effectiveness in several settings. We first compare DGL with the implementations by the original author and by existing graph DNN systems on small models and datasets. To show that DGL can scale to much larger graphs, we evaluate it on synthetic graphs. Finally, we measure the performance of our auto-batching technique on classical models. All experiments use DGL configured with Pytorch (v1.0) backend and are carried out on an AWS EC2 p3.2xlarge instance installed with one NVIDIA Tesla V100 GPU (16GB) and 8 vCPUs.

4.7.1 Speed and memory efficiency

Experimental setup. We focus on the semi-supervised setup for node classification [137]. Our datasets include three citation networks (Cora, Citeseer and Pubmed [137]) as well as the larger scale Reddit posting graph ([125]). For model selection, we follow the setup in the original papers and report the average performance from 10 rounds of experiments.

Table 4.2 evaluates the speed and memory consumption of training Graph Convolutional Network [137] and Graph Attention Network [14]. We primarily compare DGL v0.3 against PyG v1.0.3 [27][2]. As we can see from Table 4.2, DGL is consistently faster than PyG in training GAT. For GCN, DGL lags behind because it has a slightly higher python overhead cost in the wrapper codes (e.g. storage APIs) in the original setting. However when we increase the hidden layer size, DGL starts to out-perform PyG, as shown in Figure 4.4. GAT causes OOM for both DGL and PyG on Reddit so is not included.

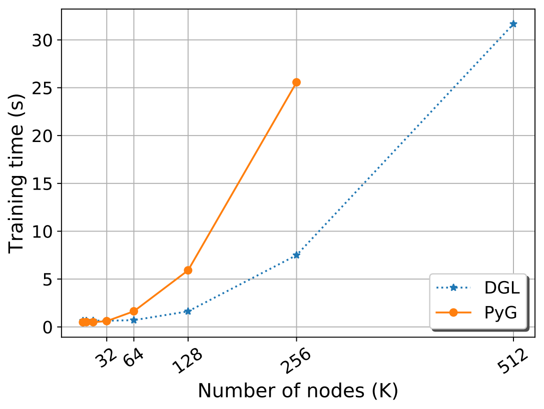
DGL's superior performance is largely owing to kernel fusion. It brings both speed and memory advantage, in particular on larger and denser graphs such as Reddit graph whose average degree is around 500. On this dataset, PyG runs out of memory while DGL can complete training by using less than 12 GB memory. For further comparison, we randomly prune out 80% of the edges in Reddit graph (dubbed as Reddit-S). Under this setup, we can measure PyG's memory consumption. In this case, DGL is still **3**$\times$ faster and saves **77%** memory.

4.7.2 Scalability

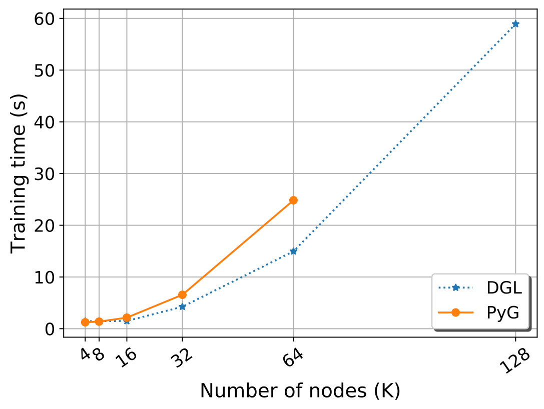
We test DGL on synthetic graphs to further evaluate the performance on larger and denser graphs. All graphs are generated using the Erdos-Renyi model. In all experiments, we train two models GCN and GAT for 200 epochs and measure the running time.

Figure 4.5 shows the scalability of DGL with growing graph size. The density of the

2. PyG compared with DGL v0.2 in their paper which does not include DGL's the kernel fusion feature.

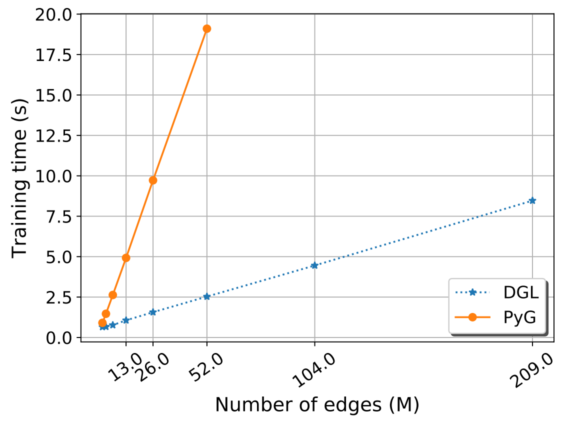


(a) GCN training time

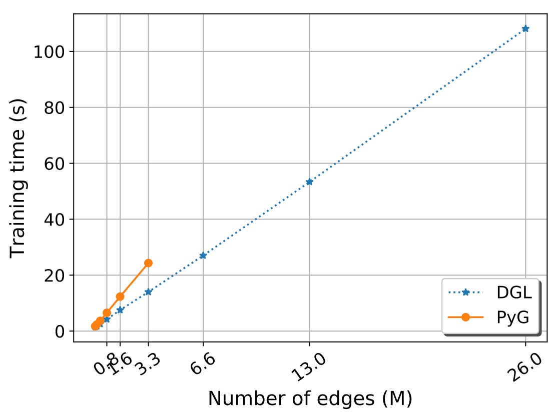


(b) GAT training time

Figure 4.5: Time to train GCN and GAT for 200 epochs on synthetic graphs of increasing number of nodes. All graphs have fixed density 0.08%.

