The Dissertation Committee for Aashaka Shah
certifies that this is the approved version of the following dissertation:

**Optimizing ML Systems without using experts**

**Committee**:

Vijay Chidambaram, Supervisor

Philipp Krähenbühl

James Bornholt

Madan Musuvathi

# Optimizing ML Systems without using experts

by

**Aashaka Shah**

**Dissertation**

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

**Doctor of Philosophy**

**The University of Texas at Austin**

**August 2023**

# Dedication

*To my family, thank you for your love and support.*

# Acknowledgments

First and foremost, I would like to thank my Ph.D. advisor Vijay Chidambaram. I was a scared first-year Ph.D. student straight out of undergrad with little research experience when I joined UT. Under Vijay's guidance and mentorship, I can say that now I am a more confident early-career researcher with a stronger handle on how to conduct research. ML Systems was as much of a new research area for Vijay as it was for me. I am truly grateful to him for being open to exploring the field and learning it along with me. His ability to identify impeccable insights and ask the correct pain-point questions in a completely different field of research put me in awe every time. Vijay always stressed the importance of communicating your work well, and taking a step back and understanding the high-level picture. Some of the best opportunities I have received till now for my academic career have only been possible because of following his advice. The COVID-19 pandemic hit right in the middle of my Ph.D. Vijay was the kindest, most supportive, and most empathetic advisor during this time. Thank you for believing in me.

I would also like to deeply thank Philipp Krähenbühl for his invaluable mentorship and collaboration throughout the majority of my Ph.D. Working with Philipp was extremely enjoyable and I always left meetings feeling more motivated than before. His advice on making presentations and writing research papers has greatly helped me refine the style of presenting my work. He also taught me the importance of believing in one's own work, which is a lesson I intend to uphold continually. I am grateful to have had the opportunity to collaborate with him.

I would also like to thank the other members of my committee James Bornholt and Madan Musuvathi for their insightful comments and feedback which has helped make this dissertation stronger.

The second part of this dissertation was work that was started during an internship at Microsoft Research. I would like to thank all my collaborators at MSR

Redmond - Saeed Maleki, Todd Mytkowicz, Madan Musuvathi, Olli Saarikivi, Rachee Singh, Meghan Cowan, and Jacob Nelson - for giving me the opportunity to work on this topic and for their guidance during and after the internship. Specifically, I would like to thank Saeed for all his help in running experiments, even if it often meant going out of his way, as well as for his constant enthusiasm for the project. I would also like to thank Todd for motivating me to go one step forward when I felt that I had hit a dead-end. I am profoundly thankful to Madan for being a steadfast champion of my work during the internship and when I was on the job market. I would also like to thank Rachee for her help in presenting TACCL to the broader audience. I would like to thank Olli for the interesting discussions on the encodings, Meghan for drawing an analogy of our work to program sketching, and Jacob for helping me understand some of the nuances of networking.

I also did an internship in Meta AI and would like to thank my collaborators there - Min Si, Pavan Balaji, Ching-Hsiang Chu, Xing Liu, and Ke Wen for all the interesting technical discussions. I would especially like to thank Min for providing amazing career advice and being as much of a mentor as a manager during the internship and even afterward.

My interest in systems research started with an internship with Subrata Mitra in Adobe Research India during my undergrad. I am extremely thankful to him for introducing me to this field. I would also like to thank my other collaborators - Chao-Yuan Wu, Melissa Wasserman, Vinay Banakar, Supreeth Shastri, Jayashree Mohan, and Soujanya Ponnapalli, and my MSR India internship manager, Satya Lokam. I learned a lot from you all. I would also like to thank the Graduate Student Committee Representative, Greg Plaxton, for always coming through on signing forms before deadlines.

I am extremely grateful to have had amazing labmates in the UTSaSLab. Rohan and Soujanya, thank you for often giving me company close to deadlines and helping me improve my work. I will miss our conversations of providing constructive critical feedback for work and talking about everything under the sun. Rohan, thanks for always making situations feel so much easier than how they felt in my head. Soujanya, thanks for laughing the hardest at my jokes, no matter how poor they were. You were the best roommate I could have ever hoped for. Sekwon, thanks for helping me boot up servers when I was working remotely and for your company in the lab when no one else was around. Jayashree, you are an inspiration for how organized someone can be. Shagha, Supreeth, Aastha, Hayley, and Yeonju, I will miss all the interesting conversations we had when hanging out.

I am also lucky to have had an amazing set of friends in UT - Devvrit, Ishan, Nilesh, Kevin, Prabal, Simran, Rishabh, Ruchika, Pragnesh, Shray, Rishabh Khincha, Yingchen, Mona, Gaurav, and so many others - who made life in Austin exciting, and also friends outside of Austin, especially Tanvi and Janhavi, who have been my constant cheerleaders.

I would like to thank my best friend, Jay Desai, for his unwavering support during the ups and downs of my Ph.D. journey. This would not have been possible without you.

Finally, I would like to thank my family. My grandparents - Dada, Dadi, Nana, and Nani - and aunts and uncles - Fiya, Fua, Mama, and Mami - took great pride in my achievements and inspired me to give my best to my work. Dada's and Nana's work ethic is something that I will always aim to emulate. I would also like to thank my cousins, Tanishka, Achal, and especially, Kushal, who were always available to hang out and gave great advice about work and networking. I would also like to thank my younger brother, Dhruvil, who made the final month before my defense so much easier with his antics and camaraderie. Most importantly, I would like to thank my parents, Mittal and Dhaval, for their constant love and support. This dissertation is dedicated to you.

## Abstract

## Optimizing ML Systems without using experts

Aashaka Shah, PhD
The University of Texas at Austin, 2023

SUPERVISOR: Vijay Chidambaram

The growth of large deep learning networks to billions and trillions of parameters has enabled them to achieve state-of-the-art results in various fields, including vision, language, speech, and game-playing. This success of deep networks has also impacted the field of databases in an interesting way - to improve performance, database indexes are now being redesigned as learned models that fit the underlying data. Both deep networks and learned indexes have high resource usage and strict throughput requirements. Minor inefficiencies in resource utilization within these machine learning (ML) systems can incur heavy costs, making it important to optimize their resource efficiency.

What makes doing this difficult is that the execution environment of ML systems is highly heterogeneous. A deep neural network is made of operators with disparate resource utilization profiles connected in different ways. It can also be executed on different types of hardware accelerators, each with distinct performance characteristics. Further, even the input workload to learned indexes can vary. For every new neural network architecture, hardware accelerator topology, or index structure workload, either an expert would be required to hand-craft solutions for efficient resource utilization from a large search space, or we would need to be satisfied with a generic solution that might leave performance on the table.

In this dissertation, we ask the question - *Is it possible to build solutions to optimize ML systems such that they perform instance-specific optimization under the hood and can be utilized by non-experts?*. We demonstrate how we can build tools to optimize the execution of deep networks and learned indexes for different use cases while minimizing manual effort. In the first part of this dissertation, we present MONeT, an automated framework that jointly optimizes different memory-saving techniques for any deep network architecture. Using MONeT, model training on a single GPU always takes less memory than a user-provided memory budget while using less compute than standalone memory-saving techniques. In the second part of this dissertation, we present TACCL, a semi-automated tool that generates efficient communication algorithms based on the hardware topology and size of data to transfer in distributed deep learning. Using TACCL, network utilization can be improved which makes distributed ML execution faster. In the third and final part of this dissertation, we present MaPLE, a parameterized learned index that can achieve high performance on a wide variety of workload patterns while maintaining a similar memory footprint as another state-of-the-art learned index.

The solutions we propose in this dissertation search from a large state space to give performant solutions for the particular use-case that match or outperform previous state-of-the-art but do not need manual tuning from an expert.

# Table of Contents

# List of Figures

# List of Tables

# Chapter 1: Introduction

Deep learning is a sub-field of machine learning that aims to learn representations of data by training artificial neural networks with multiple layers, also known as deep neural networks. The field of deep learning has advanced rapidly in the past few years, achieving state-of-the-art results on popular benchmarks in computer vision [70, 107, 51], natural language processing [34, 123, 16], and game-playing [106, 100]. Deep networks are also being used to learn insights from data in various other fields, such as computational physics, biomedicine, healthcare, auto-industry, and finance. Recently introduced conversational chatbot applications like ChatGPT [22] and Bard [12] are built over massive large language models and have taken the power of deep networks to the general population. Deep learning has thus become a powerful tool to further scientific progress, fuel industrial innovation, and improve individual productivity.

This success of deep networks has largely been driven by the trend of exponentially increasing scale of deep learning infrastructure. Massive deep networks with billions to trillions of parameters have replaced earlier deep networks that had millions of parameters. Simultaneously, the dataset sizes used to train these large networks have also grown. Deep learning is inherently compute-intensive and requires making use of expensive hardware accelerators such as GPUs, TPUs, and FPGAs. With increasing model sizes, accelerator memory becomes a bottleneck and deep networks need to be distributed across multiple accelerators. Apart from adding to the monetary cost, this also introduces network communication bottlenecks between accelerators that need to work in tandem to execute a distributed model.

Researchers and enterprises have employed fleets of expensive hardware accelerators in order to obtain a highly accurate model. Even with this compute power, training could take weeks to months to finish. In industries where deep learning models are extensively utilized, query volumes over the trained networks may range from thousands to millions or even billions of requests per minute. Inference of large networks also needs to take place on hardware accelerators in order to take advantage of parallel compute and meet low-latency expectations. Thus, deep learning is extremely resource-intensive and bottlenecks like memory and network communication need to be optimized for efficient resource utilization.

On the other hand, the landscape for deep learning is incredibly heterogeneous. Neural networks vary in the kinds of operators they use and how operators are connected with each other. Neural networks serving different modalities, such as images, video, or language, use different kinds of operators which have different resource utilization characteristics. Newer hardware accelerators and network interconnect technologies are becoming available every few years. When added to data centers that have previous generations of hardware, it introduces a new layer of heterogeneity in resource availability. Multiple techniques have been proposed over the years for reducing memory usage for deep learning. There also exist libraries like NCCL that are built to provide efficient implementations of communication primitives for distributed deep learning. However, with the broad spectrum of execution scenarios available, it is difficult to identify which technique or implementation would best serve a particular case.

The success of deep networks has also impacted the field of databases in an interesting way. An important aspect of database design is choosing a database index. Database indexes play a major role in improving the speed of retrieving data from a database and it is important to design them carefully. Traditionally, database administrators have chosen general-purpose data structures like B+ Trees [14], LSM Trees [89], and Hash Tables to store data in the form of key-value pairs. With the advent of deep learning, database indexes are being seen in a new light - as models that can use traditional ML techniques to learn the cumulative distribution function (CDF) of stored data. Different learned indexes [31, 69, 36] have successfully outperformed different traditional indexes like B+ trees and LSM-Trees for various workloads.

While learned indexes can fit the distribution of the dataset, their performance on different workloads is still constrained by their underlying data structure. For example, for write-only workloads, learned indexes whose underlying data structures store keys in a sorted manner will always underperform indexes with an append-only data structure. Meanwhile, new services and applications are coming up faster than ever and are expected to have different workload characteristics. Further, the workload characteristics of existing applications may also change with time. For every workload change, the index structure to be used will need to be either redesigned or reconfigured by experts or will suffer from average-case or even poor performance.

We have discussed that there are a wide variety of neural architectures and hardware topologies possible for deep learning workloads, which may require different memory optimization techniques and network communication algorithms. We also saw that database workloads can have different query workloads, which would require differently configured index structures. The vast range of decision options presents two distinct scenarios: achieving high performance by tailoring configurations to the specific problem with expert guidance, or settling for average-case performance and potentially encountering subpar results when employing a generalized solution. In this dissertation, we seek to answer the question: *is it possible to optimize ML systems without using an expert?* We first look into optimizing the memory usage and network communication overhead of deep learning, and finally, we look into optimizing the index structure configuration for learned indexes.

## 1.1 Memory Consumption in Deep Learning

Training deep networks is resource-intensive. In particular, the amount of GPU memory bottlenecks training many deep networks [38, 64, 23, 27]. While the tensor computation in top-of-the-line GPUs increased by 32× over the last five years, the total available memory only grew by 2.5×. This bottleneck requires either modifying the network architecture or scaling training to multiple nodes, incurring significant overheads. Different techniques exist to save memory but identifying which techniques to use requires significant experimentation and expertise.

We present **MONeT** (Memory Optimized Network Training) [103], a framework to automatically minimize the memory footprint for deep networks. MONeT *jointly* optimizes global compute-graph-level techniques (such as checkpointing) and local techniques (such as memory-efficient implementations of individual operator). At the heart of MONeT is a theoretical analysis that enables joint optimization and provides tight bounds on memory consumption. We analyze the memory consumption and computational cost of a general forward and backward pass under changing local operator implementations and a global checkpointing schedule. Specifically, we are able to tightly bound the peak memory consumption for network forward, backward, and re-computation stages. MONeT uses these constraints to optimize for the most efficient forward and backward implementation both locally and globally under a fixed memory budget. We linearize all memory bounds and express both implementation selection and checkpointing as a 0-1 integer program, which we solve using standard solvers.

We conduct extensive experiments, demonstrating that MONeT significantly outperforms existing automatic frameworks that use local or global techniques. On multiple architectures (ResNet [51], VGG [107], UNet [96], GoogleNet [114], MobileNet-V2 [98]), memory budgets (5 - 10 GB), and network configurations (multiple resolutions), MONeT consistently achieves lower memory footprints at equivalent or lower computational overhead. MONeT reduces the overall memory requirement by $3\times$ for various models, with a 9 - 16% overhead in computation. For the same computation cost, MONeT requires 1.2 - $1.8\times$ less memory than the current state-of-the-art automated checkpointing framework. The results achieved by MONeT demonstrate the power of jointly optimizing global checkpointing schedules and local operator implementations.

**Our contributions**

- We develop MONeT, a framework that bounds memory usage during training to a user-provided cap.
- We analyze how memory is used in the forward and backward passes of deep network training for checkpointing and different memory-saving techniques.
- We introduce the idea that operator optimization can work hand-in-hand with checkpointing and develop an encoding to jointly optimize both these techniques for training.

## 1.2 Network Communication in Distributed ML

Deep learning models can get too large for the resources of a single GPU and have to be distributed across multiple servers, each with several GPUs, using different parallelism strategies [105, 73, 45] for training and inference. Intermediate data and parameters of the model at each GPU need to be accumulated, shuffled, and transferred over the network between other GPUs for distributed machine learning, and result in network communication. Recent work has shown that GPU idle time spent waiting for network communication can be significant in practice [99, 71, 48, 75]. Thus, efficient communication between GPUs is the key to enabling fast distributed ML training and inference.

Modern GPU systems use message passing interface (MPI)-based *collective communication primitives*, such as ALLREDUCE, ALLGATHER, and ALLTOALL to as abstractions for inter-GPU communication. Inefficiencies in implementing collective communication primitive can cause poor network utilization, causing GPUs to remain idle until inter-GPU transfers complete [124] and reducing the overall efficiency of distributed training and inference. Newer generations of faster GPUs will only make this problem worse.

We introduce **TACCL** [104] (Topology Aware Collective Communication Library), a collective communication library that synthesizes efficient communication algorithms for a given topology and a target collective communication primitive. We encode the problem of finding optimal algorithms for communication collectives into an integer linear program (ILP) with the goal of minimizing the overall execution time. Unfortunately, this problem is NP-hard; state-of-the-art commercial solvers like Gurobi [50] can spend several days exploring the search space without finding an optimal algorithm. TACCL introduces a new abstraction called *communication sketches* that incorporate high-level intuitive inputs provided by an algorithm designer in order to constrain the search space of algorithms. TACCL also uses a novel *integer linear programming* (ILP) encoding of the collective algorithm synthesis problem that improves scalability by first solving a bandwidth-relaxed version of the problem to decide on *routing*, followed by ordering heuristics and a second bandwidth-constrained problem to find a valid *scheduling* of data transfers in the collective. In addition to significantly improving scalability, TACCL's ILP formulation allows modeling heterogeneous links with different per-message overhead characteristics.

We use TACCL to synthesize efficient algorithms for a range of collectives like Allgather, Alltoall, and Allreduce, and for different hardware backends like Azure NDv2 [10] and Nvidia DGX-2 [83]. We compare TACCL to the state-of-the-art Nvidia Collective Communication Library (NCCL). TACCL synthesized an Allgather algorithm for two Nvidia DGX-2 nodes (32 GPUs). This algorithm is up to 6.7× faster than NCCL for small-to-moderate input sizes. For large input sizes on the same hardware, TACCL synthesized a different Allgather algorithm that nearly saturates the inter-node bandwidth and is up to 25% faster than NCCL. TACCL synthesized an Alltoall algorithm for two Azure NDv2 nodes (16 GPUs) that are up to 66% faster than NCCL. Finally, we replaced NCCL with TACCL using only a two-line code change in PyTorch and found that TACCL achieves a speed-up of 17% in end-to-end training of a mixture-of-experts model that uses Alltoall and Allreduce, and a speed-up of 11% - 2× in end-to-end training of a Transformer-XL model distributed over 16 GPUs for varying batch sizes. TACCL's codebase is open-source and is actively in use by researchers at universities and practitioners at

Microsoft for Azure's GPU virtual machines.

**Our contributions**

- We develop TACCL, a system that synthesizes communication algorithms for a given topology and a target collective communication primitive.
- We introduce communication sketches, an abstraction for user inputs to guide TACCL into synthesizing efficient algorithms for a large range of hardware topologies.
- We develop a novel stage-wise encoding of the problem in TACCL's synthesizer to scale beyond single-node topologies.

## 1.3 Data Structure Configurations in Learned Indexes

Different traditional index structures have different performance characteristics - Hash Tables are used to provide average-case constant-time point lookup and insert performance. B+Tree indexes are tree-based key-value stores that provide high range-query performance while also providing good read and insert performance. LSM-tree is another key-value store design that uses append-only logs for writes and performs periodic compaction. It has a great write performance but suffers from poor read performance.

Similar to traditional indexes, the data structure of learned indexes also needs to be determined by database administrators according to the expected workload in order to achieve high performance. However, workloads tend to change as new services are introduced and old services are updated. Further, workloads may periodically change based on certain diurnal patterns.

We introduce **MAPLE**, a parameterized learned index that can achieve high performance for different types of workloads. The core idea behind MAPLE is that carefully selecting and parameterizing a handful of data structures as index components can help achieve a wide range of spectrum on the read-write performance curve while maintaining similar memory footprints. MAPLE uses two parameterized learned data structures, gapped-array and fragmented-log, as components in building the index. MAPLE also includes a neural network that is used to model the throughput of the index obtained with the different parameters of the learned index as features. Once a partial workload trace is seen, the throughput model is used to select an appropriate configuration for the MAPLE index. In this way, MAPLE's index structure can adapt to a variety of different workloads.

We evaluate MAPLE against an existing state-of-the-art updatable learned index ALEX [36] for a range of workloads. We perform our evaluations using datasets obtained from the SOSD [66] benchmark for learned indexes, which include a dataset from OpenStreetMap [88] (osm). For read-only workloads, MAPLE performs similarly to ALEX using a parameterized gapped-array, whereas, for write-only workloads, MAPLE performs up to 7.5× faster when using a fragmented-log. For an osm workload with 15% read-to-write ratio and using configurations identified by our algorithm, MAPLE is 2.1× faster than ALEX while using about 10% higher memory than ALEX.

**Our contributions**

- We develop MAPLE, a workload adaptable learned index that can obtain high performance for a variety of different workloads.
- We discuss the challenges in building a parameterized learned index.
- We identify and model the parameters of data structures that can be tuned to provide high performance for various kinds of workloads, which we use as components in building MAPLE.
- We design an algorithm to determine the configuration parameters for MAPLE by using a machine learning model to predict the index throughput.

## 1.4 Outline

The rest of this dissertation is organized as follows. Chapter 2 provides the background concepts needed for this dissertation, specifically concepts regarding deep learning training, distributed deep learning, and index structures. Chapter 3 discusses memory usage and network communication in deep learning and workload variability in databases, thus motivating the problems solved in this dissertation. Chapter 4 introduces MONeT, a framework for memory-optimized deep network training. Chapter 5 introduces TACCL, a topology-aware collective communication library. Chapter 6 introduces MAPLE, a parameterized learned index. Chapter 7 discusses prior work related to the systems introduced in this dissertation. Chapter 8 describes some avenues to extend the work in this dissertation. Finally, Chapter 9 summarizes the dissertation, highlights lessons learned, and presents concluding remarks.

# Chapter 2: Background

In this chapter, we provide the background required for various aspects of this dissertation. First, we describe deep networks and how they are trained using forward and backward passes as well as the arithmetic involved in some common deep network operators (§ 2.1, § 2.2). Next, we discuss distributed deep learning and the characteristics of network communication involved (§ 2.3, § 2.4). Finally, we discuss the properties of different index structures and the fundamental insights behind learned indexes (§ 2.5, § 2.6).

## 2.1 Deep networks and deep learning training

The goal of deep learning is to fit a non-linear function, called a deep network, onto a dataset. The deep network consists of a set of learned parameters and a directed acyclic graph (DAG) of arithmetic operators. In a trained deep network, the DAG operates on the inputs and the parameters of the network to generate an output that matches the expected or target output.

Training a deep network is an iterative process. The training dataset is divided into mini-batches of equal sizes (denoted as batch size). In each training iteration, a mini-batch is randomly chosen without replacement from the remaining training dataset and used to train the network. When the complete training set is used up in the iterations, we say that training has finished one epoch. Training continues until a fixed number of epochs have been reached or until the network has reached some accuracy. Since each iteration of deep network training performs the same DAG operations on the same input size, the performance profile across training iterations remains the same.

An iteration in deep learning training takes place as follows. First, we initialize the deep network parameters with random values. We run a forward pass over the

**Algorithm 1:** Forward Pass

```
   Input   : Inputs, $\theta$.
   Output: Output tensor
1  $D = \{\}$;                          /* Saved tensors for backward */
2  $L = \{\text{inputs}, \theta\}$;                 /* Local tensors for forward */

3  for $i = 1 \ldots N$ do
4      $x_i = \text{forward}_i(L)$;

5      Add $x_i$ to $L$;
6      Remove all tensors from $L$ that are not used later;

7      if $i \in D$ then
           ;                  /* Forward dependencies for backward */
8          Add $x_i$ to $\mathcal{D}$;

9  return $L$;
```

Figure 2.1: **Schematic overview of the forward pass..** The algorithm runs the forward pass DAG one operation at a time.

network to obtain the network output. We compare it with the target output to generate a loss value calculated using different metrics such as Mean-Squared Error and Cross Entropy Loss. We then run a backward pass in reverse, also called the gradient back-propagation step, in order to calculate the gradients of the loss with respect to the learned parameters. These gradients are then used to update the parameters using different optimization strategies like Adam [65] and Stochastic Gradient Descent [97]. We formalize the steps taken during the forward and backward pass below:

**The Forward Pass.** Alg. 1 shows a general overview of the forward pass in a deep network, as implemented in standard deep learning frameworks [62, 30, 90, 7]. The algorithm proceeds in increasing order of index $i$. Each operator $\text{forward}_i(\cdot)$ depends on a set of tensors $L$ stored in local memory. These tensors include model parameters $\Theta$, computational dependencies of the operator $\mathcal{N}_i$, and tensors stored

```
Algorithm 2: Backward Pass
  Input  : Loss gradients, inputs, $\theta$, $\mathcal{D}$.
  Output: Output tensor
1 $\hat{L}$ = {loss gradients};                /* Local backward tensors */
2 $L = \mathcal{D}$;                            /* Local forward tensors */
3 for $k = N \ldots 1$ do
4     $y_k = \text{backward}_k(\hat{L}, L)$;
5     Add $y_k$ to $\hat{L}$;
6     Remove tensors from $L$ that are not used later;
7     Remove tensors from $\hat{L}$ that are not used later;
```

Figure 2.2: **Schematic overview of the backward pass..** The algorithm runs the backward pass DAG one operation at a time. Backward pass operators have computational dependencies on the operator outputs of the forward pass.

for later forward operators, i.e. skip or residual activations [51]. The output tensor, also called output activation, obtained on execution of each operator forward$_i$ is also added to the local memory $L$. Tensors present in local memory $L$ that are no longer used in later computations can be freed. Some output activations $x_i$ are used in the backward pass and have to be saved for later. We represent $D$ to denote all activation indexes required in the backward pass and $\mathcal{D}$ to denote the set of these activations.

**The Backward Pass.** The backward pass proceeds in reverse order, as summarized in Alg. 2. backward$_k(\cdot)$ of each node $k$ depends on a set of gradient tensors $\hat{L}$ and forward pass computational dependecies $\{x_i : x_i \in \mathcal{D}_k\}$. Any gradients required by the current and later backward passes are stored in local memory $\hat{L}$. The backward operation produces a gradient for each input tensor of the original forward operation, which is added to $\hat{L}$ if required for a later backward computation. Tensors not required later are removed from $L$ and $\hat{L}$.

## 2.2 Common deep network operators

Both forward and backward passes of deep networks are DAGs, with the type of operator determining computational dependencies. We describe some commonly used operators and the computational dependencies involved in their forward and backward passes.

**Convolution.** A convolution operator takes as input an N-dimensional tensor and has two learnable parameters - a weight tensor and a bias tensor. A window the size of the weight tensor slides over the input tensor and performs a dot product of the weight tensor and the selected area of the input, which is then added to the bias in order to obtain the output tensor. There can be multiple weight tensors in a convolution operator, each concurrently performing the sliding window dot product.

In the backward pass, the backward convolution operator is actually made of two distinct operations. Both operations take in the output-gradient as input. (Note that, when we say output-gradient, we mean the gradient of the loss with respect to the output tensor.) The first operation also takes as input the weight and bias parameters and generates the input-gradient of convolution, which is propagated back to earlier operators. The second operation takes the forward convolution input and produces a parameter-gradient, that is used to update and learn the weights and bias parameters.

The forward and backward convolution operators are the most compute-intensive operators in deep learning.

**Matmul.** The matmul operator has weights and bias as learnable parameters and is used to implement a fully connected layer in a deep network. It performs a matrix multiplication of the input and the weight parameter following which the bias is added to obtain the output the matmul operation.

In the backward pass, the input-gradient is calculated by matrix multiplication of the output-gradient with the weight parameter, the weight-gradient is obtained by matrix multiplication of output-gradient and matmul input, and the bias-gradient is obtained from a sum of the output-gradient elements.

The matrix multiplication operator is second in the list of compute-intensive operators in deep learning.

**BatchNorm.** A batchnorm operator is applied to normalize the input. It has two learned parameters - weight and bias. During the forward pass, the mean and variance of the input are calculated and stored in memory. They are used to normalize the input, following which it is scaled and shifted using the weight and bias parameters respectively to produce the batchnorm output.

During the backward pass, the backward batchnorm operator takes in the output-gradient, produces the parameter-gradients by using the input, and produces the input-gradients by using the weight parameter, the stored statistics, as well as the inputs.

**ReLU.** The relu operator is used to add differentiable non-linearity to the network and does not have any parameters. It simply takes an input tensor and filters out its non-negative elements to 0 while letting the rest of the elements pass as-is.

In the backward pass, the relu backward operator also filters the output-gradient elements to generate the input-gradient elements. The filter is determined by which elements were allowed through in the forward pass. Thus, the relu backward operator also takes in the relu input to produce the input-gradient.

**MaxPool.** The maxpool operator is used to reduce the size of intermediate input tensors in the deep network DAG. It downsamples the input by taking only the maximum value from a sliding window to obtain the output. This operator does not have any parameters.

In the backward pass, the maxpool operator upsamples the output-gradient by placing each output-gradient element in the index at which the maximum value in the input was present. The other elements in the input tensor receive a gradient of zero since they did not contribute to the maximum value. Note that this means that the maxpool backward operator also needs to use the maxpool input in order to compute the input-gradient.

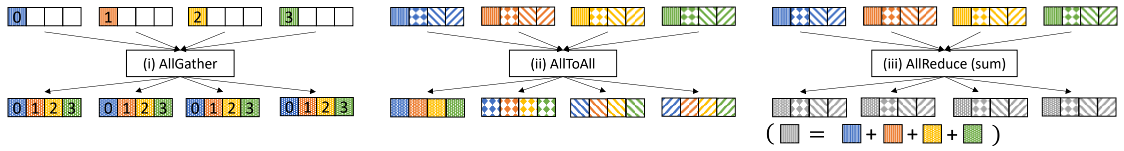


Figure 2.3: The initial and final data buffers on four GPUs participating in different collectives.

## 2.3 Distributed deep learning

As deep networks get larger, they are increasingly being distributed across multiple GPUs and servers using different distribution paradigms. For example, data parallelism involves each GPU running a forward and backward pass of the same model and aggregating their gradients at the end of each training iteration. Model parallelism involves executing different operators on different devices and using peer-to-peer communication to communicate operator outputs. Tensor model parallelism is implemented in transformer models like Megatron [105] in which a single parameter exceeds GPU memory. In this case, the operator is replicated across multiple GPUs and its parameters are sharded into multiple partitions. Each transformer block performs aggregation of activations in the forward pass and aggregation of gradients in the backward pass. Finally, we also have expert model parallelism [73, 45] where every other transformer block shuffles data with all the other experts.

## 2.4 Characteristics of network communication in distributed ML

In all paradigms of distributing deep networks, the GPUs need to accumulate, transfer, or shuffle data between each other. Deep learning frameworks use MPI-style communication collectives [40] as abstractions for the communication patterns required in distributed deep learning.

In this section, we further explain the different types of communication collectives commonly used in distributed deep learning. Multi-GPU ML workloads typically communicate using MPI-style collectives like ALLGATHER, ALLTOALL, and ALLREDUCE. Figure 2.3 shows the initial and final states of running these collectives on a 4-GPU system. In ALLGATHER, every GPU receives the data buffers of all other GPUs (left diagram in Figure 2.3). In ALLTOALL, every GPU receives different parts, or chunks, of the data buffers present on all GPUs. This effectively transposes the data chunk from buffer index to GPU index as can be seen in center diagram in Figure 2.3. In ALLREDUCE, every GPU ends up with a data buffer that has the results of performing a point-wise computation (e.g., sum in right diagram in Figure 2.3) over the same data index of all GPUs. In REDUCESCATTER, every GPU receives a part of the reduced data buffer.

The parallelism strategy for the distributed ML workload determines which collective communication primitive is used. Data parallelism and some tensor model parallelisms [105] make use of the ALLREDUCE collective to aggregate gradients and intermediate data respectively from multiple GPUs. Expert parallelism [73, 45] and common deep learning recommendation models (DLRM) [80] make use of the ALLTOALL collective to shuffle intermediate data between experts and embedding lookup data between GPUs respectively. DLRMs [80] also make use of the ALLGATHER collective and another REDUCESCATTER collective to perform embedding lookups from embedding tables sharded over multiple GPUs.

## 2.5 Data structures in database indexes

Database indexes are used to speed up access to databases. Index structures are generally designed according to the type of workload that is expected in the system. Different index structures can have different performance characteristics. We discuss some commonly used data structures that are used to build indexes and for what workloads they are used.

**B+Tree.** B+Tree indexes are tree-based key-value stores that consist of internal tree nodes that can be traversed to partition the key space. The leaf nodes in the B+Tree contain keys stored in a sorted manner and point to the location where the actual data is stored. B+tree indexes can provide high range-query performance while also providing good read and insert performance.

**LSM-tree.** LSM-tree is another key-value store design that uses append-only logs for writes and performs periodic compactions. It has a great write performance but suffers in read performance.

**Hash tables.** Hash tables are another key-value store that are great for point lookups and inserts but suffer from poor range query performance.

Based on the expected workload, a database administrator has to decide which index structure to use.

## 2.6 Learned Indexes and ALEX

**Insights behind learned indexes.** At its core, a database index maps a key to its position in the data. Using this idea, Kraska et.al. [69] proposed looking at indexes as learned models. The models can learn an approximation of the cumulative distribution function (CDF) $\mathcal{F}$ of sorted input data of size N. When a data item k is queried, its approximate position would be at $n = \mathcal{F}(k) * N$. By maintaining error bounds as $\epsilon$, the data thus needs to be searched only in the range $n - \epsilon$ to $n + \epsilon$, instead of the entire range of data.

Since data is not usually uniform and its CDF may be hard to learn, Kraska et.al. proposed building hierarchical models (called Recursive Model Index, or RMI) in order to reduce the error bounds in fitting the data. The hierarchical models in the RMI can be compared to the internal nodes of a B+Tree, which are traversed to navigate the key partitions. However, unlike in the B+Tree, the RMI does not need to store keys for comparison in the internal nodes. By performing model lookups to traverse the hierarchical models instead of having to compare against keys present in the internal nodes as in a B+Tree, RMI is able to achieve fast lookups. Further, because of not storing keys for comparison in the internal nodes, the RMI has a much lower memory footprint than B+Tree. On reaching the leaf node, a "last-mile" binary search is performed within the node's error bounds.

