Scalable Machine Learning using Dataflow Graph Analysis

by

Chien-Chin Huang

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

Courant Institute of Mathematical Sciences

New York University

September, 2019

Professor Jinyang Li

© Chien-Chin Huang

All Rights Reserved, 2019

Dedication

To Siao-Ting, Ke-Shiuan, Kitty, and my family.

Abstract

In the past decade, the abundance of computing resources and the growth of data have boosted the development of machine learning applications. Many computation frameworks, e.g., Hadoop, Spark, TensorFlow, and PyTorch, have been proposed and become widely used in the industry. However, programming large-scale machine learning applications is still challenging and requires the manual efforts of developers to achieve good performance. This thesis discusses two major issues with the existing frameworks.

First, array is a popular data abstraction for machine learning computation. When parallelizing arrays to hundreds of CPU machines, it is critical to choose a good partition strategy to co-locate the computation arrays to reduce network communication. Unfortunately, existing distributed array frameworks usually use a fixed partition scheme and requires manually partitioning if another parallel strategy is used, making it less easy to develop a distributed array program. Secondly, GPU is widely used for a popular branch of machine learning applications, deep learning. Modern GPU can be orders of magnitude faster than CPU and becomes an attractive computation resource. However, the limited memory size of GPU restricts the scale of the DNN models can be run. It is desirable to

have a computation framework to allow users to explore deeper and wider DNN models by leveraging the CPU memory.

Modern machine learning frameworks generally adopt a dataflow-style programming paradigm. The dataflow graph of an application exposes valuable information to optimize the application. In this thesis, we present two techniques to address the above issues via dataflow graph analysis.

We first design Spartan to help users parallelize distributed arrays on a CPU cluster. Spartan is a distributed array framework, built on top of a set of higher-order dataflow operators. Based on the operators, Spartan provides a collection of Numpy-like array APIs. Developers can choose the built-in array APIs or directly use the operators to construct machine learning applications. To achieve good performance for the distributed application, Spartan analyzes the communication pattern of the dataflow graph captured through the operators and applies a greedy strategy to find a good partition scheme to minimize the communication cost.

To support memory-intensive deep learning applications on a single GPU, we develop SwapAdvisor, a swapping system that automatically swaps temporarily unused tensors from GPU memory to CPU memory. To minimize the communication overhead, SwapAdvisor analyzes the dataflow graph of the given DNN model and uses a custom-designed genetic algorithm to optimize the operator scheduling and memory allocation. Based on the optimized operator schedule and memory allocation, SwapAdvisor can determine what and when to swap to achieve good performance.

Table of contents

Dedication **iii**

Abstract **iv**

List of Figures **ix**

List of Tables **xi**

1 Introduction **1**

1.1 Evolution of Computation Frameworks for Machine Learning . . 3

1.2 Challenges in Scaling Machine Learning 7

1.3 Automatic Array Partitioning with Spartan 9

1.4 Leveraging CPU Memory with SwapAdvisor for Large DNN Models 11

1.5 Contributions 12

2 Spartan Design **14**

2.1 Overview 14

2.2 Automatic Tiling Overview 15

2.2.1 What Affects Good Tiling? 15

2.2.2 Our Approach and Spartan Overview 18
2.3 Smart Tiling with High-level Operators 20
2.3.1 High-level Operators 20
2.3.2 Expression Graph Capture 24
2.3.3 Graph-based Tiling Optimizer 26
2.3.4 Additional Tiling Optimizations 31
2.4 Implementation 32

3 Spartan Evaluation 34

3.1 Experimental Setup 34
3.2 Tiling 35
3.3 Scaling 39
3.4 Comparison with Other Systems 42

4 SwapAdvisor Design 43

4.1 Background 43
4.2 Challenges and Our Approach 45
4.3 SwapAdvisor Design 50
4.3.1 Operator Schedule and Memory Allocation 51
4.3.2 Swap Planning 53
4.4 Optimization via Genetic Algorithm 56
4.4.1 Algorithm Overview 58
4.4.2 Creating New Schedules 59
4.4.3 Creating New Memory Allocation 61

4.5 Implemetation 65

5 SwapAdvisor Evaluation **66**

5.1 Experimental Setup 67

5.2 Wider and Deeper DNN Models Training 71

5.3 DNN Models Inference Evaluation 73

5.4 The Effectiveness of SwapAdvisor's Design Choices 77

6 Related Work **79**

6.1 Spartan's Related Work 79

6.2 SwapAdvisor's Related Work 83

7 Conclusion **87**

Appendix A NP-Completeness Proof of Tiling Optimizaion **89**

Bibliography **96**

List of Figures

2.1 Three tiling methods for a two-dimensional array 15

2.2 Two approaches to implement a distributed matrix multiplication . . 16

2.3 Pseudocode of Alternating Least Squares 17

2.4 Overview of Spartan's architecture 19

2.5 Implementations of add and dot in Spartan 25

2.6 Expression and tiling graphs for $Z = X + Y - X \cdot Y$ in Spartan . . . 26

2.7 Examples to build a tiling graph in Spartan 27

2.8 The maximum connectivity group first algorithm for Spartan 30

3.1 Runtime comparison between Spartan's smart tiling and the best tiling 35

3.2 The performance of Spartan's smart tiling 37

3.3 An example that smart tiling gives sub-optimal tiling 38

3.4 Scaling results for fixed input and varied number of workers for Spartan 40

3.5 Scaling input size on local cluster for Spartan 41

3.6 Scaling input size on 128 EC2 instances for Spartan 41

4.1 Different schedules and memory allocation can affect swapping . . . 46

4.2 System overview of SwapAdvisor 51
4.3 Example swap planning for a simple dataflow graph 55
4.4 Random scheduling and memory allocation for Inception-V4 57
4.5 SwapAdvisor scheduling crossover example 59
4.6 SwapAdvisor memory allocation crossover example 62

5.1 SwapAdvisor's normalized throughput relative to the ideal 68
5.2 WResNet-152-4 throughput comparison 73
5.3 99 percentile latency versus throughput of SwapAdvisor 74
5.4 SwapAdvisor's search performance 75
5.5 SwapAdvisor's search results with different search settings 75

A.1 Example node groups and edge relationship for Spartan's proof . . . 91

List of Tables

3.1 K-Means performance comparison for Spartan 42

5.1 Memory usage and dataflow graph Statistics of DNN models. 71

5.2 ResNet-152 inference time with SwapAdvisor 74

Chapter 1

Introduction

Machine learning, the study of extracting and learning patterns from the data, has been increasingly popular in the past decade. Many different algorithms have been introduced and are widely used [1, 2]. One can roughly categorize the state-of-art machine learning algorithms into two types. The first one is the algorithms that can learn from structured data and is referred to as traditional machine learning in this thesis. Another type is the algorithms which adopt artificial neural networks to learn from unstructured (or unprocessed) data and is usually referred to as deep learning.

One primary reason for the popularity of machine learning is the arrival of the so-call "Big Data" era. With the availability of a large amount of useful data, the accuracy of machine learning applications have been significantly boosted. In addition, many computation frameworks have been proposed to help scale machine learning. For example, MapReduce [3] and Spark [4] are two popular general-purpose distributed data processing frameworks. Based on these general-

purpose frameworks, MLBase [5] and Mahout [6] are developed to provide a higher-level data abstraction (array) and array libraries for machine learning developers. These frameworks are mainly used for traditional machine learning. Several specialized array frameworks, e.g., TensorFlow [7], PyTorch [8], and MXNet [9], are designed to support deep learning development. Despite these efforts, scaling machine learning applications is still challenging for programmers.

First, traditional machine learning applications generally require an enormous memory footprint to do the computation due to the vast amount of the input data. It is common to use a cluster of CPU machines to distribute the arrays to process the large inputs. One key performance factor when distributing arrays is to partition the arrays in a way such that computation data is co-located. The existing machine learning frameworks mostly require users to manually design a good array partitioning strategy. However, manual array partitioning can be painful. Ideally, a distributed array framework should support automatic array partitioning with minimal user efforts to achieve both ease-of-use and high performance.

Secondly, the computation of an artificial neural network is extremely heavy and dense, and has been shown that it is not easy to be managed by CPUs [2]. The arrival of the general-purpose GPU provides a powerful matrix (array) computation resource, boosting the development of deeper artificial neural network models, also known as deep neural networks (DNN). Unfortunately, GPU memory size is far less than CPU, restricting the opportunities for developers to explore various deeper and wider neural network models where the memory re-

quirement exceeds a single GPU memory capacity. Consequently, it is desirable to utilize the CPU memory for running a large deep learning model.

This thesis first presents Spartan, a distributed array framework for parallelizing traditional machine applications with automatic array partitioning. The second part of the thesis discusses SwapAdvisor, an array swapping framework for deep learning.

1.1 Evolution of Computation Frameworks for Machine Learning

The arrival of the "Big Data" era makes it easy to obtain a large amount of useful data. Additionally, the public availability of large-scale computing clusters (e.g., Amazon EC2, Google Cloud Engine, and Microsoft Azure) allows people to parallelize computation with abundant computing resources. However, it is challenging to manage massive data and powerful hardware without an easy-to-use and efficient distributed computation framework. Many distributed computation frameworks have been proposed to ease the burden.

General-purpose distributed computation frameworks: Since the introduction in 2004, MapReduce and its open-source counterpart, Hadoop [10], have been widely used for distributed data processing. MapReduce provides a dataflow-style, restricted programming paradigm. The distributed processes in MapReduce do not communicate with each other directly but operate on a set of read-

only key-value data collections. Programmers use two functional, data-parallel operators, “Map” and “Reduce” to process (or transform) the data collections. Internally, MapReduce performs a “Shuffle” operation to redistribute the data by keys. All the results (intermediate and final) are stored in a distributed system (e.g., GFS [11] and HDFS [12]) before being reused.

The success of MapReduce (and Hadoop) has inspired many distributed frameworks to target primitives for key-value collections (e.g., Spark [4], Dryad [13], Picolo [14], Dandelion [15], and Naiad [16]). Among them, Spark is the most notable one.

Spark works upon a set of read-only resilient distributed datasets (RDD), which reside in the distributed shared memory. The shared RDDs reduce the need to read from the distributed file system. Spark also adopts lazy evaluation to delay materializing RDDs until necessary in order to construct a dataflow graph of the application. Via analyzing the dataflow graph, Spark can optimize the computation and can further reduce the unnecessary intermediate data.

Distributed array frameworks: The popularity of MapReduce and Spark comes from the dataflow-style operators, which isolate the computation from the physical data communication and distribution, allowing users to focus on processing local data. Unfortunately, the key-value collection representation in MapReduce and Spark is designed for general usages, too low-level for machine learning developers.

In the machine learning community, array (or matrix) is the primary data abstraction as it can be used to represent the mathematical meaning of machine

algorithms. Many distributed array frameworks are designed to provide an array-based programming interface for machine learning users. We can categorize these existing array frameworks into two types.

The first type of frameworks are based on the general-purpose distributed frameworks and provide array builtins and abstraction to users. MLBase [5] and Mahout [6] are two notable libraries, which provide a set of machine learning and linear algebra algorithms. To leverage the existing distributed frameworks, ML-Base and Mahout translate the array information to the underlying framework's key-value representations. Users can directly use these array builtins to implement machine learning applications but may have to fall back to the low-level operators when designing a new algorithm which is not supported.

The second type of frameworks extend and parallelize an existing programming language. Presto (Distributed R) [17] aims to distribute a popular array programming language, R, with a set of customized data structures (e.q., distributed array) and APIs (e.q., foreach). Users write machine learning algorithms with the original R syntax but use the extension structures to store the distributed data and use the extension APIs to iterate the distributed arrays for computation. Similar to Presto, MadLINQ [18] extends a general programming language, C#, with a domain-specific language (DSL) to distribute and iterate arrays. Additionally, MadLINQ is integrated with DryadLINQ [19], a general-purpose distributed data processing framework. This design allows MadLINQ to provide a full data pipeline for machine learning developers, from processing the raw data to learning with the structured data.

Deep learning frameworks: Deep learning applications also use array (is referred to as tensor in the deep learning community) as the main data abstraction. However, there are several factors which make the discussed array frameworks less attractive for the deep learning community.

First, the computation of deep learning applications is extremely heavy. Consequently, most users adopt GPU to be the primary computation platform to speed up the development. Many existing distributed array frameworks aim to support a cluster of CPU machines and do not support GPU. Furthermore, to best utilize the GPU computation power while alleviating user programming efforts, several specialized libraries have been developed for deep learning (e.g., CUDA [20] and CUDNN [21]). Deep learning users tend to use these APIs as the underlying computation units.

More importantly, the deep learning community adopts a gradient-descent-based approach when training a deep learning model [2]. Programmers first apply the calculus chain rule layer by layer to get the gradient formulas. The gradient formulas are then converted to expressions of GPU libraries to perform the actual computation. The process is tedious and error-prone as a deep learning model can have hundreds of layers [22].

Many deep learning frameworks have been introduced to solve these issues (e.g., TensorFlow [7], PyTorch [8], and MXNet [9]). In these frameworks, CUDA and CUDNN are wrapped as the basic computation APIs. When users develop a deep neural network model, these frameworks construct a dataflow graph made by the computation APIs. With the dataflow graph capturing the mathematical

information of the model, these frameworks apply the chain rule to the dataflow graph to derive the gradients of the parameters automatically for users (the technique is referred to as backward-propagation in the deep learning community).

1.2 Challenges in Scaling Machine Learning

Although many computation frameworks have been developed (discussed in Section 1.1), scaling machine learning applications remains difficult for programmers. In this section, we present two main challenges.

Manual array partitioning is painful: When distributing traditional machine learning programs on a cluster of CPU machines, the open challenge is how to maximize the locality of access to array data spread out across the memory of many machines. To improve locality, one needs to both partition arrays smartly and co-locate computation with data. A good data locality can significantly reduce network communication. We refer to this as the "tiling" problem. Tiling is crucial for performance; programs that optimize for locality can be an order of magnitude faster than those that don't.

Existing distributed array frameworks do not adequately address the tiling problem. Most systems rely on users to manually specify array partitioning. However, manual tiling can be painful. First, developers usually use the array builtins provided by a distributed array framework to design machine learning algorithms. In order to best partition the arrays, developers need to understand how these arrays are accessed by the builtins. More importantly, a machine

learning program usually contains several array expressions, resulting in many different ways to tile the entire program. Developers needs to explore different tiling strategies to get excellent performance. Finally, a machine learning application may use multidimensional arrays (e.g., 3D images) which have more ways to partition, resulting in more complicated tiling strategies. Consequently, manual tiling is tedious and error-prone and should be avoided as possible.

Limited GPU memory size restricts the DNN model size: Deep learning community adopts stochastic gradient descent (SGD) [2] as the main algorithm to search the model parameters for a deep neural network (DNN). This design allows developers to partition the input data into batches and utilize only a single batch of data per iteration. Nevertheless, as DNN models become deeper (more layers) and wider (more parameters), the computation of DNNs has been increasingly dense. As a result, it is still very common to distribute an deep learning application to multiple GPUs.