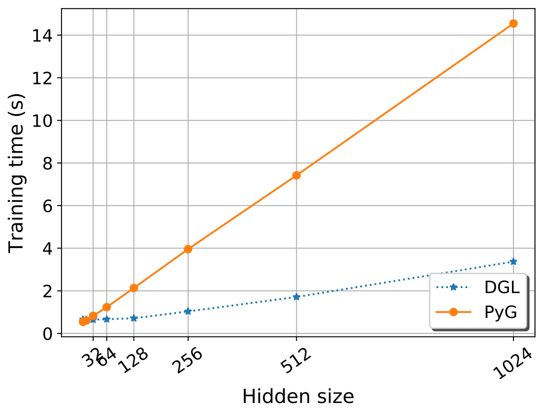


(a) GCN training time

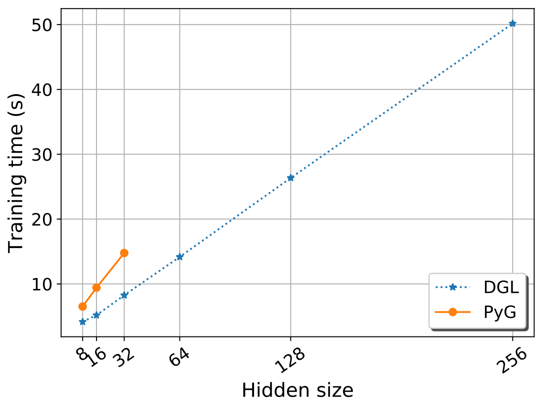


(b) GAT training time

Figure 4.6: Time to train GCN and GAT for 200 epochs on synthetic graphs of increasing density. All graphs have 32K nodes.



(a) GCN training time



(b) GAT training time

Figure 4.7: Time to train GCN and GAT for 200 epochs with increasing hidden sizes on a synthetic graph. The graph has 32K nodes and density is 0.08%.

graphs are fixed (0.008%) so the number of edges grows quadratically. DGL is much faster than PyG and uses much less memory. It can train GCN and GAT on graphs of up to 512K and 128K nodes respectively on one GPU, while PyG runs OOM. On graph of size 256K, DGL can train GCN 3.4× faster than PyG and only occupies 4.7GB memory. We then fix the graph size to be 32K and densify the graph by more edges (figure 4.6). PyG quickly runs OOM especially for GAT as it is more memory demanding on denser graphs. By contrast, DGL can scale to graphs of 209M edges and 26M edges for GCN and GAT respectively, and be 7.5× faster when both systems fit the training in one GPU. Finally, we fix the graph size and density but scale the GNN models with larger hidden dimension (figure 4.7). For GCN, DGL is always faster than PyG on models with hidden sizes larger than 32, and is 4.3× faster for hidden size equal to 1024. For GAT, DGL can fit hidden size up to 256 while PyG can only fit 32.

The advantages of DGL in both speed and memory come from kernel fusion, which avoids storing messages and the high traffic for accessing them.

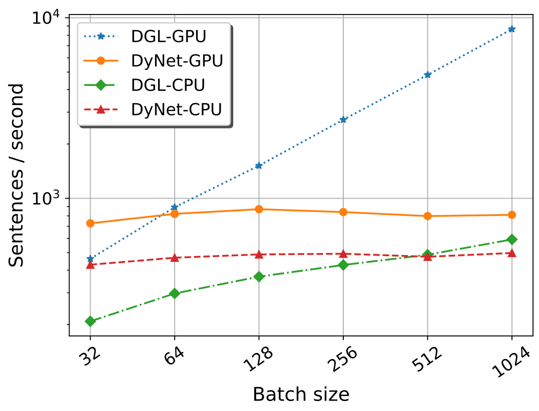


Figure 4.8: Comparison of training throughputs between DGL and DyNet with auto-batching on TreeLSTMs.

4.7.3 Auto-batching

We compare the DyNet implementation of the Stanford Sentiment Treebank regression [152] task using the same TreeLSTM model [26]. Figure 4.8 shows the training throughputs under different batch sizes. DGL is capable of achieving more throughput with larger batch size on GPU while DyNet is bottlenecked by the overhead of dataflow graph construction. As a result, with batch size 1024, DGL can train 10.67× faster and can finish one epoch in only 1.70s. On CPU, the training speeds are similar and DGL is faster when batch size is large. In addition, it is worth noting that DGL's implementation uses existing modules and operators from Pytorch, which is much more straightforward and easier for developing and debugging.

4.8 Conclusion

We present Deep Graph Library, a graph-oriented library built for deep learning on Graphs. On the ML side, we will continue to push ease of use, diversity and scale. On the system end, as the models grow deeper and larger, or structurally more complex, how to speed up them using modern hardware is another challenge; kernel fusion technique in

DGL is only a stepping stone.

Chapter 5

Conclusion and Future Work

The dissertation studied the problems of training emerging deep neural networks including DNNs scaling beyond the capacity of a single device and DNNs for graph structured data. Exploring these models is limited in current deep learning systems due to burdensome programming experience and complex system decisions. We pointed out that the insufficiency is due to the current single stage design and the lack of operator semantics. By this motivation, our proposed SMEX design extends the previous pipeline to two stages, semantic dataflow graph and execution dataflow graph, allowing the system to automatically optimize for execution with little user awareness. In addition, it introduces operator specifications into the system pipeline, enabling more extensive optimizations.