In order to keep computation cost low, the RMI is often shallow and has high fanout. In their evaluation, Kraska et.al. [69] use a two-level RMI with a large fanout ranging from 10000 to 200000. The root node at the first level has a simple to semi-complex neural network model, with up to 2 hidden layers which are 8- or 16-unit wide. The leaf nodes at the second level have simple linear models. While learned indexes use model-based computation, they are generally implemented on CPUs instead of hardware accelerators like GPUs in order to replace existing index structures without requiring changes in the hardware.

**Insights behind ALEX.** While the RMI allows high-performance lookups using a learned model, it does not support efficient inserts. ALEX [36] is an updatable learned index that uses linear models in the internal nodes and an interesting Gapped Array data structure in the leaf nodes. A Gapped Array node has interspersed gaps in-between sorted keys which allows for faster model-based reads and model-based writes.

Since the last-mile search in the Gapped Array is model-based, ALEX can allow a large leaf node without facing performance penalties. Similar to RMI, ALEX also has a low memory footprint, allowing for a higher fanout of internal nodes. With its high fanout and large leaf nodes, the trees in ALEX are largely flat. All of these factors contribute to the high performance that is provided by learned indexes.

# Chapter 3: Motivation

Based on the background material presented in Chapter 2, we now motivate this dissertation. We saw that deep learning training is done using forward and backward passes of deep networks and that MPI-style communication collectives are used for network communication in distributed machine learning. We also discussed the characteristics of different types of index structures, including learned index structures.

In this chapter, we identify the major contributors to memory usage in deep network training (§ 3.1) and discuss the limitations of some existing memory-saving techniques (§ 3.2). We then discuss the challenges of hardware heterogeneity in distributed ML (§ 3.3) and how commonly used existing communication libraries fall short in dealing with them (§ 3.4). Finally, we discuss how real-world workloads can vary with time (§ 3.5) and discuss the limitations of learned indexes in adapting to these changes (§ 3.6). We thus motivate the need to build tools that can search the state space of different techniques for optimizing memory usage and network communication algorithms in deep learning, and index configurations for learned indexes, while not needing an expert.

## 3.1 Memory requirement in deep network training

In this section, we discuss the contributions to memory usage in deep network training. Deep network parameters are normally kept in memory and contribute to memory usage. Parameter gradients computed during the backward pass also contribute to memory usage. Further, many operators, like convolutions, use temporary workspace memory that contributes to memory usage. Finally, as discussed in § 2.1, forward pass activations are stored as computational dependencies for the backward pass, and contribute to the total memory usage. We obtain the contributions of all

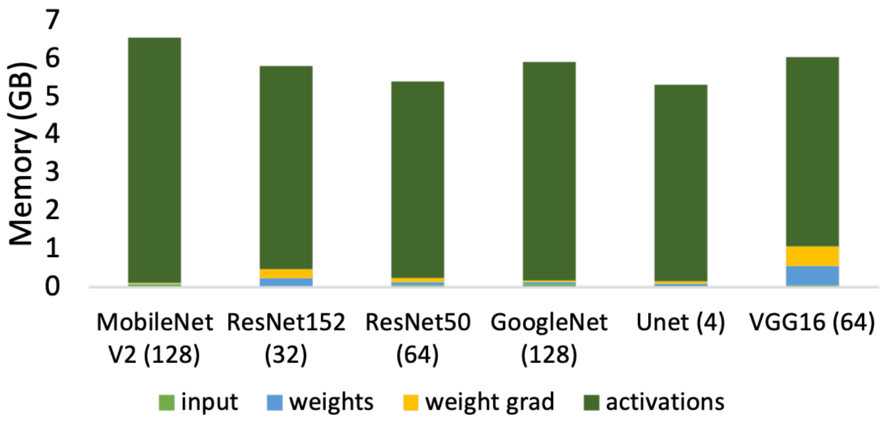


Figure 3.1: Contributions to memory usage in training deep networks.

the previously described components to memory usage when the memory consumption is at its peak. Figure 3.1 plots this data for several deep learning networks like ResNet-50, GoogleNet, VGG-16, etc., run using PyTorch on a 16-GB Nvidia P100 GPU. The batch size is mentioned in parentheses next to their names. This is the maximum possible batch size that can be run on a single GPU. We observe that the forward computational dependencies contribute the most to memory usage during deep network training. Thus, it is important to develop techniques to reduce this contribution.

## 3.2 Existing memory saving techniques

In this section, we explain some existing memory-saving techniques and discuss the tradeoffs that are involved in these techniques. There are two broad families of approaches to reduce the memory footprint of a deep network during training: operator-level implementation changes, and global, graph-level optimizations.

**Operator-Specific Optimizations.** Researchers have found creative ways to implement individual operators or groups of operators in a more memory-efficient manner. Standard deep learning frameworks [62, 30, 90, 7] provide different implementations of certain operators that trade computation for intermediate memory use. For example, the convolution operator can be implemented using multiple different algorithms, each with different compute costs and workspace memory requirements. These implementations are chosen according to local search heuristics and are not globally optimal.

Gist [58] proposes several hand-crafted optimizations such as storing only ReLU signs. RevNets [49] redesigns a ResNet [51] architecture making each network block reversible, thereby eliminating the need to store intermediate activations for back-propagation. Memory-efficient DenseNets [94] reduce memory utilized for feature maps by recomputing all intermediate feature maps during the backward pass with a small compute overhead. In-place activated batchnorm [17] or ReLU layers use output activations to compute their gradients, thus reusing a single memory buffer for the gradient computation in consecutive layers. Although these hand-crafted techniques independently result in memory savings, it is difficult to know which technique should be applied when, and different implementations perform best on different architectures.

**Checkpointing.** [24] proposed dividing a network into different segments, dropping all intermediate outputs within each segment, and recomputing them later. Chen *et al.* use $\sqrt{n}$ equal segments, trading memory savings for the cost of an extra forward pass. However, the computation costs of different operators and the memory requirement of intermediate outputs vary along the DAG, and dropping equal segments may be sub-optimal in terms of performance.

A prior work, Checkmate [59] attempts to solve the memory-usage problem in a more general setting, using a mixed-integer linear program solver to decide which layers to recompute for a given network. However, it fails to take into account local operator optimizations. In Checkmate, changes in operator implementation induce a different computation graph, and could thus not directly be optimized. We later highlight some of the difficulties of adding operator optimizations into Checkmate.

In summary, while much work has been done on local optimizations (operator

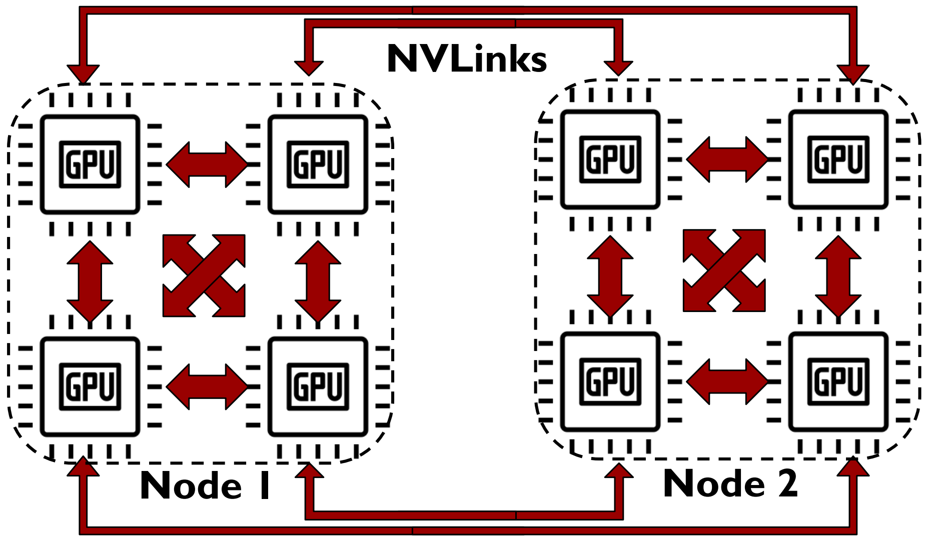


Figure 3.2: NVLink connectivity of a NDv2.

implementations) and global compute-graph-level techniques, these techniques have not been explored together. Further, it is not easy to identify which technique should be used when. *Thus, there is a need to automate choosing different operator-specific and checkpointing-based memory-saving techniques as well as their joint optimization.*

## 3.3 Hardware heterogeneity in distributed deep learning

There exists a wide variety of multi-GPU systems to meet the scaling challenges posed by growing ML models, which result in high hardware heterogeneity. Modern GPU systems, e.g., Azure NDv2 (Figure 3.2) and Nvidia DGX-2 (Figure 3.4), have the different types of interconnects: (1) **Peripheral Component Interconnect Express (PCIe)**, (2) **NVLink** [86], (3) **Infiniband** (IB) NICs [84]. A PCIe bus connects GPUs to CPUs with limited shared bandwidth (PCIe Gen3 offers $\approx 13$ GBps). PCIe connections often form a hierarchy with PCIe switches (Figure 3.3). NVLink [86], however, is a GPU to GPU *intra-node* connection with dedicated bandwidth. NVLinks are either directly connected to other GPUs (Azure NDv2 in Figure 3.2) or they are connected to other GPUs via NVSwitches [87] (Nvidia DGX2 in

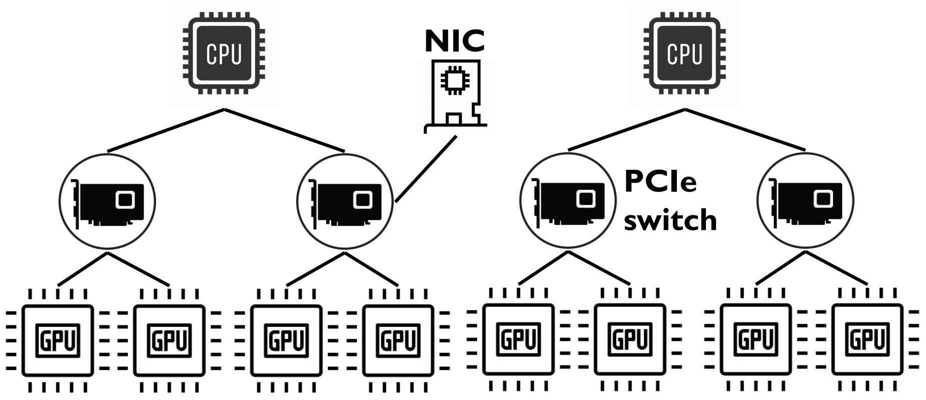


Figure 3.3: PCIe connectivity of a NDv2.

Figure 3.4). NVSwitches enable fully-connected GPU-GPU communication through NVLinks. This introduces heterogeneity in the types of hardware used for deep learning.

Further, even a cluster of machines of the same type is inherently heterogeneous. IB is a multi-node interconnect which allows GPUs to communicate with GPUs in other nodes like in the Azure NDv2 (Figure 3.3). IB NICs are usually connected to PCIe switches and GPUs may communicate directly with the NICs through Remote Direct Memory Access (RDMA) or indirectly via host memory. The IB interconnect is typically much slower than intra-node interconnects. The disparity between the speeds of the intra-node and inter-node interconnects in GPU topologies are a major cause of heterogeneity.

There also exist multiple cases where the physical topology is not fully known or documented. For example, for Azure NDv2 systems the physical topology is not fully documented: while the NVLink topology (Figure 3.2) is known to match that of Nvidia DGX1, we did not know how GPUs and the one 12.5 GBps InfiniBand NIC were connected with PCIe. PCIe peer-to-peer communication (and thus GPUDirect RDMA [4]) is not enabled on these machines, meaning that all communication happen through buffers in CPU memory over potentially shared PCIe links. Further, virtualization obscures the true PCIe topology (all 8 GPUs and the NIC appear directly

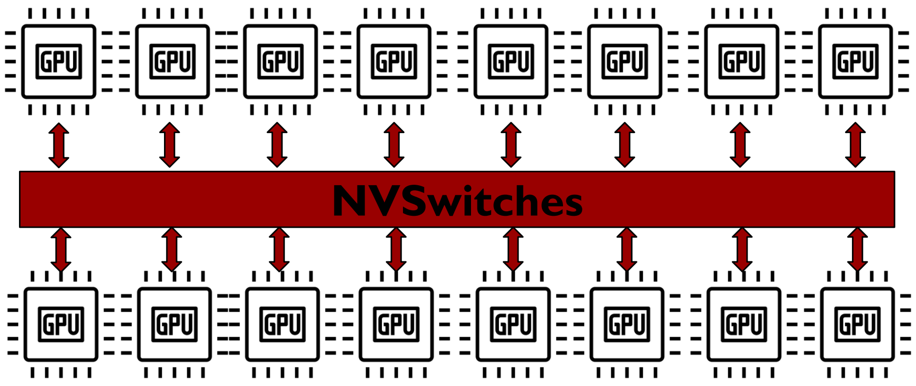


Figure 3.4: NVLink connectivity of a DGX-2.

connected to one CPU) and NUMA node and GPU IDs are not assigned consistently from VM to VM. This means that, without additional information, the software cannot avoid contention over shared PCIe links, creating interference and high variance in performance.

## 3.4 Existing network communication libraries

Collective algorithms must be designed considering the target input sizes and the heterogeneity of the target topology. However, most collective communication libraries used for distributed ML today, including the state-of-the-art NCCL [85], use pre-defined templates of collective algorithms superimposed onto a target topology.

For example, for collectives like ALLGATHER and REDUCESCATTER, NCCL identifies rings in the target topology and uses the Ring algorithm. For $n$ GPUs, this algorithm requires $n-1$ link transfer steps per data chunk and is not ideal for smaller data sizes where link transfer latencies dominate. Further, this algorithm treats the slow inter-node and fast intra-node links similarly, scheduling equal number of data transfers across both. The communication is thus bottlenecked on the slower inter-node links, when it could have benefitted by sending more node-local data (i.e. data of GPUs local to the node) over the faster intra-node links instead. For the Alltoall collective, NCCL implements the collective algorithm as peer-to-peer data transfers between all pairs of GPUs. This algorithm is topology-agnostic and often inefficient. For the Allreduce collective, NCCL chooses between two algorithms — Double-Binary-Tree [82] and Ring. This decision is made according to the communication input size and number of nodes, but might not be most accurate, as it is based on hardcoded latency and bandwidth profiling done previously by Nvidia on their machines.

Designing efficient collective algorithms requires careful analysis of the topology and its performance with different buffer sizes. Recent work [116, 18] has shown that synthesis is a promising approach for generating collective algorithms for different topologies and achieving bandwidth and latency optimality. However, scaling these approaches to multi-node (i.e. multi-machine) distributed GPU topologies has been a challenge. We measured the synthesis time for Allgather and Alltoall collectives on topologies of two Azure NDv2 nodes and two Nvidia DGX2 nodes using SCCL [18, 79]. We modified the codebase to include both topologies and attempted to synthesize the collectives with a 24-hour time limit set for each synthesis query. Given a 24-hour time limit, SCCL's `pareto-optimal` solver strategy did not finish synthesis for any combination of collective and topology. The only algorithm that SCCL could synthesize within the time limit was a latency optimal algorithm for Allgather on two NDv2 nodes.

Thus, while generic libraries like NCCL give up on performance by not being tailored to particular hardware topology and input size, synthesis-based libraries like SCCL time out when generating algorithms for multi-node systems. *Thus there is a need for a topology-aware and input-aware collective algorithms that can scale to multi-node topologies.*

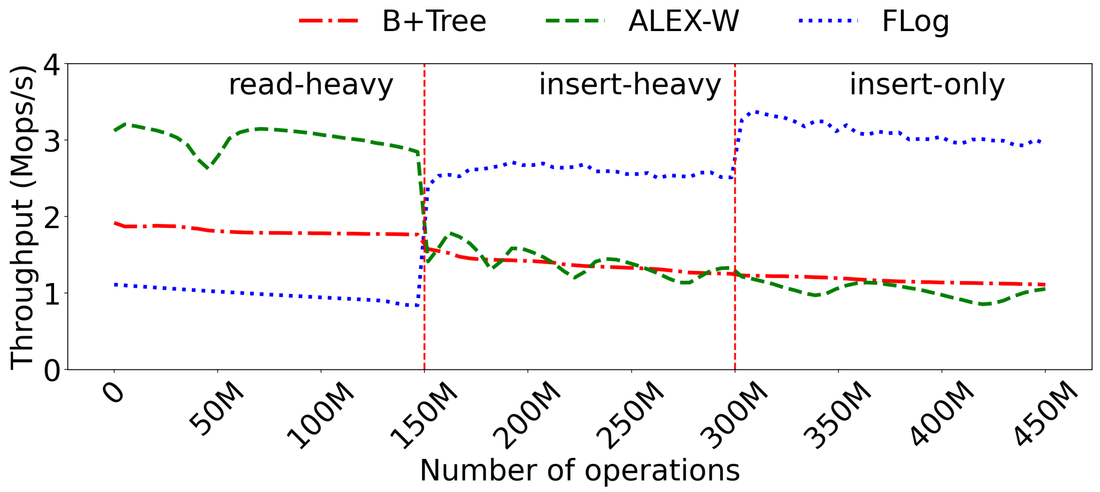


Figure 3.5: Comparing throughput of indexes for varying workload patterns for the OpenStreetMaps CellIDs dataset.

## 3.5 Workload variety in database indexes

Production services generally tend to see a wide variety of workloads ranging along the spectrum of read-only, write-heavy, and write-only workloads. There exist a bunch of workloads coming from different services such as SQL queries, deep learning applications, Apache Spark data analytics, as well as document and media servers. From the IBM Cloud Object Store traces [44], we observe that these different kinds of services have different read/write profiles. Production workloads may also change periodically with time. Some workloads may show a strong diurnal pattern of 24-hours. For example, for serving a social network, the read to write ratio usually reaches a peak of 4:1 at around 5 pm when when people might read content during off-work time, whereas it reduces to 2:1 during working hours [125, 19]. Further, new services and applications keep getting added and the workload characteristics of production databases keep changing over time [39]. This points to a need to be able to adapt to a wide variety of workload patterns over time.

## 3.6 Performance of existing learned index structures

In § 2.5, we discussed that different traditional index structures are designed to provide high performance for different types of workloads. In this section, we discuss the performance obtained by a state-of-the-art learned index, ALEX [36], as workload pattern changes.

ALEX maintains a cost model per leaf node to identify significant cost deviation from its expectations of lookups and inserts. The cost model tracks the average number of searches required for a lookup and the average number of key shifts required for an insert. Based on these, ALEX decides between the structural modifications of expanding or splitting nodes and adapts to changes in workload distribution.

However, as also noted by Wongkham et. al. [120], while ALEX performs better than other updatable learned indexes for most workloads, it is not suitable for write-heavy workloads. The performance achieved by ALEX is fundamentally limited by its underlying data structure that has to store data in a sorted manner. Let us consider a hypothetical example of a trace with varying workload patterns. We consider a real-world dataset `osm`with 400 million keys obtained from the cell ids from Open Street Map. Since synthetic datasets can be trivially fit by learned indexes, `osm`is one of the real-world datasets used in benchmarking learned indexes [77]. We bulk-load ALEX with 50 million keys and run it against a workload with 150 million read-heavy operations, followed by 150 million write-heavy operations, and finally followed by 150 million write-only operations. The read-heavy operations have a 85% read-to-write ratio and the write-heavy operations have a 15% read-to-write ratio. Figure 3.5 shows the throughput obtained by ALEX over time and compares it against a traditional B+Tree index as well as a write-optimized learned index, Fragmented Log (FLog or fragmented-log), that keeps keys unsorted and will be introduced later.

We find that ALEX performs better than B+Tree for read-heavy workloads but its performance falls as the insert fraction increases. We also see that the write-optimized Flog performs better than ALEX for write-only and write-heavy workloads.

We find that even if ALEX expects a write-heavy workload (by setting the expected insert fraction to 1, thus labelled ALEX-W), it cannot outperform an index specifically configured for a write-heavy workload. *Thus, we need to build suitable learned components that can facilitate fast data access according to different expected workloads.*

## 3.7 Summary

We motivated the need for automated tools for optimizing various aspects of ML systems without requiring an expert.

# Chapter 4: MONeT: Memory Optimization for Deep Networks

In Chapter 3, we discussed that deep network training is memory-intensive owing to the computational dependencies of backward pass operators on forward pass activations. We looked at existing memory-saving techniques and saw that they need to be handcrafted according to the network architecture by experts who understand their tradeoffs.

In this chapter, we present MONeT, an automated framework that minimizes memory footprint for deep networks by jointly optimizing global compute-graph-level memory-saving techniques (such as checkpointing) and local techniques (such as memory-efficient implementations of individual operators) [1]. MONeT encodes the peak memory usage of training into a 0-1 integer linear programming problem (ILP) which can then be solved to optimize for the most efficient forward and backward pass implementation under a fixed user-provided memory budget.

First, we discuss the goals of MONeT (§ 4.1). We then provide an overview of MONeT's design (§ 4.2), theoretically analyze memory consumption at each stage of network training (§ 3.1), and describe how MONeT formulates the ILP in order to achieve memory-constrained training at low computational overhead (§ 4.2.2, § 4.3). Finally, we describe MONeT's implementation (§ 4.4) and evaluate it against existing memory-saving techniques (§ 4.6).

## 4.1 Goals

- The forward and backward operator schedules generated by MONeT should always use less memory than the user-provided memory budget.

[1]This Chapter is based on the work, Memory Optimization for Deep Networks, published in ICLR'21 [103]

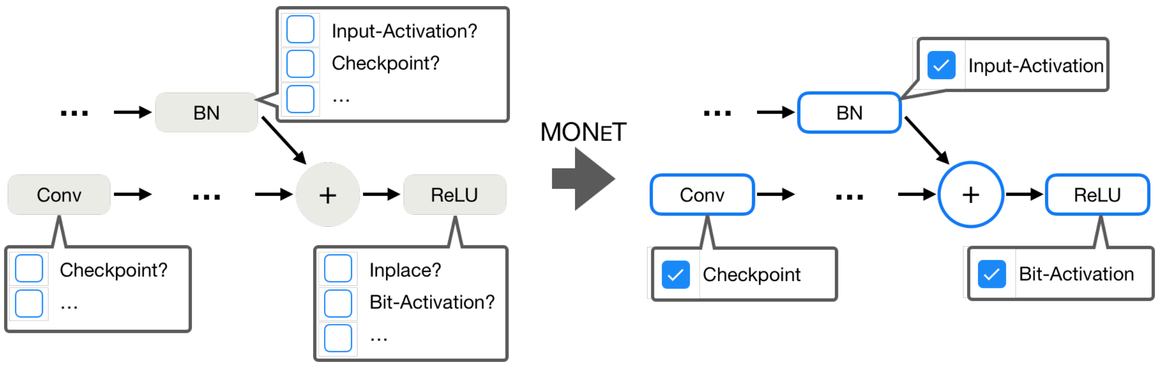


Figure 4.1: **Memory Optimized Network Training (MONeT).** MONeT is an automatic framework that minimizes the memory footprint of deep networks by jointly optimizing global and local techniques.

- The schedules generated by MONeT should introduce as little computational overhead as possible.
- MONeT should be able to generate schedules in much less time than it would take to train the deep network.
- MONeT should be able to execute the schedule it generates on real hardware.

## 4.2 Design

In order to build MONeT, we perform a theoretical analysis to provides tight bounds on memory consumption in the forward and backward pass of deep network training as well as during the recomputation stages of checkpointing. We then formulate the problem as 0-1 integer program with constraints to bound the peak memory consumption at all stages of training and an objective to minimize the computational cost of training. These terms are all expressed as linear functions of implementation selection and checkpointing. MONeT then uses these constraints to optimize for the most efficient forward and backward implementation both locally and globally under a fixed user-provided memory budget. The solution, shown in Figure 4.1, determines the schedule for checkpointing and operator implementations.

| Notation | Meaning |
|---|---|
| **Tensors** | |
| $x_i$ | Tensor which is the output of forward$_i$ in forward pass and during recomputation. |
| $y_k$ | Gradient tensor which is the output of backward$_i$ in the backward pass. |
| **Sets** | |
| $S_i^N$ | Set of stored tensors after forward$_i$ in forward pass. (N = num backward operators) |
| $L_i$ | Set of all parameters and forward tensors created till forward node $i$, required as computational dependencies for forward$_i$ and later forward passes. |
| $D_k$ | Set of forward pass tensors required as computational dependencies for backward$_k$. |
| $S^{k-1}$ | Set of stored forward pass tensors right before calling backward$_k$. |
| $\hat{L_k}$ | Set of gradient tensors created before backward node $k$, and required as computational dependencies for backward$_k$ and later backward passes. |
| $S_i^{k-1}$ | Set of stored tensors available to recomputation of forward$_i$ before computing backward$_k$. |
| $L_i^k$ | Set of all parameters and forward tensors created till forward node $i$, required as computational dependencies for forward$_i$ and later forward recomputations to be done before backward$_k$. |
| $\mathcal{I}_i$ | Set of implementations for operator forward$_i$. |
| $\hat{\mathcal{I}}_k$ | Set of implementations for operator backward$_k$. |
| **Solver variables** | |
| $s_i^N$ | Indicate if output of forward$_i$ is stored in memory in the forward pass. |
| $s_i^{k-1}$ | Indicate if output of forward$_i$ is stored in memory when computing backward$_k$. |
| $r_i^k$ | Indicate if forward$_i$ is recomputed before computing backward$_k$. |
| $\delta_{i,l}$ | Indicate if forward$_i$ uses implementation $l \in \mathcal{I}_i$ in the forward pass. |
| $\delta_{i,l}^k$ | Indicate if forward$_i$ uses implementation $l \in \mathcal{I}_i$ when recomputed before backward$_k$. |
| $\hat{\delta}_{k,l}$ | Indicate if backward$_k$ uses implementation $l \in \hat{\mathcal{I}}_k$. |
| **Memory formulations** | |
| $m_i$ | Peak memory of forward$_i$ in forward pass. |
| $\bar{m}_i^k$ | Peak memory of forward$_i$ when it is recomputed before backward$_k$. |
| $\hat{m}_k$ | Peak memory of backward$_k$. |
| **Operator costs** | |
| $c_i^l$ | Workspace memory of operator forward$_i$ executed using implementation $l \in \mathcal{I}_i$. |
| $\hat{c}_k^l$ | Workspace memory of operator backward$_k$ executed using implementation $l \in \hat{\mathcal{I}}_k$. |
| $\tau_i^l$ | Compute cost of operator forward$_i$ executed using implementation $l \in \mathcal{I}_i$. |
| $\hat{\tau}_k^l$ | Compute cost of operator backward$_k$ executed using implementation $l \in \hat{\mathcal{I}}_k$. |

Table 4.1: **Notations used in paper with explanations.** Notations with only $i$ in subscript/superscript generally relate to the forward pass, with only $k$ relate to the backward pass, and with both $i$ and $k$ relate to the recomputation phase.

### 4.2.1 Theoretical Analysis of Peak Memory Consumption

We now provide a theoretical analysis of peak memory consumption in the forward pass, backward pass, and recomputation stages. Table 4.1 provides a reference to the notations introduced in this section along with their explanations.

As discussed earlier in § 2.1, the forward pass of a deep network with parameters $\Theta$ is expressed as a directed-acyclic graph (DAG), where each node $i \in \{1, \ldots, N\}$ corresponds to an operator $\text{forward}_i$, and edges $(i, j) \in \mathbf{E}$ specify the data-flow dependencies, i.e., , the output of operator $i$ is used as input in operator $j$. Without loss of generality, computational dependency $(i, j) \in \mathbf{E}$ implies $i < j$. Let $\mathcal{N}_j = \{i : (i, j) \in \mathbf{E}\}$ be the set of all incoming edges of an operation $j$.

We will first modify our discussion of the forward pass through a network and the basic form of a backward pass done previously in § 2.1 by adding checkpointing. Checkpointing allows either saving or recomputing forward pass activations depending on a schedule $(s, r)$. The backward pass reverses all computational dependency expressed in our DAG, and induces certain dependencies on forward activations. We call these checkpoint dependencies $\mathcal{D}_k$. Recomputation from checkpoints creates a trade-off between compute cost and memory consumption. To highlight this tradeoff, we formally obtain the amount of memory consumed in the forward, backward, and recomputation phases when a checkpointing plan $(s, r)$ is already provided. This will then allow us to optimize for the ideal execution plan in § 4.2.2.

**The Forward Pass.** Alg. 3 shows a general overview of the forward pass in a deep network with checkpointing enabled. The algorithm proceeds in increasing order of index $i$. Each operator $\text{forward}_i(\cdot)$ depends on a set of tensors $L$ stored in local memory. These tensors include model parameters $\Theta$, computational dependencies $\mathcal{N}_i$, and tensors stored for later forward operators, i.e. skip or residual activations [51]. At each iteration, we add any output tensors of $\text{forward}_i$ to the local memory $L$. Early deep learning frameworks [62, 30] strictly grew the set of local tensors $L$ leading to an unnecessarily high memory consumption. Modern graph-based frameworks [90, 7]

**Algorithm 3:** Forward Pass

```
Input  : Inputs, $\theta$, a schedule $(s, r)$.
Output: Output tensor
$S^N = \{\}$;                          /* Saved tensors for backward */
$L = \{\text{inputs}, \theta\}$;                  /* Local tensors for forward */

for $i = 1 \ldots N$ do
    $x_i = \text{forward}_i(L)$;

    Add $x_i$ to $L$;
    Remove all tensors from $L$ that are not used later;

    if $s_i^N$ then
        Add $x_i$ to $S^N$;

return $L$;
```

**Algorithm 4:** Backward Pass

```
Input  : Loss gradients, inputs, $\theta$, $S^N$, $(s, r)$.
Output: Output tensor
$\hat{L} = \{\text{loss gradients}\}$;                /* Local backward tensors */
for $k = N \ldots 1$ do
    $L = S^k$;                            /* Local forward tensors */
    $S^{k-1} = \{\}$;                              /* Saved tensors */
    for $i = 1 \ldots N$ do
        if $r_i^k$ then
            $x_i = \text{forward}_i(L)$;
            Add $x_i$ to $L$;
        Remove all tensors from $L$ not used later;
        if $s_i^{k-1}$ then
            Add $x_i$ to $S^{k-1}$;                         /* use $x_i \in L$ */
    $y_k = \text{backward}_k(\hat{L}, L)$;
    Add $y_k$ to $\hat{L}$;
    Remove tensors from $\hat{L}$ that are not used later;
```

Figure 4.2: **Schematic overview of the forward and backward passes with checkpointing.** The algorithms include aggressive memory savings by greedily freeing unused tensors, and allow for a general checkpointing schedule $(s, r)$ to be executed.

reduce the memory footprint by aggressively pruning local memory $L$ and freeing any tensor that is no longer used in later computations. Some output activations $x_i$ are used in the backward pass, and have to be saved for later. We use a checkpointing schedule $s^N$ to determine which of the activations will be actually saved. Formally, $s_i^N \in \{0, 1\}$ indicates whether the output activation of node $i$ is stored during the forward pass. An activation that is not stored will be recomputed if it is needed during the backward pass.

**Analyzing peak memory consumption of the forward pass.** Only the forward$_i$ operator (Alg. 3 L. 4) allocates memory. All other operators perform mere bookkeeping on existing tensor. It is thus sufficient to study the peak memory consumption $m_i^N$ in forward$_i$ for each node $i$. Let $L_i, S_i^N$ be the set of local tensors $L$ and saved tensors $S$ while calling forward$_i$ respectively. $L_i$ includes all parameters and computational dependencies for this and later forward passes $L_i = \Theta \cup \{x_j : j \in \mathcal{N}_t \text{ for any } t \geq i \text{ and } j < i\}$. $L_i$ is constant and computed ahead of time. The schedule $s^N$ determines the set of saved tensors $S_i^N = \{x_j : s_j^N = 1 \text{ for } j < i\}$. In addition, each forward operator uses a certain amount of workspace memory $c_i$ to store intermediate results. The total memory consumption of a forward operator is thus

$$m_i = c_i + |x_i| + |S_i^N \cup L_i| = c_i + |x_i| + \sum_{x_j \in L_i} |x_j| + \sum_{j < i : x_j \notin L_i} |x_j| s_j^N, \tag{4.1}$$

where $| \cdot |$ refers to the memory consumed by a tensor or set of tensors. Most of the memory consumption is constant and does not depend on the checkpointing schedule $(s, r)$.

**The Backward Pass.** The backward pass proceeds in a reverse order, as summarized in Alg. 4. backward$_k(\cdot)$ of each node $k$ depends on a set of gradient tensors $\hat{L}$ and forward tensors $\{x_i : i \in \mathcal{D}_k\}$. Any gradients required by the current and later backward passes are stored in local memory $\hat{L}$. Dependencies $\mathcal{D}_k$ may either be stored in $S^k$ or need to be recomputed from checkpoints in $S^k$. Recomputation involves forward computation of one or more nodes, which increases computational overhead, and allows for a new set of tensors $S^{k-1}$ to be saved. After recomputation, all dependencies $\mathcal{D}_k$ are kept in memory. The backward operation produces a gradient for each input tensor of the original forward operation, which is added to $\hat{L}$ if required for a later backward computation. We aggressively remove tensors in $\hat{L}$ that are not required.