Data parallelism is the most popular way to parallelize a DNN model due to its simplicity. A data parallelized DNN program duplicates the parameter tensors to all the GPUs, and each GPU runs the same model with a different portion of the data. As a result, the implementation of a single-GPU DNN model can be used for a data-parallel environment with minimal modifications.

However, duplicating all the parameters prevents data parallelism from supporting huge DNN models where the size of parameters exceeds a single GPU's memory capacity. The parameter sizes of DNN models have doubled roughly every 2.4 years in the past decade [2] while the GPU memory capacity has only

increased by 4× (4GB to 16GB). Consequently, the GPU memory capacity limits the opportunities for the developers to explore much larger DNN models. CPU memory size is orders of magnitude larger than GPU memory; it is not uncommon for a CPU machine to have hundreds of gigabytes memory or even terabytes memory. As a result, it is desirable to leverage CPU memory to scale DNN models with either a single GPU or multiple GPUs with data parallelism.

We next discuss how to solve these two challenges with Spartan (Section 1.3) and SwapAdvisor (Section 1.4).

1.3 Automatic Array Partitioning with Spartan

We first propose Spartan to achieve automatic array partitioning (also referred to as "automatic tiling") for traditional machine learning. Spartan, a distributed array framework with smart tiling, provides the popular NumPy [23] array abstractions while achieving scalable, high performance across machines. The key innovation of Spartan is its automatic tiling mechanism: when distributing an n-dimensional array across machines, the runtime of Spartan can automatically decide which axis(es) to cut each array along and to co-locate computation with data.

A major design of Spartan is the five high-level parallel operators, including map, fold, filter, scan, and join_update. These high-level operators capture the parallel patterns of most array programs, and we use them to distribute a myriad of built-in array functions as well as user programs. A critical difference between the operators and MapReduce's (or Spark's) operators is the semantics of these

operators are array-based. The operators work on a set of distributed-arrays, instead of the lower-level, opaque data collections. The semantics of these high-level operators lead to well-defined cost profiles. The cost profile of an operator gives an estimate of the communication cost for each potential tiling strategy (e.g., row-wised and column-wised) for its inputs. Therefore, it provides crucial information to enable the runtime to perform automatic tiling. As an example, the map operator applies a user-defined function element-wise to several input arrays with the same shape. Thus, this operator achieves the best locality (and zero communication cost) if all its input arrays are partitioned in the same way. Otherwise, the cost equals to the size of those input arrays with different tiling.

At runtime, Spartan splits program execution into a series of frontend and backend steps. On the client machine, the frontend first turns a user program into an expression graph (dataflow graph) of high-level operators via lazy evaluation. It then runs a greedy search algorithm to find a good tiling for each node in the expression graph to reduce the overall communication cost. Finally, the frontend gives the tiled expression graph to the backend for execution. The backend creates distributed arrays according to the assigned tiling and evaluates each operator by scheduling parallel tasks among a collection of workers.

We have built Spartan to provide similar user interfaces as NumPy. Evaluations on a local cluster and the Amazon EC2 show that Spartan's tiling algorithm can automatically find good tiling for arrays and achieve good scalability.

1.4 Leveraging CPU Memory with SwapAdvisor for Large DNN Models

Swapping tensor data between GPU and CPU memory during deep learning computation is a promising approach to address the GPU memory limitation [24, 25, 26, 27]. Several technological trends make swapping attractive: 1) CPU memory is much larger and cheaper than GPU memory, 2) modern GPU hardware can effectively overlap communication with computation, 3) communication bandwidth between GPU and CPU is sufficiently good now and can be significantly improved with the arrival of PCIe 5.0 [28] and the wide adoption of NVLink [26, 29].

Swapping for DNN computation differs from traditional swapping (between CPU memory and disk) in that the DNN computation structure is usually known prior to execution, e.g., in the form of a dataflow graph. Such knowledge unleashes tremendous opportunity to optimize swapping performance by maximally overlapping computation and communication. Unfortunately, existing work either do not utilize this information (e.g., TensorFlow's swap extension [30]) or only use it in a rudimentary way based on manual heuristics [26, 25, 27]. For example, TFLMS [26] and vDNN [24] swap only activation tensors according to their topological sort order in the graph. SuperNeurons [25] only swaps data for convolution operations. As a result, not only do these work support only limited types of DNNs but they also fail to achieve the full performance potential of swapping. SwapAdvisor is a general swapping system which can support various

kinds of large model training and inference with limited GPU memory. For a given DNN computation, SwapAdvisor plans for what and when to swap precisely prior to execution in order to maximize computation and communication overlap.

A dataflow graph alone is not sufficient for such precise planning, which is also dependent on how operators are scheduled to execute and how the memory allocation is done. More importantly, memory allocation and operator scheduling also critically affect the best achievable swapping performance. SwapAdvisor uses a custom-designed genetic algorithm to search the space of all memory allocation and operator schedules so that the final swapping plan represents the result of joint optimization over operator scheduling, memory allocation and swapping.

SwapAdvisor can also be used for model inference. Inference has a smaller memory footprint than training. However, to save cost, one may have multiple models use a single GPU. In this setup, one can use SwapAdvisor to constrain each model to use only a fraction of the memory as opposed to time share the entire memory across models.

1.5 Contributions

In this thesis, we explore the opportunity to scale machine learning applications with minimal manual user efforts via dataflow graph analysis. In particular, this thesis makes the following contributions:

Automatic array tiling for large-scale machine learning: Design and implementation of Spartan, a distributed array framework that provides a smart tiling algorithm to partition distributed arrays effectively. With a set of carefully chosen high-level operators, Spartan provides good programmability while still achieves excellent performance.

Smart swapping to enable deeper and wider DNN models: SwapAdvisor is the first swapping system that supports various types of large DNN models for both model training and inference. Given the dataflow graph and memory usage of a DNN model, SwapAdvisor optimizes both operators and memory allocation scheduling to obtain a good swapping plan for the model.

The remainder of this thesis is organized as follows. We first present the design of Spartan in Chapter 2 and evaluate the performance of Spartan with different machine learning applications in Chapter 3. Chapter 4 details how SwapAdvisor can derive a good swapping plan by controlling the schedules and memory allocation. In Chapter 5, we show that SwapAdvisor can achieve good performance for DNN training and inference with limited GPU memory. Finally, the related work for Spartan and SwapAdvisor are presented in Chapter 6.

Chapter 2

Spartan Design

2.1 Overview

The Spartan system is comprised of many worker machines in a high speed cluster. Spartan partitions each global array into several tiles (sub-arrays) and distributes each one to a potentially different worker. We refer to the partitioning strategy as *tiling*. There are several ways to "tile" an array. For example, Figure 2.1 shows the three tiling choices for a 2D array (aka matrix).

In Spartan, an array is created by loading data from an external storage or as a result of some computation. Spartan decides the tiling choice for the array at its creation time. What is a good tiling choice? We consider the best tiling as one that incurs the minimum communication cost when the array is used in a computation – workers fetch and write as few remote tiles as possible. In next section, we examine what affects good tiling and give an overview of Spartan's approach to automatic tiling.

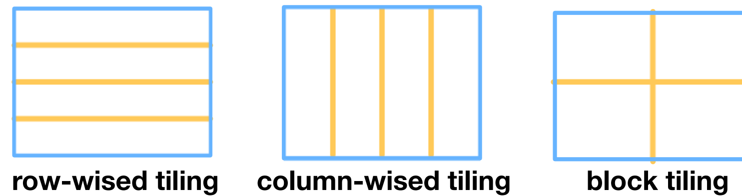


Figure 2.1: Three tiling methods for a two-dimensional array.

2.2 Automatic Tiling Overview

2.2.1 What Affects Good Tiling?

Several factors affect the tiling choice for an array. These include how the computation accesses the array, the runtime information of the array and how the array is used across the program. Below, we illustrate how each of the factors affects tiling using concrete examples.

1) The access pattern of an array. Array computation tends to read or update an array along some particular axis. This access information is crucial for determining a good tiling. Figure 2.2(a) shows the access pattern of a common implementation of matrix multiplication (aka dot). When computing $X \cdot Y = Z$, this implementation launches p parallel tasks each of which reads X row-wise and reads the entirety of Y. The task then performs a local dot and sends the result row-size to create Z. Consequently, it is best to tile both X and Z row-wise (it does not matter how Y is tiled). Other ways of tiling incur extra communication cost for fetching X and updating Z.

2) The shape and size of an array. The access pattern of an array often

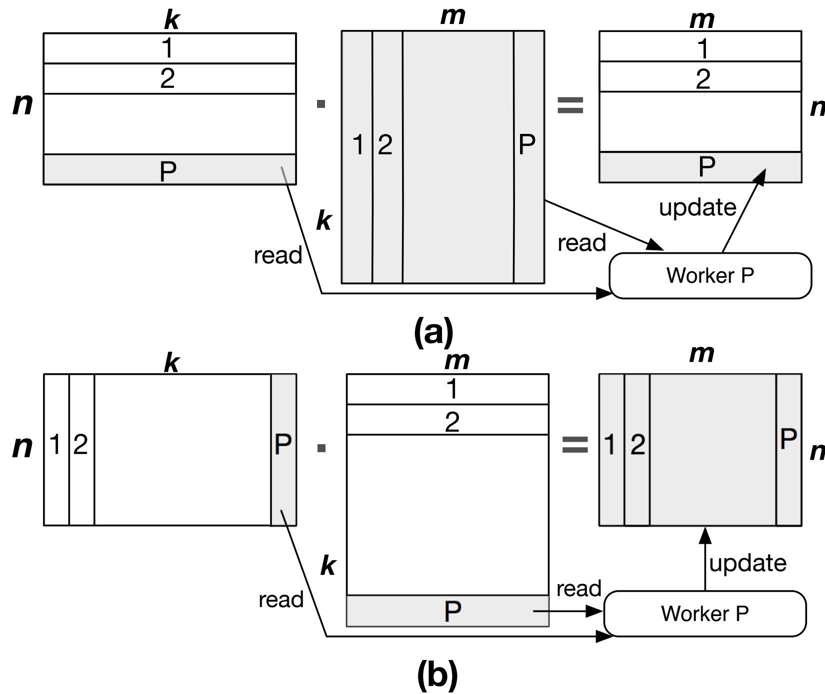


Figure 2.2: Two ways to implement matrix multiplication X·Y=Z, aka dot operation. Gray areas denote data read or updated by a single worker. In (a), each worker reads the entirety of Y across the network and performs local writes. Its per-worker communication cost is $k*m$. In (b), each worker performs local fetches and sends updates of size $n*m$ over the network. The per-worker communication cost is $n*m$.

depends on the array's shape and size. Therefore, such runtime information affects the array's tiling choice. In addition to Figure 2.2(a), there exists an alternative implementation of dot, shown as Figure 2.2(b). In this alternative implementation, each of the p parallel tasks reads X column-wise and Y row-wise to perform a local matrix multiplication and update the entirety of Z. The final Z is created by aggregating updates from all p tasks. Consequently, it is best to tile X column-wise and Y row-wise.

Whether to use Figure 2.2(a) or Figure 2.2(b) to compute $X \cdot Y = Z$ is a runtime choice that depends on the array shapes. Suppose X is an $n \times k$ matrix and Y is a $k \times m$ matrix. Figure 2.2(a) has a per task communication cost of $k * m$. This is because each task needs to fetch the entire Y across the network and can be scheduled to co-locate with the tile of X that it intends to read. By contrast, Figure 2.2(b) has a per task communication cost of $n * m$. This is because each task needs to send its update of Z over the network and can be scheduled to co-locate with the tiles of X and Y that it intends to read. Therefore, the best tiling choice depends on the shape of X. If $n > k$, the cost of Figure 2.2(a) is lower and the system computes dot using (a) whose preferred tiling for X is column-wise. If $n < k$, the cost of Figure 2.2(b) is lower and the system computes dot using (b) whose preferred tiling for X is row-wise.

```
1  func ALS(A):
2     '''
3     Alternating Least Squares
4     Input: A is a n*k user-movie rating matrix.
5     Output: U and M are factor matrices.
6     '''
7     for i from 1 to max_iter
8         U = CalculateUsersFactor(A, M)
9         M = CalculateMoviesFactor(A, U)
10    endfor
11    return U, M
```

Figure 2.3: Pseudocode of Alternating Least Squares.

3) How an array is used throughout the program. An array can be read by multiple expressions. If these expressions access the array differently, we can

reduce communication cost by creating multiple tilings for the array. In order to learn of an array's usage, the system cannot simply handle one expression at a time, but must "look ahead" in execution when determining an array's tiling. Consider the Alternating Least Squares (ALS) computation shown in Figure 2.3. ALS solves the collaborative filtering problem by decomposing the given user-item rating matrix. Consider a movie recommendation system under ALS that makes use of two parameters: users and movies. In each iteration, ALS calculates the factor for each user, based on the rating matrix, A, and a movie factor matrix (line 5 in Figure 2.3). Then, it calculates the factor for each movie based on the rating matrix, A, and users factor matrix (line 6 in Figure 2.3). Thus, ALS needs to access A along both row (users) and column (movies) in one single iteration. If the system decides on A's tiling by line 8 only, it would tile A row-wise. Later, at line 9, the system incurs communication cost when reading A column-wise. This is far from optimal. If we unroll the for loop and look at all the expressions together, we can see that A is accessed by two expressions several times (max_iterations). Thus, the best tiling is to duplicate A and tile one along row and another along column.

2.2.2 Our Approach and Spartan Overview

Like NumPy and other popular array languages, users write applications in Spartan using a large number of built-in functions and array primitives (e.g. +,*,dot, mean, etc.). Spartan implements its built-in functions using a small number of *high-level parallel operators.* The high-level operators encapsulate common

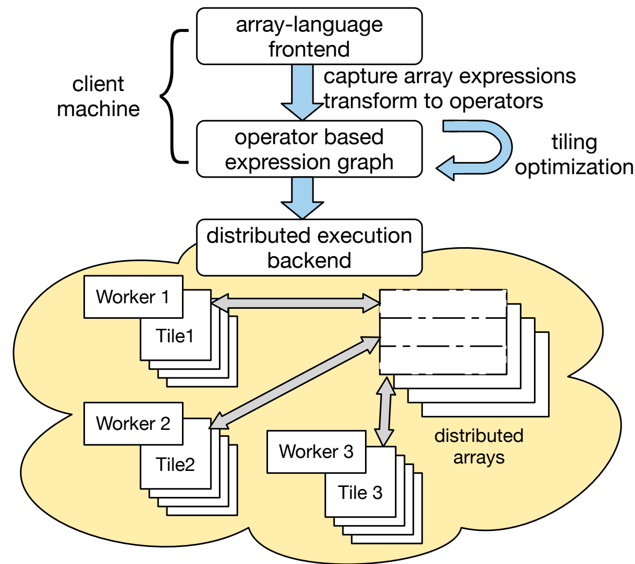


Figure 2.4: The layered design of Spartan. The frontend builds an expression graph and optimizes it. The backend executes the optimized graph on a cluster of machines. Each worker (3 workers in this figure) owns a portion of the global array.

parallel patterns and can efficiently express most types of computation. Users may also directly program using these high-level operators if their computation cannot be expressed by existing builtins.