We applied the design to two concrete systems, Tofu and DGL. Tofu supports training very large models by tensor partitioning. As discussed in §3.3, the sheer quantity of the possible partition strategies and the complexity in choosing the right one should be opaque to users. Moreover, Tofu demonstrates that the introduction of operator semantics (in Tensor Description Language (§3.4.1)) is necessary for the solution to be applicable more widely. Because searching for the best partition strategy of a whole program is hard, we proposed several techniques to shrink the search space (§3.5.2). With these improvements, Tofu can find the best parallel strategies for state-of-the-art DNNs in seconds and can train

6× larger DNNs using 8 GPUs efficiently. It is also worth noting that all these require no change to user programs and little effort in writing operator specifications.

The second part of the thesis introduces a new system DGL for training graph DNNs. Based on the common definition of graph DNNs (§4.3), we proposed the message passing programming interface as a natural form of operator specification. Besides, DGL provides flexible APIs to cover a wide range of graph DNNs (§4.5). To save both memory and training time, DGL batches the node-wise and edge-wise computation automatically and aggressively fuses them if possible (§4.6)), all of which further improve the usability of DGL. The evaluation shows that DGL is 10× faster than previous auto-batching approaches and can train DNNs efficiently on much larger graphs using one GPU.

All is said, there are still many remaining challenges which shed light on some future directions.

5.1 More General Operator Specification

Designing a flexible operator specification is challenging. As an example, Tensor Description Language (TDL) is a simple language without control flow primitives and data-dependent indexing. Furthermore, Tofu does not support sparse tensor operations due to load-imbalance, even though they can usually be described in TDL. The message passing interface in DGL can describe certain sparse operations in the notion of graph. Developing a powerful whilst natural operator specification is a promising direction.

Besides, there is no guarantee that the operator implementation matches its description. In fact, such verification is an open research problem even if the underlying implementation is open sourced. A more promising direction is to leverage recent operator code-generation tools such as TVM [67], TC [68], and TACO [153].

5.2 More Extensive System Optimizations

Tofu only supports parallelization via partition-n-reduce, which restricts each worker to perform a coarse-grained task identical to the original computation. This pattern is not applicable to all parallelizable computation (e.g. Cholesky [60]). Furthermore, the partition-n-reduce parallel strategies do not necessarily minimize communication, and do not take advantage of the underlying interconnect topology. By contrast, parallel algorithms developed for specific computation (e.g. matrix multiplication [61, 94], tensor contraction [106]) are explicitly structured to minimize communication and exploit the interconnect topology.

Furthermore, Tofu always partitions every operator and tensor across all workers. For moderately sized DNN models, partitioning across all workers lead to small GPU kernels that leave a GPU unsaturated. In such scenarios, it may be beneficial to leave certain operators un-partitioned or partially partitioned among a subset of workers. Furthermore, Tofu has no support for non-uniform partitioning when GPUs have different computing and memory capacity. Although Tofu's search algorithm tries to accommodate bandwidth differences in a hierarchical interconnect, it does not explicitly optimize communication according to the interconnect topology.

Unfortunately, Tofu's recursive search cannot be extended to address the above limitations. This is because the underlying DP algorithm cannot optimally search different device placement choices for un-partitioned, or non-uniformly-partitioned operators. Exploring stochastic or machine learning based search mechanisms [41, 66, 25] is a direction of future work.

5.3 Scale to Real World Graphs

Graphs from real world scenarios can have hundreds of billions of nodes and edges. Processing graphs of such magnitude requires advancement from both system and machine learning sides. Graph partitioning has been well studied for graph analytic algorithms [132, 154] and is recently explored in graph DNN domain [127]. However, the effectiveness of graph partitioning is constrained by the connectivity of a graph. A perhaps more promising direction is to develop stochastic algorithms[145, 5, 146, 155]. This involves several technical challenges such as how to balance the *exploration* (the urge to gather global information) and *exploitation* (the benefit from utilizing locality).

Appendix A

Recursive Partitioning Algorithm and its Correctness

A.1 Recursive partition plan

We first formally define the partition plan of a dataflow graph. Given a dataflow graph G, a partition plan P consists of the choices of how each tensor is partitioned and how each operator is paralleled. Note that the tensor can be partitioned along multiple dimensions but the number of splits should be equal to the number of GPUs.

Given 2^k GPUs, any partition plan for a dataflow graph can be realized by a sequence of recursive steps, $\langle p_1, p_2, \ldots, p_k \rangle$, where each p_i is a *basic partition plan* that partitions tensors along only one dimension among two (groups of) workers. Note that after i steps, there are 2^i identical sub-dataflow graphs whose tensors are $1/2^i$ the original size. So the p_{i+1} basic partition plan is applied to all 2^i sub-graphs.