**Analyzing the peak memory consumption of the backward pass.** Peak memory consumption $\hat{m}_k$ again only depends on the forward$_i$ (Alg. 4 L. 7) and backward$_k$ (Alg. 4 L. 12) operations. For the backward$_k$ operation, let $\hat{c}_k$ be the workspace memory, $\hat{L}_k$ be the set of gradient tensors stored, $D_k = \{x_i : i \in \mathcal{D}_k\}$ be the forward tensors used, and $S^{k-1}$ be the set of newly saved tensors. Here $\hat{L}_k$ and $D_k$ can be pre-computed. The total memory consumption for the backward$_k$ call is

$$\hat{m}_k = \hat{c}_k + |y_k| + |S^{k-1} \cup \hat{L}_k \cup D_k| = \hat{c}_k + |y_k| + \sum_{y_l \in \hat{L}_k} |y_l| + \sum_{x_i \in D_k} |x_i| + \sum_{x_i \notin D_k} s_i^{k-1} |x_i|. \quad (4.2)$$

Here again, only the last term depends on the checkpointing schedule, while the rest is a constant.

**Analyzing the peak memory consumption of the recomputation.** Finally, the peak memory $\tilde{m}_i^k$ for the forward$_i$ call (Alg. 4 L. 7) in case of recomputation (when $r_i^k$ is 1) depends on the set of local tensors $L$, checkpoint dependencies $D$, saved tensors $S$, and gradient tensors $\hat{L}$, named $L_i^k$, $D_k$, $S_i^{k-1}$, $\hat{L}_k$ respectively. Following the forward pass, when $r_i^k$ is 1:

$$\begin{aligned}\tilde{m}_i^k &= c_i + |x_i| + |\hat{L}_k| + |S_i^{k-1} \cup L_i^k \cup D_k| \\ &= c_i + |x_i| + |\hat{L}_k| + \sum_{j<i: x_j \notin L_i^k \cup D_k} s_j^{k-1} |x_j| + \sum_{j<i: x_j \in L_i^k \cup D_k} |x_j| + \sum_{j>i} s_j^k |x_j|. \quad (4.3)\end{aligned}$$

Unlike the forward pass, $L_i^k$ is no longer constant, but instead depends on past saved tensors and future recomputations in the schedule $(s, r)$: $L_i^k = \Theta \cup \{x_j : j \in \mathcal{N}_t \text{ for any } t \geq i \text{ with } r_t^k = 1 \text{ and } j < i\}$.

Next, we show how to take this formalization of the forward and backward pass and find an optimal execution plan including checkpointing schedule $(s, r)$, forward$_i$ implementations, and backward$_k$ implementations, under a fixed memory budget.

### 4.2.2 MONeT Formulation

Our goal is to find a global checkpointing schedule $(s, r)$ and local forward$_i$ and backward$_k$ implementations that jointly minimize the computation cost $\tau$ within a memory budget $M$. We show how to express this optimization in a 0-1 integer program and efficiently solve it. To this end, we linearize any peak memory consumption constraints, ensure that the checkpointing schedule is valid, and solve to minimize a computation cost objective. We keep track of the three contributors to memory and computational cost - forward pass, backward pass, and recomputation of forward operators.

**Memory Constraints.** Consider the case of basic checkpointing using only a single implementation for forward$_i$ and backward$_k$. The memory consumption of the forward (4.1) and backward (4.2) pass are linear in $s$, and thus efficiently expressed in an integer program. However, recomputation depends both on $s^{k-1}$ and $r^k$ in a non-linear manner through the local memory $L_i^k$. This joint dependence on optimization variables gives rise to quadratic constraints, which cannot directly be incorporated into an integer program. For simplicity in this derivation, we bound the set of local tensors from above, assuming every future tensor is recomputed. The upper bound $\bar{L}_i^k$ is constant, yielding a linear upper bound $\bar{m}_i^k$ of the recomputation memory $\tilde{m}_i^k$ analogous to Eq. 4.3. The set of memory constraints is thus

$$m_i \leq M \quad \forall_i \qquad \text{and} \qquad \hat{m}_k \leq M \quad \forall_k \qquad \text{and} \qquad \bar{m}_i^k \leq M \quad \forall_{k,i} \tag{4.4}$$

To enable operator optimization, we use a bit-vector $\delta$ to indicate the selection of an operator implementation. We add $\delta$ to the constraints which allows us to jointly optimize checkpointing $(s, r)$ and operator implementations $\delta$.

**Forward Operator Optimization.** Let each forward operator forward$_i$ have multiple different implementations $\mathcal{I}_i = \{a, b, c, \ldots\}$. For examples, convolution may be implemented using matrix multiplication, the Winograd algorithm [117], a Fourier transform, etc. [25]. All implementations follow the same DAG structure, and thus use the same dependencies $\mathcal{N}_i$. However, each implementation trades workspace memory $\{c_i^a, c_i^b, \ldots\}$ for computational efficiency $\{\tau_i^a, \tau_i^b, \ldots\}$ in a different manner. Our experiments show that this tradeoff is often complex.

Our goal is to represent the peak memory when using multiple forward$_i$ implementations in the forward pass and recomputation. Let $\delta_{i,a} \in \{0, 1\}$ indicate that implementation $a \in \mathcal{I}_i$ is used for forward$_i$ in the forward pass. Each forward operator should use exactly one implementation $\sum_l \delta_{i,l} = 1$. The choice of implementation determines the operator's computational cost $\sum_l \tau_i^l \delta_{i,l}$ and workspace memory $c_i = \sum_l c_i^l \delta_{i,l}$. Analogously, each recomputation of forward$_i$ during backward$_k$ chooses between implementations $\delta_{i,a}^k \in \{0, 1\}$ when needed $\sum_l \delta_{i,l}^k = r_i^k$, with equivalent cost estimates $\sum_l \tau_i^l \delta_{i,l}^k$ and workspace memory use $c_i^k = \sum_l c_i^l \delta_{i,l}^k$. In this formulation, all additional memory requirements remain linear and are directly integrated into the linear memory constraints or their linear relaxations (Eq. 4.4).

**Backward Operator Optimization.** Let each backward operator backward$_k$ have a set of different implementations $\hat{\mathcal{I}}_k = \{a, b, c, \ldots\}$. Each implementation again trades workspace memory $\{\hat{c}_k^a, \hat{c}_k^b, \ldots\}$ for computational cost $\{\hat{\tau}_k^a, \hat{\tau}_k^b, \ldots\}$. While gradient tensors follow the fixed DAG structure, different implementations may depend on different forward activations $\{\mathcal{D}_k^a, \mathcal{D}_k^b, \ldots\}$. For example, in-place activated operators [17] depend on their output activation, while regular operators use the input activation. This change in the dependency structure makes optimizing for backward-operator implementations challenging.

We again aim to represent memory in terms of implementations for each backward$_k$ operator. Let $\hat{\delta}_{k,a} \in \{0, 1\}$ indicate that implementation $a \in \hat{\mathcal{I}}_k$ is used at node $k$ in the backward pass. Each backward operator should use exactly one implementation $\sum_l \hat{\delta}_{k,l} = 1$, with a computational cost $\sum_l \hat{\tau}_k^l \hat{\delta}_{k,l}$ and workspace memory $\hat{c}_k = \sum_l \hat{c}_k^l \hat{\delta}_{k,l}$. The workspace memory adds a linear constraint to the memory consumption $\hat{m}_k$ (4.2).

The biggest changes to the optimization problem, comes from the *changing dependency structure*. $\mathcal{D}_k$ is no longer constant. Instead, the implementation of a backward operator changes the set of computational dependencies $D_k$ obtained from $\mathcal{D}_k^l$. To deal with this changing dependency structure, we use the indicator vector $\hat{\delta}_k$ to select memory contribution of dependencies from the chosen implementation. This changes the backward memory consumption to

$$\hat{m}_k = \underbrace{\sum_l \hat{c}_k^l \hat{\delta}_{k,l}}_{\hat{c}_k} + |y_k| + |\hat{L}_k| + \sum_l \hat{\delta}_{k,l}.|D_k^l \cup S^{k-1}|, \tag{4.5}$$

and the corresponding peak recomputation memory $\bar{m}_i^k$ to

$$\bar{m}_i^k = c_i + |x_i| + |\hat{L}_k| + \sum_l \hat{\delta}_{k,l}.|S_i^{k-1} \cup \bar{L}_i^k \cup D_k^l|. \tag{4.6}$$

Note, the last term of (4.5) and (4.6) are quadratic in the original optimization variables $s_i^{k-1}$, which determines $S^{k-1}$, and $\hat{\delta}_{k,l}$. However, for binary variables, it can be linearized using an auxiliary variable.

**Checkpointing Constraints.** The computational dependencies of forward and backward operators impose strict constraints on the checkpointing schedule. Any schedule violating these constraints cannot be executed, while any schedule following them can. Recomputation $r_i^k$ requires saved $s_j^{k-1}$ or recomputed $r_j^k$ dependencies $j \in \mathcal{N}_i$, and only previously stored or recomputed tensors can be saved:

$$r_i^k \leq s_j^{k-1} + r_j^k \quad \forall_{i,k,j \in \mathcal{N}_i} \qquad \text{and} \qquad s_i^{k-2} \leq s_i^{k-1} + r_i^k \quad \forall_{i,k}. \tag{4.7}$$

Furthermore, all forward tensors $\mathcal{D}_k^l$ required by backward$_k$ need to be stored or computed

$$s_i^{k-1} + r_i^k \geq \hat{\delta}_{k,l} \quad \forall_{k,l,i \in \mathcal{D}_k^l}. \tag{4.8}$$

**Objective.** Our goal is to minimize the amount of computation required for the forward and backward pass. This is represented as the sum of computational costs of all operators:

$$\underbrace{\sum_i \sum_l \tau_i^l \delta_{i,l}}_{\text{forward pass}} + \underbrace{\sum_k \sum_l \hat{\delta}_{k,l} \hat{\tau}_k^l}_{\text{backward pass}} + \underbrace{\sum_k \sum_l \tau_i^l \delta_{i,l}^k}_{\text{recomputation}} . \tag{4.9}$$

Objective (4.9) with constraints (4.4), (4.7), (4.8), and definitions (4.1), (4.5), (4.6) form our final optimization objective. It jointly solves for the optimal implementation of each forward and backward operator, as well as an efficient checkpointing schedule.

## 4.3 Detailed constraints

In this section, we explain some of the constraints used in MONeT's formulation in more detail.

### 4.3.1 In-place constraints

We show how to represent the decision of computing an operator using an in-place or out-of-place implementation. If an operator like ReLU uses an in-place implementation, its input tensor is overwritten with its output. In this case, its input tensor cannot be stored or used as input to a computation in this stage. This needs to be reflected in our constraints. We introduce two new binary variables to model in-place computations: $q_i^k$ represents if forward$_i$ is recomputed in-place when computing backward$_k$. $p_i^k$ represents that the output of forward$_i$ has been computed and will not be overwritten by any other forward node recomputations in this stage. If $q_i^k$ is true, then $p_j^k$ will be false else $p_j^k$ will be the same as $r_j^k$, where $j \in \mathcal{N}_i$. Further, $s_j^{k-1}$ will also be false if $q_i^k$ is true. This can be written in the form of boolean constraints as follows:

$$p_j^k \geq r_j^k - 2q_i^k \qquad \text{and} \qquad p_j^k \leq 2 - 2q_i^k \qquad \text{and} \qquad s_k^{k-1} \leq 2 - 2q_i^k . \tag{4.10}$$

The checkpointing constraint 4.7 changes, with $p_j^k$ replacing $r_j^k$ on the RHS. Further, $q_i^k$ (or $p_j^k$) can only be true if forward$_i$ (or forward$_j$) is actually recomputed prior to computing backward node k. Thus,

$$p_j^k \leq r_j^k \qquad \text{and} \qquad q_i^k \leq r_i^k. \tag{4.11}$$

### 4.3.2 Expanded backward pass memory constraints

We formulated the backward peak memory $\hat{m}_k$ and recomputation peak memory $\bar{m}_i^k$ as sum of memory of a set of tensors in the previous section. We now expand the memory formulation and represent it in terms of the optimization variables we use here:

$$\begin{aligned}\hat{m}_k &= \sum_l \hat{c}_k^l \hat{\delta}_{k,l} + |y_k| + |\hat{L}_k| + \sum_l \hat{\delta}_{k,l}.|D_k^l \cup S^{k-1}| \\ &= \sum_l \hat{c}_k^l \hat{\delta}_{k,l} + |y_k| + \sum_{y_l \in \hat{L}_k} |y_l| + \sum_l \sum_{x_i \in D_k^l} \hat{\delta}_{k,l}|x_i| + \sum_l \sum_{x_i \notin D_k^l} \underbrace{\hat{\delta}_{k,l} s_i^{k-1}}_{\sigma_{k,l,s}} |x_i|, \end{aligned} \tag{4.12}$$

$$\begin{aligned}\bar{m}_i^k &= c_i + |x_i| + |\hat{L}_k| + \sum_l \hat{\delta}_{k,l}.|S_i^{k-1} \cup \bar{L}_i^k \cup D_k^l| \\ &= c_i + |x_i| + |\hat{L}_k| + \sum_l \sum_{\substack{j<i: \\ x_j \notin \bar{L}_i^k \cup D_k^l}} \hat{\delta}_{k,l} s_j^{k-1}|x_j| + \sum_l \sum_{\substack{j<i: \\ x_j \in \bar{L}_i^k \cup D_k^l}} \hat{\delta}_{k,l}|x_j| + \sum_{j>i} s_j^k |x_j|. \end{aligned} \tag{4.13}$$

### 4.3.3 Complete memory constraints

We present the complete memory constraints which we use for MONeT optimization. These constraints include the recomputation variable $r_i^k$, which was excluded from the earlier discussion to make understanding simpler. As discussed in § 4.2.1, the peak memory of a forward$_i$ recomputation before computing backward$_k$ is denoted by $\tilde{m}_i^k$. This represents the recomputation memory (renamed to $m_{Ri}^k$) when forward$_i$ is actually recomputed, that is, $r_i^k = 1$. When this is not true, the peak memory ($\tilde{m}_{Si}^k$) only depends on stored checkpoints $S_i^{k-1}$, checkpoint dependencies for

$D_k$, and gradient tensors $\hat{L}_k$. Thus,

$$\begin{aligned}\tilde{m}^k_{Ri} &= c_i + |x_i| + |\hat{L}_k| + |S^{k-1}_i \cup L^k_i \cup D_k| \\ &= r^k_i c_i + r^k_i |x_i| + |\hat{L}_k| + \sum_{j<i: x_j \notin L^k_i \cup D_k} s^{k-1}_j |x_j| + \sum_{j<i: x_j \in L^k_i} r^k_i |x_j| + \sum_{j<i: x_j \in D_k - L^k_i} |x_j| + \sum_{j>i} s^k_j |x_j|.\end{aligned} \tag{4.14}$$

$$\begin{aligned}\tilde{m}^k_{Si} &= |\hat{L}_k| + |S^{k-1}_i \cup D_k| \\ &= |\hat{L}_k| + \sum_{j \le i: x_j \notin D_k} s^{k-1}_j |x_j| + \sum_{j \le i: x_j \in D_k} |x_j| + \sum_{j>i} s^k_j |x_j|.\end{aligned} \tag{4.15}$$

Local memory $L_k$ can be bounded by $\bar{L}_k$, which gives us $\bar{m}^k_{Ri}$. To add forward operator optimizations to $\bar{m}^k_{Ri}$, we recall the trade-off between workspace memory and compute time. We replace the workspace memory contributor $r^k_i c_i$ in equation 4.14 with $\sum_l \delta^k_{i,l} c^l_i$.

The complete memory constraints are:

$$m_i \le M \quad \forall_i \quad \text{and} \quad \hat{m}_k \le M \quad \forall_k \quad \text{and} \quad \bar{m}^k_{Ri} \le M \quad \forall_{k,i} \quad \text{and} \quad \tilde{m}^k_{Si} \le M \quad \forall_{k,i} \tag{4.16}$$

### 4.3.4 Constraint Linearization

The memory constraints we introduce in § 4.2.2 contain quadratic terms in the form of $x_i \cdot x_j$, with $x_i, x_j \in \{0, 1\}$. The quadratic terms cannot directly be incorporated into an integer program. However, we can linearize these terms by replacing each quadratic term $x_i \cdot x_j$ by an auxiliary variable $\alpha_{i,j} \in \{0, 1\}$ and introducing additional linear constraints $\alpha_{i,j} \ge x_i + x_j - 1$, $\alpha_{i,j} \le x_i$, and $\alpha_{i,j} \le x_j$. After this substitution for all quadratic terms, all constraints in MONeT are linear

## 4.4 Implementation

We develop MONeT in the PyTorch (v1.5.1) framework. We use PyTorch's default Autograd package for backward implementation of elementary functions when the autograd implementation is stateless. In all other cases, we implement custom forward and backward functions leveraging PyTorch ATen library functions to flexibly support multiple operators and execution schedules. Each backward operator implementation is annotated with its computational dependencies, which is generally the input or the output of its corresponding forward operator. Certain backward operators implementations may have dependencies on intermediate activations generated in the forward pass. For example, an intermediate-activated ReLU backward uses an encoded bit-mask representing the sign of forward operator's input. We annotate this as an intermediate storage node and add it to our optimization problem, with a strict recomputation dependency of the intermediate storage node on its creator node. Our operator optimizations select from different backward operator implementations, convolution algorithms, in-place operators etc. We split the convolution backward operator into two - a parameter-gradient operator followed by an input-gradient operator. Since the input-gradient operator does not have any computational dependency on the forward pass, we can aggressively free the forward input tensor right after the parameter-gradient is computed. We also reuse BatchNorm statistics in case of their recomputation. For our experiments, we limit ourselves to full precision training as quantization or lower precision computations introduce additional noise into SGD and change its convergence properties. We solve the joint optimization problem using the CVXPY [35, 8] solver with [50] backend.

**MONeT workflow.** We obtain the forward pass dependencies in MONeT by JIT tracing a model to obtain its graph. We profile each layer for workspace memory and compute cost, and obtain memory usage of the tensors from their shape and type. Note that the workspace memory for many convolution operators in VGG-16 is greater than 2GB, making it an important factor to model. Unlike prior approaches like Checkmate, we account for this workspace memory in our optimization problem, bringing the memory model very close to actual memory allocation. We phrase a boolean integer programming problem using the generated graph and the profiled compute cost and workspace memory and solve it using the CVXPY [35, 8] modeling language and GUROBI [50] solver. The solution is used to generate a schedule that can be run by the MONeT scheduler.

**Operator optimizations.** We divide operator optimizations according to the different type of implementations they select from.

- *Output-activated*: Backward calculation of operators like ReLU and BatchNorm can have computational dependency either on on their forward node's inputs or outputs.
- *Intermediate-activated*: Backward of ReLU has computational dependency on a 1-bit encoding of the sign of its forward node's input. Backward of MaxPool is calculated using an intermediate 8-bit output-shaped tensor which contains the kernel-index of the maximum element.
- *Convolution algorithms*: We choose from 8 forward and 6 backward cuDNN convolution algorithms.
- *In-place operations*: The solver can choose to do in-place computation for operators like ReLU forward. All MONeT experiments enable in-place operation selection.

## 4.5 Discussion

### 4.5.1 Adding operator optimization in other checkpointing frameworks

Adding operator optimization in other checkpointing frameworks is not straightforward. We briefly explain the difficulties of including operator selection directly into existing checkpointing framework - checkmate [59]. We will refer directly to the notation and equations in the checkmate paper (arxiv v3; 14 May 2020). The most direct way to incorporate operator selection into checkmate is to introduce an auxiliary variable $R^v_{t,i} \in \{0, 1\}$ that refers to re-computing layer $i$ at time $t$ using implementation $v$. Most constraints in equation 1 could stay the same, given $R_{t,i} = \sum_v R^v_{t,i}$, and loss (1a) $\sum_t \sum_i \sum_v R^v_{t,i} C^v_i$. Some of our operators produce a different kind of checkpoint (e.g. binary activated ReLUs), which could be handled in check-mate by splitting $S^v_{t,i}$. The main issues in Checkmate arise in the memory modeling and its relaxations (eq 4,5,7). The memory consumed by a specific checkpoint may depend on the operator implementation: DEPS[k] and USERS[i] both depend on the operator implementation (output activated, input activated, ...). In short, the checkmate computation graph is dynamic and depends on operator implementations. The most direct way to address this is to $\text{mem\_freed}_t(v_k) = \sum_v R^v_{t,i}\text{mem\_freed}_t(v_k)$ in an implementation-dependent way $\text{mem\_freed}^v_t(v_k)$, and select the right version dependent on the operator used. Likewise, we need to extend $\text{FREE}^v_{i,t,k}$ to account for different operator implementations in $R^v_{t,k}$. Likewise, the product in equation (5) will now go over all implementations $R^v_{i,j}$ using different USERS sets. This leads to a linear blowup in the number of constraints, and the number of auxiliary variables, leading to an at least quadratic expansion on computational costs. Furthermore, $\text{mem\_freed}_t(v_k) = \sum_v R^v_{t,i}\text{mem\_freed}_t(v_k)$ is a quadratic constrain that further needs to be resolved using additional auxiliary variables. Given that Checkmate already pushes the limits of current solvers, it is unlikely able to handle this explosion in constraints and variables, without significant modifications. MONeT on the other hand represents the compute-graph more compactly and efficiently integrates different operator implementations.

### 4.5.2 Applicability of MONeT to inference workloads

MONeT is mainly meant to reduce the memory usage of model training by not storing some activations that will be needed in the backward pass and by modifying implementations of the backward pass operators. It can not reduce the memory of operator parameters, which generally take up the most memory in inference workloads. MONeT's checkpointing technique may be used in specific cases where the model has heavy branching and requires activations to be present until consumed by the last forward operator. Further, MONeT's operator optimization technique could be used to select between different computation sub-graphs (instead of different implementations of a single operator), in order to reduce workspace memory usage. For example, a large matrix multiplication may be divided into sequential multiplications and additions over matrix partitions, and thus has two different operator implementations that MONeT could select from. However, we expect there to be significant modifications required in MONeT to accommodate these features. Currently, MONeT does not support recomputing operations when running the forward pass, nor does it have the functionality to identify all the different computation sub-graph choices.

## 4.6 Evaluation

In this section, we use a number of deep networks with varying user-provided memory budgets to evaluate the memory footprint and compute overhead of MONeT in relation to state-of-the-art memory saving tools for checkpointing (Checkmate) and operator-optimizations (Gist). We answer the following questions in this section:

- Does MONeT constrain memory usage when training?
- How does much computation overhead does MONeT incur as compared to other memory-saving techniques?
- How do enabling different operator optimizations in MONeT impact training time?
- How much time does MONeT need to solve the joint optimization problem?
- How many constraints and variables are present in MONeT's joint optimization problem?

We first describe our experimental methodology and our implementations of the baselines before addressing each of the above questions.

### 4.6.1 Experimental Setup

We evaluate the performance of MONeT against other tools like Checkmate and Gist on models like ResNet-50, GoogleNet, UNet, VGG-16, and MobileNet-V2.

We run our experiments on a 16 GB NVIDIA P100 GPU with the largest input batch size that fits in the GPU without any optimizations. The UNet experiments use 608×416 inputs following prior work [59]. All other experiments use 224×224 inputs following conventions [70, 107, 51].

### 4.6.2 Baseline Implementations

Since Checkmate's [59] execution engine is built for TensorFlow, and an official Gist [58] implementation is not available, we reimplement them in PyTorch for our comparisons.

**Checkmate implementation.** Our Checkmate implementation is competitive, it uses the original Checkmate solver and has the same network structure as MONeT. Checkmate does not optimize for operator implementations like convolutions, so we show its runtime using the default convolution algorithm (Checkmate-D). For a stronger comparison, we also show the runtime of a Checkmate schedule that is post-optimized to greedily run the fastest convolution algorithm (Checkmate-O). Wherever not explicitly specified, we compare with Checkmate-O. All checkpointing schedules are run using the same software implementations and costs are profiled on the same hardware (NVIDIA P100 GPUs). We have released our Checkmate implementation with the MONeT code.

**Gist implementation.** Gist [58] is an operator-based memory-efficient scheme for training DNNs. It encodes stashed forward tensors into smaller tensors which require less memory. Jain et al. [58] evaluate Gist using CNTK on an Nvidia Maxwell GTX Titan X GPU. Since we implement MONeT in PyTorch and have access to an Nvidia P100 GPU, a direct comparison with the numbers in the Gist paper is not possible. As an official Gist implementation is not available, we reimplement it on PyTorch and evaluate its execution using MONeT 's execution framework.

We implement all Gist optimizations — Binarize (intermediate encodings for ReLU-Pool layers), Sparse Storage Dense Compute (compress and store sparse convolution inputs in ReLU-Conv layers as sparse storage), Delayed Precision Reduction (storing stashed non-compressed tensors in FP-16, but computing in FP-32), and Inplace (performing ReLU operators in-place wherever possible) over MONeT's execution framework. In Gist, the Sparse Storage Dense Compute (SSDC) technique creates a sparse storage tensor in the Compressed Sparse Row (CSR) representation using the Nvidia cuSPARSE library. The dense storage is reshaped into a 256-sized column tensor before storing it in a sparse format, allowing the column index of CSR representation to be saved in 8 bits instead of using 32 bits (termed Narrow Value Optimization in the paper). We also implement SSDC using Nvidia's cuSPARSE library (function `cusparseSdense2csr`) with CUDA Toolkit version 10.1 using PyTorch's C++ extensions.

In their paper, Jain et al. [58] use the most memory-efficient convolution algorithms in Gist and compare its memory saving against a baseline which also chooses the most memory-efficient convolution algorithm. Using memory-efficient convolution algorithms, our Gist reimplementation can train VGG-16 with $0.55\times$ of the PyTorch-required memory ($1.81\times$ memory footprint), which is close to the data presented by Jain et al. [58]. However, it is 59% slower than when convolution selection is enabled, in which case it can train using $0.76\times$ of the PyTorch-required memory. Since implementing Gist using memory-efficient convolutions is not optimal in terms of compute time, we implement Gist to use PyTorch's convolution selection algorithm. For all models other than VGG-16 and UNet, we see similar memory savings for Gist with memory-efficient convolutions and with convolution-selection enabled. We have released our Gist implementation with the MONeT code.

### 4.6.3 Constraining memory usage

The main aim of MONeT is to constrain memory usage for deep network training to within a user-provided memory budget while reducing the compute overhead that comes as a tradeoff. Figure 4.3 shows the memory usage of a ResNet-50 network

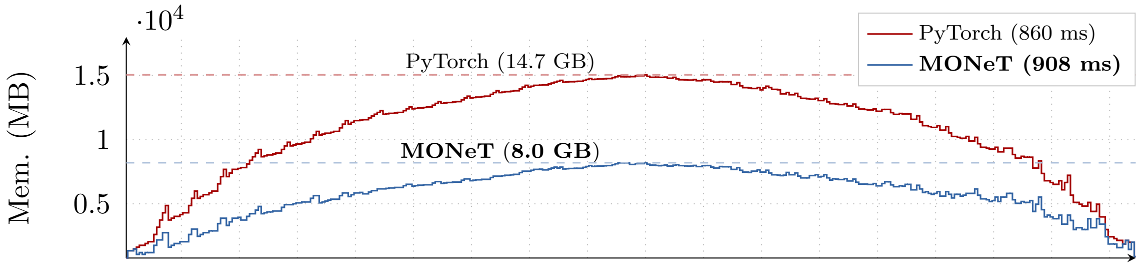


Figure 4.3: **Case study on ResNet-50.** Memory usage along execution (forward and backward). MONeT ensures that the model uses at most 8 GB of memory at any point of time during training.

| | ResNet-50 | GoogleNet | UNet | VGG-16 | MobileNet-V2 |
|---|---|---|---|---|---|
| PyTorch | 15.1 | 14.9 | 14.3 | 14.1 | 14.5 |
| Checkmate [59] | 8.2 | 10.5 | 9.1 | 9.9 | 5.8 |
| **MONeT** | **5.7** | **6.9** | **5.2** | **5.5** | **4.8** |

Table 4.2: **Memory usage comparison (in GB) for a fixed compute overhead for Checkmate and MONeT.** At 10% compute overhead over PyTorch, MONeT uses **2-3**× less memory than PyTorch. At the same overhead, MONeT can train models using 1.2-1.8× less memory than Checkmate.

with PyTorch and compares it to MONeT with a memory budget of 8 GB. MONeT ensures that the memory usage of the network is capped at 8 GB at all times.

### 4.6.4 Computation overhead

**Comparison against checkpointing frameworks.** Table 4.2 compares the memory savings obtained by MONeT and Checkmate for five different models when computational overhead over PyTorch is fixed to be 10%. MONeT schedules use 2-3× less memory than PyTorch. For the same computational overhead, MONeT uses 1.2-1.8× less memory than Checkmate.

Figure 4.4 shows detailed runtime-memory trade-offs of MONeT to PyTorch and Checkmate for different models. We plot the average iteration time of training as % overhead over PyTorch for MONeT and Checkmate schedules. The memory budgets range from 5 GB to 10 GB, or equivalently, 0.33× to 0.70× PyTorch memory

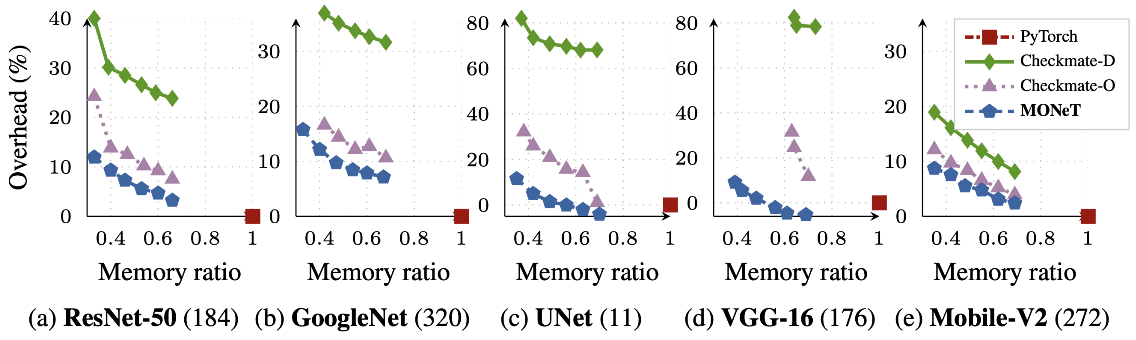


Figure 4.4: **Comparing MONeT with PyTorch and Checkmate.** MONeT reduces memory by 3× compared to PyTorch, with 9-16% compute overhead. It achieves a better memory-compute trade-off than default Checkmate-D and conv-optimized Checkmate-O.

| | VGG-16 (176) | | ResNet50 (184) | | GoogleNet (320) | | MobileNetV2 (256) | | UNet (11) | |
|---|---|---|---|---|---|---|---|---|---|---|
| | mem | overhead | mem | overhead | mem | overhead | mem | overhead | mem | overhead |
| Gist | 0.76 | 44.34 | 0.58 | 105.69 | 0.52 | 35.94 | 0.69 | 153.98 | 0.73 | 38.26 |
| **MONeT** | **0.39** | 9.11 | **0.33** | 11.94 | **0.33** | 15.77 | **0.34** | 8.80 | **0.35** | 11.51 |

Table 4.3: **Memory ratio and overhead (%) over PyTorch for Gist and MONeT.** MONeT obtains 1.4×-2.1× higher memory savings over Gist across models. The number in parenthesis after the model name shows the batch size.

consumption. The batch size for these models is mentioned in parentheses. For all models, MONeT reduces memory usage by 3× (0.33 memory ratio) as compared to baseline PyTorch with $9 - 16\%$ compute overhead. For the same memory budget, MONeT schedules are up to 34% faster than Checkmate schedules. Note that we measure the empirical performance of the schedules running on GPUs instead of just providing a simulation of runtime and memory using the solver values; this is important since Checkmate does not consider workspace cost and overestimates its savings.

For networks with individual memory-intensive layers, like VGG-16, operator optimization becomes even more important for reducing memory; Checkmate can reduce memory for VGG-16 only up to 7 GB, whereas MONeT with its optimizations is able to run VGG-16 with 5.5 GB memory. The small runtime improvement of MONeT schedules over PyTorch for VGG-16 and UNet at higher memory budgets is mainly because of choosing faster convolution algorithms. MobileNet-V2 uses depthwise convolutions and hence does not significantly benefit from joint convolution optimization. As a result, the performance of MONeT and Checkmate is closer for MobileNet-V2. We provide additional results for MONeT on a memory-intensive model, 3D-UNet [29], in Appendix A.3, for which we observe a consistent memory reduction to 0.54× of PyTorch memory with an overhead of 8.86%.

**Comparison against tools for operator-optimizations.** Table 4.3 shows the comparison of MONeT with Gist. While MONeT can determine a range of memory-runtime tradeoffs, purely operator-optimization-based schemes like Gist only provide a single memory-runtime data point. For MONeT, we show the memory-runtime data point with the most memory saving. MONeT uses 1.4×-2.1× less memory than Gist for multiple architectures while maintaining full precision. Overall, Gist provides impressive memory savings but incurs a high computation cost to achieve the savings.