Spartan uses a layered approach which splits the execution into frontend and backend steps, shown in Figure 2.4. The frontend, running on a client machine, captures user code and turns it into an expression graph whose nodes correspond to the high-level operators. Next, the frontend runs a tiling optimizer to determine good tiling for each node in the expression graph. Finally, the frontend sends the tiled expression graph to the backend. The backend provides high performance distributed implementations of high-level operators. For each operator, it schedules a collection of tasks running on many compute machines.

The tasks create, fetch and update distributed in-memory arrays based on the tiling hint determined by the optimizer.

Spartan's high-level operators and its layered design help collect the necessary information for automatic tiling. First, by expressing various types of computation in a small set of high-level operators, the data access pattern is made explicit for analysis (§2.2.1 (1)). Second, the frontend dynamically captures the expression graph with runtime information about the shape of input and intermediate arrays (§2.2.1 (2)). Third, the expression graph represents a large execution context, thereby allowing the frontend to understand how an array is used by multiple expressions. This is crucial for good tiling (§2.2.1 (3)).

2.3 Smart Tiling with High-level Operators

This section describes the design of Spartan, focusing on those parts crucial for automatic tiling. Specifically, we discuss high-level operators (Section 2.3.1), how Spartan's frontend turns an array program into a series of expression graphs (Section 2.3.2), the basic tiling algorithm (Section 2.3.3) and additional optimizations (Section 2.3.4).

2.3.1 High-level Operators

A high-level operator in Spartan is a parallel computation that can be parameterized by some user-defined function [1]. The operators are "functional" in nature:

1. The user-defined function must be free of side-effects and deterministic.

they take arrays or views of arrays as input and generate a new one without modifying existing arrays in place. Spartan supports views of arrays like NumPy. A view is an interface that allows users to manipulate arrays (e.g., swapping axes, slicing) without copying data. When reading a tile of a view, Spartan translates the shape and location from the view to those of the underlying array to fetch data.

High-level operators are crucial to Spartan's smart tiling, but what operators should we use? There are two considerations in choosing them. First, each operator should capture a general parallel pattern that can be used to implement many builtins. Second, each operator should have restricted semantics that correspond to a well-defined cost profile for different ways of tiling its input and output. This enables the captured expression graph to be analyzed to identify good tiling choices.

Spartan's current collection of five high-level operators is the result of many design iterations based on our experience of building various applications and builtins. Below, we describe each operator in turn and also discuss its (communication) cost w.r.t. different tiling choices.

- D=map($f_{map}, S_1, S_2, \ldots$) applies function f_{map} in parallel tile-wise over input arrays, $S_1, S_2, \ldots$, and generates output array D with the same shape. The total cost is zero if all inputs have the same tiling. Otherwise, the cost is the total size of all input arrays whose tiling differs from S_1.

  As an example usage of map, Figure 2.5(line 4–7) shows the implementation of Spartan's built-in array addition function which simply uses map with

f_{map} as Numpy's addition function.

- D=filter(f_{pred}, S) creates a view of S that excludes elements that do not satisfy the given predicate f_{pred}. Alternatively, filter can take a boolean array in place of f_{pred}. Since filter creates a view without copying actual data, the cost is zero.
- D=fold($f_{accum}, S, axis$) aggregates input array S using the commutative and associate function f_{accum} along the $axis$ dimension. For example, if S is a $m \times n$ matrix, then folding it along axis=0 creates a vector of n elements. Spartan performs the underlying folding in parallel using up to m tasks. The cost of fold is zero if S is tiled along the $axis$ dimension, otherwise, the cost is $S.size$.
- D=scan($f_{accum}, S, axis$) computes cumulative aggregates using f_{accum} over the $axis$ dimension of S. Unlike fold, its output D has the same shape as the input. The cost profile of scan is the same as fold.
- D=join_update($f_{join}, f_{accum}, S_1, S_2, \ldots, axis_1,\ axis_2, \ldots, output_shape$) is more complex than previous operators. This operator treats each input array S_i as a group of tiles along the $axis_i$, The shapes of the input arrays must satisfy the requirement that they have the same number of tiles along their respective $axis_i$. Spartan joins each tile among different groups and applies f_{join} in parallel. Function f_{join} generates some update to be written to output D at a specified location. Multiple workers running f_{join} may concurrently update to the same location of D; such conflicts are automatically resolved by applying f_{accum}.

As an example of join_update, consider the matrix multiplication implementation in Figure 2.2(b), where S_1 is a $n \times k$ matrix and S_2 is a $k \times m$ matrix. Figure 2.5 (lines 20–22) uses join_update which divides S_1 into k column vectors and S_2 into k row vectors. The f_{join} (aka dot_udf) is called in parallel for each column vector of S_1 joined with the corresponding row vector of S_2. It performs a local dot product of the joined column and row to generate an $n \times m$ output tile. All updates are aggregated together using the addition accumulator to create the final output.

A special case of join_update is when some input array S_i has $axis_i = -1$. In this case, the entire array S_i will be joined with each tile of other input arrays. Figure 2.5 (lines 23-25) uses this special case of join_update to realize the alternative matrix implementation of Figure 2.2(a).

The cost of join_update consists of two parts, 1) the cost to read the input arrays. 2) the cost of updating the output array. If an input array S_i is partitioned along $axis_i$, the input cost for S_i is zero, otherwise, the cost is $S_i.size$. Since the size and shape of output array created by f_{join} is unknown to Spartan, it assumes a default update cost, $D.size$.

In addition to the five high-level operators, Spartan also provides several primitives to create distributed arrays or views of arrays.

- D=newarray($shape, init_method$) creates a distributed array with a given shape. The array can be initialized in several ways, 1) by loading data from an external storage, 2) by some computation, e.g. random, zeros.
- D=slice($S, region$) creates a view over a specified region in array S. The

region descriptor specifies the start and end of the sliced region along each dimension.

- D=swapaxis($S, axis_1, axis_2$) creates a view of array S by swapping the axes $axis_1$ and $axis_2$. The commonly used built-in transpose function is implemented using this operator. The output view D has a different tiling from S. For example, if S is a column-tiled matrix, then $D = swapaxis(S, 0, 1)$ is effectively a row-tiled matrix.

There is no cost for newarray, newarray and swapaxis (the cost of newarray reading from an external storage is unrelated to tiling).

Based on the high-level operators, Spartan supports 70+ Numpy builtins. Figure 2.5 shows two implementations of Spartan's builtins, add and dot.

Although Spartan's map and fold resemble the "map" and "reduce" primitives in the MapReduce world [3, 10, 4, 13], they are more restrictive. Spartan only allows f_{map} to write a tile in the same location of the output array as its input tile location and not some arbitrary location. Similarly, fold can only reduce along some *axis* as opposed to over arbitrary keys in a key value collection. Such restriction is necessary for them to have a well-defined cost profile.

2.3.2 Expression Graph Capture

During a user program's execution, Spartan's frontend captures array expressions via lazy evaluation and turns them into a series of expression graphs [31, 32]. In an expression graph, each node corresponds to a high-level operator and an edge from one node to another shows the data dependency between them. Fig-

```
1  import numpy
2  import spartan
3
4  # Spartan's parallel implementation of
5  # element-wise array addition
6  def add(a, b):
7      return spartan.map(a, b, f_map=numpy.add)
8
9  # User-defined f_join function
10 def dot_udf(input_tiles):
11   output_loc = spartan.location(0,0)
12   output_data = numpy.dot(input_tiles[0], input_tiles[1])
13   return output_loc, output_data
14
15 # Spartan's parallel implementation of
16 # matrix multiplication
17 def dot(a, b):
18   if a.shape[0] <= a.shape[1]:
19     return spartan.join_update(S=(a, b), axes=(1, 0),
20                               shape=...
21                               f_join=dot_udf,
22                               f_accum=numpy.add)
23   else:
24     return spartan.join_update(S=(a, b), axes=(0, -1), ...)
```

Figure 2.5: Implementations of add and dot in Spartan.

ure 2.6(a) shows an example expression graph. Expression graphs are acyclic because Spartan's high-level operators create immutable arrays.

The frontend stops growing an expression graph only when forced: this occurs in a few situations: (1) when a variable is used to determine the control flow, (2) when a variable is used for program output, (3) when a user explicitly requests evaluation. The use of lazy evaluation leads to an implicit form of loop unrolling: as long as there is no data dependent control flow, expression graph will continue

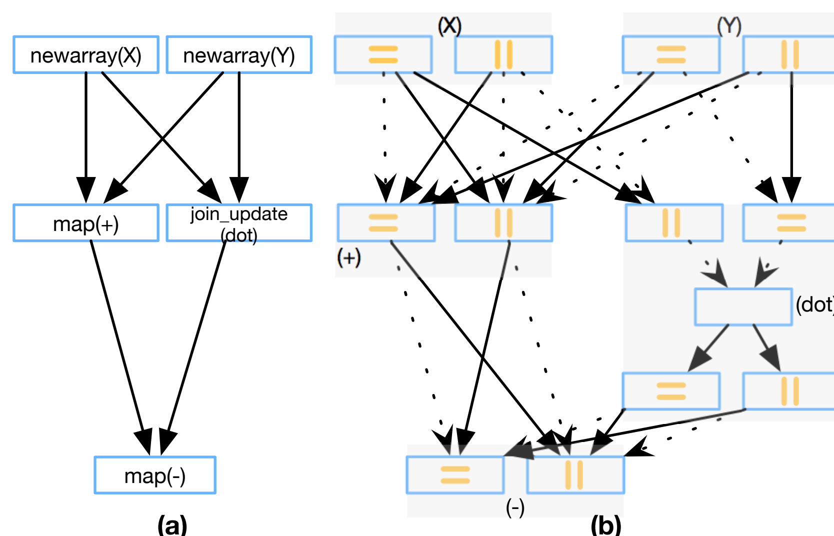


Figure 2.6: The expression graph and its corresponding tiling graph for $Z = X + Y - X \cdot Y$.

growing until pre-configured limits.

2.3.3 Graph-based Tiling Optimizer

Spartan supports "rectangular" tiles: an n-dimensional array can be partitioned along any one dimension (e.g. row-wise, column-wise), or partitioned along two or more dimensions (e.g. block-wise tiling). Some existing work [33] explored other possible shapes that are more efficient for its applications.

Given an expression graph of high-level operators, the goal of the tiling optimizer is to choose a tiling for each operator node to minimize the overall cost. This optimization problem is NP-Complete(we show the proof in Appendix A). It is also not practical to find the best tiling via brute force since the expression

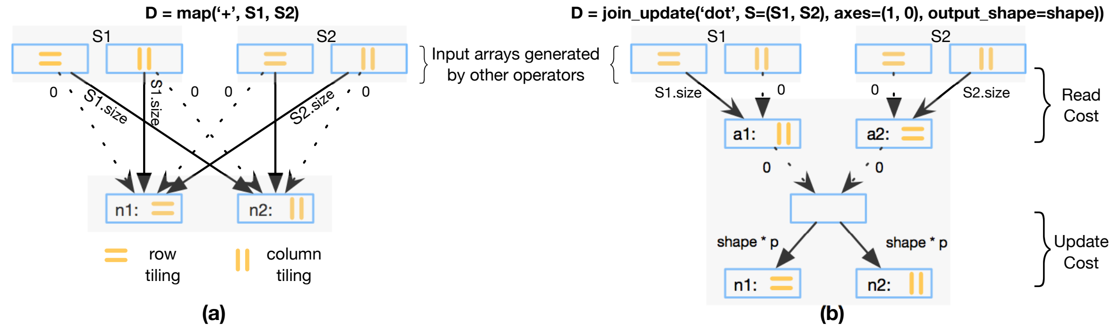


Figure 2.7: Two examples of building the tiling graph. (a) A plus expression, (S1 + S2), implemented by map operator (b) A dot expression, dot(S1, S2), implemented by join_update operator.

graph can be very large. Therefore, we propose a graph-based approximation algorithm to identify a good tiling quickly.

The algorithm works in two stages. First, it constructs a tiling graph based on the expression graph and the cost profile of each operator. Next, it uses a greedy strategy to search for a low cost tiling combination.

1) Constructing the tiling graph. The goal of the tiling graph is to *expose the tiling choices and cost in the expression graph.* For each operator in the expression graph, the optimizer transforms it into a *node group*, i.e. a cluster of several tiling nodes, each representing a specific choice to tile the operator's output or intermediate steps. The weight of each edge that connects two tiling nodes represents the underlying cost if the two operators are tiled according to the tiling nodes.

Figure 2.7 shows how a map operator, corresponding to $D = S_1 + S_2$, is transformed. To keep the figure simple, we assume that all arrays are two dimensional

with two tiling choices: row-based or column-based. And all dotted lines represent zero edge weights. As Figure 2.7 shows, the map operator becomes two nodes in the tiling graph, each representing a different way to tile its output D. Similarly, each of the map operator's input arrays S_1 and S_2 (which are likely outputs from the previous operators) also correspond to two nodes. For map, there is a well-defined way to label the weights among nodes, as illustrated in Figure 2.7. For example, if S_2 is tiled column-wise and D is tiled row-wise, the weight between the corresponding two nodes is S_2.size because workers have to read S_2 across the network to perform the map. fold and scan are treated similarly as map, but with edge weights labeled according to their own tiling cost profiles.

Next, we discuss the transformation of join_update. For this operator, we use some intermediate tiling nodes ($a_1, a_2 \ldots$ in Figure 2.7(b)) to represent the reading cost during the join. A placeholder node is used to represent the join stage. We use another set of tiling nodes ($n1, n2$ in Figure 2.7(b)) to capture the update cost to the output array. Unfortunately, Spartan can not know the precise update cost of join_update without executing the user-defined f_{join} function. Thus, we provide a default update cost according to the common update cost pattern observed in the applications implemented by join_update. If join_update is performed within a loop, the optimizer can adjust the edge cost of the tiling graph according to the actual cost observed during the previous execution of the join_update.

Figure 2.7(b) shows the tiling graph used for the matrix multiplication func-

tion implemented in join_update. This implementation corresponds to the data access pattern shown in Figure 2.2(b). As shown in Figure 2.5, the join axes for the first and second arrays are column and row respectively. The edge weight for S_i is 0 if it matches the join axis and is $S_i.size$ otherwise. The cost is $S_i.size$ is because each worker needs to update the entirety of the result matrix. The edge weights for n_1 and n_2 are both $p * output_shape$.

Figure 2.6 gives an example showing a specific array execution ($Z = X + Y - X \cdot Y$)) and its corresponding expression graph and tiling graph. We omitted the details of other edge weights to keep the graph readable.

2) Searching for a good tiling. Deciding a tiling choice for an operator corresponds to picking one node among the corresponding node group in the underlying tiling graph and different combinations of tiling nodes pose different costs. As a result, the next step for the tiling optimizer is to analyze the tiling graph and find a combination of tiling choices that minimizes the overall cost. The tiling optimizer adopts a greedy search algorithm. The heuristic is to decide the tiling for the node group with the maximum connectivity first. Here, connectivity of a node group is the number of its adjacent node groups. When deciding a tiling for a node group X, the algorithm chooses the one resulting in the minimum cost for X. Why does this heuristic work? The cost of a tiling for an operator depends on the tiling choices of its adjacent operators. Thus, an operator with more adjacent operators has a higher impact on overall cost. Consequently, the algorithm should first minimize the cost of node groups with higher connectivity[2].