To see a concrete example, consider a dataflow graph of one matrix multiplication `C=A*B`. Suppose there are 4 GPUs, an example partition plan sequence $\langle p_1, p_2 \rangle$ is as follows:

$$p_1 = \{\mathtt{A} \mapsto 0, \mathtt{B} \mapsto 0, \mathtt{C} \mapsto 0, * \mapsto 1\}$$

$$p_2 = \{\mathtt{A} \mapsto 1, \mathtt{B} \mapsto 0, \mathtt{C} \mapsto 0, * \mapsto 0\}$$

Here, `A` $\mapsto 0$ means that row dimension is chosen to partition the matrix; $* \mapsto 1$ means the second partition-n-reduce strategy is chosen for the multiplication operator. Because p_1 has partitioned each matrix into two sub-matrices, p_2 is applied on the sub-matrices. The sequence represents a plan where `A` is partitioned into a 2x2 grid and `B` and C are partitioned into four row strips.

Tofu's recursive partition algorithm chooses a sequence of partition plans $A = \langle a_1, a_2, \dots, a_k \rangle$ in k recursive steps and we want to show that this sequence is no worse than the optimal sequence $O = \langle o_1, o_2, \dots, o_k \rangle$.

A.2 Region Analysis

Recall in Sec 3.4.2, we use symbolic interval to analyze the access pattern of an operator. Let $\mathcal{X}_1, \dots, \mathcal{X}_n$ and $\mathcal{Y}_1, \dots, \mathcal{Y}_m$ be the symbolic upper bound of each output index and access range of each input dimension, respectively. The analysis produces following affine transformation:

$$\begin{pmatrix} \mathcal{Y}_1 \\ \vdots \\ \mathcal{Y}_m \end{pmatrix} = \begin{pmatrix} \alpha_{11} & \alpha_{12} & \dots \\ \vdots & \ddots & \\ \alpha_{m1} & & \alpha_{m(n+1)} \end{pmatrix} \begin{pmatrix} \mathcal{X}_1 \\ \vdots \\ \mathcal{X}_n \\ 1 \end{pmatrix} \tag{A.1}$$

Here, we consider a restricted form of affine transformation:

Assumption#1. Each output index is used to access only one dimension for each input tensor. The same output index can be used in multiple input tensors such as element-wise operators `lambda i : A[i] + B[i]`, but `lambda i: A[i, i]` is not considered. In practice, we do not encounter any such example when investigating operators in MXNet and Tensorflow.

Assumption#2. We only consider input dimensions that scale linearly with one output index (i.e, in the form of $\mathcal{Y}_i = \alpha_i \mathcal{X}_j$). This restriction rules out the partition-n-reduce strategies such as halo exchange in convolution, but still includes many strategies such as partitioning on channel dimension. Whether such assumption is necessary or not requires further study.

Consider an operator that has output shape $\mathcal{X}_1 \times \ldots \times \mathcal{X}_n$. The shape of any of its input tensors can be represented as $\alpha_1 \mathcal{X}_{\pi_1} \times \ldots \times \alpha_d \mathcal{X}_{\pi_d}$, where d is the number of dimensions, $\alpha_1 \ldots \alpha_d$ are constants and π is a permutation of $1 \ldots n$.

A.3 Communication cost

Theorem 1. *Let p be any basic partition plan, the communication cost incurred by p is proportional to the total tensor size in the dataflow graph.*

Proof. Communication happens in two situations:

- The selected partition-n-reduce strategy requires input region that is not available locally.
- The selected partition-n-reduce strategy produces output region that is assigned to other devices.

Consider an operator whose output shape is $\mathcal{X}_1 \times \ldots \times \mathcal{X}_n$ and the partition plan p chooses to partition the dimension i into halves. For the first case, the communication required to fetch one of the input tensor is either:

$$\frac{1}{2}\Pi_{j=1}^{d} \alpha_j \mathcal{X}_{\pi_j}$$

if i is not included in $\pi_1 \ldots \pi_d$ (i.e, the whole tensor is needed), or

$$\frac{1}{4}\Pi_{j=1}^{d} \alpha_j \mathcal{X}_{\pi_j}$$

, otherwise.

The same analysis can be applied to the second case. Because the communication cost of each individual operator is proportional to the tensor size, the total cost is proportional to the total tensor size of the dataflow graph. □

Let cost(P) be the total communication cost of a partition plan sequence P. Due to symmetry, the cost can be calculated by aggregating the within-group communication cost incurred by each basic partition plan:

$$\text{cost}(P) = \sum_{i=1}^{k} 2^{i-1}\text{cost}(p_i) \tag{A.2}$$

We can then show that the following commutativity property holds:

Theorem 2. *cost*$(\langle p_1, p_2\rangle) =$ *cost*$(\langle p_2, p_1\rangle)$*, where* p_1 *and* p_2 *are basic partition plans.*

Proof. The case is trivial if $p_1 = p_2$. Let G be the unpartitioned dataflow graph; G_1 and G_2 be the partitioned graph by p_1 and p_2; $S(G)$ be the total tensor size of a dataflow graph G. Because every tensor is partitioned by half, $S(G_1) = S(G_2) = \frac{1}{2}S(G)$. By theorem 1, we then have:

$$\begin{aligned}
\text{cost}(\langle p_1, p_2\rangle) &= \text{cost}(p_1) + 2 * \text{cost}(p_2) \\
&= \alpha_1 S(G) + 2 * \alpha_2 S(G_1) \\
&= \alpha_1 * 2 * S(G_2) + 2 * \alpha_2 \tfrac{1}{2} S(G) \\
&= \alpha_2 S(G) + 2 * \alpha_1 S(G_2) \\
&= \text{cost}(p_2) + 2 * \text{cost}(p_1) \\
&= \text{cost}(\langle p_2, p_1\rangle)
\end{aligned}$$

□

Let the per-step cost be $\delta_i = 2^{i-1}\text{cost}(p_i)$. We can easily prove theorem **??**.