While we get similar memory-saving results for reimplemented-Gist as shown by its authors for VGG-16, our compute overhead results are higher. This could be because of evaluations on different frameworks (PyTorch v/s CNTK) and different GPU models (Nvidia P100 v/s Nvidia Maxwell GTX Titan X). Gist uses dense to sparse conversion using `cusparseSdense2csr` in one of its techniques. For the first ReLU-Conv layer in VGG-16 (shape `(2207744,256)`), this function takes 144ms, which itself is 10% of the VGG-16 execution time. We see similar results for other networks. To ensure a fair comparison, we focus on the maximum memory savings obtained by MONeT with Gist, while reporting the compute overhead for completeness.

### 4.6.5 Ablation experiments

Figure 4.5 shows additional ablation experiments. We show the % compute overhead over PyTorch on ResNet-50, GoogleNet, and VGG-16 for different types of

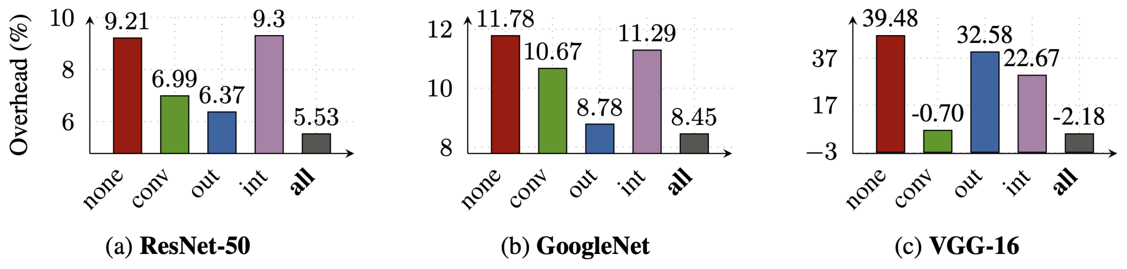


Figure 4.5: **Ablation results for memory ratio 0.53.** Lowest compute overhead across models is seen only when all optimizations are jointly optimized.

MONeT checkpointing schedules with a memory budget of 8 GB - with no operator optimizations enabled, with only one type of operator optimization enabled (conv-optimized, output-activated optimized, intermediate-activated optimized), and with all optimizations enabled. Schedules which do not jointly optimize convolution algorithms are run with greedily post-optimized convolution algorithm. Plots for other models look similar to that of ResNet-50 and GoogleNet. The only difference between 'none' and 'conv' is that convolution algorithms are jointly optimized in the latter. However, this fact leads to significant improvement in compute time for all cases. Similarly, output-activated optimization also provides significant benefits over vanilla checkpointing, since it effectively reduces the number of recomputations required. For memory-intensive networks, intermediate-activated optimization becomes more important. Jointly optimizing all strategies together gives the least computational overhead. See Appendix A.1 for detailed ablation plots.

### 4.6.6 Solver time

For our evaluations, we capped the solver time to 24 hours for both MONeT and Checkmate, and ran the schedule thus obtained on MONeT's execution framework. At tighter memory budgets for non-linear models like ResNet-50 and GoogleNet, Checkmate is unable to find a feasible solution within a couple of hours. In contrast to Checkmate, MONeT finds the execution plans efficiently. For all the models and memory limits that we evaluate, MONeT reaches a 5% close-to-optimal solution

| | 5 GB | 6 GB | 7 GB | 8 GB | 9 GB | 10 GB |
|---|---|---|---|---|---|---|
| **ResNet-50** | | | | | | |
| Checkmate | - | 8.96 | 12.01 | 10.78 | 4.54 | 2.98 |
| MONeT-NoOp | 1.18 | 0.46 | 0.14 | 0.09 | 0.06 | 0.07 |
| MONeT | **7.24** | **3.84** | **0.73** | **0.70** | **0.31** | **0.11** |
| **GoogleNet** | | | | | | |
| Checkmate | - | 12.72 | 4.56 | 4.32 | 3.92 | 0.86 |
| MONeT-NoOp | 0.10 | 0.11 | 0.07 | 0.07 | 0.07 | 0.07 |
| MONeT | **3.53** | **0.47** | **0.54** | **0.31** | **0.25** | **0.24** |
| **MobileNet-V2** | | | | | | |
| Checkmate | 2.16 | 2.88 | 1.16 | 0.29 | 0.34 | 0.14 |
| MONeT-NoOp | 0.11 | 0.04 | 0.02 | 0.02 | 0.04 | 0.08 |
| MONeT | **0.37** | **0.28** | **0.52** | **0.05** | **0.06** | **0.03** |
| **UNet** | | | | | | |
| Checkmate | **0.149** | **0.031** | **0.022** | **0.020** | **0.010** | 0.009 |
| MONeT-NoOp | 0.048 | 0.002 | 0.002 | 0.002 | 0.002 | 0.002 |
| MONeT | 0.363 | 0.064 | 0.028 | 0.027 | 0.024 | **0.006** |
| **VGG-16** | | | | | | |
| Checkmate | - | - | - | **0.002** | **0.002** | **0.001** |
| MONeT-NoOp | - | - | - | 0.001 | 0.000 | 0.000 |
| MONeT | - | **0.003** | **0.003** | 0.003 | 0.003 | 0.003 |

Table 4.4: **Solver time (in hours) to reach 5% close to optimal solution.** MONeT-NoOp reaches a 5% close-to-optimal solution 1.6×-117× faster than Checkmate. MONeT gets close to 5% of the optimal solution only in a few hours, and up-to 16× faster than Checkmate for larger models.

within a few hours or sometimes even minutes. Table 4.4 shows the time it takes for the solver to reach 5% close to the optimal solution, for Checkmate, MONeT-NoOp (MONeT with checkpointing enabled but operator-optimization disabled), and MONeT. MONeT-NoOp converges to a close-to-optimal solution 1.6×-117.4× faster than Checkmate. For larger models, MONeT's solver converges to a close-to-optimal solution up to 27× faster than Checkmate. Note that running a solver is a one-time cost for a model - once a MONeT schedule has been solved for, it can be used by everyone to train the model for different purposes with different batch sizes. The cost (typically seconds to hours) is tiny compared to the efforts and costs to develop a model for distribution in most cases. We also discuss the time taken by the solver to reach 2% close to the optimal solution in Appendix A.2.

| | | Checkmate | | MONeT-NoOp | | MONeT | |
|---|---|---|---|---|---|---|---|
| | Fwd ops | Constraints | Variables | Constraints | Variables | Constraints | Variables |
| GoogleNet | 215 | 719,327 | 519,252 | 221,630 | 104,673 | 781,747 | 362,640 |
| ResNet-50 | 175 | 473,592 | 344,659 | 344,652 | 167,238 | 487,842 | 229,431 |
| Mobilen.V2 | 153 | 337,316 | 247,033 | 153,828 | 74,579 | 241,478 | 115,047 |
| UNet | 67 | 65,715 | 48,744 | 32,273 | 15,982 | 73,624 | 36,548 |
| VGG-16 | 40 | 25,334 | 18,968 | 12,772 | 6,306 | 21,918 | 11,234 |

Table 4.5: **ILP statistics for Checkmate, MONeT-NoOp, and MONeT.** MONeT-NoOp has on average 50% fewer constraints and 67% fewer variables than Checkmate. MONeT has a slightly higher number of constraints, on average 40% fewer variables than Checkmate.

### 4.6.7 ILP statistics in MONeT's formulation

For different models, Table 4.5 shows the solver statistics after presolving for the problem formulated by Checkmate, MONeT-NoOp, and MONeT for a 10 GB memory limit. It shows the number of forward operators in the model and the number of constraints and variables for each solver. MONeT-NoOp, which is MONeT with only checkpointing enabled and without using operator optimization, has on average 50% fewer constraints and 67% fewer variables than Checkmate. Jointly-optimized MONeT has a slightly larger number of constraints and on average 40% fewer variables than Checkmate. MONeT's formulation is more efficient and might be the reason that it reaches a good solution faster than Checkmate.

## 4.7 Summary

In this chapter, we presented MONeT, a framework to automatically reduce memory requirements for training deep networks. MONeT jointly optimizes local (operator-level) and global (graph-level) optimizations to yield a compute- and memory-efficient checkpointing schedule. MONeT reduces memory usage by 3× over PyTorch, with a $9-16\%$ compute overhead. It uses 1.2-1.8× less memory than the state-of-the-art automated checkpointing framework for the same computational cost. Experimental results show that MONeT leads to better memory-computation trade-offs compared to the state-of-the-art. MONeT is open-source and available at `https://github.com/utsaslab/monet`.

# Chapter 5: TACCL: Guiding Collective Algorithm Synthesis using Communication Sketches

In Chapter 3, we discussed how GPUs communicate with each other over the network using MPI-style communication collectives in distributed deep learning. We looked at some existing libraries for collective communication and discussed the challenges they face in building collective algorithms for heterogeneous hardware and different input sizes.

In this chapter, we present TACCL, a semi-automated tool that enables algorithm designers to guide a synthesizer into generating efficient collective algorithms for a given hardware topology and communication collective [1]. TACCL uses a novel abstraction of communication sketch and a novel formulation of the algorithm-synthesis problem in order to scale beyond single-node topologies.

First, we present the goals of TACCL (§ 5.1) and motivate TACCL's core components (§ 5.2). We briefly explain how TACCL models the physical topology of the GPU systems (§ 5.3). We then describe its design (§ 5.4) and problem formulation (§ 5.5) in detail, and discuss the implementation of TACCL's backend (§ 5.6). Finally, we evaluate TACCL against an existing state-of-the-art library for collective communication (§ 5.8).

## 5.1 Goals

**Efficient collectives.** TACCL should be able to generate fast and efficient collective communication algorithms.

**Scalability.** TACCL should be able to generate collective algorithms for multi-node

[1]This Chapter is based on the work, TACCL: Guiding Collective Algorithm Synthesis using Communication Sketches, published in NSDI'23 [104]

topologies.

**Generality across different topologies.** TACCL should be applicable for different kinds of hardware topologies.

**Data-size awareness.** TACCL should be able to generate collective algorithms that perform well for the expected size of data transfers.

## 5.2 TACCL components

In order to generate efficient collectives, we need to first identify the physical topology and link profiles and then determine the route and schedule that data chunks should take. The search space of possible algorithms to implement a collective is intractably large and cannot be explored via brute force. Deciding whether or not to route data chunks from $n$ GPUs over $l$ links in a topology has $O(2^{n \times l})$ combinations. As we scale to multi-node topologies, $n$ as well as $l$ will also scale, increasing the exponent quadratically. The search space explodes further if we consider the problem of ordering data sends at each link along with deciding routing for the data. We argue that high-level inputs from a human algorithm designer help reduce the search space to make algorithm synthesis more tractable. In the most extreme case, the user would hand-write the entire algorithm. However, handcrafting data routing and scheduling over links to implement a collective is complex and requires many design choices. Instead, users can provide input in the form of an intuitive *communication sketch* around which a collective algorithm can be synthesized. It is important to ensure that providing inputs to TACCL is a low-effort activity, but can discard large parts of the search space to achieve improvements in running time of the synthesis engine.

TACCL synthesizes a collective algorithm by deciding the route that each data chunk in the collective should take in the topology as well as the ordering of chunks at every link. Even with communication sketches that reduce the search space for the synthesizer, this decision problem is NP-hard and the complexity increases exponentially with the number of GPUs. To make the problem more tractable, we first relax the synthesis problem to solve just the routing of all data chunks and then heuristically order chunks sent over the same links according to bandwidth constraints. TACCL's synthesizer design along with communication sketches help TACCL synthesize efficient collectives for multi-node topologies.

## 5.3 Physical Topologies of GPU systems

In order to effectively sketch algorithms for TACCL to synthesize, algorithm designers must understand the *physical topology* of the target multi-GPU system. However, the performance characteristics of their heterogeneous links are sparsely documented and for some cloud offerings [9] even the topology is not given. To address this, TACCL includes a *physical topology profiler* to measure performance characteristics of links and to disambiguate the topologies of some multi-GPU systems.

**$\alpha$-$\beta$ Cost Model and Link Profiling.** TACCL uses the well-known $\alpha$-$\beta$ [52] cost model to model link performance. $\alpha$ is the latency of a link and $\beta$ is the inverse of its bandwidth. The cost of sending a chunk of size $s$ along a link is $\alpha + \beta \cdot s$. $\alpha$ and $\beta$ are affected by both the interconnect hardware as well as the software stack running the collective algorithm. TACCL helps users select these values with a *link profiler*, which measures and infers the $\alpha$ and $\beta$ costs of different types of links in GPU systems.

The profiler empirically derives the $\alpha$ and $\beta$ parameters of different links in the network by performing peer-to-peer data transfers between GPUs. We send $n$ chunks one after another on a link and measure the time to transfer. As per the $\alpha - \beta$ cost model, the time to transfer is $n \cdot (\alpha + \beta \cdot s)$. We then send $n$ chunks all at once on the link and attribute that time to be $\alpha + n \cdot \beta \cdot s$. Using several measurements of time to transfer, we solve for $\alpha$ and $\beta$. Table 5.1 shows the $\alpha$ and $\beta$ values for NDv2 and DGX-2 systems. Using these values, we expect that for transfers between

| Link | Azure NDv2 | | Nvidia DGX-2 | |
|---|---|---|---|---|
| | $\alpha$ (us) | $\beta$ (us/MB) | $\alpha$ (us) | $\beta$ (us/MB) |
| NVLink | 0.7 | 46 | 0.7 | 8 |
| InfiniBand | 1.7 | 106 | 1.7 | 106 |

Table 5.1: Experimentally obtained $\alpha$ and $\beta$ costs for Azure NDv2 and Nvidia DGX-2 nodes.

two Azure NDv2 nodes over InfiniBand (IB), sending two 32 KB chunks together as a single 64 KB chunk will be 17% faster as compared to sending two 32 KB chunks one after the other.

**Inferring Multi-GPU Topologies** In many cases, the physical topology for the multi-GPU system may not be known or may be hidden due to virtualization. TACCL helps determine the topology of the system with a topology profiler.

For example, consider the use-case of in Azure NDv2 systems. Their physical topology is not fully documented: while the NVLink topology (Figure 3.2) is known to match that of Nvidia DGX1, the details of how GPUs and the one 12.5 GBps InfiniBand NIC are connected with PCIe is not known. PCIe peer-to-peer communication (and thus GPUDirect RDMA [4]) is not enabled on these machines, meaning that all communication happen through buffers in CPU memory over potentially shared PCIe links. Further, virtualization obscures the true PCIe topology (all 8 GPUs and the NIC appear directly connected to one CPU) and NUMA node and GPU IDs are not assigned consistently from VM to VM. This means that, without additional information, software cannot avoid contention over shared PCIe links, creating interference and high variance in performance.

To determine the PCIe topology, TACCL's profiler sends bandwidth and latency probes between the two CPUs and between pairs of GPUs. It answers the following questions:

- Which CPU is nearest to the NIC? We find this by the latency of loopback RDMA operations to each CPU.

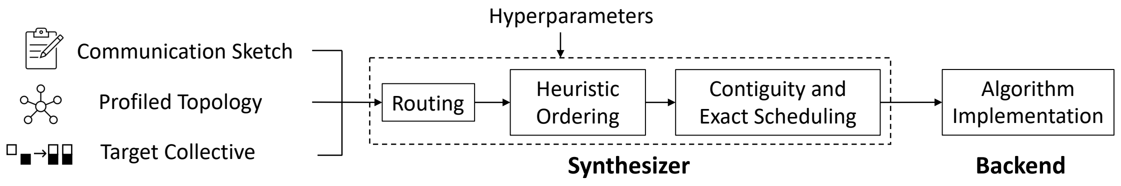


Figure 5.1: **TACCL workflow.** TACCL's novel synthesizer takes as input a communication sketch, profiled topology, and target collective along with synthesizer hyperparameters to generate an algorithm for the collective. The synthesized algorithm is implemented on the hardware cluster using TACCL's backend.

- Which GPUs share a PCIe switch? We find all pairs of GPUs that get low bandwidth in a simultaneous copy to the CPU, indicating contention.
- Which GPUs share a PCIe switch with the NIC? We find which GPUs get low bandwidth in copies from the CPU closest to the NIC while it is doing an RDMA loopback copy between buffers on the CPU.

With this profiling information we were able to deduce the PCIe topology (Figure 3.3) of Azure NDv2 systems. Each CPU has two PCIe switches connecting to two GPUs each, and the InfiniBand NIC is connected to one of these switches. Additionally, by running the profiler on every new NDv2 VM TACCL is able to select one of the NVLink topology's four automorphisms and set the `CUDA_VISIBLE_DEVICES` environment variable such that the NIC is always placed close to GPU 0.

## 5.4 Design

Figure 5.1 shows an overview of TACCL's design. We introduce communication sketches as a new form of sketching [110] that serve as an effective tool for users to communicate interesting aspects of collective communication algorithms to synthesis backends. TACCL's novel stage-wise synthesizer takes in the communication sketch, a profiled hardware topology, and the target collective as input and outputs a collective algorithm that is lowered into an executable format by TACCL's backend. In this section, we will discuss the two main components of TACCL - communication sketches and synthesizer - that enable it to generate efficient collective algorithms for large-scale topologies.

### 5.4.1 Communication Sketches

Sketching approaches must strike a balance between allowing users to omit implementation details while still providing enough direction for the synthesis to scale. In our experience, *routing* is an aspect of collective communication that we often have intuitions about, while reasoning about *scheduling* tends to be tedious and better left to synthesis. Moreover, properties about scheduling are routing dependent since the order of operations is only relevant when routes intersect, which makes them harder to express. Meanwhile, interesting properties about routing are expressible globally, e.g., "never send over InfiniBand from this GPU". We thus ask the algorithm designer (user) for three low-effort inputs as a part of the communication sketch:

1. Specifying the *logical topology* as a subset of the actual physical topology that the algorithm will operate on. For example, the outgoing links of all but one GPU can be removed in the logical topology to force all data going to remote GPUs to be relayed through one GPU.

2. Annotating switches inside the topology for the synthesizer to take certain *switch-hyperedge policies.*

3. Providing *algorithm symmetry* based on the symmetries in the topology and the collective.

We explain each part of the communication sketch below and provide an example of a sketch written for TACCL.

**Logical Topology.** The core of TACCL's communication sketches is a *logical topology*, a subset of the physical topology in which the user may omit links that they wish the routing to avoid. As a result, a logical topology has as many nodes as the

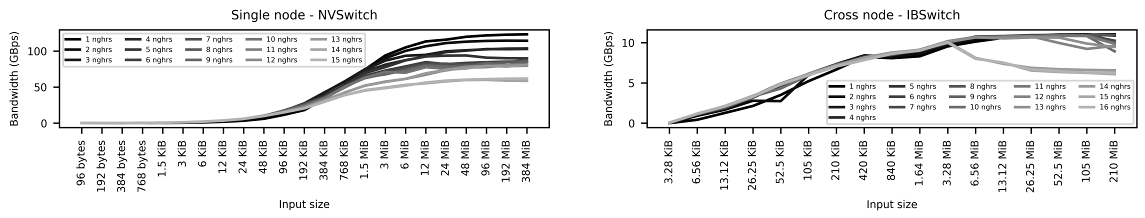


Figure 5.2: Multi-connection with varying number of GPU neighbors and data volume.

physical topology. A logical topology inherits the cost model produced by TACCL's profiler for the physical topology. A logical topology omits NICs and switches and uses switch-hyperedges, abstracting them away into links between GPUs. The reason is twofold: TACCL runtime is embedded inside NCCL runtime and NCCL has no direct control over NIC or switch use, and it allows TACCL to reason over a smaller graph thus enhancing scalability. We discuss the implications of this later in this section.

**Example 5.4.1** (Distributed sketching for NDv2 clusters)**.** For distributed collective communication with NDv2 systems, some PCIe connections must be used since the NIC is connected to GPUs over PCIe (Figure 3.3). Care must be taken in choosing which links to use, as some PCIe links are oversubscribed and due to lack of GPUDirect RDMA [4] on these systems, all communication must pass through host memory. Obtaining maximum throughput systems requires a logical topology that avoids conflicting flows on the oversubscribed PCIe links. To build a logical topology for a cluster of NDv2 systems, a pair of receiver and sender GPUs is selected for each NDv2 such that the selected GPUs and the NIC are connected to the same PCIe switch.

**Switch-Hyperedges.** In a switched fabric with full bisectional-bandwidth, like the NVSwitch or IBSwitch fabrics in DGX-2 and NDv2 systems, nodes can simultaneously communicate at the full bandwidth of their ingress or egress links. However, as the number of connections through a switch, originating from a single GPU or NIC increases, the resulting queuing delays increase the latency. Figure 5.2 plots the accumulated ingress/egress bandwidth of exchanging varying volume of data (upto 200-400 MB) for different number of connections over NVSwitches in a DGX2 node (left) and over IBSwitches among four DGX2 nodes (right). In both cases, the bandwidth drops as the number of connections increases despite the volume of data remaining constant. However, for small input sizes, the difference for different number of connections is not significant. In TACCL, a logical topology does not model switches and therefore, the effect of number of connections cannot be captured.

TACCL addresses this performance impact by *switch-hyperedges* in the synthesizer as a way to control the number of connections to between GPUs and switches. A switch-hyperedge replaces a switch by a set of direct links in a logical topology that will be imposed for the entire runtime of an algorithm. Note that the synthesizer still has the freedom to search over which direct links it will impose. To control the number of direct links for each switch-hyperedge, TACCL provides three policies for a user: (1) *maximize* the number of links (`uc-max`), (2) *minimize* the number of links (`uc-min`), and (3) ignore freely choose any number of links. These policies are enforced by adding the number of unique connections over the switch to the objective function.

**Example 5.4.2** (Sketching for congestion)**.** Figure 5.3 shows a physical topology of three GPUs connected by a switch, where each GPU can communicate with any other GPU.

Figure 5.4 shows a logical topology with a switch-hyperedge that TACCL may choose with maximizing number of connections policy. This is desired for small data sizes that result in low likelihood of congestion at the switch with large number of connections as Figure 5.2 shows.

In Figure 5.5 TACCL has minimized number of connections, which effectively results in a Ring topology. This is more desired for larger data sizes, as restricting the number of logical connections limits the congestion in the switch (Figure 5.2).

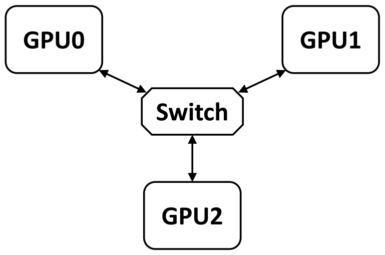


Figure 5.3: Physical topology with a switch

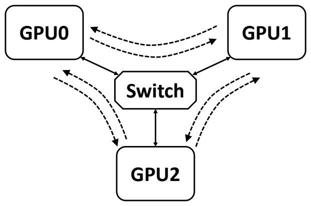


Figure 5.4: Connections with maximizing strategy

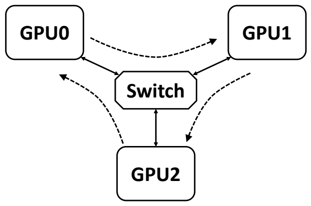


Figure 5.5: Connections with minimizing strategy

**Algorithm Symmetry.** Many collective communication algorithms are symmetric in a variety of ways. For example, ring algorithms follow a ring symmetry or in hierarchical algorithms, the local phases inside all machines are symmetric to each other. Inspired by this, TACCL offers a generic way to enforce algorithms to be symmetric.

The user may enforce a symmetry by supplying an *automorphism* of the logical topology and collective, i.e., a permutation of the ranks and chunks that maintains the structure of the topology and the collective pre- and post-conditions, and a *partition* of the ranks such that the automorphism maps each subset of ranks to some subset of the partition. TACCL will then restrict synthesis to algorithms with the same symmetry for all chunk transfers.

**Example 5.4.3.** Consider a cluster of two NDv2 systems and the task of synthesizing an Allgather. A hierarchical symmetry may be specified with an automorphism composed of a the permutation $[8, \ldots, 15, 0, \ldots, 7]$ for both chunks and ranks, and a partition $\{\{0, \ldots, 7\}, \{8, \ldots, 15\}\}$. Now if an algorithm performs a send of chunk 0 from rank 0 to rank 1, then it must also include a send of chunk 8 from rank 8 to rank 9. However, sends between GPUs in different NDv2s, e.g., between 0 and 8, are not affected by the symmetry.

Since the internal topologies of NDv2 systems are identical, enforcing this symmetry is reasonable and helps TACCL scale to larger distributed topologies. Meanwhile, TACCL still has the freedom to synthesize the top-level algorithm and connect the systems to each other as it best can.

**Writing a communication sketch.** A communication sketch comprises of a logical topology, switch-hyperedge strategy, symmetry information, input size, and other hyperparameters. We give an example of how users can provide an intuitive communication sketch input to the TACCL synthesizer in Appendix B.1.

### 5.4.2 Synthesizer

Once the user has written a communication sketch, they are ready to call TACCL's synthesizer. This section describes the synthesis process TACCL uses, as well as additional hyperparameters available to the user.

GPUs participating in a communication collective partition their initial data into $C$ equal chunks where $C$ is a hyperparameter selected by the user. TACCL's synthesizer routes and schedules these chunks. Given a communication sketch and a collective, the synthesizer decides chunk transfer schedules across every link in the network, such that each chunk reaches its destination GPUs as specified by the collective.

TACCL encodes this problem as a mixed integer linear program (MILP) with binary and continuous decision variables. The encoding has a continuous variable called *start_time* for every chunk and GPU to indicate when a chunk is available at a GPU. A binary variable *is_sent* for all chunk and link pairs denotes if a chunk is sent over a link. Another continuous variable *send_time* indicates when a chunk is sent over a link. The encoding has bandwidth and correctness constraints to ensure the correctness of a chunk transfer schedule. The objective of the MILP is to minimize *time* which is a continuous variable indicating the maximum time among all chunks that must reach their destination GPUs.

Additionally, TACCL's synthesizer also decides if it should merge some chunks and transfer them contiguously as one large buffer over a link. Sending $n$ chunks contiguously in one send instruction over a link requires paying only one $\alpha$ latency cost whereas sending $n$ chunks one after the other requires paying $n \times \alpha$ latency costs. Note that this does not change the $\beta$ bandwidth cost. However, sending $n$ chunks separately over a link enables TACCL to order them such that subsequent dependent sends from the destination of the link could be scheduled earlier. TACCL's synthesizer navigates this trade-off to minimize the time. TACCL uses this feature only for IB transfers due to their high $\alpha$ cost and ignores it for NVLinks due to their lower latency.

MILP problems in general are NP-hard. Luckily, there are solvers such as Gurobi [50] that apply heuristics to solve MILPs in a feasible way. However, this requires careful consideration regarding the number of variables and constraints in the formulation. In TACCL's formulation, transferring chunks over a link cannot overlap and an ordering among them is required. Therefore, potentially a binary decision is needed for every pair of chunks that may traverse a link. If we assume there are $C$ chunks for a collective problem, there are $O(C^2)$ such decisions per link. Moreover, as the number of nodes increase, the number of links increase linearly (larger topology) and the number of chunks for a collective increases linearly (ALLGATHER) or even quadratically (ALLTOALL). This large set of variables and constraints leads to infeasible solver time and memory requirements.

To solve this problem, we divide the synthesis into three parts. First, the synthesizer solves an optimization problem to determine the path used by every chunk without fixing any ordering among chunks, then it heuristically orders the chunks over every link, and finally, it solves another optimization problem to determine chunk contiguity. Complete formal descriptions of each step are discussed later in § 5.5.

**Step 1: Routing** solves a MILP for finding the path of each chunk independent of other chunks, allowing chunks sent over a link to overlap. The objective of this MILP is to minimize the time, which we constrain to be the maximum of two sets of variables. (1) for each link, the number of chunks that traverse that link multiplied by the transfer time of a chunk over that link. (2) for the path of each chunk, the summation of transfer times of the chunk along every link in the path. Note that this is only a lower bound on the time since we do not consider link contention or chunk ordering. According on the terminology introduced by Leighton, Maggs, and Rao in their seminal paper [72] that studied the issue of scheduling packets given their paths, our first variable provides a measure of *congestion* and our second variable provides a measure of *dilation*. In that paper, the authors proved that for store-and-forward packet routing, it is possible to schedule data transfers in $O(C + D)$ time when given paths with congestion $C$ and dilation $D$. Inspired by this, we obtain paths that minimize both congestion and dilation.

TACCL also constrains each chunk's path to be via GPU ranks that are on the shortest paths from their sources to their destinations using the links the user decided to include in the logical topology. If the communication sketch specifies an algorithm symmetry, TACCL adds the constraints for the symmetric sends. Replacing switches with switch-hyperedges is also applied in this step. For each switch-hyperedge, a user-provided policy on the number of unique connections to/form a switch is applied. TACCL can also add switch-hyperedge policies on its own based on the chunk size - if the chunk size is larger than 1 MB, TACCL uses `uc-min` strategy, and otherwise uses `uc-max`.

TACCL uses Gurobi [50] to solve this MILP and the solution gives every chunk a start_time for each GPU along its path. Clearly this step solves chunk routing, but only partially solves the chunk scheduling and contiguity problem and requires follow-up steps (explained next) to account for ordering the chunks sent over a link as well as minimizing $\alpha$ costs of sends. However, by using this technique, TACCL's synthesizer is able to reduce binary variables needed from $O(C^2)$ to $O(C)$ per link.

**Step 2: Heuristic Ordering** decides the chunk ordering sent on each link based on a heuristic. Note that this step is not an MILP and solely solved by a greedy algorithm. Regardless of when each chunk becomes available at a GPU, this step assigns a total order on the chunks sent over a link $l = (src, dst)$. This is decided by two heuristic functions. (1) chunks which need to traverse the longest path from *src* to their final GPU, have higher priority. (2) In case there is tie in (1), chunks which have traversed the shortest path from their initial GPU to *src*, have higher priority. This ordering will be used in Step 3 to assign the final schedules.

**Step 3: Contiguity and Exact Scheduling** solves an MILP problem to decide which chunks to send contiguously and gives the exact schedule. The path to be taken by chunks and their ordering over links have already been determined by the previous steps which are added as constraints to this MILP. The start_time and send_time variables are reassigned in this step by considering both the $\alpha$ and $\beta$ costs for each transfer. In this step, the synthesizer allows either sending one chunk at a time or sending multiple chunks contiguously. This offers a trade-off between (1) sending the chunks that are available at the same time for a link according to the ordering in step 2 so that the subsequent sends can be scheduled earlier or (2) sending the chunks contiguously in one send instruction to save the latency cost. The objective of this MILP is to minimize the total time by enforcing all constraints which in TACCL solved by Gurobi [50]. The solution gives the exact schedule for each chunk.

**Synthesizer Hyperparameters.**

TACCL's synthesizer has some additional parameters that control the synthesis process. These are provided by the user to the synthesizer through the communication sketch.

- Buffer Size: TACCL needs the size of input/output buffers of a collective for the $\alpha$-$\beta$ cost model. In ML workloads the input/output buffer size is a known fixed value.

- Chunk Partitioning: The data buffer at each GPU at the start of the collective can be partitioned into multiple equal chunks. Each chunk is considered as an atomic scheduling unit by the synthesizer and different chunks of the same data buffer can be routed over different links. The semantics of a collective forces a minimum number of chunks such as ALLTOALL which needs at least as many chunks as the number of GPU for each buffer. On one hand, using the minimum number of chunks is often times ideal for finding latency-optimal algorithms. On the other hand, providing a higher number of chunks allows the synthesizer to better utilize the links that might be idle otherwise which is better for finding bandwidth-optimal algorithms.

## 5.5 Synthesizer Formulation

As explained earlier, TACCL's synthesizer has routing, heuristic ordering, and contiguity and exact scheduling stages. We provide a detailed description of each of these stages in this section. We first formally introduce some terms that we will use later. Let $\mathcal{C}$ denote the set of chunks that are required to be routed in the algorithm for collective *coll*. Let $\mathcal{R}$ denote the set of GPU ranks involved in *coll*. Let *coll*.precondition and *coll*.postcondition denote the precondition and post-condition of the collective respectively.The tuple $(c, r) \in coll.\text{precondition}, c \in \mathcal{C}, r \in \mathcal{R}$, if chunk $c$ is present at rank $r$ at the start of the collective. Similarly, the $(c, r) \in coll.\text{postcondition}$ if chunk $c$ has to be present at rank $r$ at the end of the collective. Further, let $\mathcal{L}$ denote the set of links, such that $(r1, r2) \in \mathcal{L}, r1 \in \mathcal{R}, r2 \in \mathcal{R}$ if there exists a link from rank $r1$ to rank $r2$ in the logical topology determined by the topology and communication sketch. Let $\mathcal{S}_r^{send}$ denote the set of switched destinations for rank $r$, such that $dst \in \mathcal{S}_r^{send}$ if link $(r, dst)$ is a part of a switch-hyperedge. Similarly, $\mathcal{S}_r^{recv}$

denotes the set of switched sources for rank $r$, such that $src \in \mathcal{S}_r^{recv}$ if link $(src, r)$ is a part of a switch-hyperedge. $\alpha(r1, r2)$, $\beta(r1, r2)$ are the alpha and beta costs respectively of the link $(r1, r2) \in \mathcal{L}$. The term $lat(r1, r2)$ is the sum of $\alpha(r1, r2)$ and $\beta(r1, r2)$ cost of the link, which denotes the total transfer cost of a single chunk over link $(r1, r2)$. Table 5.2 lists the variables that the TACCL's synthesizer solves for. We will describe each variable in detail in this section.