2. Another natural heuristic is to search the node group with largest array size first. Unfortunately, this algorithm does not perform well according to our experiments.

```
1  func FindCost(NodeGroup G, TileNode T)
2      # Find the cost for tiling node T of G
3      cost = 0
4      foreach NodeGroup g in G.connectedGroups():
5          if IsView(g, G):
6              cost += FindCost(g, g.viewTileNode(T))
7          else:
8              edgeCost = INFINITY
9              foreach Edge e in g <-> T
10                 edgeCost = min(edgeCost, e.cost)
11             endfor
12             cost += edgeCost
13         endif
14      endfor
15      return cost
16
17 func FindTiling(TilingGraph G)
18     # Find good tiling for every operator in G.
19     GroupList = SortGroupByConnectivity(G)
20     foreach NodeGroup x in GroupList
21         minCost = INFINITY
22         goodTiling = NONE
23         foreach TileNode y in x
24             cost = FindCost(x, y)
25             if cost < minCost:
26                 minCost = cost
27                 goodTiling = y
28             endif
29         endfor
30         x.chosenTiling = goodTiling
31         # Other Group can only connect to goodTiling.
32         x.removeAllConnectedEdgesExcept(goodTiling)
33     endfor
34     return G
```

Figure 2.8: The maximum connectivity group first algorithm to find good tiling based on the tiling graph.

Figure 2.8 shows the pseudo code for the tiling algorithm. Given a tiling graph G, the algorithm processes node groups in the order of edge connectivity (Line 19–20). For each node group (x in Line 20), the algorithm calculates the cost of each tiling node and chooses the tiling node with the minimum cost (Line 23–29). After deciding the good tiling ($x.chosenTiling$ in Line 30) for node group x, the algorithm removes all edges connected to all other tiling nodes (Line 32). This implies that the algorithm can't freely choose tiling for adjacent node groups of x any more – it must consider the chosen tiling of x.

$FindCost$ obtains the cost of a tiling node (T in Line 1) by calculating the sum of the minimum edge weight between each adjacent node group and T (Line 4–14). If the adjacent node group is a view operator such as swapaxis, its tiling node will be decided by T. To get accurate cost affected by T, the algorithm should also consider the adjacent node groups for its view operators. As a result, $FindCost$ recursively finds the cost of the view node group (Line 5–6). The result corresponds to the best possible cost for tiling node T.

The complexity of the tiling algorithm is $O(E * N)$ where E is the number of edges in the tiling graph and N is the number of node groups. It is not guaranteed to find the optimal tiling. However, we find that the greedy strategy works well in practice (Section 3.2).

2.3.4 Additional Tiling Optimizations

Duplication of arrays. As the ALS example in Fig 2.3 shows, some arrays may be accessed along different axes several times. To reduce communication, Spartan

supports duplication of arrays and tiles each replica along different dimensions. To support duplication in the tiling optimizer, we add a “duplication tile“ node to each node group in the underlying tiling graph. As duplication of arrays increases memory consumption. Spartan allows users to specify the memory budget for duplicating arrays to limit memory usage. Whenever the optimizer chooses to “duplicate tile“ which causes an operator’s output to be duplicated, it deducts from the memory budget. The optimizer will not choose duplication tiling without enough memory budget.

2.4 Implementation

Since NumPy is wildly popular in machine learning and scientific computing, our implementation goal is to replicate the “feel” of NumPy as much as possible. Our prototype currently supports 70+ most commonly used Numpy builtins.

The Spartan frontend, written in Python, captures expression graph and performs tiling optimization (Section 2.3). The Spartan backend, consists of one designated master and many worker processes on a cluster of machines. Below, we provide more details on the major backend components:

Execution engine. The backend provides efficient implementations of all high-level operators. Given an expression graph, the master is responsible for coordinating the execution of one node (a high-level operator) at a time. To execute a node, the master first creates an output array with the given tiling hint and then schedules a set of tasks to run user-defined parameter functions in parallel according to the data locality. Locality here means the task is executed

on the worker that stores its input source tile. If the node corresponds to a join_update, scan or fold, the backend also associates a user-defined accumulator function with the output array to aggregate updates from multiple workers.

User-defined parameter functions are written in Python NumPy and process one tile instead of one element at a time. Like MatLab, NumPy relies on high performance C-based linear algebra libraries like BLAS [34] or LAPACK [35]. As a result, the local execution of parameter functions in each worker is efficient.

Distributed, tiled arrays. Each distributed array is partitioned into a set of tiles according to its tiling hint and stored in workers' memory. To create an array, the master assigns each of its tile to a worker (e.g. in a round-robin fashion) and distributes the tile-to-worker mapping to all workers so everybody can access remote tiles without consulting the master. If two arrays of the same shape have identical hints, the master ensures that tiles corresponding to the same region in both arrays are co-located in the memory of the same worker.

Fault tolerance. To recover from worker failure in the middle of a long computation, the backend checkpoints in-memory arrays to durable storage. Our implementation currently adopts the simplest design: after finishing an entire operator, the master periodically instructs all workers to save their tiles and also saves its own state.

Chapter 3

Spartan Evaluation

In this chapter, we measured the performance of our smart tiling algorithm. We also evaluated the scalability of applications and compared against other open-source distributed array frameworks.

3.1 Experimental Setup

We evaluated the performance of Spartan on both our local cluster as well as Amazon EC2. The local cluster is a heterogeneous setup consisting of eleven machines: 6 machines have 8-core AMD Opterons with 16GB of RAM, and 5 machines have 4-core Intel Xeons with 8GB of RAM. The machines are connected by gigabit Ethernet. For the EC2 experiments, we use 128 spot instances of the older generation m2.xlarge. Each of these instances has 17.1GB memory and 2 virtual CPUs. The network performance is rated as “moderate”, which is approximately 300Mbps according to our measurements.

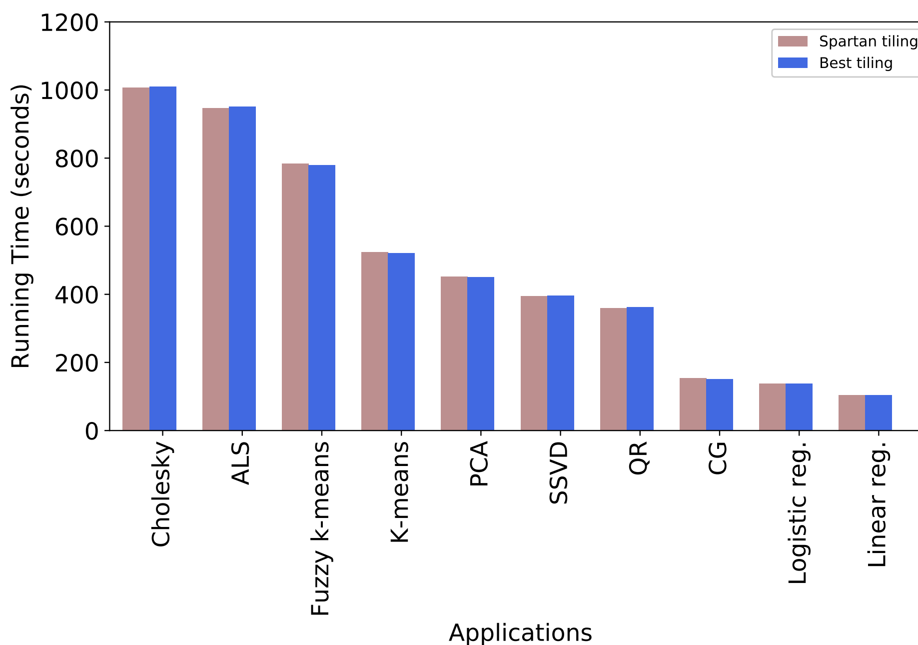


Figure 3.1: Running time comparison between the smart tiling and the best tiling for 10 applications.

Unless otherwise mentioned, we ran multiple worker processes on each machine, one associated with each CPU core. We use 12 applications as our benchmarks. They include algorithms from machine learning, data mining and computational finance.

3.2 Tiling

Smart Tiling Evaluation for Applications: We compared the running time of applications with the tiling generated by smart tiling against the best tiling – the tiling that incurs the minimum communication cost. The best tiling can be pre-calculated by using a brute-force algorithm to traverse the expression graph

and search the minimum communication cost among all possible tiling choices. The experiment runs on 128 EC2 instances. Figure 3.1 only shows 10 applications because the computational finance ones operate on one-dimensional arrays which can only be tiled along one axis. For applications which are not perfectly scalable such as ALS and Cholesky, we set the sample sizes up to 10 million. For others, the sample sizes are up to 1 billion due to the memory limitation.

These applications show various kinds of tiling patterns. First, many applications contain expressions or operators that require runtime shape and axis information to best tile matrices, e.g. dot and join_update. Smart tiling analyzes the runtime information and gives the best tiling for the applications such as row-wise tiling for Regression and block tiling for Cholesky decomposition. Second, some program flows pass the intermediate matrices to expressions that change the view of tiling, e.g. swapaxis. Smart tiling identifies the best tiling through the global view of computation. Example applications include SSVD and PCA. Finally, some applications, like ALS, access matrices along different axes several times. As described in Section 2.2.1, the best tiling for these applications is duplication tiling.

Figure 3.1 shows that Spartan's smart tiling is able to give the best tiling and improve the performance for all applications. Note that the application running time of the best tiling and Spartan's smart tiling are not the same; sometimes Spartan's smart tiling even outperforms the best tiling. The difference is caused by the instability of Amazon EC2. Spartan's optimizer makes the same choices as the best tiling for all applications.

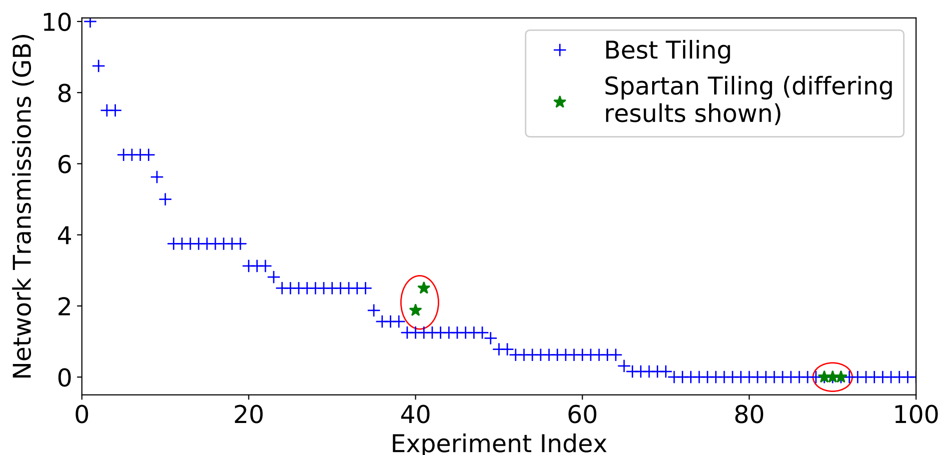


Figure 3.2: Network transmission cost comparison between smart tiling and the best tiling for 100 randomly generated programs. Sorted by network transmission for readability only (array sizes are randomly chosen from a set and there is no relation between experiment index and network transmission).

A bad tiling can result in huge network transmission. For instance, if the tiling of the input arrays for logistic regression is partitioning along the smaller dimension, workers need to remotely fetch the matrix which is more than 512GB in the evaluation (4GB network transmission per instance in one iteration which result in approximately an extra 110 seconds in our environment). Another interesting example is ALS. Simply row-wise or column-wise tiling can result in 40% performance degradation compared to duplication tiling. Moreover, the running speed of smart tiling is fast. For example, the brute-force algorithm needs more than 500 seconds to analyze a 14-operators ALS while Spartan's smart tiling derives the same result in 0.06 seconds.

Smart Tiling Evaluation for Randomly Generated Programs: Although smart tiling gives the best tiling for applications we implemented, there is no guar-

antee that smart tiling performs well for various kinds of applications. Therefore, we examined the performance of smart tiling for randomly generated programs. Each array dimension is randomly chosen from 128K to 512K. These programs contain various numbers and types of operators Spartan has supported. The number of operators per program ranges from 2 to 15.

Figure 3.2 shows the network transmission cost of 100 randomly generated programs with the tiling given by smart tiling and the best tiling. The result shows that Spartan's smart tiling can give the best tiling for most programs. It is also fast compared to the brute-force algorithm. For all programs, smart tiling needs less than 0.1 seconds while the brute-force algorithm spends 1900 seconds when the program contains 15 operators.

```
1 def sub_optimal_case_pattern(SIZE):
2       A = expr.rand((SIZE, SIZE))
3       B = expr.rand((SIZE, SIZE))
4       C = A + B
5       D = expr.transpose(A) + expr.transpose(B)
6       E = C + D
```

Figure 3.3: An example that Spartan's smart tiling gives sub-optimal tiling.

Figure 3.3 shows the pattern residing in those programs that smart tiling gives sub-optimal tiling. The best tiling for Figure 3.3 is to tile D column-wise and other operators row-wise. However, smart tiling inspects the tiling cost for C first and then for D because of the maximum connectivity. It finds that row-wise tiling costs zero for both operators. Therefore, smart tiling partitions both C and D row-wise and thus gives sub-optimal tiling due to the conflict views (caused

by transpose) of C and D.

Although smart tiling cannot give the best tiling for these programs, this sub-optimal case rarely happens. Smart tiling produces a conflict view only when a program exhibits two patterns simultaneously: 1) Two operators have different views of tiling from the same input arrays. 2) Both operators have more connectivity than their input arrays. As Figure 3.2 shows, only 5 out of 100 random generated programs satisfy both requirements. For three of them, the best tiling needs zero network transmission while the smart tiling needs around 0.01 GB network transmission. The number is not large because these expressions include fold which reduces the size of matrices. For the other two instances, the best tiling requires 1.3 GB but the smart tiling consumes 1.9GB and 2.6GB respectively.

3.3 Scaling

We evaluated the scalability of all applications in two ways. First, the applications use fixed-size inputs and run across a varying number of workers. Second, the applications use inputs whose sizes are scaled linearly with the number of workers. All results are normalized by the 8 workers baseline cluster size to show the relative savings (comparing with 1 worker is not fair because there is no communication for only 1 worker). All inputs are synthetic data.