Proof. Suppose there exists a sequence $\langle p_1, \ldots, p_i, p_{i+1}\rangle$ such that $\delta_i > \delta_{i+1}$. By theorem 2,

$$\text{cost}(\langle p_1, \ldots, p_i, p_{i+1}\rangle) = \text{cost}(\langle p_1, \ldots, p_{i+1}, p_i\rangle)$$

Because $\delta_i > \delta_{i+1}$, we have

$$\text{cost}(\langle p_1, \dots, p_{i+1} \rangle) < \text{cost}(\langle p_1, \dots, p_i \rangle)$$

This means applying p_{i+1} instead of p_i at step i is a more optimized partitioning, which contradicts with the per-step optimality of the dynamic programming algorithm. □

A.4 Optimiality proof

Theorem 3. *The recursive algorithm is optimal under assumption #1 and #2.*

Proof. Let $A = \langle a_1, a_2, \dots, a_k \rangle$ be the partition sequence produced by the recursive algorithm and $O = \langle o_1, o_2, \dots, o_k \rangle$ be the optimal sequence. By theorem 2, we can reorder the sequence so that the per-step costs of both sequences are non-descending.

We prove by contradiction. Suppose $\text{cost}(A) > \text{cost}(O)$. Then there must exist a step i such that:

$$\text{cost}(\langle a_1, \dots, a_i \rangle) \leqslant \text{cost}(\langle o_1, \dots, o_i \rangle) \tag{A.3}$$

$$\text{cost}(\langle a_1, \dots, a_i, a_{i+1} \rangle) > \text{cost}(\langle o_1, \dots, o_i, o_{i+1} \rangle) \tag{A.4}$$

Let G_a and G_o be the partitioned dataflow graphs after applying $\langle a_1, \dots, a_i \rangle$ and $\langle o_1, \dots, o_i \rangle$, respectively. Every tensor is only 2^i of the size of original tensor so $S(G_a) = S(G_o)$. Finally, by theorem 1, we have

$$\begin{aligned}
\text{cost}(\langle a_1, \dots, a_i, a_{i+1} \rangle) &> \text{cost}(\langle o_1, \dots, o_i, o_{i+1} \rangle) \\
&= \text{cost}(\langle o_1, \dots, o_i \rangle) + 2^i \alpha_o S(G_o) \\
&\geq \text{cost}(\langle a_1, \dots, a_i \rangle) + 2^i \alpha_o S(G_a) \\
&= \text{cost}(\langle a_1, \dots, a_i, o_{i+1} \rangle)
\end{aligned}$$

Hence, applying o_{i+1} at step $i+1$ produces strictly less communication cost than applying a_{i+1}, which contradicts the optimality of the dynamic programming algorithm.

□

Bibliography

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[2] 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*, pages 770–778, 2016.

[3] Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara Sainath, and Brian Kingsbury. Deep neural networks for acoustic modeling in speech recognition. *Signal Processing Magazine*, 2012.

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[5] 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*, pages 974–983. ACM, 2018.

[6] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International Conference on Machine Learning*, 2017.

[7] 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 effi-

cient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[8] 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 *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[9] Facebook. Caffe2: a lightweight, modular, and scalable deep learning framework, 2017.

[10] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, volume 5, pages 1–6, 2015.

[11] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[12] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8), 2019.

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[14] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. *International Conference on Learning Representations*, 2018.

[15] 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.

[16] Chen Meng, Minmin Sun, Jun Yang, Minghui Qiu, and Yang Gu. Training deeper models by gpu memory optimization on tensorflow. In *Proc. of ML Systems Workshop in NIPS*, 2017.

[17] Audrunas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. Memory-efficient backpropagation through time. In *Advances in Neural Information Processing Systems*, pages 4125–4133, 2016.

[18] James Martens and Ilya Sutskever. Training deep and recurrent networks with hessian-free optimization. In *Neural networks: Tricks of the trade*, pages 479–535. Springer, 2012.

[19] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[20] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. Deep learning with COTS HPC systems. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1337–1345, 2013.

[21] Minjie Wang, Tianjun Xiao, Jianpeng Li, Jiaxing Zhang, Chuntao Hong, and Zheng Zhang. Minerva: A scalable and highly efficient training platform for deep learning. In *NIPS Workshop, Distributed Machine Learning and Matrix Computations*, 2014.

[22] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. In *arXiv:1404.5997*, 2014.

[23] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. Tetris: Scalable and efficient neural network acceleration with 3d memory. *ACM SIGOPS Operating Systems Review*, 51(2):751–764, 2017.

[24] Zhihao Jia, Sina Lin, Charles R. Qi, and Alex Aiken. Exploring hidden dimensions in parallelizing convolutional neural networks. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, pages 2279–2288, 2018.

[25] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *arXiv preprint arXiv:1807.05358*, 2018.

[26] Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.

[27] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. *CoRR*, abs/1903.02428, 2019.

[28] Moshe Looks, Marcello Herreshoff, DeLesley Hutchins, and Peter Norvig. Deep learning with dynamic computation graphs. *arXiv preprint arXiv:1702.02181*, 2017.