### 5.5.1 Routing

The main aim of the routing stage is to give us the path that every chunk takes in the collective. Our objective is to minimize the time (denoted by continuous variable *time*) it takes to reach the post-condition of the collective.

$$\text{Minimize} \quad time \tag{5.1}$$

The time taken for the collective algorithm is the latest time at which a chunk becomes available on a rank that is in the post-condition of the collective. We use a continuous variable $start[c, r]$ to denote the time that chunk $c$ becomes available on rank $r$, and end up with the following constraints for *time*

$$time \geq start[c, r] \quad \forall (c, r) \in coll.\text{postcondition} \tag{5.2}$$

For chunks on ranks that belong to the collective's precondition, we set the start time to zero.

$$start[c, r] = 0 \quad \forall (c, r) \in coll.\text{precondition} \tag{5.3}$$

We also add correctness constraints in our formulation for routing - chunks are sent from a GPU rank only after they have been received on that rank. We introduce a continuous variable $send[c, src, r]$ to denote the time of sending chunk $c$ from rank $src$ to rank $r$ and add the following constraint to our formulation:

$$send[c, src, r] \geq start[c, src] \quad \forall c \in \mathcal{C} \quad \forall (src, r) \in \mathcal{L} \tag{5.4}$$

We use a binary variable $is\_sent[c, src, r]$ to indicate if chunk $c$ is sent over the link $(src, r)$ in our algorithm. We note that the routing stage does not strictly respect bandwidth constraints of any link - the generated solution may send two chunks simultaneously over a link at the time cost of one chunk. The chunk start time on a rank will be determined only by the chunk send time on the source, independent of other chunk transfers on the link (eq. 5.5). LHS→RHS in the equation signifies an indicator constraint, i.e., if LHS is 1, RHS will hold.

$$\begin{aligned} is\_sent[c, src, r] \rightarrow start[c, r] &= send[c, src, r] + lat(src, r) \\ &\forall c \in \mathcal{C} \quad \forall (src, r) \in \mathcal{L} \end{aligned} \tag{5.5}$$

Instead of bandwidth constraints, this encoding uses *relaxed bandwidth constraints.* They are expressed by aggregating the link transfer time of all chunks sent over a link and using it to to lower bound the total time of the algorithm (eq. 5.6). For switched connections, the total time is lower bounded by the sum of link transfer times of all chunks sent over all switched outgoing links from a source, and also by the sum of link transfer times for chunks received from all incoming links to a destination (eq. 5.7 and eq. 5.8).

$$time \geq \sum_{c \in C} (lat(src, r) * is\_sent[c, src, r]) \quad \forall (src, r) \in \mathcal{L} \tag{5.6}$$

$$time \geq \sum_{c \in C} \sum_{dst \in \mathcal{S}_r^{send}} (lat(r, dst) * is\_sent[c, r, dst]) \quad \forall r \in \mathcal{S}_{send} \tag{5.7}$$

$$time \geq \sum_{c \in C} \sum_{src \in \mathcal{S}_r^{recv}} (lat(src, r) * is\_sent[c, src, r]) \quad \forall r \in \mathcal{S}_{recv} \tag{5.8}$$

Based on the communication sketch, we also add constraints for `uc-max` and `uc-min` strategies for switch-hyperedges to maximize and minimize the number of links utilized in a switch respectively. We introduce a new binary variable $is\_util[src, r]$ for links $(src, r)$ that are a part of a switch-hyperedge. This variable is 1 if any chunk is sent over link $(src, r)$, and 0 otherwise.(eq. 5.9 and eq. 5.10). According to the switch-hyperedge strategy, we add this variable, weighted with a small constant $\gamma$, to the objective function (eq. 5.11). $\gamma$ is negative for `uc-max` and positive for `uc-min`.

$$is\_util[src, r] >= is\_sent[c, src, r] \quad \forall c \in \mathcal{C} \forall (src, r) \in \mathcal{L} \tag{5.9}$$

$$is\_util[src, r] <= \sum_{\forall c \in \mathcal{C}} is\_sent[c, src, r] \quad \forall (src, r) \in \mathcal{L} \tag{5.10}$$

$$\text{Minimize} \quad time + \gamma \times \left( \sum_{(src,r):\text{switched links}} is\_util[src, r] \right) \tag{5.11}$$

We also add symmetry constraints according to the symmetry offsets provided by user in the communication sketch. For a chunk $c$ and link $(src, r)$, we identify a rotationally symmetric chunk $\hat{c}$ and link $(\hat{src}, \hat{r})$ and add the following constraints:

$$start[c, r] = start[\hat{c}, \hat{r}] \tag{5.12}$$

$$send[c, src, r] = send[\hat{c}, \hat{src}, \hat{r}] \tag{5.13}$$

$$is\_sent[c, src, r] = is\_sent[\hat{c}, \hat{src}, \hat{r}] \tag{5.14}$$

Further, for chunks that start on one node and have a final destination on another node, we add inter-node transfer constraints which specify that at least one inter-node link will be used to transfer that chunk.

$$\sum_{(r_1,r_2)\in\mathcal{L}:r_1\in\text{node}_1,r_2\in\text{node}_2} is\_sent[c, r_1, r_2] \geq 1 \tag{5.15}$$

### 5.5.2 Ordering Heuristics

We start the heuristic ordering by determining the paths each chunk takes using the solution of the path encoding. We then consider the first link in every path as a candidate for scheduling a chunk transfer. Using heuristics like *chunk-with-shortest-path-until-now-first* and *chunk-with-longest-path-from-now-first*, we select a

| MILP Variables | Explanation |
|---|---|
| **Routing** | |
| *time* | time spent in the collective algorithm |
| $start[c, r]$ | time at which chunk $c$ becomes available at GPU $r$ |
| $send[c, src, r]$ | time at which chunk $c$ is sent from GPU $src$ to GPU $r$ |
| $is\_sent[c, src, r]$ | indicates if chunk $c$ is sent from GPU $src$ to GPU $r$ |
| $is\_util[src, r]$ | indicates if any chunk is sent from GPU $src$ to GPU $r$ |
| **Contiguity** | |
| $is\_together[c, o, r]$ | indicates if chunks $c$ and $o$ are sent to GPU $r$ together from the same source, thus sharing the bandwidth and reducing the latency cost of transfer |

Table 5.2: Variables used in TACCL's MILP formulation. Variables with prefix *is_* are binary variables and others are continuous variables.

path (and thus a chunk) which should be scheduled in this round. We keep a running estimate of link time, which is the earliest time at which a chunk can be scheduled over the link. We also keep a running estimate of chunk time, which is the earliest time at which the next link transfer can be scheduled for a chunk. At the start, the link time for every link is 0 and the chunk time for every chunk is 0. When a path is chosen in the first round, the chunk associated with the path is scheduled to traverse the first link in the path. The link time of that link increases by link latency and chunk time of that chunk increases by link latency. The link candidate from the selected path is also updated to be the next link in the path. For the next rounds, we decide which path's candidate link to schedule next using the tracked link and chunk times along with the scheduling heuristics. This keeps going until we have scheduled a data transfer over all the links in all the paths. We find that the best heuristics differ for architectures with NVLinks and those with NVSwitches, in terms of whether to start selecting links to schedule in the same order as the paths or in the opposite order of the paths. The heuristic ordering has the following three outputs:

- chunk_order$(r_1, r_2)$, an ordered list of chunks transferred along each link $(r_1, r_2)$. If chunk $c_1$ is present before chunk $c_2$ in chunk_order$(r_1, r_2)$, it denotes that $c_1$ is scheduled to be sent before $c_2$ over link $(r_1, r_2)$.

- switch_send_order($r$), an ordering on the chunks sent from a switch source $r$ to any of the switch destinations $dsts$. If $(c_1, dst_1)$ is present before tuple $(c_2, dst_2)$ in switch_send_order($r$), it means that a send of $c_1$ over link $(r, dst_1)$ should be scheduled before a send of chunk $c_2$ over link $(r, dst_2)$.
- switch_recv_order($r$), an ordering on the chunks received on a switch destination $r$ from any of the switch sources $srcs$. If $(c_1, src_1)$ is present before tuple $(c_2, src_2)$ in switch_recv_order($r$), it means that a receive of $c_1$ over link $(src_1, r)$ should be scheduled before a receive of chunk $c_2$ over link $(src_2, r)$.

### 5.5.3 Contiguity and Exact Scheduling

Finally, we describe the formulation for the contiguity and exact scheduling stage. Given the link and switch ordering from the heuristic ordering stage, the aim of this stage is to find the sweet spot in the trade-off between lower link latency by sending multiple data chunks contiguously as a big data chunk and reduced pipelining benefits due to the big data-chunk transfer. We provide the main set of constraints in our formulation below, leaving out other less important constraints.

Our objective is still to minimize the time of the collective and constraints eq. 5.1-eq. 5.4 must still hold in this formulation. We add a new binary variable $is\_together(c_1, c_2, r)$ for all chunks $c_1$ and $c_2$ that are sent over the same link to rank $r$. If $is\_together(c_1, c_2, r)$ is 1, chunks $c_1$ and $c_2$ are sent as a single data-chunk over a link to rank $r$.

$$\begin{gathered} is\_together[c, o, r] \rightarrow send[c, src, r] = send[o, src, r] \\ \forall c, o \in \text{chunk\_order}(src, r) \quad \forall (src, r) \in \mathcal{L} \end{gathered} \tag{5.16}$$

The transfer time of a data chunk $c$ along a link $(src, r)$ will be determined by all other data chunks that it has to travel together with:

$$\begin{aligned} lat[c, src, r] = &\alpha(src, r) + \beta(src, r) * \\ &\Big( \sum_{o \in \text{chunk\_order}(src, r)} is\_together[c, o, r] \Big) \\ &\forall c \in \text{chunk\_order}(src, r) \quad \forall (src, r) \in \mathcal{L} \end{aligned} \tag{5.17}$$

$$\begin{aligned} start[c,r] =& send[c,src,r] + lat[c,src,r] \\ & \forall c \in \text{chunk\_order}(src,r) \quad \forall (src,r) \in (L) \end{aligned} \tag{5.18}$$

We also add strict bandwidth constraints for this formulation, allowing only one data chunk per link transfer time if the data chunks are not sent contiguously over the link. Let $pos(c, src, r)$ determine the position of chunk $c$ in the chunk_order$(src, r)$, then

$$\begin{aligned} \neg is\_together[c,o,r] \rightarrow send[o,src,r] \geq & send[c,src,r] \\ +lat[c,src,r] \quad & \forall c \in \text{chunk\_order}(src,r) \\ & \forall o \in \text{chunk\_order}(src,r) \\ \text{if} \quad \text{pos}(o,src,r) \geq \text{pos}(c,src,r) \quad & \forall (src,r) \in \mathcal{L} \end{aligned} \tag{5.19}$$

Similarly, we add bandwidth constraints for switch, allowing a source to send data to only one switched destination at a time, and a receiver to receive data from only one switched sender at a time. Let $sw - pos - send(c, r, dst)$ determine the position of tuple $(c, dst)$ in the switch_send_order$(r)$, and let $sw - pos - recv(c, src, r)$ determine the position of tuple $(c, src)$ in the switch_recv_order$(r)$, then,

$$\begin{aligned} send[o,r,dst_o] \geq & send[c,r,dst_c] + lat[c,r,dst_c] \\ & \forall (c,dst_c) \in \text{switch\_send\_order}(r) \\ & \forall (o,dst_o) \in \text{switch\_send\_order}(r) \\ \text{if} \quad \text{sw-pos-send}(o,r,dst_o) \geq & \text{ sw-pos-send}(c,r,dst_c) \\ & \forall r \in \mathcal{S}^{send} \end{aligned} \tag{5.20}$$

$$\begin{aligned} send[o,src_o,r] \geq & send[c,src_c,r] + lat[c,src_c,r] \\ & \forall (c,src_c) \in \text{switch\_recv\_order}(r) \\ & \forall (o,src_o) \in \text{switch\_recv\_order}(r) \\ \text{if} \quad \text{sw-pos-recv}(o,src_o,r) \geq & \text{ sw-pos-recv}(c,src_c,r) \\ & \forall r \in \mathcal{S}^{recv} \end{aligned} \tag{5.21}$$

## 5.6 Backend

The synthesizer described above generates an abstract algorithm that specifies the order in which the nodes communicate the various chunks. The goal of the backend is to implement this abstract algorithm. To do so, we extend NCCL [85] with an *interpreter* which we call TACCL runtime. While any communication algorithm can be trivially implemented using NCCL's point-to-point sends and receives, TACCL runtime enables us to execute the entire algorithm in a single kernel launch, eliminating multiple launch overheads. In addition, by reusing NCCL transport mechanisms, TACCL runtime is able to support all of NCCL's communication backends such as IB, Ethernet, NVLink, and PCIe.

### 5.6.1 TACCL runtime

The input to TACCL runtime[2] is a TACCL-EF program, which is an XML format for representing collective algorithms. TACCL-EF programs operate on three buffers: input, output and scratch. For each buffer, the program specifies the number of chunks it will be sliced into such that all chunks are equal size. Every step of the algorithm is expressed in terms of these chunks.

The program is divided into a set of GPU programs made up of threadblocks. Each threadblock is made up of a series of steps that are executed sequentially, with each step specifying an instruction and operands as indices into the input/output/scratch buffers. The current instruction set includes sends, receives (with optional reduction), and local copies. To simplify the implementation of TACCL runtime, each threadblock can send to and receive from at most one GPU. Additionally, threadblocks within a GPU can synchronize by indicating that one step depends on another step, which will cause the interpreter to wait until the dependency has completed before executing the dependent step.

[2]Link to code: `https://github.com/microsoft/msccl`

The TACCL runtime extends NCCL and it is backward compatible with its API. Therefore, integrating TACCL runtime into machine learning frameworks such as PyTorch is a single line change wherein that change swaps the third-party NCCL library for TACCL runtime. This allows TACCL to dynamically swap in collective algorithms generated for any training/inference workload using `torch.distributed`.

### 5.6.2 Lowering to TACCL runtime

To target TACCL-EF, abstract algorithms are lowered to the executable format. The sets of sends operating on abstract chunks that comprise the steps of the algorithm are transformed into pairs of send and receive operations operating on concrete buffer indices. Furthermore, these operations are placed sequentially into threadblocks and any necessary dependencies recorded between them.

**Buffer allocation.** Input and output buffers are preallocated by the user and passed to the collective. Scratch buffers are allocated by the TACCL runtime per TACCL-EF. Chunks are indices in the input, output and scratch buffers. For chunks that are common for both the input and the output buffers (e.g. as in ALLGATHER) a local copy from input to the output buffer is performed at the end.

**Instruction generation.** The operations of the abstract algorithm are split into two instructions for the sender and receiver GPU, and chunks are translated into buffer references and indices according to the buffer allocation.

**Dependency insertion.** TACCL transforms a synthesized algorithm into the asynchronous execution model of TACCL-EF and dependencies for each buffer index are inserted to ensure that the data dependencies present in the abstract algorithm are honored.

**Threadblock allocation.** Instructions are grouped such that all of them are either sending to at most one GPU and/or receiving from at most another GPU (possibly different). Order of the instructions inside a group should follow the order of the abstract algorithm. TACCL allocates a threadblock for each group of instructions.

**Instances.** NCCL and consequently TACCL runtime cannot saturate the bandwidth of a link in a topology using a single threadblock. Thus, TACCL generates multiple instances of the algorithm to maximize the performance. This is done by subdividing each chunk into $n$ subchunks that follow the same path as the parent chunk. All groups of instructions and their threadblocks are duplicated $n$ times and executed in parallel. § 5.8.3 explores the performance implications of choices of $n$.

## 5.7 Discussion

We reflect on our experiences building TACCL, describe problems we encountered, how we solved them, and how TACCL can be used for inference.

**Representing switches in continuous encoding.** It is difficult to model how link bandwidth will be shared over switches when encoding time as a continuous variable (as is done in TACCL). To adress this challenge, we do not allow a GPU to send data to two different GPUs on the switch at the same time in our encoding.

**Synthesizing combining collectives.** Combining collectives are collectives that combine chunks like REDUCESCATTER and ALLREDUCE. It is not straightforward to encode combining collectives in a formulation. TACCL synthesizes combining collectives by utilizing synthesis of non-combining collectives, similar to the technique used by SCCL [18]. REDUCESCATTER can be implemented as an "inverse" of ALLGATHER— a send from a source GPU in ALLGATHER is instead received and reduced on the source GPU. However, simply inverting the sends does not work — a GPU may simultaneously send on different links in an ALLGATHER, but it cannot reduce all receives together in the inverse case. We thus order the inverse sends using heuristic ordering followed by contiguity encoding in order to synthesize REDUCESCATTER. ALLREDUCE is synthesized directly by concatenating REDUCESCATTER with an ALLGATHER algorithm.

**Applicability of TACCL for inference.** In the evaluation presented later for end-to-end models with TACCL, we only show results for training. However, Large

Language Models (LLMs) are extremely large with hundreds of billions of parameters [105, 16] and their inference also needs to be distributed between various multi-GPU machines.

Simply throwing money at the problem and scaling the execution hardware is not enough. Quoting a Senior AI Scientist at NVIDIA, Dr. Jim Fan, ”In the future, every 1% speedup on LLM inference will have similar economic value as 1% speedup on Google Search infrastructure” [41]. We expect that LLM inference workloads will spend more time waiting on network communication because they have lighter computation requirements than training workloads. Further, with the increasing scale of distribution, the size of data to be communicated reduces, making latency-aware collective algorithms important.

Since TACCL algorithms can reduce network communication overhead and can generate input-aware collective algorithms with the help of communication sketches and the $\alpha - \beta$ model, we expect it to provide significant performance improvements, leading to high cost savings.

## 5.8 Evaluation

In this section, we evaluate algorithms obtained with TACCL for various collectives and hardware. We seek to answer the following question:

- How do TACCL collective algorithms perform against state-of-the-art NCCL?
- How is the performance of algorithms synthesized by TACCL impacted with changing synthesizer inputs?
- Do faster TACCL algorithms impact end-to-end training deep networks?
- How much time is required for synthesizing algorithms using TACCL?

We briefly describe our experimental setup before addressing each of the above questions.

### 5.8.1 Experimental Setup

We evaluate algorithms obtained with TACCL for Allgather, Alltoall, and Allreduce collectives on a cluster of 32 GPUs comprised of two Nvidia DGX-2 nodes or up to four Azure NDv2 nodes. To compare performances, algorithm bandwidth [81] measurement is used which is calculated by input buffer size divided by execution time. We synthesize TACCL algorithms by exploring different communication sketches and comparing them against the popular Nvidia Collective Communication Library (NCCL) (v.2.8.4-1) (§ 5.8.2). We also analyze how different communication sketches impact the performance of the algorithms synthesized by TACCL. In particular, we perform ablation studies by varying the inter-node connections in the logical topology, changing synthesizer hyperparameters, and changing the number of instances used when lowering to TACCL-EF (§ 5.8.3). To evaluate how TACCL's speedups translate to end-to-end performance, we use algorithms generated by TACCL in two large language models, Transformer-XL and BERT (§ 5.8.4). Finally, we discuss the synthesis time required by TACCL to generate these algorithms (§ 5.8.5). All our communication sketches for DGX-2 and NDv2 use a hierarchical symmetry like the one in Example 5.4.3.

We believe our focus on up to 32 GPUs covers a large section of important use cases: in an internal cluster of DGX-2 nodes at Microsoft, the sum of GPUs in jobs of at most 32 was 93.7% of all jobs in the second half of 2021.

### 5.8.2 Standalone Experiments

#### 5.8.2.1 Allgather

**Allgather on DGX-2.** Figure 5.6(i) shows the algorithm bandwidth for TACCL's synthesized algorithms on two DGX-2 nodes for each output buffer size and plots it against that of NCCL. We show the speedup of TACCL's algorithms over NCCL on the right Y-axis of the plot. We used two different sketches for this topology which will be explained next.

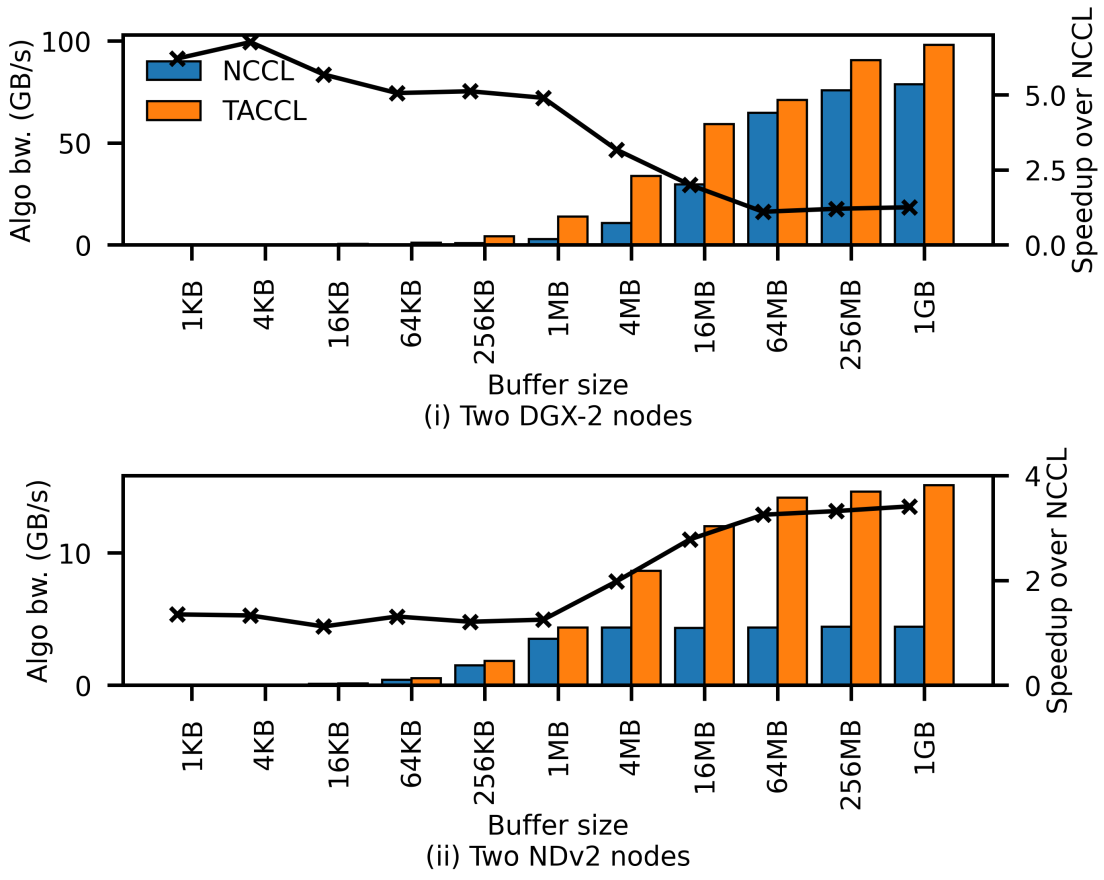


Figure 5.6: Allgather comparisons of NCCL to TACCL's best algorithm at each buffer size.

A DGX-2 node has 16 V100 GPUs (Figure 3.4) where each pair of GPUs share a PCIe switch with a NIC. This makes it natural to assign one GPU in a pair to be a receiver and the other to be a sender by eliminating outgoing and incoming links, respectively, in the logical topology. We design a sketch (*dgx2-sk-1*) that uses this logical topology, sets chunk size to 2MB, uses two chunk partitions for each buffer, and the sets switch-hyperedge policy to `uc-min`. With this sketch, TACCL synthesizes an Allgather algorithm for two DGX-2 nodes. This algorithm almost saturates the inter-node bandwidth during the entire run of the algorithm and provides a 20%−25% speedup over NCCL for large buffer sizes in the 256MB - 1GB range.

Next, we design a sketch (*dgx2-sk-2*) for smaller sizes. This sketch allows both GPUs in a pair to utilize the shared NIC. However, local GPU $i$ on each node is only allowed to send/receive to/from local GPU $i$ on the other node. Since the IB is shared, we double the $\beta$ cost for each IB transfer to $2 * \beta_{IB}$ cost. In this sketch, chunk size is set to 1KB and the switch-hyperedge policy is `uc-max`. Using this sketch TACCL synthesizes an algorithm that is $4.9\times - 6.7\times$ faster than NCCL in the 1KB - 1MB range, and $10\% - 3.8\times$ faster than NCCL in the 2MB - 64MB range. On inspecting this algorithm, we found that TACCL's synthesized algorithm overlaps inter-node sends with intra-node all-pair ALLGATHER of node-local data chunks followed by an intra-node all-pair ALLGATHER of the node-external chunks received over IB.

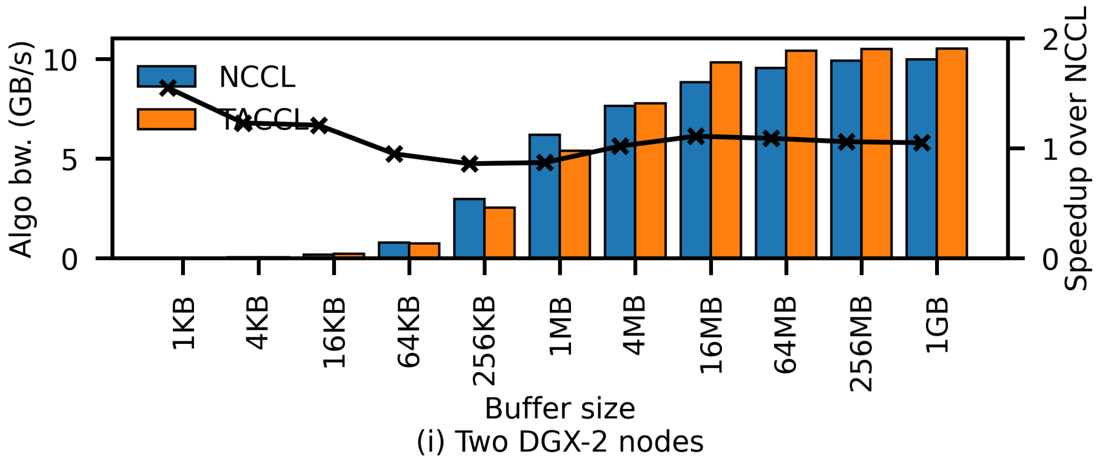
Figure 5.6(i) shows the algorithm bandwidth and the speedup over NCCL baseline for the best of these two sketches for each output buffer size.

**Allgather on NDv2.** The sketch we used, *ndv2-sk-1*, uses the logical topology discussed in Example 5.4.1, in which a sender and a receiver GPU were dedicated such that they are on the same PCIe switch as the NIC. We use a single instance when lowering algorithms into TACCL-EF for data sizes 1MB and below, and use 8 instances for larger data sizes. Figure 5.6(ii) compares the synthesized algorithms to NCCL on two Azure NDv2 nodes. TACCL's synthesized algorithms are $12\% - 35\%$ faster than NCCL for buffer sizes of 1KB - 1MB, and $61\% - 3.4\times$ faster than NCCL for sizes larger than 1MB. These algorithms better saturate the inter-node bandwidth thanks to the dedicated send/receiver GPUs.

We similarly synthesize ALLGATHER algorithms for four NDv2 nodes and present the results in Figure B.1(i) in Appendix B.2. These algorithms are $10\% - 2.2\times$ faster than NCCL depending on buffer size.

#### 5.8.2.2 Alltoall

**Alltoall on DGX-2.** We explore the synthesis of ALLTOALL algorithms by reusing the *dgx2-sk-2* communication sketch designed in the previous section. Figure 5.7(i)



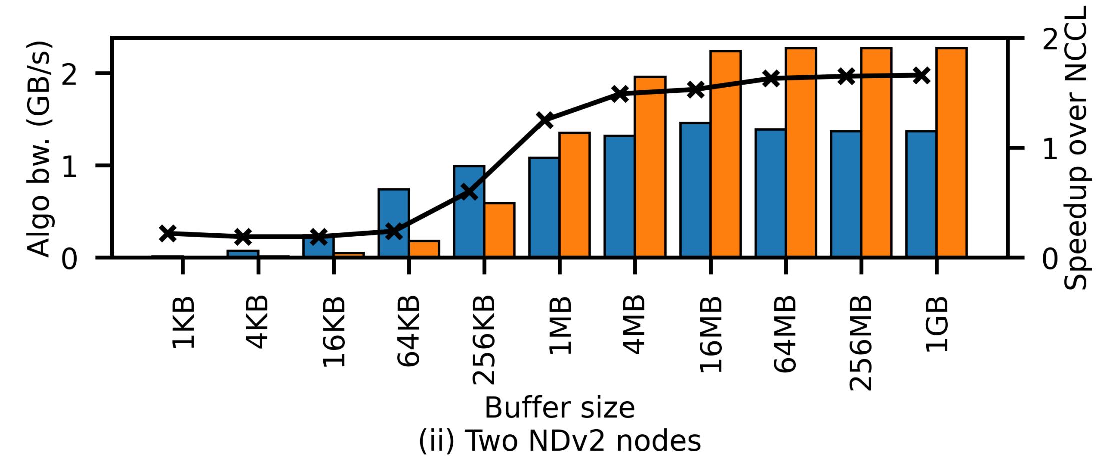


Figure 5.7: ALLTOALL comparisons of NCCL to TACCL's best algorithm at each buffer size.

compares the resulting algorithm on two DGX-2 nodes. The synthesized algorithm using this sketch performs up-to 15% faster than NCCL for batch sizes of 2MB and larger. For this sketch, TACCL's synthesizer coalesces chunks sent in inter-node transfer in this algorithm, which reduces the latency of transfers over IB. TACCL also uses a communication sketch with chunk size set as 1KB and a logical topology where GPUs have links to all other GPUs connected via the NIC (*dgx2-sk-3*). This algorithm is up-to 55% faster than NCCL for small buffer sizes ranging from 1KB to 16KB.

**Alltoall on NDv2.** Figure 5.7(ii) shows a comparison of TACCL's best algorithms for ALLTOALL on two Azure NDv2 nodes against NCCL. We reuse the communication sketch *ndv2-sk-1* and set the chunk size to 1MB. The generated algorithms run 53% − 66% faster than NCCL for buffer sizes between 16MB - 1$GB$ We explore another sketch (*ndv2-sk-2*) with a logical topology in which all GPUs in a node are fully-connected to all the GPUs in the other node and set chunk size as 1KB. The algorithm generated by TACCL using this sketch performs up-to 12% faster than NCCL for buffer sizes from 1KB to 128KB.

For four NDv2 nodes, TACCL's synthesized algorithms uses communication sketch *ndv2-sk-1* and they are up-to 46% faster than NCCL for buffer size greater than 1MB, as shown in Figure B.1(ii) in Appendix B.2.

#### 5.8.2.3 Allreduce

**Allreduce on DGX-2.** As discussed in § 5.7, TACCL composes REDUCESCATTER with ALLGATHER to implement ALLREDUCE and an algorithm for REDUCESCATTER can be constructed by inverting an ALLGATHER algorithm. Figure 5.8(i) shows the performance of TACCL algorithms on two DGX-2 nodes. The ALLREDUCE synthesized from the ALLGATHER using *dgx2-sk-2* is 49% − 6.4× faster than NCCL for buffer sizes ranging from 1KB - 4MB. TACCL's generated algorithms by using other communication sketches like *dgx2-sk-1* are 2% − 37% faster than NCCL for buffer sizes ranging from 16MB - 256MB. For buffer sizes of 512MB and greater, our algorithms are at most 9% slower than NCCL. This is because NCCL uses the more optimized fused communication instructions (such as receive-reduce-copy-send) in its ALLREDUCE communication which are unavailable in TACCL's lowering. We leave these such further optimizations for future work.