Fixed input size. Figure 3.4 shows the running time of 12 applications on the local cluster. The number of workers used in the experiments increases from 8 to 64. The dotted lines corresponding to $\frac{1}{2}$, $\frac{1}{4}$ or $\frac{1}{8}$ ratio represent the ideal

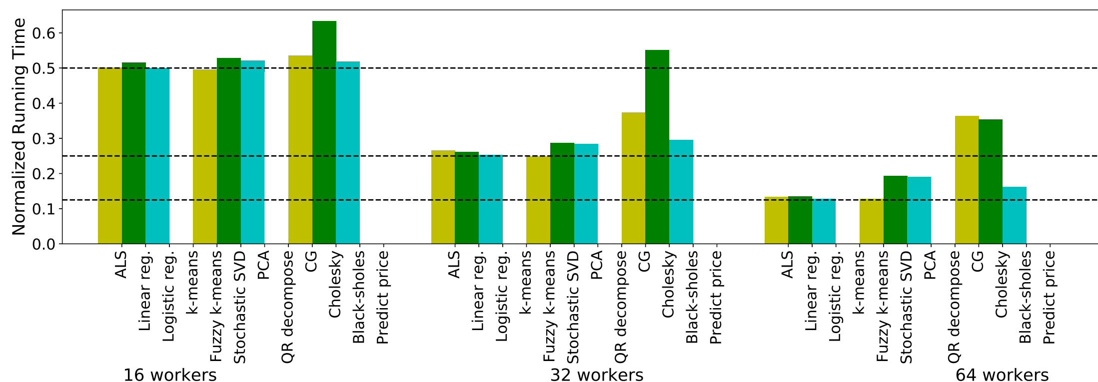


Figure 3.4: Fixed input size, varying number of workers. Normalized running time is calculated by dividing 8 worker running time on local cluster.

scaling for 16, 32, and 64 workers.

The evaluation shows that the running time of many applications achieves perfect scaling. Some of them do not scale well due to the inefficiencies of the underlying algorithms. CG has many dependent folds that reduce to one value on one worker. Cholesky also has many dependent steps: the parallelism available in each step grows and shrinks, thus Cholesky cannot always utilize all workers.

Scaling input size. Figure 3.5 shows the performance for 16 and 64 workers. Ideal scaling corresponds to a flat line of 1.0.To examine the scalability on a larger-scale system, we ran the experiment on EC2. Figure 3.6 illustrates the experiment running up to 256 workers. The result is similar to that of Figure 3.5 except for ALS. There are three matrices in ALS, rating matrix, sample matrix and item matrix. While Spartan's smart tiling can reduce the reading cost of rating matrix by duplication, ALS still needs to randomly fetch sample matrix and item matrix in each iteration and results in large communication. Thus, ALS is not scalable for large-scale datasets.

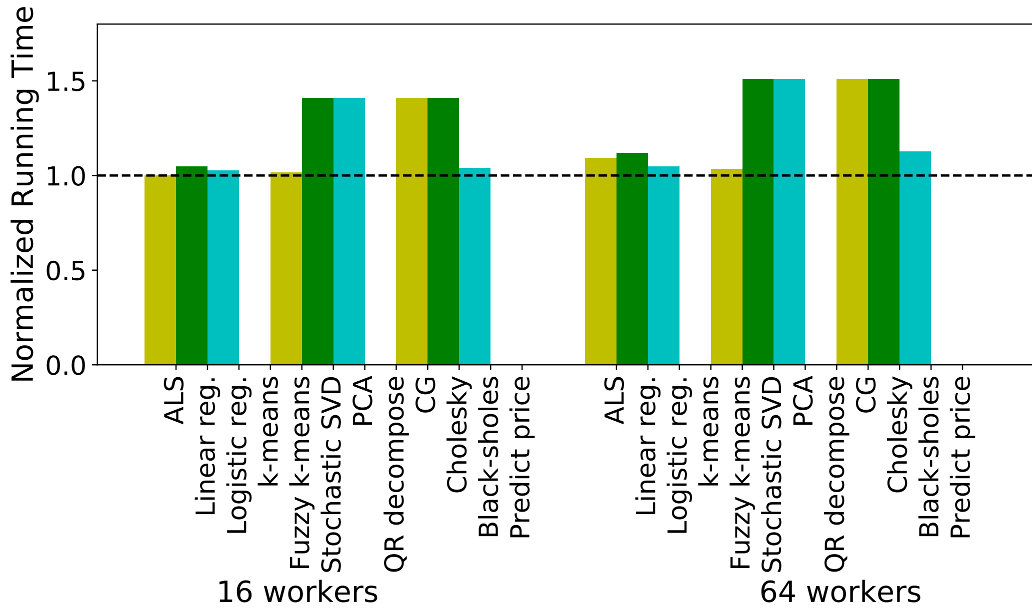


Figure 3.5: Scaling input size on local cluster.

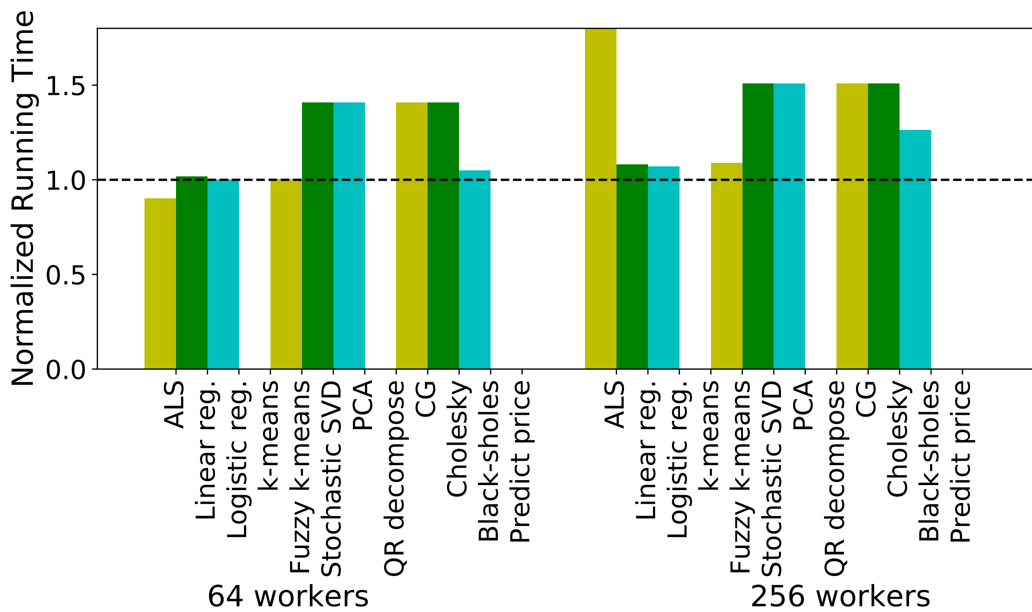


Figure 3.6: Scaling input size on 128 EC2 instances.

	Running Time (seconds)	Sample Size
Spartan	523.95s	1 billion
Presto	882.47s	1 billion
SciDB	2573.83s	10 million

Table 3.1: K-Means performance comparison with Presto and SciDB on 128 instances EC2. The dataset for Spartan and Presto contains 1 billion points, 50 dimensions and 128 centers. The dataset for SciDB contains 10 million points.

3.4 Comparison with Other Systems

We compared the performance of Spartan's k-means with the implementation of Presto (also called Distributed R) and SciDB. The synthetic dataset contains 1 billion samples with 50 dimensions and 128 centers for Presto and Spartan while only 10 million samples for SciDB.

Table 3.1 shows that the performance of Spartan is 1.7x faster than Presto. Though both Spartan and Presto partition the arrays row-wise which is the best tiling, Presto requires users to explicitly assign the tiling while Spartan needs no user hints. Thus, the performance difference of Spartan and Presto comes from the backend library and implementation. We have verified this by running k-means only on a single worker.

Unlike Spartan and Presto, SciDB is not an in-memory distributed system and thus has much slower performance. The basic partition unit in SciDB is a chunk. It is important for SciDB to select the correct chunk size to reduce disk I/O. However, in Spartan, we focus on how to reduce the network communication.

Chapter 4

SwapAdvisor Design

In this chapter, we first present the necessary backgroud of swapping tensors for DNN models (Section 4.1). Section 4.2 justifies the design choices for SwapAdvisor. We detail the system design and search algorithm of SwapAdvisor in Section 4.3 and 4.4. Section 4.5 discusses the implementation of SwapAdvisor.

4.1 Background

DNN training and inference are usually done on a GPU, which is attached to a host CPU via a high-performance bus, e.g., PCIe and NVLink. GPU uses different memory technology with higher bandwidth but limited capacity, e.g. 16GB on the NVIDIA V100. By contrast, it is common for CPUs to be equipped with hundreds of gigabytes of memory. Therefore, it is attractive to swap data between GPU and CPU memory, in order to support training and inference that otherwise would have been impossible given the GPU memory constraint.

Modern DNNs have evolved to consist of up to hundreds of layers, which are usually composed together in a sophisticated non-linear topology. Programming frameworks such as TensorFlow/MXNet express DNN computation as a dataflow graph of tensor operators. DNN's memory consumption falls into 3 categories:

1. Model parameters. In DNN training, parameters are updated at the end of an iteration and used by the next iteration. Parameter tensors are proportional to a DNN model's "depth" (the number of layers) and "width" (the size of a layer). For large models, these dominate the memory use.
2. Intermediate results. These include activation, gradient and error tensors, of which the latter two are only present in training but not in inference.
3. Scratch space. Certain operator's implementation (e.g. convolution) requires scratch space, up to one gigabyte. Scratch space is a small fraction of total memory use.

Existing work use manual heuristics based on the memory usage patterns of different categories. For example, prior work do not swap parameters[1], but only swap activation to the CPU [26, 24]. Without parameter swapping, prior work cannot support DNNs whose parameters do not fit in the GPU memory. Furthermore, designs based on manual heuristics miss opportunities for performance improvements as modern DNN dataflow graphs are too complex for analysis by humans.

SwapAdvisor is a general swapping mechanism in which any tensor can be swapped in/out under memory pressure. More importantly, we aim to move

1. The only exception being SuperNeuron [25] which swaps convolution but not other types of parameters.

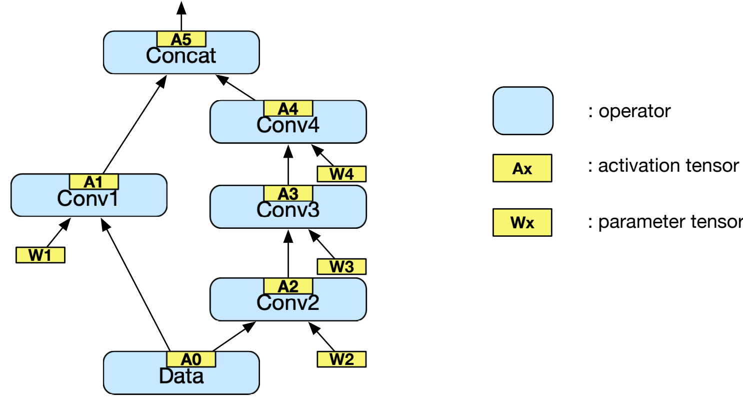
away from manual heuristics and to automatically optimize for the best swapping plan given an arbitrarily complex dataflow graph. We focus the discussion on swapping for a single GPU, but our design can be used in a multi-GPU training setup which replicates the model on different GPUs using data parallelism.

4.2 Challenges and Our Approach

A good swapping plan should overlap communication and computation as much as possible. The opportunities for overlapping come from swapping out a (temporarily) unused tensor to make room for swapping in an out-of-memory tensor before the latter is required for operator execution. We aim to maximize such overlapping by carefully planning for what and when to swap with the help of the dataflow graph.

Prior work attempt to find a good swapping plan heuristically based on the dataflow graph structure alone [24, 26, 25]. However, this is not enough. In particular, we argue that there are two critical factors affecting swap planning:

- *Memory allocation.* DNN computation uses a wide range of tensor sizes, from a few KB to hundreds of MB. To improve speed and reduce internal fragmentation, frameworks such as MXNet use a memory pool which pre-allocates a number of fixed-size tensor objects in various size classes. As a result, swapping happens not just when the GPU memory is full, but when there is no free object in a particular size class. Therefore, how to configure the memory pool for allocation can critically affect swapping performance.



(a) The partial dataflow graph of a model. All tensors are 1MB, except for A2 and A5 which are 2MB. The GPU memory is 10MB.

Time	t1	t2	t3	t4	t5	t6	t7
Computation	Data	Conv1	Conv2	Conv3	Conv4	Concat	
Swapin	W_1	W_2	W_3	W_4	A_1		
Swapout			A_1	A_0	A_3	A_2	
Memory Usage	A_0W_1	$A_0A_1W_1W_2$	$A_0A_1A_2W_2W_3$	$A_0A_2A_3W_3W_4$	$A_1A_2A_3A_4W_4$	$A_1A_2A_4A_5$	

(b) Schedule the left branch first. Allocate five memory objects, each is 2MB.

Time	t1	t2	t3	t4	t5	t6	t7
Computation	Data	Conv2	Conv3	Conv4		Conv1	Concat
Swapin	W_2	W_3	W_4		W_1		
Swapout				A_2	A_2	A_3	A_0
Memory Usage	A_0W_2	$A_0A_2W_2W_3$	$A_0A_2A_3W_3W_4$	$A_0A_2A_3A_4W_4$	$A_0A_2A_3A_4W_1$	$A_0A_1A_3A_4W_1$	$A_0A_1A_4A_5$

(c) Schedule the right branch first. Allocate five memory objects, each is 2MB.

Time	t1	t2	t3	t4	t5	t6	t7
Computation	Data	Conv2	Conv3	Conv4	Conv1	Concat	
Swapin	W_2	W_3	W_4	W_1			
Swapout				A_2	A_2	A_3	
Memory Usage	A_0W_2	$A_0A_2W_2W_3$	$A_0A_2A_3W_3W_4$	$A_0A_2A_3A_4W_1W_4$	$A_0A_1A_2A_3A_4W_1$	$A_0A_1A_3A_4$	

(d) Schedule the right branch first. Allocate nine memoryobjects, one is 2MB and others are 1MB each.

Time	t1	t2	t3	t4	t5	t6	t7
Computation	Data	Conv1	Conv2	Conv3	Conv4		Concat
Swapin	W_1	W_2	W_3	W_4			
Swapout				A_0	A_2	A_2	A_3
Memory Usage	A_0W_1	$A_0A_1W_1W_2$	$A_0A_1A_2W_2W_3$	$A_0A_2A_3W_3W_4$	$A_1A_2A_3A_4W_4$	$A_1A_2A_3A_4$	$A_1A_3A_4A_5$

(e) Schedule the left branch first. Allocate nine memory objects, one is 2MB and others are 1MB each.

Figure 4.1: Different schedules and memory allocation can affect swapping.

- *Operator scheduling.* Modern DNNs have complex dataflow graphs as the layers no longer form a chain, but contain branches, joins and unrolled loops. As a result, there are many different potential schedules for executing operators. The order of execution can profoundly affects the memory usage and thus the performance of swapping.

Example. We use an example to show how memory allocation and scheduling affect swapping. The example is based on a portion of the dataflow graph of a toy neural network, as illustrated in Figure 4.1(a). For simplicity, the dataflow graph only shows the forward propagation and omits the backward part. This branching structure is common in modern CNNs [36, 37].

In Figure 4.1(a), blue rounded rectangles represent operators and small yellow rectangles represent tensors. A tensor is labelled as A_x (activation tensor) or W_x (parameter tensor). Suppose all tensors are 1MB, except for A_2 and A_5, which are 2MB (because A_2, A_5 are used to join two paths). Thus, the memory consumption is 12MB[2]. Suppose the GPU's memory capacity is 10MB, and it takes one unit of time to execute an operator or to transmit 1MB data between GPU and CPU.