[29] Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, et al. Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980*, 2017.

[30] Alex Krizhevsky. Cuda-convnet. `https://code.google.com/p/cuda-convnet/`.

[31] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[32] Cxxnet: a fast, concise, distributed deep learning framework. `https://github.com/dmlc/cxxnet`.

[33] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.

[34] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, 2010.

[35] NVIDIA. cudnn: Gpu accelerated deep learning.

[36] Intel. Intel math kernel library.

[37] Jeff Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Symposium on Operating System Design and Implementation (OSDI)*, 2004.

[38] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *European Conference on Computer Systems (EuroSys)*, 2007.

[39] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

[40] Mxnet gluon. `https://gluon.mxnet.io/`.

[41] 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. *arXiv preprint arXiv:1706.04972*, 2017.

[42] NNVM. `https://github.com/dmlc/nnvm`.

[43] Open neural network exchange. `https://onnx.ai/`.

[44] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. Relay: a new ir for machine learning frameworks. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 58–68. ACM, 2018.

[45] Multi-level intermediate representation. `https://mlir.llvm.org/`.

[46] Olivier Breuleux and Bart van Merriënboer. Automatic differentiation in myia. 2017.

[47] Bart van Merrienboer, Dan Moldovan, and Alexander Wiltschko. Tangent: Automatic differentiation using source-code transformation for dynamically typed array programming. In *Advances in Neural Information Processing Systems*, pages 6256–6265, 2018.

[48] 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 *Neural Information Processing Systems (NIPS)*, 2012.

[49] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *USENIX OSDI*, 2014.

[50] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[51] Priya Goyal, Piotr Dollar, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. In *arXiv:1706.02677*, 2017.

[52] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, 2014.

[53] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. In *arXiv:1605.07146*, 2016.

[54] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, and Mohammad Norouzi. Google's neural machine translation system: Bridging the gap between human and machine translation. In *arxiv.org:1609.08144*, 2016.

[55] Xuan Yang, Jing Pu, Blaine Burton Rister, Nikhil Bhagdikar, Stephen Richardson, Shahar Kvatinsky, Jonathan Ragan-Kelley, Ardavan Pedram, and Mark Horowitz. A systematic approach to blocking convolutional neural networks. *arXiv preprint arXiv:1606.04209*, 2016.

[56] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 380–392. IEEE, 2016.

[57] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

[58] Rafal Józefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the limits of language modeling. *CoRR*, abs/1602.02410, 2016.

[59] Google Cloud. Tpu: System architecture.

[60] Zhengping Qian, Xiuwei Chen, Nanxi Kang, Mingcheng Chen, Yuan Yu, Thomas Moscibroda, and Zheng Zhang. MadLINQ: large-scale distributed matrix computation for the cloud. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, 2012.

[61] L. E. Cannon. *A cellular computer to implement the Kalman Filter Algorithm*. PhD thesis, Montana State University, 1969.

[62] Ken Kennedy and Ulrich Kremer. Automatic data layout for distributed-memory machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(4):869–916, 1998.

[63] Ulrich Kremer. Np-completeness of dynamic remapping. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers, Delft, The Netherlands*, 1993.

[64] Jingke Li and Marina Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *Frontiers of Massively Parallel Computation, 1990. Proceedings., 3rd Symposium on the*, pages 424–433. IEEE, 1990.

[65] Jingke Li and Marina Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of parallel and distributed computing*, 13(2):213–221, 1991.

[66] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V. Le, and Jeff Dean. A hierarchical model for device placement. In *ICLR*, 2018.

[67] 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 *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA, 2018. USENIX Association.

[68] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. In *arXiv:1802.04730v2*, 2018.

[69] Arnaud J Venet. The gauge domain: scalable analysis of linear inequality invariants. In *International Conference on Computer Aided Verification*, pages 139–154. Springer, 2012.

[70] Radu Rugina and Martin Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *ACM Sigplan Notices*, volume 35, pages 182–195. ACM, 2000.

[71] Xueguang Wu, Liqian Chen, and Ji Wang. An abstract domain to infer symbolic ranges over nonnegative parameters. *Electronic Notes in Theoretical Computer Science*, 307:33–45, 2014.

[72] Chien-Chin Huang, Qi Chen, Zhaoguo Wang, Russell Power, Jorge Ortiz, Jinyang Li, and Zhen Xiao. Spartan: A distributed array framework with smart tiling. In *USENIX Annual Technical Conference*, 2015.

[73] J.A Bondy and U.S.R. Murty. *Graph Theory with Applications.* Elseyier Science Publishing, 1976.

[74] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[75] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[76] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[77] Taro Sekiyama, Takashi Imamichi, Haruki Imai, and Rudy Raymond. Profile-guided memory optimization for deep neural networks. *arXiv preprint arXiv:1804.10001*, 2018.

[78] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–13. IEEE, 2016.

[79] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[80] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

[81] H. Cui, J. Cipar, Q. Ho, J.K. Kim, S. Lee, A. Kumar, J.Wei, W. Dai, G. R. Ganger, P.B. Gibbons, G. A. Gibson, and E. P. Xing. Exploiting bounded staleness to speed up big data analytics. In *USENIX Annual Technical Conference*, 2014.