**Allreduce on NDv2.** These algorithms are based on the ALLGATHER synthesized from the *ndv2-sk-1* sketch and use two versions with 1 and 8 instances. Figure 5.8(ii) compares them to NCCL on two NDv2 nodes. The single instance TACCL algorithm

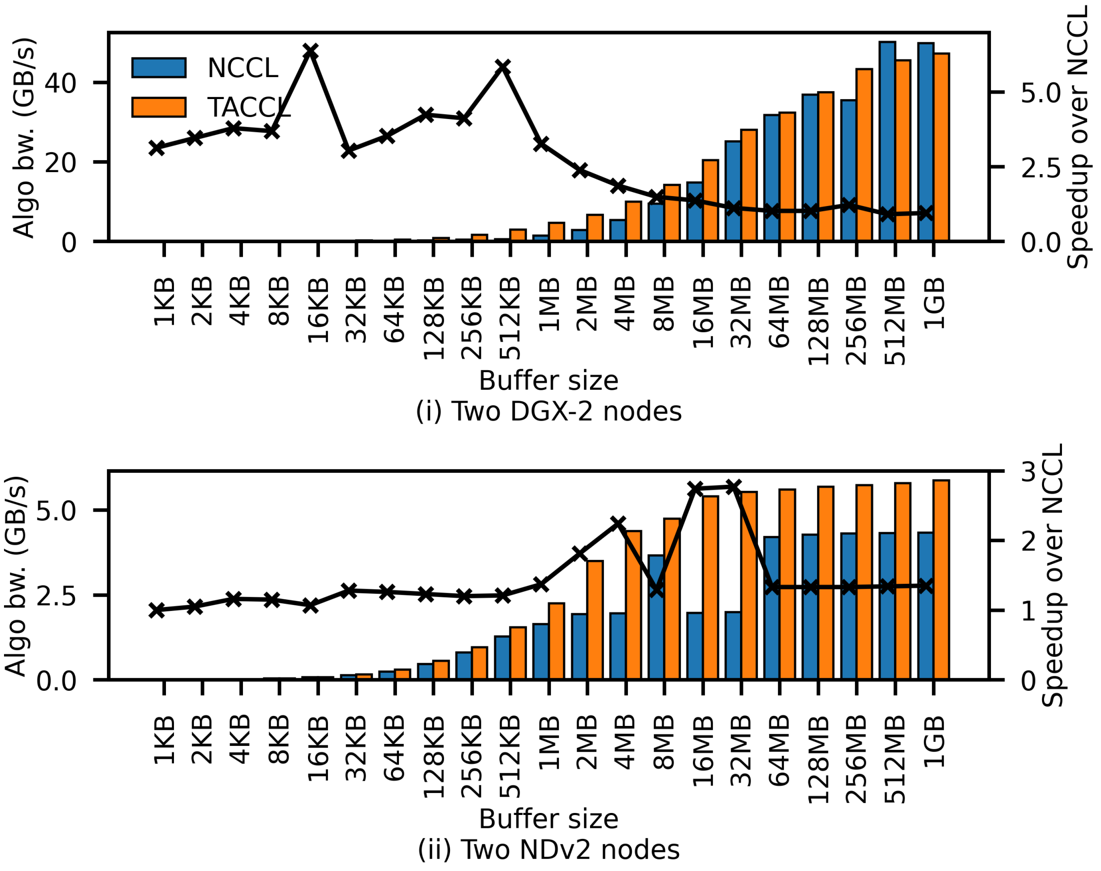


Figure 5.8: Allreduce comparisons of NCCL to TACCL's best algorithm at each buffer size.

outperforms NCCL's Allreduce by up to 28% for buffer sizes of up to 1MB, while the 8 instance algorithm outperforms NCCL by $28\% - 2.7\times$ for larger sizes.

On 4 NDv2 nodes, as shown in Figure B.1(iii) in Appendix B.2, the TACCL algorithms are up to 34% faster than NCCL for small buffer sizes and $1.9\times - 2.1\times$ faster than NCCL for larger buffer sizes.

### 5.8.3 Impact of Varying Synthesizer Inputs

In this section, we explore modifications to communication sketches, as well as the synthesizer hyperparameters and the instances for the lowering, in order to

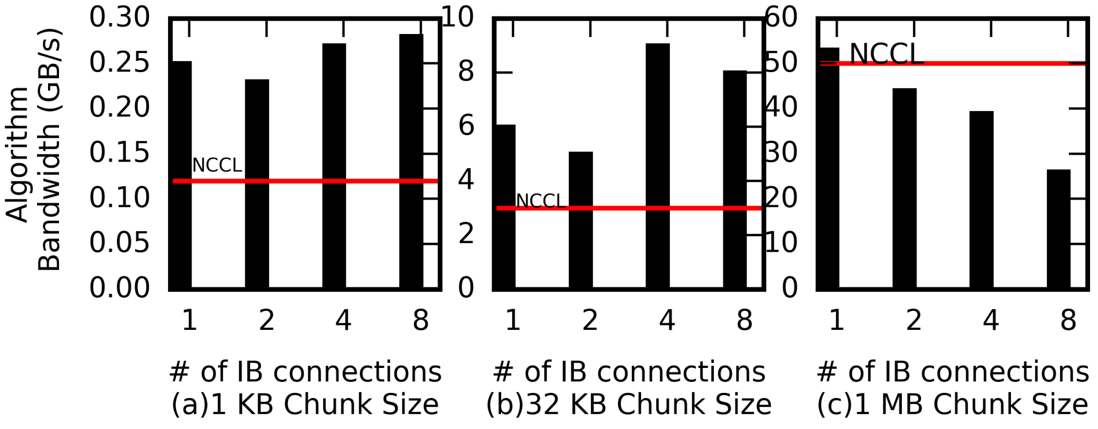


Figure 5.9: Logical topology

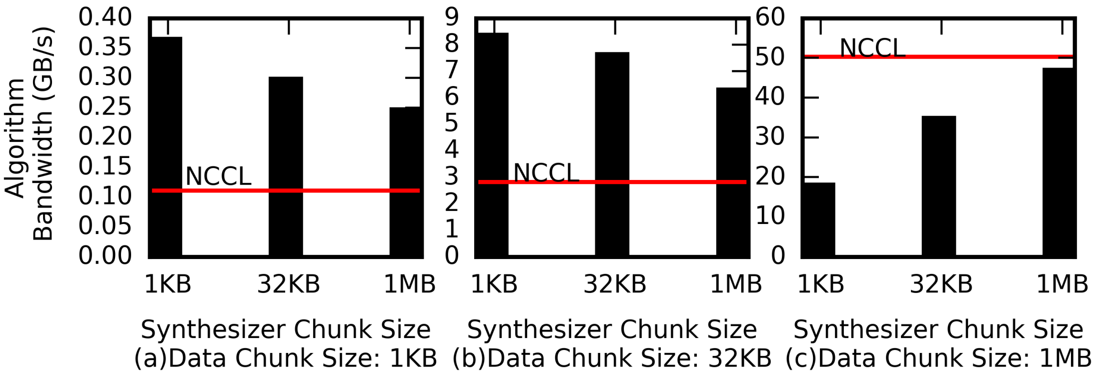


Figure 5.10: Chunk size

understand their impact on the performance of the synthesized algorithms. Our aim is to demonstrate that the controls offered by TACCL have intuitive effects on the resulting algorithms, which is necessary for effectively communicating user intuition to TACCL.

We present our analysis for the ALLGATHER collective on two Nvidia DGX-2 nodes. Unless mentioned otherwise, we use the following communication sketch as the baseline: same logical topology as *dgx2-sk-1*, chunk size set to 1MB, data partitioning set to 1, and the switch-hyperedge policy set to `uc-max`.

**Changing logical topology.** We create a logical topology with a dedicated sender

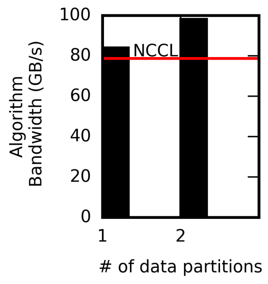


Figure 5.11: Data partition

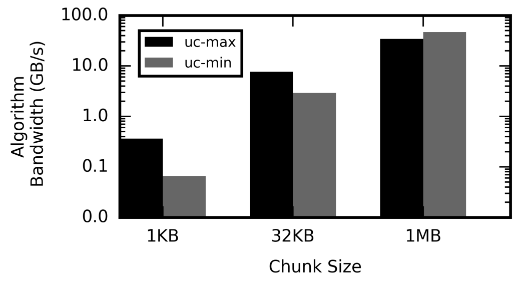


Figure 5.12: Switch-hyperedge strategies

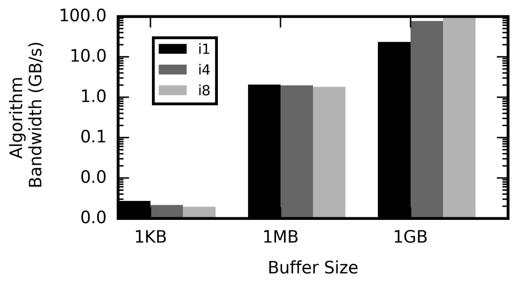


Figure 5.13: Runtime instances

and receiver GPU similar to *dgx-sk-1* except we allow a sender to be connected to $n$ different receivers in the other node. Figure 5.9 shows the algorithm bandwidth of ALLGATHER obtained by varying $n$, the number of IB connections per GPU, for a fixed chunk size of 1KB, 32KB, and 1MB. For a 1KB chunk size, we found the algorithm that uses 8 IB connections per NIC performs better than algorithms using fewer connections. As the chunk size increases to 32KB and 1MB, the optimal number of IB connections per NIC reduces to 4 and 1, respectively. The benefits of link sharing shrink as the chunk size increases and $\beta$-cost starts dominating over the $\alpha$-cost.

**Changing transfer cost using chunk size.** We analyze the sensitivity of TACCL's synthesizer to the data size provided in the communication sketch when its algorithms are applied on a communication using a different data size. Figure 5.10 shows the performance of ALLGATHER algorithm for three different chunk sizes (1KB, 32KB, and 1MB). Algorithms generally perform well for a range of data sizes close to what they have been synthesized for. We recommend trying a small set of nearby sizes to ensure the best performance.

**Changing data partitioning.** Figure 5.11 shows the algorithm bandwidth of algorithms generated by partitioning data on each GPU into a single or two chunks. We set the switch-hyperedge policy to `uc-min` and fix number of instances to 8. At a large buffer size of 1GB, the algorithm generated for two data chunks utilizes bandwidth better as compared to the algorithm generated for a single data chunk per GPU.

**Changing switch-hyperedge policy.** Figure 5.12 shows the algorithm bandwidth for algorithms generated and evaluated for 1KB, 32KB, and 1MB chunks. The algorithm bandwidth is displayed in log-scale. We vary the switch-hyperedge policy between `uc-max` and `uc-min`. For smaller buffer sizes, the `uc-max` configuration performs better than `uc-min`, whereas for larger buffer sizes, `uc-min` performs better than `uc-max`.

**Changing number of instances.** Figure 5.13 shows algorithm bandwidth with instances ranging from 1 to 8. The switch-hyperedge policy for these algorithms is set to `uc-min`. Increasing the number of instances improves bandwidth utilization — multiple threadblocks seem to be needed to keep the six NVLinks in a V100 busy. However, a larger number of threadblocks also increases latency, which we suspect is due to unfavorable scheduling of synchronization related memory operations onto the NVLinks at the start of each send. Since latency cost dominates for small buffer sizes, using a large number of instances only increases the latency cost. As the buffer size increases, the bandwidth improvements due to more instances become predominant. Since switch-hyperedge policy and number of instances have a similar relation with chunk sizes, we always run `uc-max` algorithms with a single instance and `uc-min` algorithms with 8 instances.

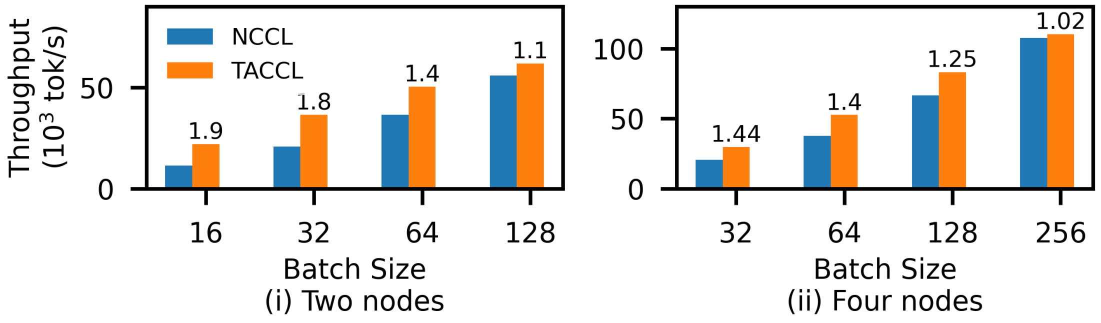


Figure 5.14: **Comparison of TACCL against NCCL for Transformer-XL model.** Training throughput using TACCL's collective algorithms on Transformer-XL compared against NCCL on 2 and 4 Azure NDv2 nodes. Speedup over NCCL is mentioned on top of the bars.

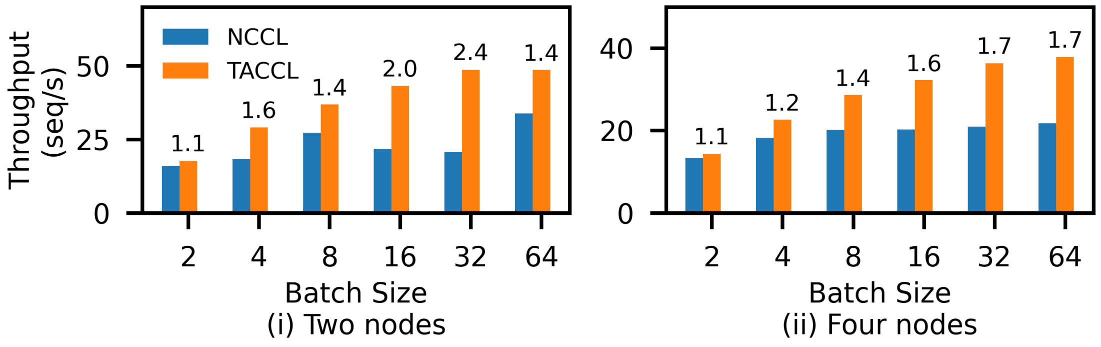


Figure 5.15: **Comparison of TACCL against NCCL for BERT model.** Training throughput using TACCL's collective algorithms for BERT compared against NCCL on 2 and 4 Azure NDv2 nodes. Speedup over NCCL is mentioned on top of the bars.

### 5.8.4 End-to-End Training.

We evaluate TACCL on distributed training of two large language models, Transformer-XL [32, 6] and BERT [33, 5], on two (and four) Azure NDv2 nodes, i.e. 16 (and 32) GPUs. Transformer-XL uses data parallelism and whereas BERT uses model parallelism. The typical transfer sizes for Allreduce in Transformer-XL is in the 20 - 40MB range, and for BERT it is about 2MB. Both models communicate with `torch.distributed` and, as explained in § 5.6, using TACCL algorithms in them is

| AllGather | | AlltoAll | | AllReduce | |
|---|---|---|---|---|---|
| Sketch | Time(s) | Sketch | Time(s) | Sketch | Time(s) |
| *dgx2-sk-1* | 35.8 | *dgx2-sk-2* | 92.5 | *dgx2-sk-1* | 6.1 |
| *dgx2-sk-2* | 11.3 | *ndv2-sk-1* | 1809.8 | *dgx2-sk-2* | 127.8 |
| *ndv2-sk-1* | 2.6 | *ndv2-sk-2* | 8.4 | *ndv2-sk-1* | 0.3 |

Table 5.3: Synthesis time for TACCL algorithms for different collectives using different communication sketches.

quite straightforward.

We lower the algorithm synthesized by the synthesizer into TACCL-EF with 1 and 8 instances, and show the performance of both against NCCL. Figure 5.14 and Figure 5.15 show the training throughput obtained by using TACCL's collective algorithms for communication instead of NCCL for Transformer-XL and BERT respectively for different batch sizes. TACCL speeds up training of Transformer-XL by $11\% - 1.94\times$ on 2 nodes and by $2\% - 1.44\times$ on 4 nodes. The speedup for BERT is $12\% - 2.36\times$ on 2 nodes and $7\% - 1.74\times$ on 4 nodes. Depending on the memory available per GPU and on how the batch size affects model accuracy, any of these batch sizes might be chosen for use in practice.

We also use algorithms synthesized by TACCL for Alltoall and Allreduce collectives for training an internal Microsoft's mixture-of-experts workload on two NDv2 nodes. The Alltoall and Allreduce sizes required for this model are $\approx$ 6MB and $\approx$ 256MB, respectively. TACCL improves the end-to-end throughput of this model by 17%.

### 5.8.5 Synthesis Time

Table 5.3 shows the total time it takes for TACCL to synthesize algorithms for different collectives using some of the communication sketches mentioned in § 5.8.2. In most cases synthesis takes from seconds to a few minutes, making it amenable to a user-in-the-loop approach. When synthesizing an Alltoall collective using some communication sketches, TACCL's contiguity encoding may take more time in proving the optimality of a feasible solution. We put a time limit of 30 minutes on the contiguity encoding in these cases. The contiguity encoding for sketch *ndv2-sk-1* reaches this timeout, but a feasible solution was already found in 4min 14s. We have also been able to synthesize an ALLGATHER for 80 GPUs (10 NDv2 nodes) in under 8 minutes.

## 5.9 Summary

In this chapter, we presented TACCL, a topology and input-size aware collective communication library for multi-node distributed machine learning training and inference. TACCL uses user-provided communication sketches to guide the synthesis of collective algorithms. Using a three-step technique of relaxed routing, heuristic ordering, and contiguity and exact scheduling, TACCL generates efficient collectives for multi-node topologies. The algorithms thus generated are up to $6.7\times$ faster than the state-of-the-art NCCL and result in $11\% - 2.3\times$ faster end-to-end training time. TACCL is open-sourced and available at `https://github.com/microsoft/taccl`.

# Chapter 6: MAPLE: Parameterized Learned Index

In Chapter 2, we discussed that there exist a wide variety of workloads and that index structure performance depends on the underlying data structure they employ. In this chapter, we present MAPLE, a parameterized learned index that can obtain high performance for a wide range of workloads.

We first present the goals of MAPLE (§ 6.1). We then discuss the characteristics of different data structures that we explore in order to build MAPLE. We then discuss MAPLE's design (§ 6.3). Finally, we evaluate MAPLE against an existing state-of-the-art learned index (§ 6.5).

## 6.1 Goals

- MAPLE should be designed to identify and expose only a few, but important, parameters.
- MAPLE should be able to achieve a wide range of spectrum on the read-write performance curve.
- MAPLE should achieve high performance on different types of workloads and datasets.

## 6.2 Data structures for learned index

Prior work (ALEX) [36] has shown that it is possible to build an updatable learned index that, analogous to the B+Tree, first performs a model-based tree traversal to reach a leaf node, and then performs a "last-mile" search on the leaf node. This last-mile search comprises 54% - 83% of the lookup time for ALEX for different datasets. For inserts, this contribution would be even greater. Thus, the data structures used as leaf nodes in learned indexes ultimately determine the speed of access

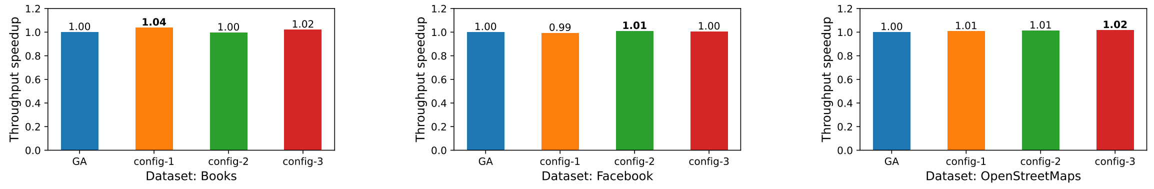


Figure 6.1: Performance improvements over cost model based gapped-array for different gapped-array configurations on different datasets.

to data. In order to achieve the goals of the MAPLE, we first need to discuss the data structures that will be used to store data and identify how they can be parameterized to achieve different points on the read-write performance curve. Here, we discuss three data structures that we explore and their lookup and insert performance as well as their contribution to memory footprint.

### 6.2.1 Gapped Array

The Gapped Array (gapped-array) data structure was introduced in ALEX [36] and provides fast read performance and good write performance. A gapped-array node has interspersed gaps in-between sorted keys which allows for faster model-based reads and model-based writes. A linear model is first trained using closed-form linear regression on keys in the node. Keys are then placed in the gapped-array according to the position predicted by the model. The gap density in a gapped-array is kept to be around $20\% - 40\%$, allowing keys to be spread apart and be placed close to the predicted positions. A bitmap is used to mark the positions of gaps and keys.

**Lookups.** Whenever a key has to be looked up, the same model is used to obtain its position, thereby having a very close approximation of the key position. Lookups are done by performing an exponential search around the approximate position provided by the model. The exponential search involves exponentially increasing the search boundary for as long as the boundary condition is satisfied and then performing a binary search within the identified boundaries.

**Inserts.** In the case of inserts, the lookup position is first obtained for the key to be inserted. If there is a gap in that position, the key is inserted there, and otherwise, the keys are shifted to the nearest gap to make space for the new insert. Once the density of keys in a gapped-array reaches 80%, the node is either expanded or split into two nodes. ALEX makes this decision based on its cost model which models cost using a linear combination of the expected number of exponential searches in the node for lookups and inserts, and the expected number of shifts in the node for inserts. When the empirical cost of a node becomes higher than its expected cost, ALEX decides to split the node.

We explore different parameters in a gapped-array in order to achieve different tradeoffs in read-write performance and to check if it can be used to better adapt to write-heavy workloads. ALEX has a knob for the expected insert fraction in the workload. We vary it from 0 to 1 in increments of 0.1 for two different datasets (Amazon Books and OpenStreetMaps Cellids) to obtain the average read and write latency for different expected insert fractions. We notice that this only improves insert latency by about $2\% - 3\%$ when the knob is turned from a read-only to write-only workload expectation. We then expose the maximum average number of exponential search iterations (`Msi`) and the maximum average number of shifts (`Msh`) as parameters of the gapped-array. We track the number of exponential searches upon key lookups and inserts and perform structural modifications during inserts to the gapped-array node in case the average exponential search required in a node exceeds the maximum of the expected exponential search or the `Msi` configuration set. Using a higher `Msi` reduces the frequency of structural modifications required but also increases the iterations of exponential searches required. This shows around 4% improvement in cases where the number of search iterations expected is already close to one. However, it cannot be increased indiscriminately as that would affect the average-case performance of inserts and lookups. We also track the number of shifts required when inserting keys and perform structural modifications in case the average shifts required exceed the maximum of the expected shifts or the `Msh` configuration set.

We only consider the `Msi` and `Msh` parameters in this dissertation. By making

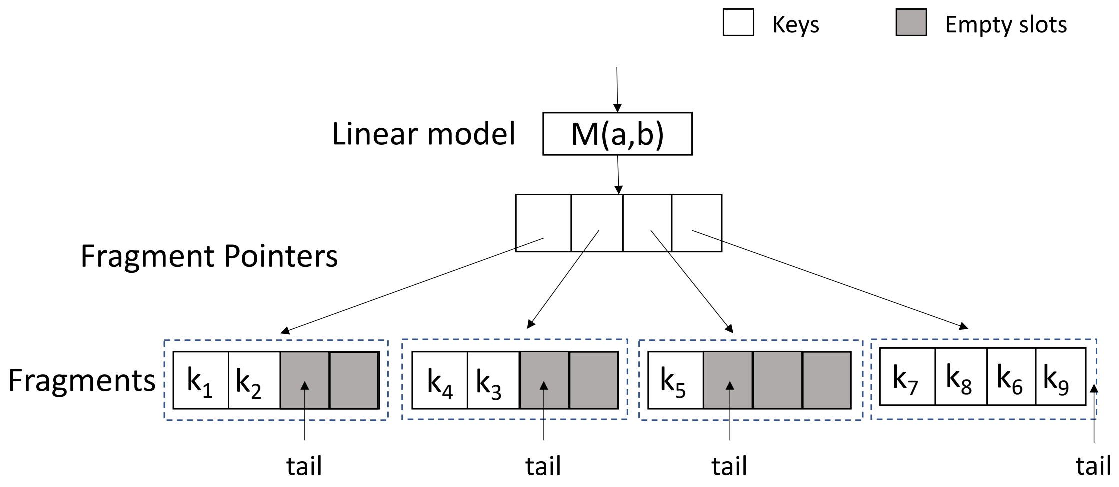


Figure 6.2: The Fragmented Log Data Structure.

use of the `Msi` and `Msh` parameters, we are able to tune the insert performance of gapped-array, though only by a little. Figure 6.1 shows the improvement in performance obtained by three different configurations on three different datasets for the same workload (100% inserts) as compared to a gapped-array in MAPLE which uses ALEX's cost model (GA). We notice different configurations perform better for different datasets. However, the performance gain for inserts is not too high. In fact, for a given workload and dataset, a couple of configurations seem to outperform all other configurations in terms of read and write latency (by a small margin), and we are unable to obtain the wide range of read-curve performance spectrum that we would like. This motivates the need to introduce a new learned data structure that has fast inserts and is still tunable to achieve good read performance.

### 6.2.2 Fragmented Log

We introduce a new write-optimized learned data structure called *Fragmented Log* (fragmented-log). As shown in Figure 6.2, a fragmented-log node consists of multiple fragments, each fragment storing un-sorted dense keys at the start and empty slots for new keys at the end. A sorted order exists between the fragments of the

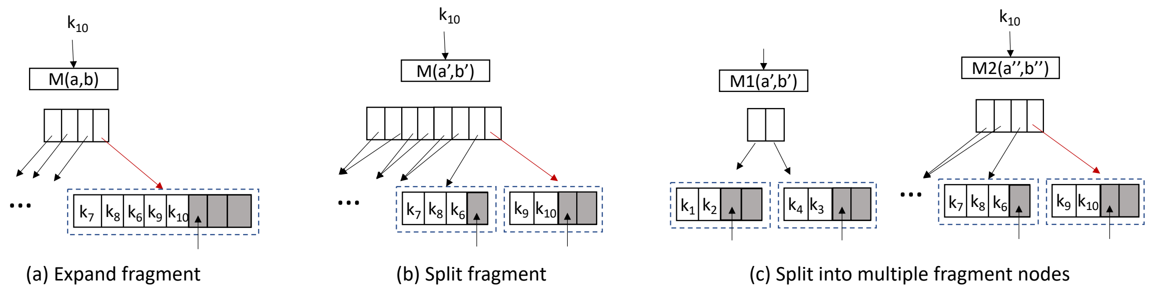


Figure 6.3: Cases on inserting a key into a full fragment.

log, and each fragment of a node is assigned a partition of the key space allocated to the node. An array of fragment pointers is used to access different fragments in a fragmented-log. A fragmented-log node also stores a tail array that points to the first empty slot in each fragment.

**Lookups.** When a key is read in fragmented-log, a linear model is used to map the key to the fragment that is assigned the key-space partition in which this key lies. The key is then linearly searched from the identified fragment.

**Inserts.** On key inserts, the fragment to insert into is identified using a linear model, and the key is simply appended to the fragment. In case the fragment is full, it is either expanded to make space for the key or split into multiple fragments, one of which will be used to insert the key. In case of fragment split, either the number of fragments in a single fragmented-log node increases or the entire fragmented-log node is split into multiple nodes. Figure 6.3 shows the different cases that occur when key $k_{10}$ is inserted into the full fragment shown in Figure 6.2. The fragmented-log exposes different parameters that determine which of these actions will be taken.

**Exposed parameters and performance characteristics.** Owing to the unsorted nature of data in a fragment, lookups need to take place using linear search and are thus not particularly fast. We expose the maximum fragment size (`mf-size`) as a parameter of the fragmented-log. By setting the `mf-size`, we cap the number of keys per fragment, thus providing control over read performance. On inserts, we ensure

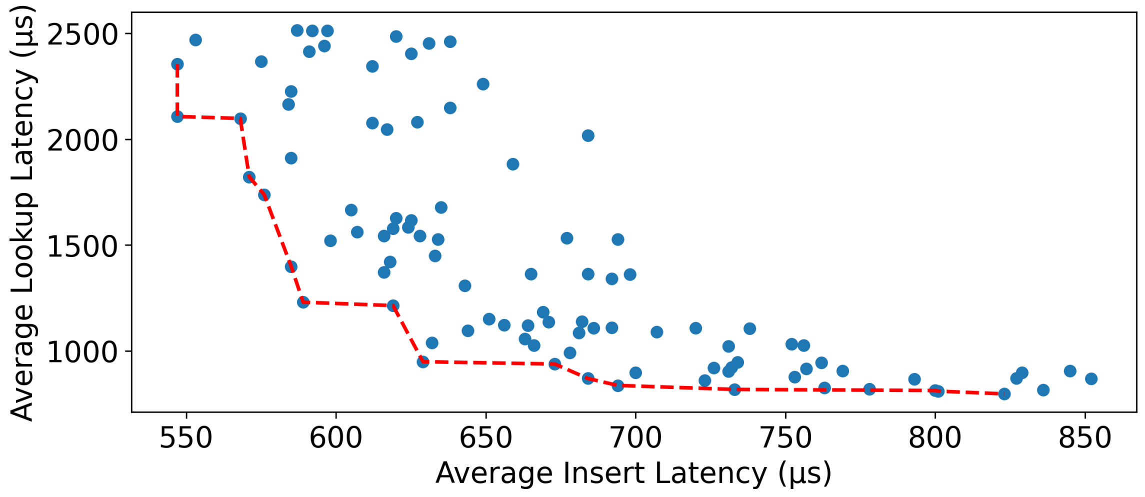


Figure 6.4: Insert and lookup latency for different fragmented-log configurations.

that the number of keys in a fragment does not exceed `mf-size` by either multiplying the number of fragments in the node or splitting the node into two. Reducing the `mf-size` would lead to a higher read throughput. However, it would trade-off write performance because structural modifications would be required at a higher frequency.

We also expose the maximum number of fragments (`Mnf`) allowed in each fragmented-log node as a parameter. On inserts, if the maximum fragment size has been achieved, the fragmented-log node will undergo structural modifications. The parameter `Mnf` determines whether the node will increase the number of fragments as a part of the modification, or split into two. The higher the `Mnf`, the more the number of fragment pointers. If the fragmented-log stores a highly skewed key space, the fragment usage will also be skewed, in which case, a larger `Mnf` would only serve to increase memory usage contribution from fragment pointers. On the other hand, for a more uniform key distribution, a higher `Mnf` reduces the number of fragmented-log nodes needed in a tree, thus also reducing the tree height.

When the maximum number of fragments is reached, the node will split into multiple nodes with smaller fragments. We expose the initial number of fragments (`nf`) in each new node as another parameter of fragmented-log. Based on the `nf`, the number of new nodes created is determined.

By modifying the three parameters - `mf-size`, `Mnf`, and `nf`, it is possible to achieve a range of data points on the read-write performance spectrum. For a particular workload and dataset, we obtain the read and write latency of about 100 different fragmented-log configurations. As can be seen in Figure 6.4, we are able to achieve a tradeoff between read and write performance for different configurations. We also plot the Pareto front for this data and note that many configurations are sub-optimal and do not lie on the Pareto front. Thus, fragmented-log exposes important parameters that can be used to tune its performance and it is important to identify the right configurations.

**Memory contribution.** Each fragment is accessed using a fragment pointer, which adds to the memory usage in the index structure. Since the size of each fragment is capped in a fragmented-log, it is possible to obtain bounds on the number of fragments and thus the memory contributions from each fragment pointer and its associated metadata.

Let us assume $N$ keys need to be indexed, the maximum number of keys allowed in a fragment is `mf-size`, and a fraction $d$ of empty slots are reserved in fragments to allow inserts. We refer to the number of fragments as $F$. Then, the minimum value for $F$ is when the keys are uniformly distributed and all fragments are filled with the maximum allowed keys. Thus, at minimum, there will be $N/(\texttt{mf-size} \times d)$ fragments. The maximum value of $F$ would be when keys are distributed in such a way that one fragment has $\texttt{mf-size} \times d$ keys and all other fragments have a single key. In this case, there would be $N - \texttt{mf-size} \times d + 1$ fragments, resulting in much higher memory usage. However, this worst case only occurs when keys are exponentially increasing in powers of two, which is an unlikely scenario. By partitioning the key space with the help of several fragmented-log nodes, we are able to obtain a much simpler key distribution per fragmented-log node, so that it requires fewer fragments to model, and thus does not lead to a memory usage explosion.

For each fragment, we need to store pointers to its key and value array. We also need to store metadata like the last filled position and size of the fragment. In total, the added memory contribution to the index due to the fragments is $21 \times F$ bytes. The fragmented-log node does not need to store a bitmap of existing keys since keys are densely packed. Deletion of keys happens by swapping with the last filled key, thus also maintaining the dense structure. Instead of maintaining a bitmap like other gap-based data structures that we will discuss later, the fragmented-log node can use memory for storing fragment pointers instead. Further, unlike gap-based data structures that only allow nodes to be filled to a certain density, the fragmented-log node can be filled to capacity, thereby efficiently utilizing memory.

### 6.2.3 Minimal Perfect Hash Functions (MPHF)

Minimal perfect hash functions [92, 43, 74], are built on static data and injectively map all $n$ keys of the data to integers from 1 to $N$. They can thus be used to provide worst-case O(1) lookup times when keys are placed in the positions obtained from the perfect hash function.

We explore using a particular perfect hashing function, called PTHash [92], to obtain good read performance in MAPLE. Internally, the PTHash function partitions keys into skewed buckets based on key hashes and generates a ‘pilot‘ per bucket, which when XOR’d with the hashed key gives the key index.

**Hash function construction.** Constructing a perfect hash function is computationally expensive. We modify the method of hash function construction of PTHash by adding a model-based bucket selection technique. PTHash partitions keys into skewed buckets based on key hashes, and generates a ‘pilot‘ per bucket, which when XOR’d with the hashed key gives the key index. Instead of using key hashes to determine the bucket, we use linear models to identify the buckets for keys. This speeds up construction time by 3× for sorted data.