A parameter tensor is initially in the CPU memory and must be swapped into GPU memory before being used. We can swap out a parameter tensor without copying it to CPU memory as it is not changed during the forward pass. By contrast, there is no need to swap in an activation tensor (because it's created by operators) but it must be copied to CPU memory upon swap-out because it

2. We do not consider memory reuse in the example as the partial dataflow graph does not include the backward pass which forbids many reuse cases

is needed in the backward pass.

There are many ways to allocate memory and schedule execution for Figure 4.1(a). We show 2 example schedules: *left-first* executes operators on the left branch first, and *right-first* executes the right branch first. We show 2 example memory allocations: *coarse-grained* allocates 5 memory objects of 2MB each, and *fine-grained* allocates 8 memory objects of 1MB each and 1 object of 2MB. Together, there are 4 combinations of schedule/allocation and we show the best swapping plan under each combination, in Figures 4.1(b) to (e). As GPU-CPU communication is duplex and concurrent with GPU execution, each table's top 3 rows give the timeline of actions for GPU computation, swap-in (from CPU to GPU), and swap-out (from GPU to CPU) respectively. The last table row shows the tensor objects that are currently resident in the GPU memory.

Let's contrast Figure 4.1(c) and (d) to see why memory allocation affects swap planning. Both (c) and (d) have the same right-first scheduling. However the total execution time of (c) is one unit time longer than that of (d). Specifically, in Figure 4.1(c), GPU sits idle in time slot t_5 while operator $Conv_1$ waits for its parameter W_1 to be swapped in. It's not possible to swap in W_1 earlier because the coarse-grained memory pool of five 2MB objects is full at time t_4. One cannot swap out any of the 5 GPU-resident objects earlier: A_3, W_4 and A_4 are input/output tensors needed by the currently running operator $Conv_4$, while A_0 is needed as input for the next operator $Conv_1$. A_2 is being swapped out but the communication takes two units of time due to its larger size. Figure 4.1 uses a fine-grained memory pool with eight 1MB objects and one 2MB object. As a

result, it can swap in W_1 needed by operator $Conv_1$ one unit time earlier, at t_4, because there is still space in the memory pool.

We contrast Figure 4.1(d) and (e) to see why scheduling affects swap planning. Both (d) and (e) use the same fine-grained memory pool. However, Figure 4.1(e) takes one unit of time longer than (d) because of its left-first schedule. In Figure 4.1(e), GPU is idle for the time slot t_6 as operator $Concat$ waits for 2MB tensor A_2 to complete swapping in order to make room for its 2MB output A_5. It is not possible to swap out A_2 any time earlier as it is the input of operator $Conv_3$ which executes at time t_4. By contrast, Figure 4.1(d)'s right-first schedule is able to execute $Conv_3$ earlier at t_3, thereby allowing A_2 to be swapped out earlier.

Our approach. Since memory allocation and operator scheduling critically affect swapping performance, we derive a swapping plan (aka which tensors to swap in/out and when) assuming a given dataflow graph as well as a corresponding memory allocation scheme and an operator schedule (Section 2.3). Specifically, the swap plan optimizes computation and communication overlapping by swapping out tensors not needed for the longest time in the future and prefetching previously-swapped out tensor as early as possible.

We search the space of possible memory allocations and operator schedules to find a combination with the best swapping performance. Instead of using manual heuristics to constraint and guide the search, we adopt genetic algorithm [38, 39, 40] to search for a good combination of memory allocation and operator scheduling. Genetic algorithms have been used for many NP-hard combinatorial

problems [41, 42] and have been applied for scheduling in parallel systems [43]. To enable effective exploration of a vast search space, we must be able to quickly evaluate the overall performance (i.e. end-to-end execution time) of a swap plan under any combination of memory allocation/scheduling. We found it too slow to perform the actual execution on real frameworks. Therefore, we estimate the performance by running the swap plan under a dataflow engine simulator. The simulator uses measured computation time for each operator as well as GPU-CPU communication bandwidth so that it can estimate the execution time of a dataflow graph under a given scheduling, memory allocation, and swap plan. The running time of our simulator on a CPU core is orders of magnitude faster than that of actual execution, reducing the search time for a model to less an hour. The simulator enables SwapAdvisor's genetic algorithm to directly optimize the end-to-end execution time.

4.3 SwapAdvisor Design

Overview. Figure 4.2 gives the architecture of SwapAdvisor, which is integrated with an existing DNN framework (MXNet in our implementation). Given a dataflow graph, SwapAdvisor picks any legitimate schedule and memory allocation based on the graph as initial values, and passes them to the swap planner to determine what tensors to swap in/out and when. The result of the swap planner is an augmented dataflow graph which includes extra swap-in and swap-out operators and additional control flow edges. The additional edges are there to ensure the final execution order adheres to the given schedule and the planner's

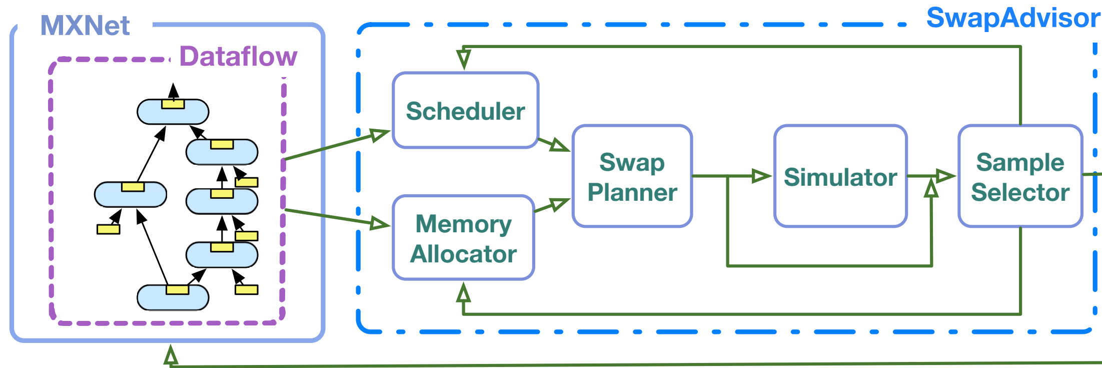


Figure 4.2: System overview of SwapAdvisor

timing of swaps.

For optimization, the augmented graph is passed to SwapAdvisor's dataflow simulator to estimate the overall execution time. The genetic algorithm-based search measures the performance of many memory allocation/schedule combinations, and proposes new allocation/schedule candidates for the swap planner. Once a swap plan has been sufficiently optimized, the final augmented dataflow graph is given to the framework for actual execution.

This section describes swap planner's inputs (Section 4.3.1) and explain how the planner maximizes performance given a specific schedule and memory allocation (Section 4.3.2). A later section (Section 4.4) discusses genetic algorithm-based optimization.

4.3.1 Operator Schedule and Memory Allocation

In addition to the dataflow graph, the swap planner takes as input an operator schedule and memory allocation.

Operator schedule. Given an acyclic dataflow graph G, an operator schedule is any topological sort ordering of nodes in G. When using a single GPU, the framework can issue operators to the GPU according to the schedule to keep the GPU busy. Indeed, frameworks such as MXNet commonly perform topological sort to schedule operators.

NVIDIA's recent GPUs support multiple "streams". SwapAdvisor uses 3 streams: one for performing GPU execution, one for swapping out tensors to the CPU, and one for swapping in tensors from the CPU. Since GPU-CPU communication is duplex, all three streams can proceed concurrently when used in this manner. By contrast, if one is to use multiple streams for computation, those streams cannot execute simultaneously if there is not enough GPU compute resource for parallel execution. We have observed no performance benefits in using more than one stream for computation for all the DNN models that we've tested. This observation is also shared by others [24].

Memory allocation. We need to configure the memory pool and specify memory allocation for a given dataflow graph. The memory pool consists of a number of different size-classes each of which is assigned a certain number of fixed-size tensor objects. Given a dataflow graph G which contains the sizes of all input/output tensors needed by each operator, a memory allocation scheme can be defined by specifying two things: 1) the mapping from each tensor size in G to some size class supported by the memory pool. 2) the set of supported size classes as well as the number of tensor objects assigned to each class. As an example, the coarse-grained allocation scheme in Figure 4.1(b)(c) has only one

size-class (2MB) with 5 objects, and maps each 1MB or 2MB tensor to the 2MB size-class. The fine-grained scheme in Figure 4.1(d)(e) has two size-classes (1MB and 2MB) with 8 and 1 objects respectively, and maps each 1MB tensor to the 1MB size-class and each 2MB tensor to the 2MB size-class.

4.3.2 Swap Planning

The swap planner is given the dataflow graph as well as a valid operator schedule and memory allocation scheme. Its job is to find the swap plan with the best performance under the given schedule/allocation combination. In particular the swap planner decides: 1) which memory-resident tensors to swap out under memory pressure, 2) when to perform swap-in or swap-out.

Which tensors to swap out? At the high level, the swap planner uses Belady's strategy [44] to pick the tensor that will not be needed for the longest time in the future to swap out. Seeing into the future is possible as the planner is given the schedule. Belady's strategy is optimal for cache replacement and also works well in our context as it gives the planner sufficient time to swap the tensor back before its next use. Concretely, the planner scans each operator according to the order in the schedule and keeps track of the set of input/output tensor objects that become resident in memory as a result of executing the sequence of operators. Upon encountering memory pressure when adding a tensor of size s (i.e. there is no free object in size-class of s), the planner chooses the tensor from the same size-class as s to swap out. If there are multiple candidates, the

planner chooses one that will be used in the furthest future.

There is a caveat when using Belady's strategy in our setting. Suppose tensor T_i–which is last used by operator op_i–is chosen to make room for tensor T_j, an input tensor to the current operator op_j. Thus, the earliest time T_i can be swapped out is when op_i finishes. If operators op_i and op_j are too close in time in the schedule, there is little time to swap in T_j before it's needed by operator op_j as its memory space is not available until after T_i is swapped out. As a remedy, when choosing a candidate tensor to swap out, the planner picks among those who are most recently used at least a threshold of time ago.

DNN training is iterative, but the swap planner is given the dataflow graph for a single iteration only. At the end of each iteration, all tensors other than the parameter tensors can be discarded. However, to ensure that the same swap plan can be used across many iterations, we must ensure that the set of parameter tensors in the GPU memory at the end of an iteration is the same as in the beginning. To achieve this, we perform a double-pass, i.e. scan the schedule to plan for swapping twice. In the first pass, we assume no parameter tensors are in the GPU memory, and must be swapped in before their first usage. At the end of the first pass, a subset of parameter tensors become residents in the memory, which we refer to as the initial resident parameters. We then do a second pass assuming the initial resident parameters are present in memory in the beginning of the schedule. In the second pass, if there is additional memory pressure that did not happen in the first pass, we remove a parameter tensor from the set of initial residents to resolve the pressure. The final swap plan's initial

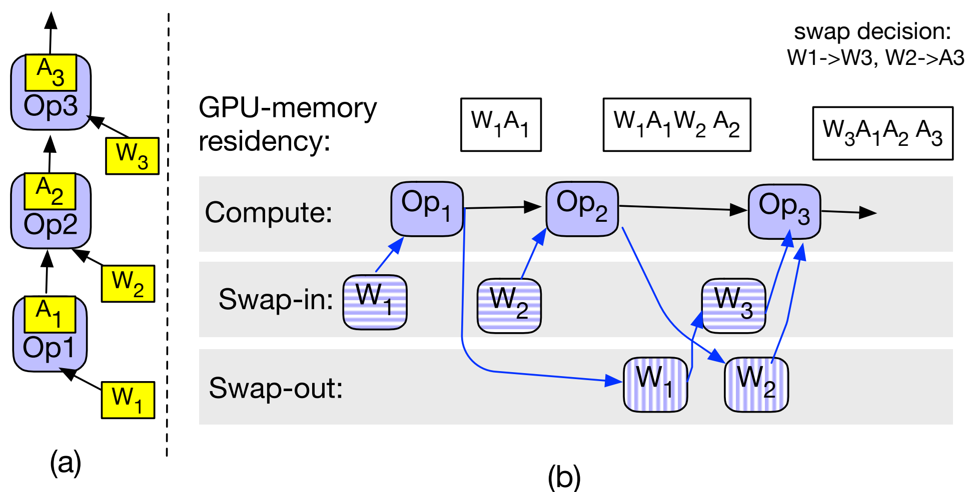


Figure 4.3: Example swap planning for a simple dataflow graph. All tensors have unit size and total GPU memory is 4 units.

GPU-resident parameters do not include those removed in the second pass.

When to swap in and out? Our previously discussed selection strategy has determined pairs of tensors to swap-out and swap-in if necessary in order to execute each operator according to the schedule. To maximize computation and communication overlap, we want to complete a pair of swap-out and swap-in as early as possible in order not to block the execution of the corresponding operator in the schedule. However, we must also ensure that the timing of swap-in/out is safe.

We illustrate how the planner controls swap timing using an example 3-node dataflow graph (Figure 4.3(a)) and the schedule op_1, op_2, op_3 (Figure 4.3(b)). For simplicity, we assume all tensors in the example are 1 unit in size and the total GPU memory size is 4 units. In order to execute op_1, we must swap in the

parameter tensor W_1, thus the planner adds a new dataflow node for swapping in W_1 which is to be run on the GPU stream dedicated for swap-ins. Similarly a swap-in node for W_2 is added. We note that there is sufficient memory to hold the input/output tensors of both op_1 and op_2. However, in order to run op_3, we need room to swap in W_3 and to allocate space for $A3$. The planner chooses W_1 to make room for W_3 (referred to as $W_1 \rightarrow W_3$) and chooses W_2 to make room for A_3 (referred to as $W_2 \rightarrow A_3$). Let's consider the case of $W_1 \rightarrow W_3$ first. The planner adds two dataflow nodes W_1(swap-out) and W_3(swap-in). A control flow edge from W_3(swap-in) to op_3 is added to ensure that operator execution starts only after W_3 is in GPU memory. An edge from W_1(swap-out) to W_3(swap-in) is added to ensure that swap-in starts only when the memory becomes available upon the completion of the corresponding swap-out. Additionally, an edge from op_1 to W_1(swap-out) is included as W_1 cannot be removed from the memory until op_1 has finished using it. The case of $W_2 \rightarrow A_3$ is similar, except that the planner does not need to add a swap-in node for A_3 because it is created by the operator. The resulting augmented dataflow graph can be passed to the framework's dataflow engine for execution.