[82] J. Wei, W. Dai, A. Qiao, H. Cui, Q. Ho, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E.P. Xing. Managed communication and consistency for fast data-parallel iterative analytics. In *ACM Symposium on Cloud Computing (SoCC)*, 2015.

[83] Henggang Cui, Hao Zhang, Gregory R. Ganger, Phillip B. Gibbons, and Eric P. Xing. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Eurosys*, 2016.

[84] Jin Kyu Kim, Qirong Ho, Seunghak Lee, Xun Zheng, Wei Dai, Garth Gibson, and Eric Xing. Strads: A distributed framework for scheduled model parallel machine learning. In *Eurosys*, 2016.

[85] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015.

[86] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[87] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014.

[88] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Advances in neural information processing systems*, pages 4107–4115, 2016.

[89] Edward Anderson, Zhaojun Bai, J Dongarra, A Greenbaum, A McKenney, Jeremy Du Croz, S Hammerling, J Demmel, C Bischof, and Danny Sorensen. LAPACK: A portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pages 2–11. IEEE Computer Society Press, 1990.

[90] Jaeyoung Choi, Jack J Dongarra, Roldan Pozo, and David W Walker. Scalapack: A scalable linear algebra library for distributed memory concurrent computers. In *Frontiers of Massively Parallel Computation, 1992., Fourth Symposium on the*, pages 120–127. IEEE, 1992.

[91] Jack Poulson, Bryan Marker, Robert A. van de Geijn, Jeff R. Hammond, and Nichols A. Romero. Elemental: A new framework for distributed memory dense matrix computations. *ACM Trans. Math. Softw.*, 39(2):13:1–13:24, feb 2013.

[92] Jaroslaw Nieplocha, Robert J Harrison, and Richard J Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10(2):169–189, 1996.

[93] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[94] Robert A. van de Geijn and Jerrell Watts. Summa: Scalable universal matrix multiplication algorithm. Technical report, Austin, TX, USA, 1995.

[95] Edgar Solomonik, Devin Matthews, Jeff R Hammond, John F Stanton, and James Demmel. A massively parallel tensor contraction framework for coupled-cluster computations. *Journal of Parallel and Distributed Computing*, 74(12):3176–3190, 2014.

[96] Calvin Lin and Lawrence Snyder. ZPL: An array sublanguage. In *Languages and Compilers for Parallel Computing*, pages 96–114. Springer, 1994.

[97] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 2007.

[98] UPC Consortium. UPC language specifications, v1.2. Technical report, Lawrence Berkeley National Lab, 2005.

[99] Joe B. Buck, Noah Watkins, Jeff LeFevre, Kleoni Ioannidou, Carlos Maltzahn, Neoklis Polyzotis, and Scott Brandt. Scihadoop: array-based query processing in hadoop. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[100] Murray Stokely, Farzan Rohani, and Eric Tassone. Large-scale parallel statistical forecasting computations in r. In *JSM Proceedings, Section on Physical and Engineering Sciences*, Alexandria, VA, 2011.

[101] SparkR: R frontend for Spark. `http://amplab-extras.github.io/SparkR-pkg`.

[102] Mingxing Zhang, Yongwei Wu, Kang Chen, Teng Ma, and Weimin Zheng. Measuring and optimizing distributed array programs. *Proc. VLDB Endow.*, 9(12):912–923, August 2016.

[103] Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Arvind K. Sujeeth, Christopher De Sa, Christopher Aberger, and Kunle Olukotun. Have abstraction and eat performance, too: Optimized heterogeneous computing with parallel patterns. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, 2016.

[104] 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.

[105] Tyler Denniston, Shoaib Kamil, and Saman Amarasinghe. Distributed halide. In *Principles and Practice of Parallel Programming (PPoPP)*, 2016.

[106] Edgar Solomonik, Devin Matthews, Jeff Hammond, and James Demmel. Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 813–824. IEEE, 2013.

[107] So Hirata. Tensor contraction engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories. *The Journal of Physical Chemistry A*, 107(46):9887–9897, 2003.

[108] Stefan C. Müller, Gustavo Alonso, Adam Amara, and André Csillaghy. Pydron: Semi-automatic parallelization for multi-core and the cloud. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 645–659, Broomfield, CO, October 2014. USENIX Association.

[109] David E Hudak and Santosh G Abraham. Compiler techniques for data partitioning of sequentially iterated parallel loops. In *ACM SIGARCH Computer Architecture News*, volume 18, pages 187–200. ACM, 1990.

[110] Kathleen Knobe, Joan D Lukas, and Guy L Steele Jr. Data optimization: Allocation of arrays to reduce communication on simd machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, 1990.

[111] Michael Philippsen. *Automatic alignment of array data and processes to reduce communication time on DMPPs*, volume 30. ACM, 1995.

[112] Igor Z Milosavljevic and Marwan A Jabri. Automatic array alignment in parallel matlab scripts. In *Parallel Processing, 1999. 13th International and 10th Symposium on Parallel and Distributed Processing, 1999. 1999 IPPS/SPDP. Proceedings*, pages 285–289. IEEE, 1999.

[113] J Ramanujam and P Sadayappan. Compile-time techniques for data distribution in distributed memory machines. *Parallel and Distributed Systems, IEEE Transactions on*, 2(4):472–482, 1991.

[114] J Ramanujam and P Sadayappan. A methodology for parallelizing programs for multicomputers and complex memory multiprocessors. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 637–646. ACM, 1989.