**Inserts.** We provide a small write buffer to accommodate writes in PTHash. On key inserts, we append the key to the write buffer. If the buffer is full, we coalesce the buffer keys with the read-only array and reconstruct the hash function.

**Lookups.** On key lookups, we query the hash function to give us a key index. If the key is not present in that index, we search the write buffer linearly.

In our experiments with PTHash, we found that requiring a per-node auxiliary data structure to store pilots made the PTHash leaf node much less cache efficient than the Gapped Array. Further, the memory usage of the MAPLE tree with PTHash nodes increased because it had to store pilots for each bucket in each node. In our evaluation of MAPLE later, we only consider the Gapped Array and Fragmented Log data structures. However, this exploration of PTHash shows a way to add model-based components to already existing data structures.

## 6.3 Design

In this section, we discuss the design of MAPLE, starting with an overview of MAPLE and how it manages to achieve the goals mentioned above. We then describe the throughput prediction model used by MAPLE to obtain candidate configurations for the parameters.

### 6.3.1 Overview

MAPLE is a parameterized learned index that can be used to achieve high performance for a wide range of workloads. Figure 6.5 shows the workflow of MAPLE. The first time that MAPLE sees an input workload, it queries its evaluator with a subset of the trace. We expose different parameters for leaf node data structures in MAPLE that determine how the underlying data is stored and how fast it can be accessed or modified. The evaluator uses a machine-learning model to predict the throughput of all the different configurations of the parameters on the trace subset. It then obtains the top-few configurations for MAPLE according to the predicted throughput and executes the trace subset with them on the hardware in order to

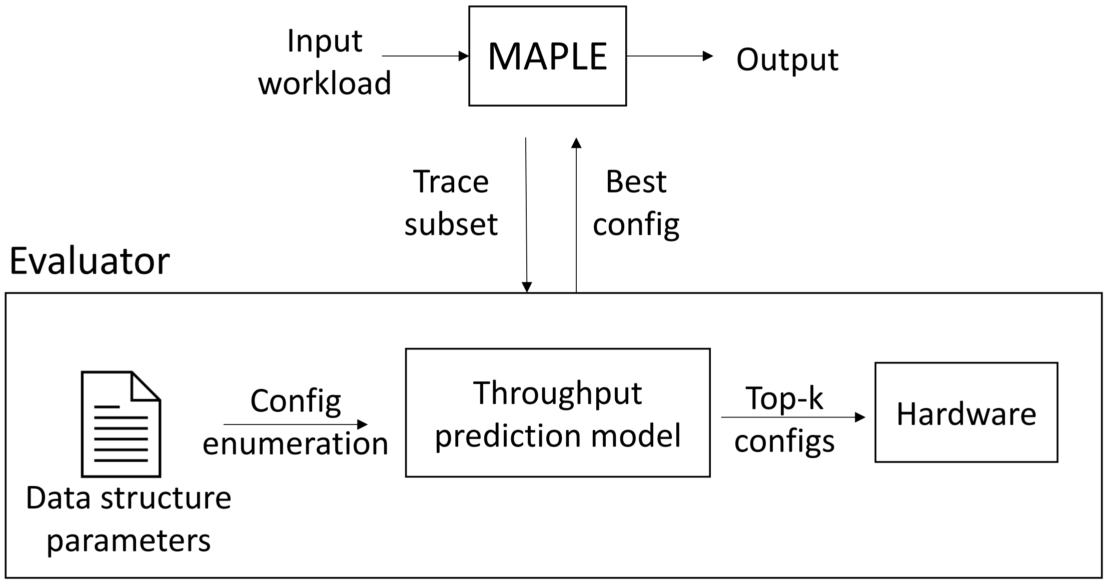


Figure 6.5: Overview of MAPLE workflow.

identify the best-performing configuration for the trace. MAPLE then uses this configuration to load the data and respond to requests.

### 6.3.2 Selecting MAPLE parameters

MAPLE exposes different parameters that determine how the underlying data is stored and how fast it can be accessed or modified. First, it exposes the type of data structure that stores the key-value data. MAPLE allows selecting between a gapped-array data structure(§ 2.6) and the novel fragmented-log data structure as a node type in the index. Based on the type, each data structure is further parameterized. For each data structure, we identify a handful of parameters as described earlier that determine how well it performs.

### 6.3.3 Throughput Prediction Model

In order to identify the best configuration of parameters for MAPLE, we train a machine learning model to predict the throughput of the workload given the configuration parameters. We observe that both the workload distribution as well as the dataset distribution impact which configurations are best for MAPLE. Thus, we build a convolutional neural network model that takes in a subset of the trace as an image in order to generate trace embeddings. We also discretize the parameters into smaller buckets and obtain an embedding of these bucketed configurations, which we add and concatenate with the trace embeddings. We then train a multi-layered neural network classifier to predict the throughput of the (trace, configuration) pair. By querying the trained throughput model again with multiple candidate configurations for a given workload and dataset, we can obtain a prediction of the throughput.

## 6.4 Discussion

We discuss some challenges we faced, how we solved them, and the insights we obtained when building MAPLE.

**In-memory systems are sensitive to control.** We found that performing various control-related activities in the in-memory environment adds significant overheads in practice, whose effects are felt especially in the scenarios where learned indexes already provide high-speed data access. We list overheads that are incurred as a part-and-parcel of MAPLE's functionality and discuss how we carefully implement MAPLE to reduce these added overheads.

- Overhead of maintaining different data structures: Since data is stored and accessed differently in fragmented-log and gapped-array in MAPLE, we need a way to determine the correct code path for an operation. We initially used derived node classes to implement gapped-array and fragmented-log and take advantage of polymorphism and transparent vtable lookups. However, we saw up to 14% degradation of performance when deriving two types of nodes from a base datanode class, as compared to using the datanode class to implement a gapped-array node. Instead, we implement both fragmented-log and gapped-array as a part of the same node class, and use an attribute *node_type* to switch between their data access implementations. Here too, the switch condition and associated branch prediction misses do add to overheads. However, the performance with this implementation with both node types enabled is faster than using a derived class implementation. We note the more simple a dataset is to model, the faster its accesses are, and thus these overheads are seen more prominently in such cases.

- Overhead of monitoring to maintain configuration parameters: MAPLE monitors node parameter values and performs structural modifications when they exceed their configured threshold. However, this monitoring itself has a cost associated with it. We find that monitoring parameters when performing lookups in a MAPLE tree of gapped-array reduced throughput by up to 3% with different datasets. We note that as long as MAPLE is bulk-loaded with the correct configurations and all inserts aim to preserve the property that parameter values are lower than the threshold, it is not required to perform any monitoring of parameters for lookups. Thus, lookups in MAPLE do not lead to structural changes.

**Linear search in Fragmented Log can allow partial ordering.** We make an interesting observation regarding the lookup technique in the Fragmented Log data structure. Keys in a fragment of a fragmented-log are unsorted and lookups in a fragment perform linear search in order to obtain the position of the key. Since we are scanning the keys and comparing them against the key to be looked-up, we are ultimately performing a subpart of the algorithm that might be used to sort the fragment. It would be interesting to explore the different ways to make use of this insight to obtain an adapting fragmented-log index structure in case of heavy reads.

For example, one way to introduce partial order in the fragmented-log on key lookup could be to swap the first key larger that the looked-up key with the looked-up key on linear search.

## 6.5 Evaluation

In this section, we aim to answer the question of whether MAPLE can achieve better performance than baseline ALEX. We also discuss the memory usage of MAPLE as compared with ALEX.

### 6.5.1 Experimental Setup

We run experiments on an Intel(R) Xeon(R) Silver 4314 machine with two sockets. We use `numactl` to run MAPLE on a fixed core and use memory from the same node as the core. We use AVX-2 instructions to perform linear searches in fragmented-log. We downsample the trace image to $256 \times 256$ dimensions before using it in the throughput prediction model. For all experiments on ALEX, we set its expected insert fraction to the actual insert fraction of the workload.

### 6.5.2 Datasets and workloads

**Datasets.** We use the SOSD [66] suite of datasets to benchmark MAPLE. Commonly used benchmarks like the Yahoo! Cloud Serving Benchmark (YCSB) tend to have a fairly uniform key distribution, which a learned index can overfit easily. Instead, SOSD benchmark is particularly built to benchmark learned indexes. We use three real-world datasets from SOSD - 1) Amazon (Books), a dataset containing book popularity data from Amazon, 2) Facebook (FB) data, a dataset containing randomly sampled user-ids from Facebook, and 3) OpenStreetMap (OSM) CellIDs, a dataset containing cellids from OpenStreetMaps. These datasets have 200 million unsigned 64-bit integer keys. We use a constant payload of unsigned 64-bit with the keys.

**Workloads.** For all experiments, we bulk-load 10M keys in the index structure

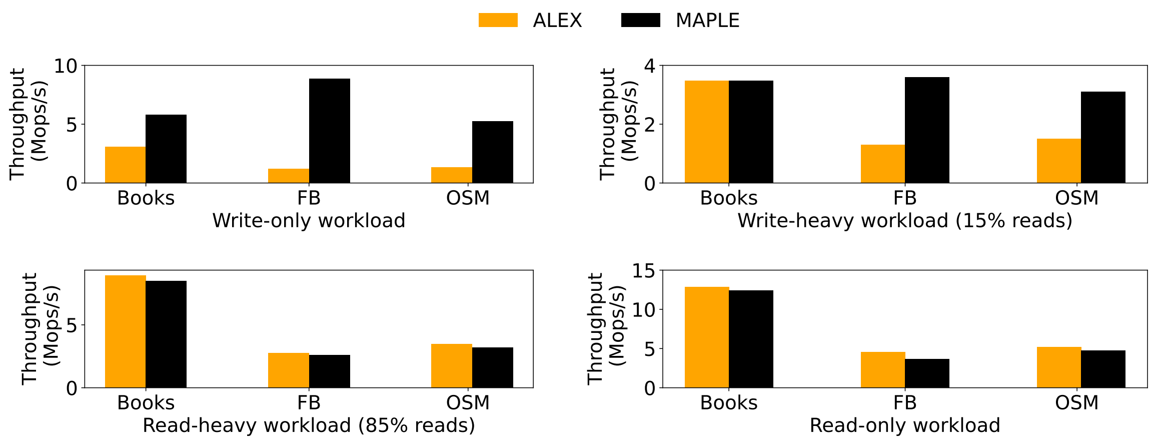


Figure 6.6: Performance of MAPLE as compared to ALEX on different workloads.

and perform 200M trace operations. We benchmark for different workload patterns, from write-only (insert fraction = 1), and write-heavy (insert fraction = 0.85), to read-heavy (insert fraction = 0.15), and read-only (insert fraction = 0).

### 6.5.3 Throughput comparison

We compare the throughput obtained by MAPLE against ALEX for different workloads and datasets in Figure 6.6. For a write-only workload, MAPLE chooses to use a configuration with fragmented-log, and obtains an 88%, 7.5×, and 3.9× performance improvement over ALEX for the books, fb, and osm datasets respectively. The books dataset is much simpler to model as compared to the other two datasets, resulting in a smaller tree and fewer exponential search iterations and key shifts in ALEX. Thus the performance improvements from using fragmented-log in MAPLE are not as pronounced for the books dataset as compared to the fb and osm datasets. For a write-heavy workload, MAPLE chooses to use a different configuration with fragmented-log, and obtains similar performance as ALEX for books, 2.8× speedup over ALEX for fb, and 2× speedup over ALEX for the osm dataset. For a read-heavy workload, MAPLE uses different configurations of gapped-array for all datasets. Since MAPLE has overheads due to maintaining different data structures,

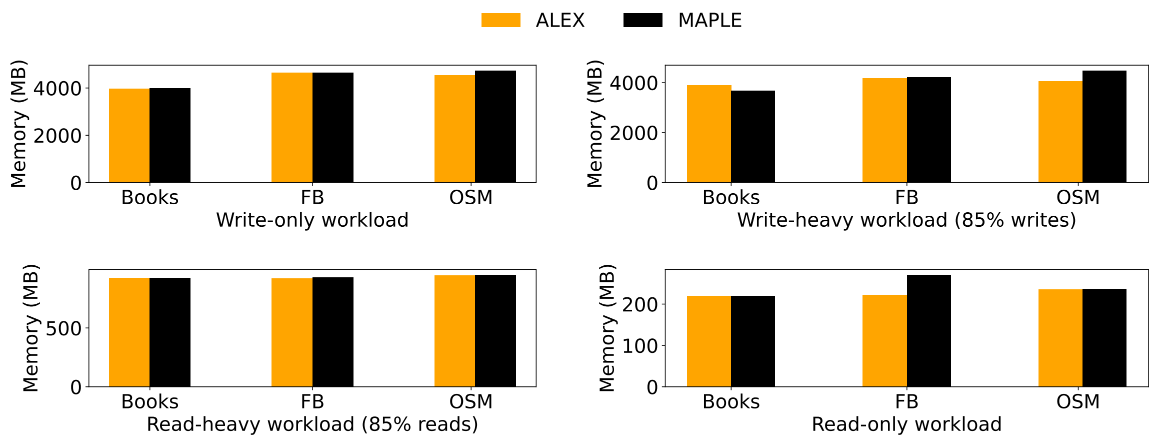


Figure 6.7: Memory usage of MAPLE as compared to ALEX on different workloads.

it suffers a performance drop of 4% − 8% as compared to ALEX. For a read-only workload, MAPLE chooses different configurations of gapped-array for books and osm, and has performance drop of 3% and 8% respectively for the datasets. However, for the fb dataset, MAPLE spuriously chooses a fragmented-log configuration and suffers from around 20% of performance drop. This could be because the fb trace is hard to represent in an image owing to the presence of extremely big outlier keys, resulting in the throughput model not being able to accurately predict configuration throughput for the workload.

### 6.5.4 Memory usage

We show the memory used at the end of a workload by MAPLE as compared to ALEX for different workloads and datasets in Figure 6.7. The memory used by MAPLE using fragmented-log or gapped-array is within 4% of that of ALEX for most workloads and datasets. For the books dataset in the write-heavy workload, MAPLE uses 5% less memory than ALEX. For the osm dataset in the write-heavy workload and fb dataset in the read-only workload, MAPLE uses 10% and 22% more memory than ALEX respectively. Currently, MAPLE's throughput prediction model and configuration selection strategy do not consider memory usage as an objective.

The next fastest configuration of MAPLE from the top-10 for the osm dataset in the write-heavy workload also has a 2× speedup over ALEX with only a 2% higher memory usage than ALEX. In the future, we can explore predicting memory usage in the prediction model or filtering out high-memory configurations from the top-k configurations.

## 6.6 Summary

MAPLE is a parameterized learned index structure that can provide efficient read and write performance by using different learned structures as components for its leaf nodes. For write-heavy and write-only workloads, MAPLE can achieve similar and up to 7.5× performance speedup with the help of the novel fragmented-log data structure while having performance comparable to the state-of-the-art updatable learned index on different datasets.

# Chapter 7: Related Work

In this chapter, we discuss systems, libraries, and other research that are related to this dissertation.

## 7.1 Memory usage bottlenecks in deep network training

We first reiterate some of the prior work done in reducing memory consumption bottlenecks in deep network training.

**Optimizing deep network operators.** Gist [58] proposes several hand-crafted optimizations such as storing only ReLU signs. RevNets [49] redesigns a ResNet [51] architecture making each network block reversible, thereby eliminating the need to store intermediate activations for backpropagation. Memory-efficient DenseNets [94] reduce memory utilized for feature maps by recomputing all intermediate feature maps during the backward pass with a small compute overhead. In-place activated batchnorm [17] or ReLU layers use output activations to compute their gradients, thus reusing a single memory buffer for the gradient computation in consecutive layers. Although these hand-crafted techniques independently result in memory savings, it is difficult to know which technique will perform better when, and different implementations perform best on different architectures. In contrast, MONeT automatically finds the best implementation for each forward and backward operator given a memory budget.

**Mixed precision training.** Mixed precision training [78] uses half-precision (FP16) instead of single precision (FP32) for all tensors and arithmetic during training, reducing the memory by nearly half. While training at precision lower than FP16 results in loss of training quality [11], prior work like backpropagation with approximate activations [20] carefully quantize certain intermediate outputs (activations) to 4 bits, resulting in significant memory savings. MONeT is orthogonal to mixed precision training. It currently only has operators that compute in full-precision, but it is possible to add half-precision operations to MONeT as well.

**Dropping intermediate outputs.** [24] proposed dividing a network into different segments, dropping all intermediate outputs within each segment, and recomputing them later. Chen *et al.* use $\sqrt{n}$ equal segments, trading memory savings for the cost of an extra forward pass. Checkmate [59] solves the problem in a more general setting, using a mixed-integer linear program solver to decide which layers to recompute for a given network. Like Checkmate, MONeT optimizes a checkpointing schedule, but on a different computation graph that allows for the optimization of an entire execution plan jointly to find a checkpointing schedule as well as the best implementation of each forward and backward operator.

## 7.2 Network communication overhead in distributed deep learning

We now discuss prior work done to build efficient collective algorithms. We also discuss prior sketching approaches in other areas.

**High-Performance Computing (HPC).** The MPI standard provides a set of collective communication algorithms that enable efficient distributed computations of interconnected nodes [40]. The HPC community has focused on the efficient implementation of these MPI collective algorithms [93, 115] and demonstrated how to build optimized algorithms for specific interconnects, like mesh, hypercube, or fat-tree [101, 15, 13]. In contrast to TACCL, these prior works assume homogeneous interconnects and are often only focused on bandwidth optimality. Hybrid algorithms [21, 13] combine bandwidth- and latency-optimal algorithms based on input sizes, but only for mesh networks.

**NCCL.** NCCL [85] is a GPU implementation of a subset of the standard MPI collectives, optimized for NVLINK and Infiniband interconnects. While NCCL uses the topology of GPU connections and NIC placement along with buffer size to decide between two main types of communication algorithms — Ring and Tree, it is agnostic to the exact performance profile of the links, and thus (as we show earlier) is often multiple times slower than TACCL's topology aware collectives.

**Synthesis-based approaches.** Recent works like SCCL [18], Blink [116], and Plink [76] specialize algorithms for the underlying topology. SCCL solves an integer programming encoding based on discrete-time values in the form of steps and rounds of the algorithm in order to achieve the pareto-frontier of latency- and bandwidth-optimal algorithms. SCCL is able to synthesize a novel pareto-optimal Allgather algorithm for an Nvidia DGX1 node, but its restrictive formulation constrains it to only synthesize algorithms for single-node topologies. TACCL on the other hand synthesizes collective algorithms for multi-node topologies. Blink uses a heuristic spanning-tree packing algorithm to maximize bandwidth utilization within a node and a hierarchical approach across the node. Blink has good performance over NCCL in the case when NCCL cannot create rings spanning all GPUs inside a node. TACCL, on the other hand, generates algorithms for the full multi-node topology and outperforms NCCL when using the entire node of GPUs. Plink constructs a logical topology based on bandwidth and latency probes of the physical topology to avoid oversubscribed and congested links and searches for a reasonable clustering of nodes for a two-level hierarchical reduction strategy. Plink builds that hierarchical reduction from known primitives and does not search over the space of possible algorithms.

**Hierarchical approaches.** There are also hierarchical approaches to implement collectives [28, 102, 76, 116]. For example, Horovod [102] implements an Allreduce by a local ReduceScatter, a global Allreduce, and then a local Allgather. These methods do not search over possible algorithms and instead pick from a known set of decompositions. Concurrent to our work, Ningning et al. [122] use syntax-guided synthesis to combine base MPI primitives among a subset of nodes to hierarchically generate larger MPI primitives for the entire network. In contrast, TACCL uses a fine-grained approach for algorithm synthesis while using communication sketches for scalability. Combining these two complementary approaches is an interesting opportunity for future work.

**Network flow problems.** Network flow problems have used linear programming to solve routing and scheduling problems for traffic engineering [53, 60, 63, 108, 3] and topology engineering [109]. These techniques, however, cannot be used for generating collective algorithms since communication collectives do not follow all flow properties. Non-source GPUs in a collective can send the same chunk over different links in parallel while having received that chunk only once, which violates an important flow-conservation property used extensively in network flow problem literature. TACCL on the other hand makes use of communication sketches and an encoding relaxation technique to solve a continuous-time integer linear programming that faithfully models communication collectives.

Earlier works [113] solve a store-and-forward packet routing problem, in which packets must be routed to their destinations in over an arbitrary N-node network in minimum time. They first solve a relaxed linear problem to obtain a set of paths, followed by path filtering to obtain a schedule with time steps proportional to congestion and dilation of the path. Similar to TACCL, the approach of finding paths is to minimize the congestion and dilation metrics of the paths. However, TACCL differs from this work in how it encodes both the routing and scheduling problems. Further, collective algorithms have many differences as compared to store-and-forward routing. For example, routing of packets in collective algorithms can take place as trees instead of paths, and the underlying network has heterogeneous link profiles.

**Sketching approaches.** Program sketching [110, 61, 111] is a popular technique that has been applied to a variety of problems from synthesizing stencil computations [112], converting hand drawings to images [42] to social media recommendations [26]. Our work builds on this body of work to use sketching to effectively search a large space of communication algorithms.

## 7.3 Workload-adaptable Index Structures

We finally discuss some of the work done in building index structures to obtain high performance for different workloads and datasets.

**Updatable learned indexes.** Earlier learned indexes such as the Recursive Model Index (RMI) [69] and Radix Spline [67] are read-only and cannot efficiently support updates. More recently, updatable learned indexes such as the FITing-Tree [47], PGM-Index [46], ALEX [36], and LIPP [121] have been introduced.

- The FITing-Tree uses a B+Tree for tree traversal and has different segments as leaf nodes. Each segment in a FITing-Tree is represented by a linear function that predicts a key's approximate position within the error bounds specified when building the FITing-Tree. In order to allow inserts, the FITing-Tree uses a sorted delta buffer per segment. When full, the buffer is merged with the segment, which may then be split into multiple segments if the error threshold is crossed. While the FITing-Tree has good lookup performance and reduces space utilization by multiple orders of magnitude compared to traditional index structures, its insert performance is comparable to or lower than B+Tree, and thus it should not be used for write-heavy workloads.

- The PGM-index uses piece-wise linear functions with a fixed $\epsilon$ error threshold for both internal as well as leaf nodes. In contrast to FITing-Tree, it uses an LSM-based strategy for inserts where it builds a new PGM-index by merging smaller sub-indexes on key inserts. While PGM-indexes have good insert performance, MAPLE instead uses the fragmented-log data structure for write-heavy workloads which can be tuned to also improve read performance. In the future, we can explore how an LSM-based strategy can be used for MAPLE leaf nodes.

- As discussed earlier, ALEX uses a Gapped Array data structure in order to absorb key inserts. Based on the workload, MAPLE can decide to either use Gapped Array or Fragmented Log data structure.

- LIPP is another learned index that uses gaps in leaf nodes to accommodate inserts. LIPP ensures that its model-based tree traversal always points to the precise position of the key, and thus eliminates the need for a last-mile search as required in ALEX. However, prior work [120] has shown that the performance gain in LIPP is largely from trading off space for speed and ALEX can perform similarly or better than LIPP if it uses as much memory as LIPP. We show that MAPLE has similar memory usage as ALEX while being able to achieve high read and write performance according to the workload.

**Automated and semi-automated index structure construction.** Works like self-designed data structures [55, 54, 56, 57] and GENE [37] eschew handcrafted index data structures and instead automate the design of data structures from a range of different design choices. For example, the Data Calculator [55] identifies all the different types of data structure designs possible in terms of data layout and data access, and introduces learned cost models that can be used by data structure designers to identify the best-performing design for the expected hardware, data, and query workload. Initial results show that it is possible to design efficient and interesting data structures that combine different design primitives. Similarly, in GENE, the authors allow using multiple design primitives in the index structure. They differentiate between the logical part of the index, which partitions the key space, and the physical part of the index, which actually stores data, and use a genetic algorithm to obtain the best combination of design choices for a given workload. However, GENE's implementation currently only supports read-only queries.

With MAPLE, our main focus is to reduce the number of data structures to reason about to only a handful and parameterize them to allow obtaining good performance on a wide read-write spectrum of workloads. We restrict the internal nodes of MAPLE to model nodes that allow model-based traversals. We also use the same configuration of parameters for all leaf nodes. We believe that this approach allows the building of more practical index structures that can be easily used and adopted.

In the future, it would be interesting to explore allowing different configurations for different leaf nodes in MAPLE as is done in the Data Calculator and GENE.

**Parameter tuning in existing indexes.** LSM-Tree based index structures expose a large number of parameters to the user that can be tuned to obtain a wide range of performance characteristics. Multiple works, such as Dremel [125] aim to identify the optimal configuration for the parameterized index. In contrast to such works, we design learned data structures in MAPLE that have only a handful of parameters and obtain the fastest configuration from those parameters.

# Chapter 8: Future Work

In this chapter, we outline the directions in which our work can be extended in the future. In particular, we discuss three main directions - adapting MONeT's formulation for optimizing power consumption, using TACCL's communication sketches for other networking applications, and finally, using MAPLE's data structures to build a practical self-designing index structure.

## 8.1 Extension to MONeT's formulation

Training deep networks on edge devices like mobile phones and micro-controllers is becoming more and more commonplace. This is because sending the data of edge devices to large data centers for training is tricky owing to privacy concerns, battery drainage, and even lack of internet connections on the edge devices [91]. These edge devices often have small memory sizes and strict power requirements.

The main objective of MONeT is to minimize computation in deep network training while keeping memory usage below a fixed memory budget. We believe that MONeT's objective function can be very easily adapted to instead minimize power consumption. By doing so, we can easily obtain a schedule of checkpointing and operator implementations for any deep network which will fit within the memory budget of the edge device, while having minimal power consumption.

## 8.2 Extension of TACCL's communication sketches

TACCL introduced the novel abstraction of communication sketches, which are simple inputs provided by the user to communicate intuitive information to the synthesizer without needing much domain knowledge. We believe that the abstraction of communication sketches can also be used for other distributed applications that use data center networking.

Further, the expressivity of a communication sketch can directly help guide a synthesizer to generate collective algorithms. It would be interesting to explore automated search techniques, such as reinforcement learning, to generate different communications sketches, such that we can remove any reliance on a user and can still provide high performance for different hardware topologies and data transfer sizes.

## 8.3 Extension of MAPLE's data structures

Self-designing index structures (discussed in § 7.3) combine nodes with different design primitives, such as storing data in a sorted or un-sorted manner, or performing a search using linear search, binary search, or exponential search, within the same index structure. The design choices can then be tuned over a design continuum to adapt to workload changes. However, the search space for possible design changes is huge, and enabling the complete design continuum along with their transitions would require immense research and development effort [68].

We envision that it is possible to build a practical self-designing learned index structure by using MAPLE's data structures. The key enabler behind this idea is that it is possible to change a few parameters of the leaf node data structure in MAPLE to obtain different performance characteristics. By allowing leaf nodes in a single MAPLE tree to have different parameters and different data structure types, and introducing lightweight policies for transitions, MAPLE could fit well to even more varied different workload patterns.

# Chapter 9: Conclusion

Today, machine learning systems have become critical for various applications and use cases in the industry. These systems are resource-intensive and require to efficiently utilize existing resources. However, these systems can run on extremely variable and heterogeneous settings in production, and different solutions may be required for different cases. Optimizing such systems would require significant expertise. In this dissertation, we presented solutions that optimize various aspects of ML systems without needing an expert.

We introduced MONeT, a framework to automatically reduce memory requirements for training deep networks. MONeT jointly optimizes local and graph-level optimizations to yield a compute- and memory-efficient checkpointing schedule. MONeT reduces memory usage by 3× over PyTorch, with a $9-16\%$ compute overhead. It uses 1.2-1.8× less memory than the state-of-the-art automated checkpointing framework for the same computational cost. Our experimental results show that MONeT leads to better memory-computation trade-offs compared to the state-of-the-art.

Next, we introduced TACCL, a topology and input-size aware collective communication library for multi-node distributed machine learning training and inference. TACCL uses user-provided communication sketches to guide the synthesis of collective algorithms. Using a three-step technique of relaxed routing, heuristic ordering, and contiguity and exact scheduling, TACCL generates efficient collectives for multi-node topologies. The algorithms thus generated are up to 6.7× faster than the state-of-the-art NCCL and result in $11\% - 2.3\times$ faster end-to-end training time.

Finally, we introduced MAPLE, a parameterized learned index that can provide high performance for a wide variety of workloads. MAPLE uses two parameterized learned data structures, gapped-array and fragmented-log, as components in building the index, and outperforms existing state-of-the-art learned index by up to

7.5× for different datasets in a write-only workload.

## 9.1 Lessons Learned

In this section, we discuss some of the lessons that we can take from this dissertation to inform future work.

**Joint optimization is important.** We saw that jointly optimizing memory-saving techniques in MONeT gave the best performance results. In fact, when techniques are used independently, they may even ”step on each other” and degrade performance. Joint optimization of different techniques is important if they use a common input, such as the computational graph in the case of MONeT.

**Simple human inputs can go a long way.** TACCL's communication sketches can help reduce the search space for collective algorithms significantly, thus making the problem more tractable. While the reduction in search space varies according to the communication sketch, just sketching the logical topology in one particular communication sketch described previously, *ndv2-sk1*, is able to reduce the number of links to make decisions about by 3×.

**Data structure design and parameters are important for performance.** We saw that the Gapped Array data structure is inherently limited in its write performance because of having to keep data in a sorted manner. Further, by tuning the parameters of a Fragmented Log, it is possible to obtain a wide range of read and write performance. Data structure design and parameters can significantly impact the performance of an index structure and must be tuned well.

## 9.2 Closing Remarks

As deep networks grow popular and generate higher compute demands than before, hardware accelerators are becoming more and more varied. Startups like Cerebras [1] and Graphcore [2] are building new types of accelerators, and companies like NVIDIA and Google are rapidly advancing their own accelerator technologies. This has greatly heterogenized the space of hardware accelerators. Further, newer interconnect technologies are being introduced in data centers. With optical circuit switches [95], it will be possible to directly change how different hardware accelerators are organized in a cluster. Neural network architectures in deep learning are also constantly evolving. With neural architecture search [126] being used in many cases to automate the generation of neural architecture, common patterns of executions and optimizations may no longer be valid. Similarly, database applications over the past few years have significantly changed - with new workloads coming from machine learning, blockchains, and even new requirements due to data protection regulations like GDPR. It is important to build a parameterized index that can adapt to all these different workloads patterns. Optimizing machine learning systems to the fullest in this heterogeneous environment would require expertise and a cycle of significant experimentation every time the environment changes. This dissertation takes steps in solving these challenges by introducing solutions that can optimize different aspects of ML systems without using experts.

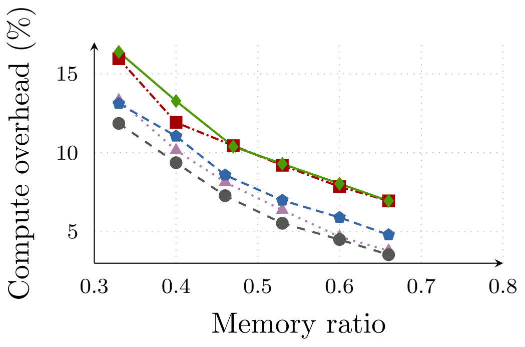
# Appendix A: Appendix for MONeT

## A.1 Detailed ablations

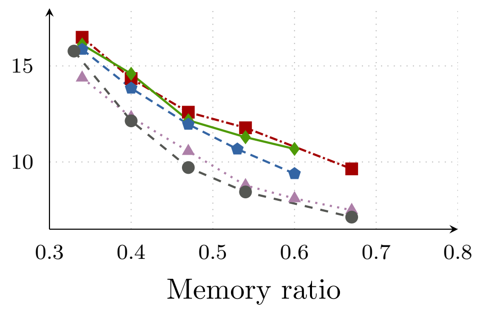
Figure A.1 shows a detailed plot of our ablation experiments comparing the compute overhead of variants of MONeT across a range of memory limits. Y-axis shows the compute overhead over PyTorch and X-axis shows the memory ratio to a PyTorch model. All variants which are not conv-optimized are greedily post-optimized to use the fastest convolution. We see that MONeT with no operator optimization (NoOp) is generally slower than the other variants for all models and memory limits. Convolution and output-activated optimizations are both important in reducing compute overhead. MobileNet-V2 uses depthwise separable convolutions, and hence does not significantly benefit from convolution-optimization. Further, MobileNet-V2 has `hardtanh` operators instead of ReLU operators, for which we have not implemented intermediate-activated backward optimization. Interemediate-activated optimizations provide memory savings in memory-intensive models, allowing models like VGG-16 to reach memory savings which are not attainable by other optimizations. All optimizations together result in the least compute overhead for any model or memory limit.