4.4 Optimization via Genetic Algorithm

With the design of Section 4.3, one can randomly generate a combination of schedule and memory allocation for the swap planner to find a good swapping scheme. But, how good it is if we randomly create the schedule and memory allocation? Figure 4.4 shows the results of using random samples. For the first

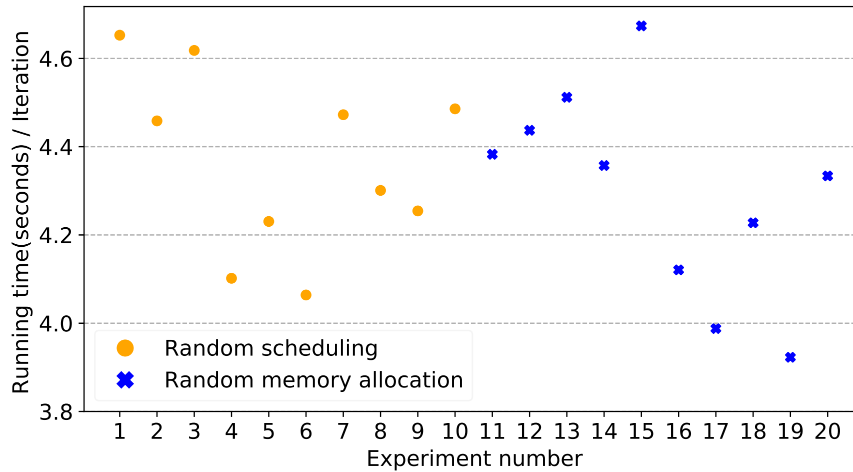


Figure 4.4: Random scheduling and memory allocation for Inception-V4 with wide scale 4 and batch size 32.

half of the experiments, we use fixed memory allocation and randomly permute the schedule. The second half of the experiments are with a fixed schedule and different randomly generated memory allocation. The best and worst samples in the figure have up to 30% performance difference. The result enforces our discussion in Section 4.2 that we should also optimize the schedule and memory allocation.

In this section, we discuss how to adopt the genetic algorithm to find a good schedule and memory allocation. For the model demonstrated in Figure 4.4, SwapAdvisor manages to find a result with 50% performance improvement than the best result shown in Figure 4.4.

4.4.1 Algorithm Overview

Genetic algorithm (GA) aims to evolve and improve an entire population of individuals via the nature-inspired mechanisms such as crossover, mutation, and selection [38, 39, 40]. In SwapAdvisor, an individual's chromosome consists of two components: an operator schedule and a memory allocation. The first generation of individuals are created randomly and the size of the population is decided by a hyper-parameter, N_p.

To create a new generation of individuals, we perform crossover and mutation on the chromosomes of the current generation. A crossover takes a pair of parent chromosomes and produces new individuals by combining the features from parents so that children can inherit "good" characteristics (probabilistically) from the parents. In SwapAdvisor, each crossover generates two new schedules and two memory allocation, thereby resulting in 4 children. We then perform mutation on the children which is essential for GA to escape the local minimum and avoid premature convergence [38, 39, 40]. The resulting mutated children are given to the swap planner to generate the augmented dataflow graph with swapping nodes. We use a custom-built dataflow simulator to execute the augmented graph and obtain the execution time, which is used to measure the quality of an individual. Finally, the GA selects N_p individuals among the current population to survive to the next generation.

Selection methodology How to select individuals to survive is crucial in GA. If we choose only the best individuals to survive, the population can lose diversity

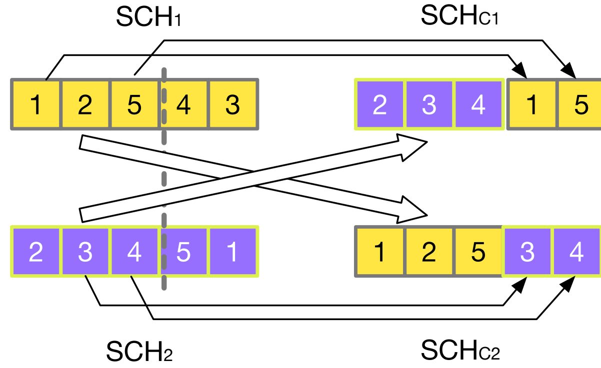


Figure 4.5: Cross over SCH_1 and SCH_2 for a toy dataflow graph. There are five nodes in the dataflow graph.

and converges prematurely.

SwapAdvisor's selection takes into account the quality of an individual to determine the probability of its survival. Suppose an individual's execution time is t, we define its normalized execution time as $t_{norm} = (T_{Best} - t)/T_{Best}$, where T_{Best} is the best time among all individuals seen so far. The survival probability of an individual is decided by the $softmax$ function.

$$Prob_i = \frac{e^{t_{norm_i}}}{\sum_{j=1}^{S} e^{t_{norm_j}}} \text{ for } i = 1 \dots S \tag{4.1}$$

In the equation, S is the population size before selection (usually larger than N_p). We use the softmax-based selection because our experiments show that it reaches more stable results compared to the popular tournament selection [38, 39, 40].

4.4.2 Creating New Schedules

Encoding As a schedule is a topological ordering of the dataflow graph (G), it is natural to encode a schedule as a list, SCH, where each element in SCH is a

node id in G.

Crossover We borrow the idea from [43] to create two child schedules. by crossing over two parent schedules, SCH_1 and SCH_2. We explain via an example, shown in Figure 4.5. First, a crossover point, CR, is chosen randomly. In the example, $CR = 3$. To create child SCH_{C1}, the crossover takes a slice of SCH_2 ($SCH_2[1 \ldots CR] = [2, 3, 4]$) to be the first part of SCH_{C1}. The nodes not in the SCH_{C1} are filled in according to their order in SCH_1. In the example, nodes 1 and 5 are not in $SCH_{C1} = [2, 3, 4]$, thus we fill them in the remaining slots of SCH_1, in that order. SCH_{C2} can be created via the same approach but with different parts from SCH_1 and SCH_2, as shown in the bottom of Figure 4.5. The algorithm guarantees SCH_{C1} and SCH_{C2} are the topological ordering of G [43].

Mutation A simple way to mutate a schedule is to change a node's position in the list randomly as long as the result remains a topological ordering [43]. However, we have empirically observed that GA works better if we mutate multiple nodes in one mutation (e.g., more than 2x performance improvement for RNNs).

SwapAdvisor's mutation algorithm mimics a dataflow scheduler. It maintains a ready set, containing all the nodes which are ready to run (all the predecessor nodes are executed). The core function of the mutation algorithm is to choose a node from the ready set based on following two conditions. First, with a probability $\mathcal{P}$, a mutation happens. In such a case, the algorithm randomly chooses a node from the ready set. Otherwise, the algorithm selects the node from the ready set which is executed earliest in the original schedule (not mutated).

A chosen node is viewed as "executed". The algorithm terminates when all the nodes are "executed". The mutation algorithm generates a new schedule which mostly follows the input schedule but with some nodes scheduled differently.

4.4.3 Creating New Memory Allocation

Encoding The memory allocation controls how to map each size to a size class and how many objects to assign to each size class. Although it's natural to use a hash map to map tensor sizes to size classes, doing so loses the relative size information between different tensor sizes, making it more difficult to do efficient crossover. We use two lists, CLS and CNT, to represent the tensor size-class mapping.

Let TS be the sorted list of unique tensor sizes observed in the dataflow graph. CLS is a list with the same length as TS, and the i_{th} item in CLS ($CLS[i]$) is a positive integer representing the size-class for tensors with size $TS[i]$. Thus, the number of the size-classes is $Max(CLS)$. CNT is a list representing the number of tensor objects allocated for each size-class. Consequently, the length of CNT is $Max(CLS)$. As an example, the dataflow graph of Figure 4.1 has only two different tensor sizes, thus $TS = [1MB, 2MB]$. The coarse-grained allocation with five 2MB objects corresponds to $CLS = [0, 0]$, indicating that both 1MB and 2MB sizes are mapped to the same size-class with id 1 and object size is 2MB. $CNT = [5]$ contains the number of objects assigned to each size class from $id = 0...Max(CLS)$.

The number of potential CLS lists is $O(N^N)$ where N is the number of unique

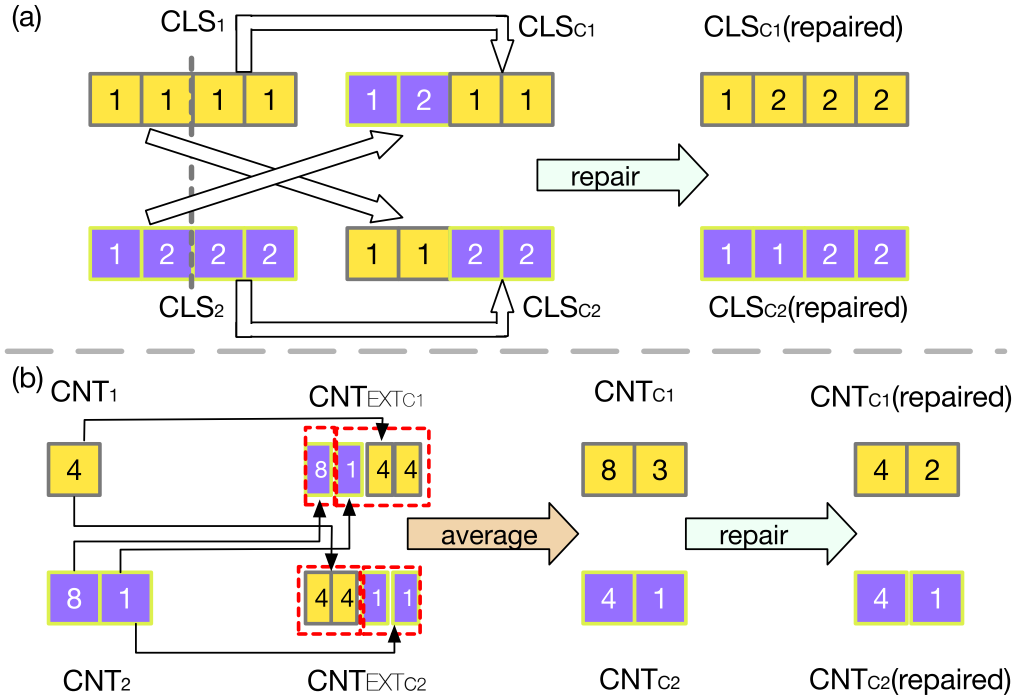


Figure 4.6: Cross over (CLS_1, CNT_1) and (CLS_2, CNT_2) for a toy dataflow graph. There are four different sizes of tensors, 1MB, 2MB, 3MB, and 4MB. Assume the GPU memory is 12MB.

tensor sizes [45]. Such a gigantic search space can seriously cripple the performance of GA. We prune the search space by imposing the restriction that CLS must be a monotonically increasing sequence and each $CLS[i]$ is equal to or one greater than $CLS[i-1]$. The intuition is that the allocation is more efficient when consecutive sizes are mapped to the same or adjacent size classes. This restriction cuts down the search space from $O(N^N)$ to $O(2^N)$.

Crossover We first explain how to cross over two CLS using an example, shown in Figure 4.6. There are two CLS before crossover (CLS_1 and CLS_2) and four

different tensor sizes: 1MB, 2MB, 3MB and 4MB. $CLS_1 = [1, 1, 1, 1]$ means that all four tensor sizes belong to the same size-class, 4MB, and $CNT_1 = [4]$ indicates that there are four 4MB tensor objects allocated. $CLS_2 = [1, 2, 2, 2]$ means that there are two size-classes, 1MB and 4MB. There are eight 1MB tensor objects and one 4MB tensor object as CNT_2 is $[8, 1]$.

The crossover randomly picks a crossover point, CR to partition the parent lists, CLS_1 and CLS_2. The first child size-class mapping CLS_{C1} is made by concatenating $CLS_2[1 \ldots CR]$ and $CLS_1[CR + 1 \ldots N]$. The second child size-classes mapping CLS_{C2} is made by concatenating $CLS_1[1 \ldots CR]$ and $CLS_2[CR + 1 \ldots N]$. Figure 4.6(a) shows the crossover for CLS. In the figure, CR is 2 and the results, are CLS_{C1} ($[1, 2, 1, 1]$) and CLS_{C2} ($[1, 1, 2, 2]$). Note that CLS_{C1} is not monotonically increasing. Thus, we repair it to ensure the new size-class mapping is still valid. Our repair increases the elements in a problematic CLS by the minimal amount so that the resulting sequence becomes valid. In Figure 4.6(a), we would increase the 3rd and 4th elements of CLS_{C1} by 1, so that the repaired CLS_{C1} becomes $[1, 2, 2, 2]$.

The same crossover scheme cannot be directly used for CNT as its length depends on the content of the corresponding CLS. As a result, we extend a CNT to an extended CNT_{EXT} which has the same length as CLS (and TS). CNT captures how many tensor objects are allocated for each size-class, while CNT_{EXT} indicates how many tensor objects can be used for each tensor size. For example, in Figure 4.6, CLS_2 is $[1, 2, 2, 2]$ which means 1MB belongs 1MB size-class and 2MB, 3MB, and 4MB belong to 4MB size-class. CNT_2 is $[8, 1]$

and CNT_{EXT_2} can then be viewed as $[8, 1, 1, 1]$. CNT_{EXT_1} is $[4, 4, 4, 4]$. We apply the same technique to cross over the extended CNT. Figure 4.6(b) shows the resulting extended CNTs for child1 and child2. For example, $CNT_{EXT_{C1}}$ is $[8, 1, 4, 4]$ and the last three element belong to the same size-class (according to CLS_{C1}). We average all the elements in the same size class to get the count for that size-class. Thus the resulting CNT_{C1} is $[8, 3]$.

Similar to CLS, a new CNT may need to be repaired. For example, CNT_{C1} is invalid, as the memory consumption is 20MB $(8 * 1 + 3 * 4)$, exceeding the 12MB memory capacity. We repair a CNT by decreasing each element inverse-proportionally to the element's corresponding size-class. For example, the original CNT_{C1} is $[8, 3]$ and the repaired CNT_{C1} is $[4, 2]$, as the first size-class is 1MB and the second size-class is 4MB.

Mutation Similar to the scheduling mutation, we mutate more than one element in CLS (and CNT). An element is mutated with the probability of $\mathcal{P}$

To mutate the i_{th} element in CLS, we can either increase it by 1 or decrease it by 1. If $CLS[i]$ equals to $CLS[i-1]$, we can increase $CLS[i]$ by 1 as decreasing it breaks the monotonic increasing feature. If $CLS[i]$ equals to $CLS[i-1]+1$, we decrease $CLS[i]$ by 1. Note that, all the elements after the i_{th} element also need to be increased or decreased to maintain the monotonic increasing feature.

To mutate an element in CNT, we use a Gaussian distribution with the original value as the mean. With Gaussian distribution, the mutated value is close to the original value most of the time but can have a large variation with a small chance. The mutated CNT and CLS can exceed the memory limit. We

use the same methodology as crossover to repair them.

4.5 Implemetation

SwapAdvisor is implemented based on MXNet 1.2. The genetic algorithm components and simulator are written in Python (4.5K LoC). We use a parallel implementation of the genetic algorithm – each process performs crossover, mutation, and simulation of a portion of the samples on a CPU core.

The augmented dataflow graph generated by SwapAdvisor is feed to MXNet. We modify MXNet's scheduler to ensure all swap-in operations and swap-out operations are executed in two separated GPU streams other than the computation stream. A new memory allocator is implemented in order to make MXNet to follow the memory allocation results from SwapAdvisor. The total modification of MXNet is 1.5K LoC.

Chapter 5

SwapAdvisor Evaluation

In this chapter, we evaluate the performance of SwapAdvisor. The followings are the highlights of our results:

- SwapAdvisor can achieve 53%-99% of the training throughput of the ideal baseline with infinite GPU memory when training various large DNNs. SwapAdvisor outperforms the online swapping baseline up to 80× for RNNs and 2.5× for CNNs.