[115] David Bau, Induprakas Kodukula, Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. Solving alignment using elementary linear algebra. In *Languages and Compilers for Parallel Computing*, pages 46–60. Springer, 1995.

[116] ERIKH D'HOLLANDER. Partitioning and labeling of index sets in do loops with constant dependence vectors. In *1989 International Conference on Parallel Processing, University Park, PA*, 1989.

[117] Chua-Huang Huang and Ponnuswamy Sadayappan. Communication-free hyperplane partitioning of nested loops. *Journal of Parallel and Distributed Computing*, 19(2):90–102, 1993.

[118] Y-J Ju and H Dietz. Reduction of cache coherence overhead by compiler data layout and loop transformation. In *Languages and Compilers for Parallel Computing*, pages 344–358. Springer, 1992.

[119] Qingda Lu, Christophe Alias, Uday Bondhugula, Thomas Henretty, Sriram Krishnamoorthy, Jagannathan Ramanujam, Atanas Rountev, Ponnuswamy Sadayappan, Yongjian Chen, Haibo Lin, et al. Data layout transformation for enhancing data locality on nuca chip multiprocessors. In *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*, pages 348–357. IEEE, 2009.

[120] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S Yu. A comprehensive survey on graph neural networks. *arXiv preprint arXiv:1901.00596*, 2019.

[121] Sara Sabour, Nicholas Frosst, and Geoffrey E. Hinton. Dynamic routing between capsules. *CoRR*, abs/1710.09829, 2017.

[122] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[123] Marinka Zitnik, Monica Agrawal, and Jure Leskovec. Modeling polypharmacy side effects with graph convolutional networks. *Bioinformatics*, 34(13):i457–i466, 2018.

[124] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. *arXiv preprint arXiv:1703.06103*, 2017.

[125] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, pages 1024–1034, 2017.

[126] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

[127] 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 *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 443–458, 2019.

[128] Alibaba. Euler. `https://github.com/alibaba/euler`, 2019.

[129] Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference.* Elsevier, 2014.

[130] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1-7):107 – 117, 1998. Proceedings of the Seventh International World Wide Web Conference.

[131] 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 *SIGMOD '10: Proceedings of the 2010 international conference on Management of data.*

[132] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[133] 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 *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[134] 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)*, 2016.

[135] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T Riffel, et al. Gunrock: Gpu graph analytics. *ACM Transactions on Parallel Computing (TOPC)*, 4(1):3, 2017.

[136] Samuel R Bowman, Jon Gauthier, Abhinav Rastogi, Raghav Gupta, Christopher D Manning, and Christopher Potts. A fast unified model for parsing and sentence understanding. *arXiv preprint arXiv:1603.06021*, 2016.

[137] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR)*, 2017.

[138] David K Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P Adams. Convolutional networks on graphs for learning molecular fingerprints. In *Advances in neural information processing systems*, pages 2224–2232, 2015.

[139] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. Aligraph: A comprehensive graph neural network platform. *arXiv preprint arXiv:1902.08730*, 2019.

[140] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

[141] Zhitao Ying, Jiaxuan You, Christopher Morris, Xiang Ren, Will Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. In *Advances in Neural Information Processing Systems*, pages 4805–4815, 2018.

[142] Wengong Jin, Regina Barzilay, and Tommi Jaakkola. Junction tree variational autoencoder for molecular graph generation. *arXiv preprint arXiv:1802.04364*, 2018.

[143] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710. ACM, 2014.

[144] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864. ACM, 2016.

[145] Jie Chen, Tengfei Ma, and Cao Xiao. Fastgcn: fast learning with graph convolutional networks via importance sampling. *arXiv preprint arXiv:1801.10247*, 2018.

[146] Jianfei Chen, Jun Zhu, and Le Song. Stochastic training of graph convolutional networks with variance reduction. *arXiv preprint arXiv:1710.10568*, 2017.

[147] Zhilin Yang, Bhuwan Dhingra, Kaiming He, William W Cohen, Ruslan Salakhutdinov, Yann LeCun, et al. Glomo: Unsupervisedly learned relational graphs as transferable representations. *arXiv preprint arXiv:1806.05662*, 2018.

[148] Yujia Li, Oriol Vinyals, Chris Dyer, Razvan Pascanu, and Peter Battaglia. Learning deep generative models of graphs. *arXiv preprint arXiv:1803.03324*, 2018.

[149] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[150] Rodrigo Nogueira and Kyunghyun Cho. End-to-end goal-driven web navigation. In *Advances in Neural Information Processing Systems*, pages 1903–1911, 2016.

[151] Rajarshi Das, Shehzaad Dhuliawala, Manzil Zaheer, Luke Vilnis, Ishan Durugkar, Akshay Krishnamurthy, Alex Smola, and Andrew McCallum. Go for a walk and arrive at the answer: Reasoning over paths in knowledge bases using reinforcement learning. *arXiv preprint arXiv:1711.05851*, 2017.

[152] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642, 2013.

[153] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):77, 2017.

[154] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Transactions on Parallel Computing (TOPC)*, 5(3):13, 2019.

[155] Adam Lerer, Ledell Wu, Jiajun Shen, Timothee Lacroix, Luca Wehrstedt, Abhijit Bose, and Alex Peysakhovich. Pytorch-biggraph: A large-scale graph embedding system. *arXiv preprint arXiv:1903.12287*, 2019.