## A.2 More details on solver time

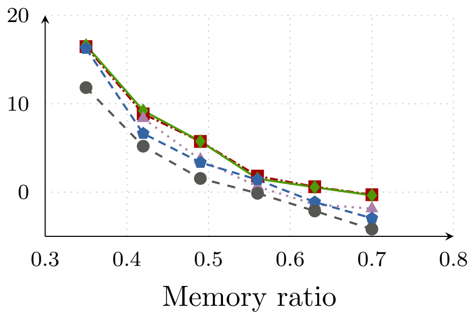
In § 4.6, we show the time it takes for the solver to reach 5% close to the optimal solution for Checkmate, MONeT-NoOp (MONeT with checkpointing enabled but operator-optimization disabled), and MONeT. We also provide Table A.1 which shows the time taken by the solver to reach 2% close to the optimal solution. We note that it has a similar behavior as the time taken by the solver to reach 5% close to the optimal solution. MONeT-NoOp converges to 2% close-to-optimal solution 1.3×-139× faster than Checkmate. For larger models, MONeT's solver converges to



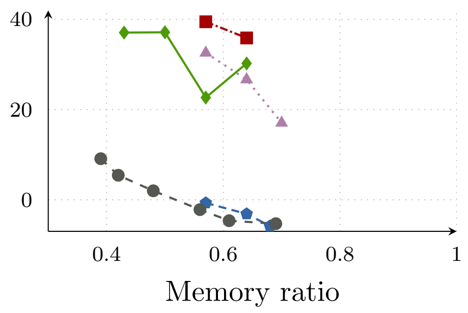
(a) **ResNet-50** (184)



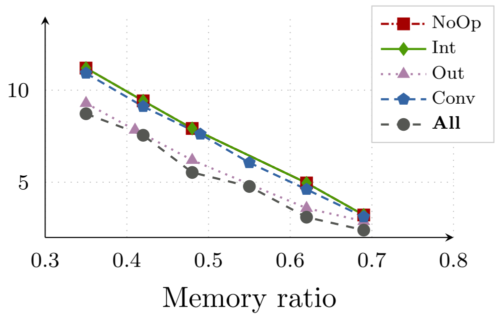
(b) **GoogleNet** (320)



(c) **UNet** (11)



(d) **VGG-16** (176)



(e) **Mobile-V2** (272)

Figure A.1: **Ablation results on ResNet-50, GoogleNet, UNet, VGG-16, MobileNet-V2.**

| | 5 GB | 6 GB | 7 GB | 8 GB | 9 GB | 10 GB |
|---|---|---|---|---|---|---|
| **ResNet-50** | | | | | | |
| Checkmate | - | **16.44** | 13.43 | 11.91 | 5.74 | 3.81 |
| MONeT-NoOp | - | 2.06 | 1.28 | 0.16 | 0.08 | 0.07 |
| MONeT | - | - | **12.64** | **3.00** | **3.60** | **0.62** |
| **GoogleNet** | | | | | | |
| Checkmate | - | 15.08 | **4.93** | 5.04 | 3.92 | 0.90 |
| MONeT-NoOp | 0.10 | 0.11 | 0.07 | 0.07 | 0.07 | 0.07 |
| MONeT | - | **5.47** | 5.34 | **0.31** | **0.25** | **0.24** |
| **MobileNet-V2** | | | | | | |
| Checkmate | **2.16** | **2.88** | **1.16** | 0.29 | 0.34 | 0.14 |
| MONeT-NoOp | 0.43 | 0.37 | 0.02 | 0.02 | 0.10 | 0.09 |
| MONeT | 9.49 | 5.33 | 1.53 | **0.14** | **0.18** | **0.05** |
| **UNet** | | | | | | |
| Checkmate | **0.243** | **0.031** | **0.027** | **0.021** | **0.011** | **0.009** |
| MONeT-NoOp | 0.181 | 0.003 | 0.003 | 0.002 | 0.002 | 0.002 |
| MONeT | 5.001 | 0.204 | 0.164 | 0.069 | 0.083 | 0.027 |
| **VGG-16** | | | | | | |
| Checkmate | - | - | - | **0.003** | **0.002** | **0.001** |
| MONeT-NoOp | - | - | - | 0.001 | 0.000 | 0.000 |
| MONeT | - | **0.004** | **0.006** | 0.004 | 0.003 | 0.003 |

Table A.1: **Solver time (in hours) to reach 2% close to optimal solution.** MONeT-NoOp reaches a 2% close-to-optimal solution 1.3×-139× faster than Checkmate. MONeT reaches a 2% close-to-optimal solution within a few hours in most cases, and up to 27× faster than Checkmate for larger models.

a 2% close-to-optimal solution up to 16× faster than Checkmate. At tighter memory limits for MobileNet-V2, the Checkmate solver reaches 2% close-to-optimal solution faster than MONeT, but is still much slower than MONeT-NoOp.

## A.3 Applicability to memory-intensive models

To further show MONeT's applicability to memory-intensive models, we evaluate it on 3D-UNet [29], a fully-convolutional model for volumetric images. Figure A.2 presents the runtime-memory trade-off for MONeT on 3D-UNet. We used a commonly used 3D-UNet implementation [118, 119] with training configuration sim-

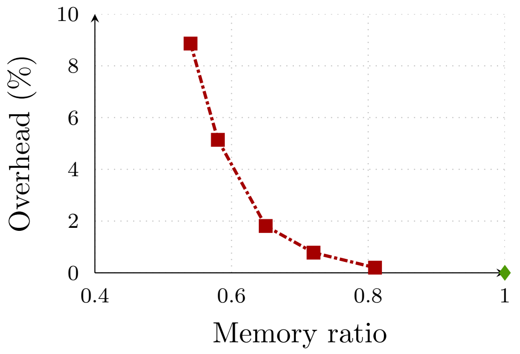


Figure A.2: **Runtime-memory trade-off curve for 3D-UNet using MONeT.** The green point denotes the PyTorch baseline.

ilar to `3DUnet_confocal_boundary` provided in the repository and a batch size of 22, which just fits on a 16 GB P100 GPU. MONeT reduces memory usage to 0.54× of PyTorch, while incurring 8.86% overhead in compute time. At a memory ratio of 0.81, MONeT incurs almost no computational overhead, because it makes use of operator optimizations and is able to bring down the recomputation cost to zero.

# Appendix B: Appendix for TACCL

## B.1 Writing a communication sketch

Here, we show an example of the communication sketch *dgx2-sk-1* used in the evaluation to synthesize an ALLGATHER algorithm for 2 Nvidia DGX-2 nodes (each node has 16 GPUs and 8 NICs, every two GPUs in the node share a NIC).

The sketch annotates the NVSwitch in each node and sets a switch-hyperedge strategy to minimize the number of links (denoted by `uc-min`). Further, the inter-node sketch fixes the sender and receiver GPUs in a node for inter-node data transfers. In our example, the odd-numbered GPUs sharing a NIC are chosen as senders and the even-numbered GPUs are chosen as receivers for inter-node communication. The user also annotates how the inter-node relay GPUs would split the inter-node bandwidth using a *beta_split* attribute. Since only a single GPU per NIC is chosen in our example to perform inter-node send and similarly receive, the bandwidth is not split. Optionally, the user can also map chunks to sender GPUs so that only mapped GPUs are used for inter-node transfers for the chunk. The *chunk_to_relay_map* attribute defines the parameters for the mapping function. The communication sketch also allows users to play with rotational symmetry for data routing. Given a symmetry offset and a group size, a chunk transfer over a link is set to be equivalent to a rotationally symmetric chunk over a rotationally symmetric link. In our example of the *symmetry_offset* attribute, using $[2, 16]$ fixes an intra-node symmetry with an offset of two, and using $[16, 32]$ fixes a symmetric data transfer pattern between the two DGX-2 nodes. Hyperparameters like input data partitioning and input size can also be provided via the communication sketch.

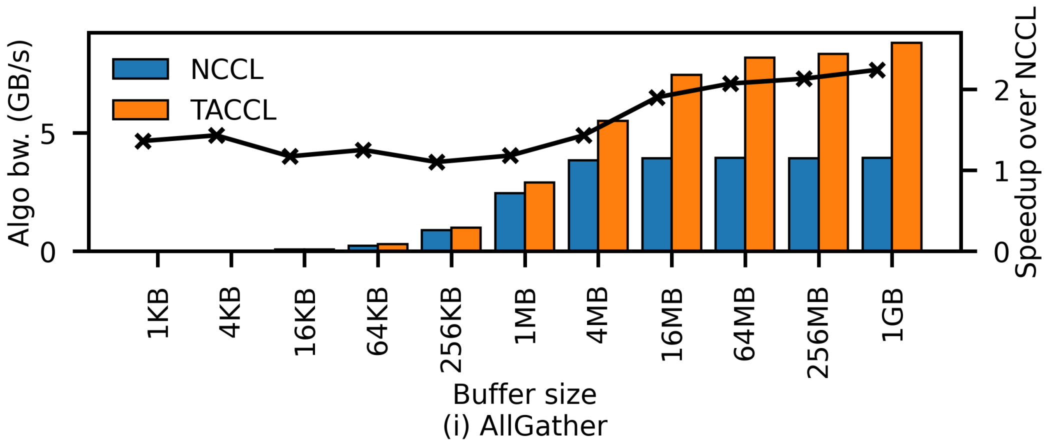
Listing B.1: Example sketch *dgx2-sk-1* for ALLGATHER

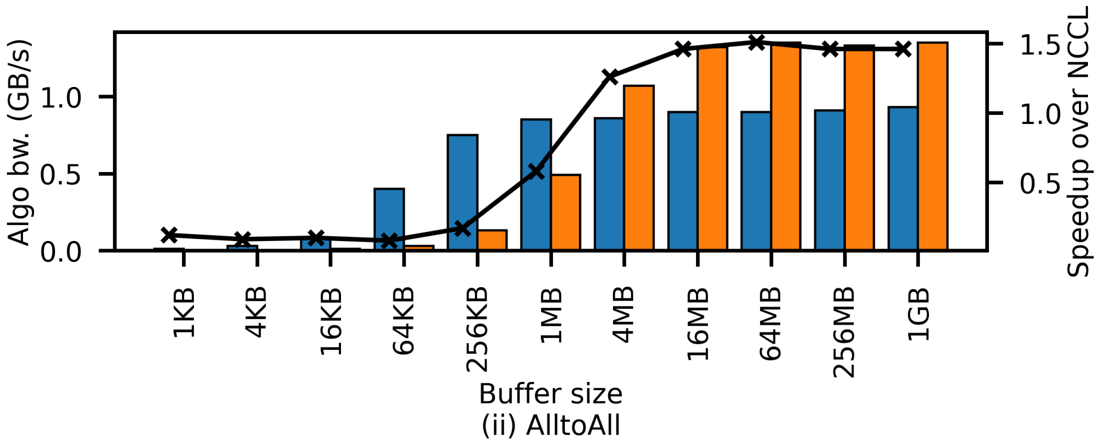
```
{
    // sketch for intra-node policy
    "intranode_sketch": {
        "strategy": "switch",
        "switches":
           [[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]],
        "switch_hyperedge_strategy": ["uc-min"]
    },

    // sketch for communication policy between any two nodes
    "internode_sketch": {
        "strategy": "relay",
        "internode_conn": {"1" : [0], "3" : [2], "5" : [4],
           "7" : [6], "9" : [8], "11" : [10], "13" : [12],
           "15" : [14]}, // "i": [j1, j2] implies GPU i in a
           node will only send data to GPU j1 and j2 of
           another node
        "beta_split": {"1": 1, "3": 1, "5": 1, "7" : 1, "9" :
            1, "11" : 1, "13" : 1, "15" : 1}, // "i": n
           implies inter-node sends from a GPU i of a node
           will use 1/n-th of the inter-node bandwidth
        "chunk_to_relay_map": [2,1] // maps chunk to a sender
            relay GPU. [r1,r2] means chunk c will be send to
           another node via GPU (rp//r1)*r1 + r2, where rp is
            the precondition GPU for chunk c
    },

    // enforces rotational symmetry.
    // [(o,g), ..]: o is the rotational offset and g is the
       group size for the rotational symmetry.
    // : eg. send(c,src,r) == send( (c + o)%g, (src + o)%g, (
       r + o)%g)
    "symmetry_offsets": [[2, 16], [16, 32]],

    "hyperparameters": {
        "input_chunkup": 2, // Data at each GPU is
           partitioned into 2 chunks that can be
           independently routed
        "input_size": "1M"
    }
}
```

## B.2 Standalone Experiments on Four Azure NDv2 Nodes

Figure B.1 shows additional algorithm bandwidth and the speedup over NCCL graphs of TACCL for Allgather, Alltoall, and Allreduce on 4-node NDv2 cluster. We synthesize all collectives using the *ndv2-sk-1* communication sketch (see § 5.8.2 for details), and lower them using 1 or 8 instances. We plot the best of the two algorithms over different buffer sizes.

TACCL's Allgather algorithms are $10\% - 2.2\times$ faster than NCCL across all buffer sizes. For Alltoall, the *ndv2-sk-1* sketch is most effective for large buffer sizes, and helps generate algorithms that are up to 46% faster than NCCL for buffer size greater than 1MB. TACCL Allreduce algorithms are up-to 34% faster than NCCL for small buffer sizes and $1.9\times - 2.1\times$ faster than NCCL for larger buffer sizes.





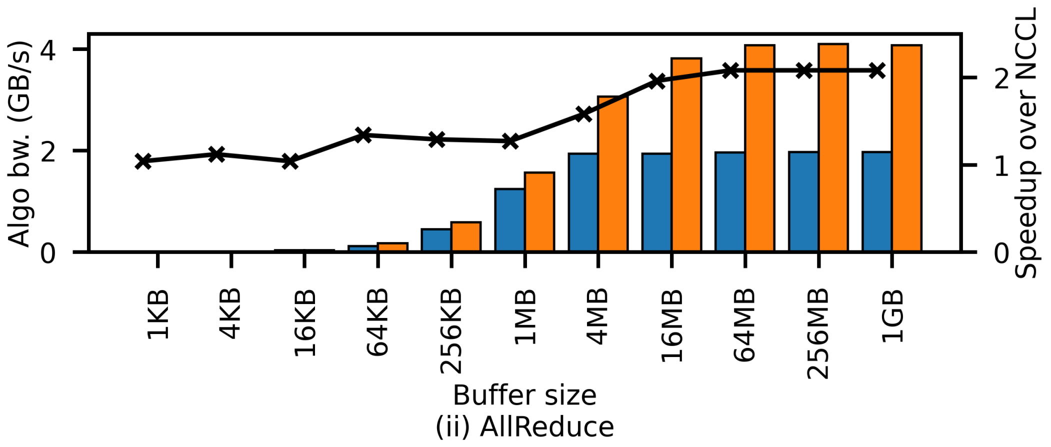


Figure B.1: **Algorithm bandwidth comparison of TACCL against NCCL for four Azure-NDv2 nodes.** TACCL algorithms compared against NCCL (left Y-axis) and their speedup over NCCL (right Y-axis) for Allgather, Alltoall, and Allreduce collectives on four NDv2 nodes.

# References

[1] URL `https://www.cerebras.net/`. Last Accessed August 2023.

[2] URL `https://www.graphcore.ai/`. Last Accessed August 2023.

[3] Cost-effective cloud edge traffic engineering with cascara. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, April 2021. URL `https://www.usenix.org/conference/nsdi21/presentation/singh`.

[4] GPUDirect RDMA, 2021. https://developer.nvidia.com/gpudirect.

[5] Megatron-LM. https://github.com/NVIDIA/Megatron-LM, 2022.

[6] Transformer-XL. https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/L XL, 2022.

[7] 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 Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *OSDI*, 2016. URL `https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi`.

[8] Akshay Agrawal, Robin Verschueren, Steven Diamond, and Stephen Boyd. A rewriting system for convex optimization problems. *Journal of Control and Decision*, 5(1):42–60, 2018.

[9] Azure ND-series. Azure ND-series, 2021. https://docs.microsoft.com/en-us/azure/virtual-machines/nd-series.

[10] Azure NDv2-series. Azure NDv2-series, 2021. https://docs.microsoft.com/en-us/azure/virtual-machines/ndv2-series.

[11] Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. Scalable methods for 8-bit training of neural networks. In *NeurIPS*, 2018. URL `http://papers.nips.cc/paper/7761-scalable-methods-for-8-bit-training-of-neural-networks`.

[12] Bard. Google bard. https://bard.google.com/. Accessed July 2023.

[13] Michael Barnett, Rick Littlefield, David G Payne, and Robert van de Geijn. Global combine on mesh architectures with wormhole routing. In *[1993] Proceedings Seventh International Parallel Processing Symposium*, pages 156–162. IEEE, 1993.

[14] Rudolf Bayer and Edward McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, pages 107–141, 1970.

[15] Shahid H Bokhari and Harry Berryman. Complete exchange on a circuit switched mesh. In *1992 Proceedings Scalable High Performance Computing Conference*, pages 300–301. IEEE Computer Society, 1992.

[16] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[17] Samuel Rota Bulò, Lorenzo Porzi, and Peter Kontschieder. In-place activated batchnorm for memory-optimized training of dnns. In *CVPR*, 2018. URL `http://openaccess.thecvf.com/content_cvpr_2018/html/Bulo_In-Place_Activated_BatchNorm_CVPR_2018_paper.html`.

[18] Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. Synthesizing optimal collective algorithms. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 62–75, 2021.

[19] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, modeling, and benchmarking {RocksDB}{Key-Value} workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, 2020.

[20] Ayan Chakrabarti and Benjamin Moseley. Backprop with approximate activations for memory-efficient network training. In *NeurIPS*, 2019.

[21] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert Van De Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007.

[22] chatgpt. Chatgpt. https://chat.openai.com/. Accessed July 2023.

[23] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L. Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *IEEE TPAMI.*, 2018.

[24] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *CoRR*, abs/1604.06174, 2016. URL `http://arxiv.org/abs/1604.06174`.

[25] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, 2014. URL `http://arxiv.org/abs/1410.0759`.

[26] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Using program synthesis for social recommendations. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, CIKM

’12, page 1732–1736, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450311564. doi: 10.1145/2396761.2398507. URL `https://doi.org/10.1145/2396761.2398507`.

[27] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *CoRR*, 2019. URL `http://arxiv.org/abs/1904.10509`.

[28] Minsik Cho, Ulrich Finkler, Mauricio Serrano, David Kung, and Hillery Hunter. Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy. *IBM Journal of Research and Development*, 63(6):1:1–1:11, 2019.

[29] Özgün Çiçek, Ahmed Abdulkadir, Soeren S Lienkamp, Thomas Brox, and Olaf Ronneberger. 3D U-Net: learning dense volumetric segmentation from sparse annotation. In *MICCAI*, 2016.

[30] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NeurIPS workshop*, 2011.

[31] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. From WiscKey to bourbon: A learned index for Log-Structured merge trees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 155–171. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL `https://www.usenix.org/conference/osdi20/presentation/dai`.

[32] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.

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

[34] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *NAACL*, 2019.

[35] Steven Diamond and Stephen Boyd. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 17 (83):1–5, 2016.

[36] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. Alex: an updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 969–984, 2020.

[37] Jens Dittrich, Joris Nix, and Christian Schön. The next 50 years in database indexing or: the case for automatically generated index structures. *Proceedings of the VLDB Endowment*, 15(3):527–540, 2021.

[38] Chao Dong, Chen Change Loy, Kaiming He, and Xiaoou Tang. Image super-resolution using deep convolutional networks. *IEEE TPAMI.*, 2016.

[39] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Evolution of development priorities in key-value stores serving large-scale applications: The {rocksdb} experience. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 33–49, 2021.

[40] Jack Dongarra et al. MPI: A message-passing interface standard version 3.0. *High Performance Computing Center Stuttgart (HLRS)*, 2(5):32, 2013.

[41] Economic value of LLM inference speedup. Economic value of LLM inference speedup. https://twitter.com/DrJimFan/status/1666132981839437825, 2023. Last Accessed August 2023.

[42] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. Learning to infer graphics programs from hand-drawn images. *Advances in neural information processing systems*, 31, 2018.

[43] Emmanuel Esposito, Thomas Mueller Graf, and Sebastiano Vigna. Recsplit: Minimal perfect hashing via recursive splitting. In *2020 Proceedings of the Twenty-Second Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 175–185. SIAM, 2020.

[44] Ohad Eytan, Danny Harnik, Effi Ofer, Roy Friedman, and Ronen Kat. It's time to revisit {LRU} vs.{FIFO}. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, 2020.

[45] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *CoRR*, abs/2101.03961, 2021. URL `https://arxiv.org/abs/2101.03961`.

[46] Paolo Ferragina and Giorgio Vinciguerra. The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment*, 13(8):1162–1175, 2020.

[47] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. Fiting-tree: A data-aware index structure. In *Proceedings of the 2019 international conference on management of data*, pages 1189–1206, 2019.

[48] Nadeen Gebara, Manya Ghobadi, and Paolo Costa. In-network aggregation for shared machine learning clusters. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages

829–844, 2021. URL https://proceedings.mlsys.org/paper/2021/file/eae27d77ca20db309e056e3d2dcd7d69-Paper.pdf.

[49] Aidan N. Gomez, Mengye Ren, Raquel Urtasun, and Roger B. Grosse. The reversible residual network: Backpropagation without storing activations. In *NeurIPS*, 2017. URL http://papers.nips.cc/paper/6816-the-reversible-residual-netwo

[50] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2022. URL https://www.gurobi.com.

[51] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[52] Roger W. Hockney. The communication challenge for mpp: Intel paragon and meiko cs-2. *Parallel Computing*, 20(3):389–398, 1994. ISSN 0167-8191. doi: https://doi.org/10.1016/S0167-8191(06)80021-9. URL https://www.sciencedirect.com/science/article/pii/S0167819106800219.

[53] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. *SIGCOMM Comput. Commun. Rev.*, 43(4):15–26, August 2013. ISSN 0146-4833.

[54] Stratos Idreos, Kostas Zoumpatianos, Manos Athanassoulis, Niv Dayan, Brian Hentschel, Michael S Kester, Demi Guo, Lukas M Maas, Wilson Qin, Abdul Wasay, et al. The periodic table of data structures. *IEEE Data Eng. Bull.*, 2018.

[55] Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Michael S Kester, and Demi Guo. The data calculator: Data structure design and cost synthesis from first principles and learned cost models. In *Proceedings of the 2018 International Conference on Management of Data*, pages 535–550, 2018.

[56] Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, et al. Design continuums and the path toward self-designing key-value stores that know and learn. In *CIDR*, 2019.

[57] Stratos Idreos, Kostas Zoumpatianos, Subarna Chatterjee, Wilson Qin, Abdul Wasay, Brian Hentschel, Mike Kester, Niv Dayan, Demi Guo, Minseo Kang, et al. Learning data structure alchemy. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 42(2), 2019.

[58] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. Gist: Efficient data encoding for deep neural network training. In *ISCA*, 2018. URL `https://doi.org/10.1109/ISCA.2018.00070`.

[59] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Kurt Keutzer, Ion Stoica, and Joseph E. Gonzalez. Checkmate: Breaking the memory wall with optimal tensor rematerialization. *CoRR*, 2019. URL `http://arxiv.org/abs/1910.02653`.

[60] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined wan. *SIGCOMM Comput. Commun. Rev.*, 43(4):3–14, August 2013. ISSN 0146-4833.

[61] Jinseong Jeon, Xiaokang Qiu, Jeffrey S Foster, and Armando Solar-Lezama. Jsketch: sketching for java. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 934–937, 2015.

[62] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *ACM MM*, 2014. doi: 10.1145/2647868.2654889. URL `https://doi.org/10.1145/2647868.2654889`.

[63] Srikanth Kandula, Ishai Menache, Roy Schwartz, and Spandana Raj Babbula. Calendaring for wide area networks. In *SIGCOMM'14*.

[64] Jiwon Kim, Jung Kwon Lee, and Kyoung Mu Lee. Accurate image super-resolution using very deep convolutional networks. In *CVPR*, 2016.

[65] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[66] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. Sosd: A benchmark for learned indexes. *arXiv preprint arXiv:1911.13014*, 2019.

[67] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. RadixSpline: a single-pass learned index. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2020, Portland, Oregon, USA, June 19, 2020*, pages 5:1–5:5, 2020. doi: 10.1145/3401071.3401659. URL `https://doi.org/10.1145/3401071.3401659`.

[68] Tim Kraska. Towards instance-optimized data systems. *Proceedings of the VLDB Endowment*, 14(12), 2021.

[69] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 international conference on management of data*, pages 489–504, 2018.

[70] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *NeurIPS*, 2012.

[71] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. ATP: In-network aggregation for multi-tenant learning. In *18th USENIX Symposium on Networked Systems Design and Implemen-*

*tation (NSDI 21)*, pages 741–761. USENIX Association, April 2021. ISBN 978-1-939133-21-2. URL https://www.usenix.org/conference/nsdi21/presentation/lao.

[72] Frank Thomson Leighton, Bruce M Maggs, and Satish B Rao. Packet routing and job-shop scheduling in o (congestion+ dilation) steps. *Combinatorica*, 14 (2):167–186, 1994.

[73] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *CoRR*, abs/2006.16668, 2020. URL https://arxiv.org/abs/2006.16668.

[74] Antoine Limasset, Guillaume Rizk, Rayan Chikhi, and Pierre Peterlongo. Fast and scalable minimal perfect hashing for massive key sets. *arXiv preprint arXiv:1702.03154*, 2017.

[75] Liang Luo, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. Parameter hub: A rack-scale parameter server for distributed deep neural network training. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 41–54, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450360111. doi: 10.1145/3267809.3267840. URL https://doi.org/10.1145/3267809.3267840.

[76] Liang Luo, Peter West, Jacob Nelson, Arvind Krishnamurthy, and Luis Ceze. Plink: Discovering and exploiting locality for accelerated distributed training on the public cloud. In *Proceedings of Machine Learning and Systems 2020*, pages 82–97. 2020.

[77] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. Benchmarking learned indexes. *arXiv preprint arXiv:2006.12804*, 2020.

[78] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory F. Diamos, Erich Elsen, David García, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. In *ICLR*, 2018. URL `https://openreview.net/forum?id=r1gs9JgRZ`.

[79] Microsoft SCCL. Microsoft sccl, 2021. https://github.com/microsoft/sccl.

[80] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, et al. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 993–1011, 2022.

[81] NCCL Tests. Nccl tests, 2021. https://github.com/NVIDIA/nccl-tests.

[82] NCCL Tree Algorithm. NCCL Tree Algorithm, 2019. https://developer.nvidia.com/blog/massiv
scale-deep-learning-training-nccl-2-4.

[83] Nvidia DGX Systems. Nvidia DGX Systems, 2021. https://www.nvidia.com/en-us/data-center/dgx-systems/.

[84] Nvidia Infiniband. Nvidia InfiniBand, 2021. https://www.nvidia.com/en-us/networking/infiniband-adapters/.

[85] NVIDIA NCCL. Nvidia nccl, 2021. https://github.com/nvidia/nccl.

[86] Nvidia NVLink. Nvidia NVLink and NVSwitch, 2021. https://www.nvidia.com/en-us/data-center/nvlink/.

[87] Nvidia NVSWITCH. NVIDIA NVSWITCH The World's Highest-Bandwidth On-Node Switch , 2021. https://images.nvidia.com/content/pdf/nvswitch-technical-overview.pdf.

[88] OpenStreetMap contributors. Planet dump retrieved from https://planet.osm.org . `https://www.openstreetmap.org`, 2017.

[89] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33:351–385, 1996.

[90] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, 2019. URL `http://papers.nips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library`.

[91] Shishir G Patil, Paras Jain, Prabal Dutta, Ion Stoica, and Joseph Gonzalez. Poet: Training neural networks on tiny devices with integrated rematerialization and paging. In *International Conference on Machine Learning*, pages 17573–17583. PMLR, 2022.

[92] Giulio Ermanno Pibiri and Roberto Trani. Pthash: Revisiting fch minimal perfect hashing. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 1339–1348, 2021.

[93] Jelena Pješivac-Grbović, Thara Angskun, George Bosilca, Graham E Fagg, Edgar Gabriel, and Jack J Dongarra. Performance analysis of mpi collective operations. *Cluster Computing*, 10(2):127–143, 2007.

[94] Geoff Pleiss, Danlu Chen, Gao Huang, Tongcheng Li, Laurens van der Maaten, and Kilian Q. Weinberger. Memory-efficient implementation of densenets. *CoRR*, 2017. URL `http://arxiv.org/abs/1707.06990`.

[95] Arslan Sajid Raja, Sophie Lange, Maxim Karpov, Kai Shi, Xin Fu, Raphael Behrendt, Daniel Cletheroe, Anton Lukashchuk, Istvan Haller, Fotini Karinou, et al. Ultrafast optical circuit switching for data centers using integrated soliton microcombs. *Nature communications*, 12(1):5867, 2021.

[96] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *MICCAI*, 2015. doi: 10.1007/978-3-319-24574-4_28. URL `https://doi.org/10.1007/978-3-319-24574-4_28`.

[97] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[98] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[99] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtárik. Scaling distributed machine learning with {In-Network} aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808, 2021.

[100] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

[101] David S Scott. Efficient all-to-all communication patterns in hypercube and mesh topologies. In *The Sixth Distributed Memory Computing Conference, 1991. Proceedings*, pages 398–399. IEEE Computer Society, 1991.

[102] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow, 2018.

[103] Aashaka Shah, Chao-Yuan Wu, Jayashree Mohan, Vijay Chidambaram, and Philipp Krähenbühl. Memory optimization for deep networks. *arXiv preprint arXiv:2010.14501*, 2020.

[104] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. {TACCL}: Guiding collective algorithm synthesis using communication sketches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 593–612, 2023.

[105] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, abs/1909.08053, 2019. URL `http://arxiv.org/abs/1909.08053`.

[106] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

[107] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015. URL `http://arxiv.org/abs/1409.1556`.

[108] Rachee Singh, Manya Ghobadi, Klaus-Tycho Foerster, Mark Filer, and Phillipa Gill. Radwan: Rate adaptive wide area network. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 547–560, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355674. doi: 10.1145/3230543.3230570. URL `https://doi.org/10.1145/3230543.3230570`.

[109] Rachee Singh, Nikolaj Bjorner, Sharon Shoham, Yawei Yin, John Arnold, and Jamie Gaudette. Cost-effective capacity provisioning in wide area networks with shoofly. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 534–546, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383837. doi: 10.1145/3452296.3472895. URL `https://doi.org/10.1145/3452296.3472895`.

[110] Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, USA, 2008. AAI3353225.

[111] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, page 404–415, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595934510. doi: 10.1145/1168857.1168907. URL `https://doi.org/10.1145/1168857.1168907`.

[112] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. Sketching stencils. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 167–178, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595936332. doi: 10.1145/1250734.1250754. URL `https://doi.org/10.1145/1250734.1250754`.

[113] Aravind Srinivasan and Chung-Piaw Teo. A constant-factor approximation algorithm for packet routing, and balancing local vs. global criteria. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 636–643, 1997.

[114] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Go-

ing deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[115] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[116] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Nikhil Devanur, Jorgen Thelin, and Ion Stoica. Blink: Fast and generic collectives for distributed ml. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 172–186, 2020. URL `https://proceedings.mlsys.org/paper/2020/file/43ec517d68b6edd3015b3edc9a11367b-Paper.pdf`.

[117] Shmuel Winograd. *Arithmetic complexity of computations*. Siam, 1980.

[118] Adrian Wolny. PyTorch 3D-UNet. `https://github.com/wolny/pytorch-3dunet`, 2019.

[119] Adrian Wolny, Lorenzo Cerrone, Athul Vijayan, Rachele Tofanelli, Amaya Vilches Barro, Marion Louveaux, Christian Wenzl, Susanne Steigleder, Constantin Pape, Alberto Bailoni, Salva Duran-Nebreda, George Bassel, Jan U. Lohmann, Fred A. Hamprecht, Kay Schneitz, Alexis Maizel, and Anna Kreshuk. Accurate and versatile 3d segmentation of plant tissues at cellular resolution. *bioRxiv*, 2020.

[120] Chaichon Wongkham, Baotong Lu, Chris Liu, Zhicong Zhong, Eric Lo, and Tianzheng Wang. Are updatable learned indexes ready? *arXiv preprint arXiv:2207.02900*, 2022.

[121] Jiacheng Wu, Yong Zhang, Shimin Chen, Jin Wang, Yu Chen, and Chunxiao Xing. Updatable learned index with precise positions. *arXiv preprint arXiv:2104.05520*, 2021.

[122] Ningning Xie, Tamara Norman, Dominik Grewe, and Dimitrios Vytiniotis. Synthesizing optimal parallelism placement and reduction strategies on hierarchical systems for deep learning. *CoRR*, abs/2110.10548, 2021. URL https://arxiv.org/abs/2110.10548.

[123] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. In *NeurIPS*, 2019.

[124] Zhen Zhang, Chaokun Chang, Haibin Lin, Yida Wang, Raman Arora, and Xin Jin. Is network the bottleneck of distributed training? In *Proceedings of the Workshop on Network Meets AI & ML*, pages 8–13, 2020.

[125] Chenxingyu Zhao, Tapan Chugh, Jaehong Min, Ming Liu, and Arvind Krishnamurthy. Dremel: Adaptive configuration tuning of rocksdb kv-store. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6 (2):1–30, 2022.

[126] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.