- When being used for model inference, SwapAdvisor reduces the serving latency up to 4× compared to the baseline which time-shares the GPU memory.

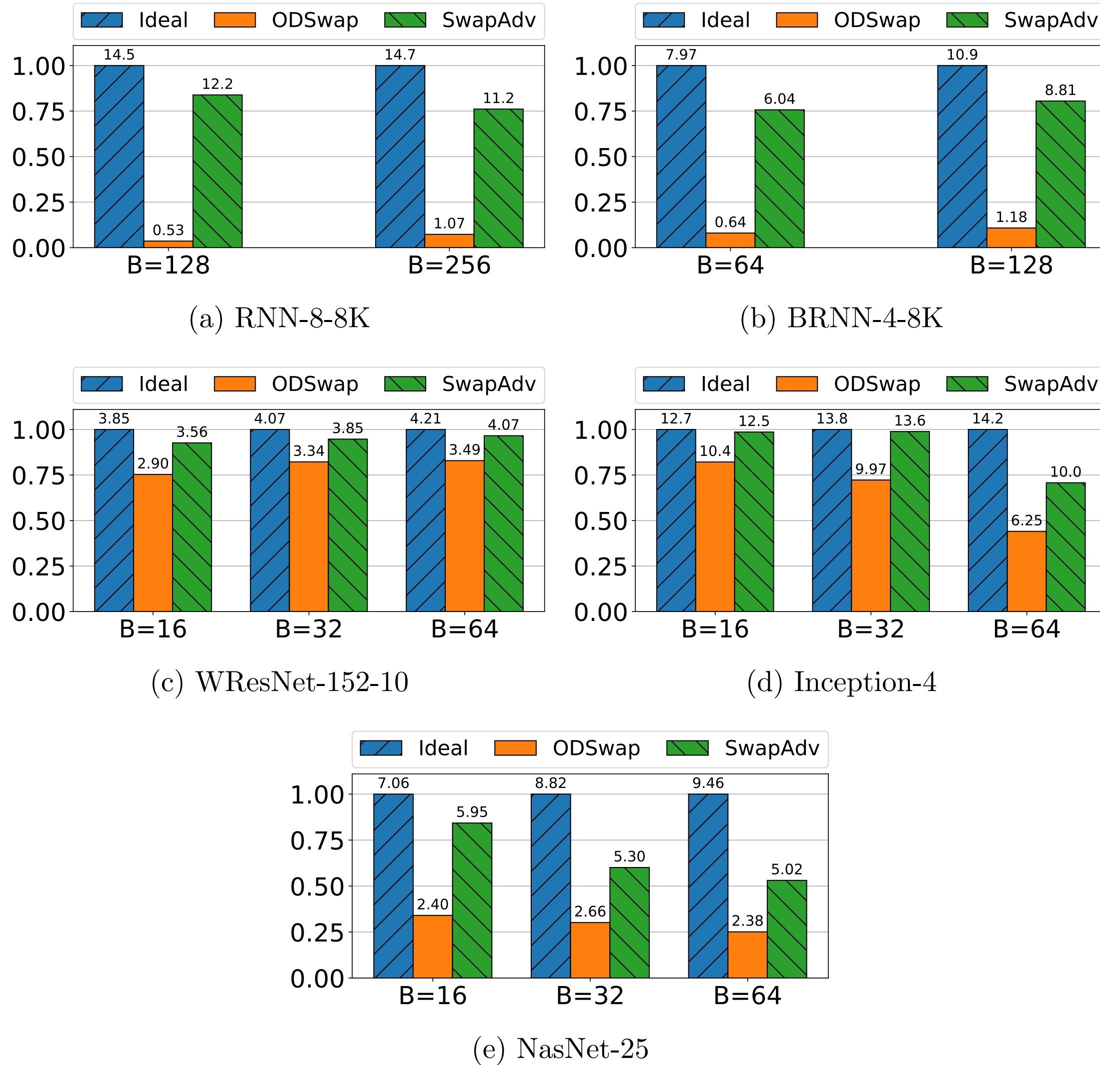
- SwapAdvisor's joint optimization improves the training throughput with swapping from 20% to 1100% compared to only searching memory allocation or only searching scheduling.

5.1 Experimental Setup

Testbeds We run SwapAdvisor on an EC2 c5d.18xlarge instance with 72 virtual CPU cores and 144GB memory. The results of SwapAdvisor are executed on an EC2 p3.2xlarge GPU instance. The p3.2xlarge instance has one NVIDIA V100 GPU with 16GB GPU memory and 8 virtual CPU cores with 61GB CPU memory. The PCIe bandwidth between the CPU and GPU is 12GB/s unidirectional and 20GB/s bidirectional. For experiments with more than 61GB memory consumption, we use a p3.8xlarge instance with 244 GB CPU memory but utilize only a GPU.

Genetic algorithm parameters All parameters of the genetic algorithm are determined empirically. The sample size is set to 144 to allow evenly distributing search tasks to the 72 CPU cores. The effectiveness of the mutation probability varies with different DNN models. However, 10% is a good start search point for our evaluation. Although it is possible to get a better result with a longer search time, SwapAdvisor can find good results for all of our evaluated DNN models in 30 mintues. Thus, we set 30 minutes to be the search time limit. If the search converge earler (no improvement for more than 5 minutes), we terminate the search to save the computation resources.

Evaluated DNN models ResNet [22] is one of the most popular CNNs. A ResNet contains several residual blocks; a residual block has several convolution operators. The activation of a residual block is combined with activation from



(a) RNN-8-8K

(b) BRNN-4-8K

(c) WResNet-152-10

(d) Inception-4

(e) NasNet-25

Figure 5.1: Normalized throughput relative to the ideal performance. Each group of bar represents one batch size. The number on each bar shows the absolute throughput in samples/sec. X axis shows the different batch sizes.

the previous block to form the final output. We use ResNet-152, a 152 layers ResNet, for the inference experiments. Wide ResNet [46] is a widened version of ResNet. The channel size of convolution operators are multiplied by a wide scale. Due to the large memory consumption, the original work applies wide ResNet on small images (32x32) dataset CIFAR-10 instead of ImageNet (224x224) which is used by ResNet. In our training experiments, the input images are the same size as ImageNet. We denote a wide ResNet model as WResNet-152-X, a 152-layers wide ResNet with X wide scale.

Inception-V4 [36] is another popular CNN model. An Inception-V4 model contains several types of inception cells; an inception cell has many branches of convolution and the activation tensors of all the branches are concatenated to form the final output. We enlarge the model by adding a wide scale parameter similar to WResNet. We use the notation Inception-X to denote an Inception-V4 model with wide scale X.

Unlike manually designed ResNet and Inception-V4, NasNet [37] is crafted by a deep reinforcement learning search. Thus, the model structure of NasNet is more irregular. A NasNet model consists of a chain of Reduction and Normal cells, with residual connection (same as ResNet) between consecutive cells. A Reduction or Normal cell is like an inception cell but with different branch structures. The Normal cells are repeated by $3R$ times. In the original design, the max R is 7. In our experiments, we train NasNet-25, a NasNet with $R = 25$.

RNN [47] is a DNN for training sequence input (e.g., text). A layer of RNN consists of a list LSTM cells where the input of a cell is the corresponding element

in the sequence (e.g., character). Bidirectional-RNN (BRNN) [48] is a variation of RNN. A hidden layer in BRNN contains two sub-layers. The input for the first sub-layer is the original sequence, and the input for the second sub-layer is the reversed sequence. The activation tensors of the two layers are concatenated to form the final activation. Each sub-layer has its own parameters. We use the notations RNN-L-XK (BRNN-L-XK) to denote a RNN with L layers and the parameter size of a layer is XK.

Baselines We compare SwapAdvisor with two baselines. The first one is the ideal baseline, denoted as *ideal*. For the ideal baseline, we assume the GPU memory is infinite. We implement the ideal baseline by directly reusing the GPU memory without considering computation correctness. The performance of ideal is the best a swapping system can achieve.

The second baseline is an online swapping system, On-Demand swap, denoted as *ODSwap*. ODSwap swaps out GPU memory when the GPU memory is insufficient to run the next node. It chooses the tensor to swap out using LRU algorithm. There is a separate prefetch thread that runs concurrently with the execution thread. The prefetch thread decides which tensor to prefetch based on the topology of the dataflow graph. Unlike SwapAdvisor, ODSwap does not control the scheduling; it is just an extension to MXNet's memory management. MXNet's memory management does not support dynamic memory allocation. We implement a memory allocator (based on the buddy algorithm) to allow ODSwap to dynamically allocate memory. The design choices require no search decisions like SwapAdvisor does.

5.2 Wider and Deeper DNN Models Training

Table 5.1 shows the statistics of the models evaluated in this section. Each row shows the memory usage, number of operators, and number of different tensor sizes. The batch size for a model is the largest one in Figure 5.1.

RNN performance Figure 5.1a and 5.1b show the throughput for RNN-8-8K and BRNN-4-8K. SwapAdvisor achieves 70-80% of the ideal performance for RNN and BRNN, while the throughput of ODSwap is only less than 1% of ideal. For RNN and BRNN, the parameter tensors are shared by the LSTM cells in the same layer. Thus, a schedule which executes LSTM cells from different layers results in terrible swapping performance as the system has to prepare memory for different large parameters. Unfortunately, randomly generating a topological ordering almost always results in such a schedule, as is MXNet's default schedule. Thus ODSwap has poor performance. SwapAdvisor is able to find a swap-friendly schedule through genetic algorithm.

Model	MemUsage	OPs	TensorSizes
WResNet-152-10	180GB	882	26
Inception-4	71GB	830	64
NasNet-25	193GB	5533	65
RNN-8-8K	118GB	8594	7
BRNN-4-8K	99GB	9034	9

Table 5.1: Statistics of DNN models. The batch size for CNN models is 64, is 256 for RNN, and is 128 for BRNN.

CNN performance Figure 5.1c - 5.1e show the throughput for WResNet-152-10, Inception-4, and NasNet-25. Table 5.1 shows that WResNet-152-10 uses astonishingly 180GB memory, but both SwapAdvisor and ODSwap perform well; SwapAdvisor achieves 95% of the ideal performance, and ODSwap achieves 80% of ideal. WResNet-152-10 has only 26 different tensor sizes, making it less difficult to do the memory allocation. More importantly, unlike RNN/BRNN, the topology of the dataflow graph of WResNet more resembles to a line – only a jump link for a residual block. Thus, the scheduling choice may affect little to the final results.

On the other hand, Inception-4 and NasNet-25 have more than 60 different tensor sizes, making it harder to do memory management. The topology of the dataflow graph for Inception-V4 and NasNet is also more complicated as discussed in Sec 5.1. SwapAdvisor achieves 20% - 150% performance improvement compared to ODSwap.

Note that, for Inception-4 and NasNet-25, SwapAdvisor can achieve 80% performance of the ideal baseline, when the batch size is 16. However, SwapAdvisor cannot achieve more than 65% of the ideal performance when the batch size is 64. Both Inception-4 and NasNet-25 have many large activation tensors (>500MB) when the batch size is 64. Together with large number of tensor sizes (> 60), it can be difficult for SwapAdvisor to search a good pool configuration to minimize swapping overhead. NasNet-25 also has more than 9000 nodes in the graph, making it hard to schedule. As a result, SwapAdvisor achieves only 53% of the ideal baseline for NasNet-25 with batch size 64.

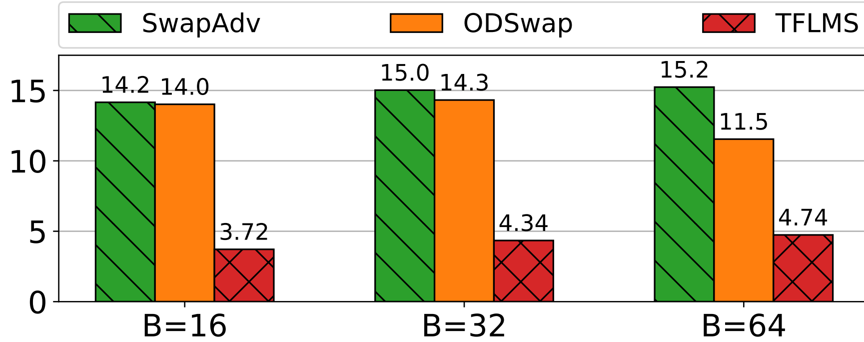


Figure 5.2: WResNet-152-4 throughput comparison

Comparing with TFLMS. We also compare SwapAdvisor and ODSwap with TFLMS [26], an swapping extension to TensorFlow. Unfortunately, TFLMS cannot support models in Figure 5.1. We evaluate WResNet-152-4 as this is the largest executable one for TFLMS. Figure 5.2 shows that for all three batch sizes, SwapAdvisor is at least 3X better than TFLMS, while ODSwap is at least 2X better than TFLMS. TFLMS performs poorly due to not swapping out parameter tensors (which are large in WResNet-152-4). The design reduces the GPU memory capacity to store activation tensors and causes more swapping of activation tensors.

5.3 DNN Models Inference Evaluation

DNN model inference uses far less GPU memory than training a model as there is no back-propagation. However, SwapAdvisor is still useful for model inference in several cases.

Table 5.2 shows how SwapAdvisor can reduce the memory requirement for

MemSize/Batch	1	16	32	64
64MB	**0.024s**	N/A	N/A	N/A
128MB	**0.022s**	**0.077s**	N/A	N/A
192MB	**0.018s**	**0.044s**	N/A	N/A
256MB	***0.017s***	**0.043s**	**0.130s**	N/A
320MB	*0.017s*	**0.042s**	**0.073s**	N/A
512MB	*0.017s*	***0.040s***	***0.067s***	**0.238s**
640MB	*0.017s*	***0.040s***	***0.067s***	***0.123s***
1024MB	0.017s	0.040s	0.067s	0.121s

Table 5.2: ResNet-152 inference time with different batch sizes and GPU memory sizes.

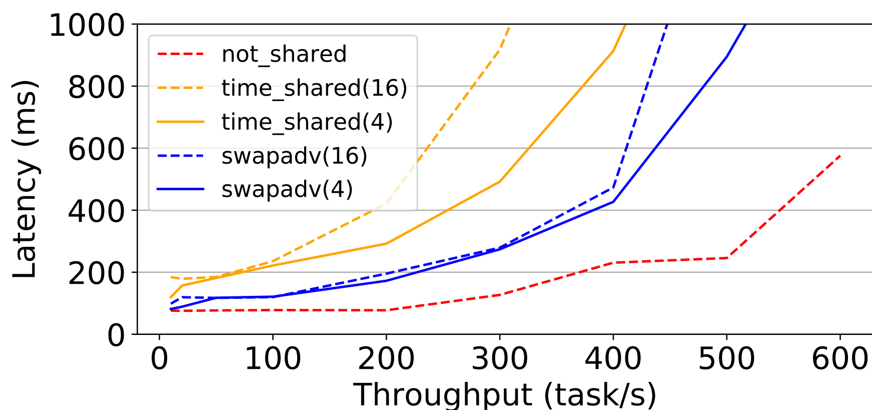
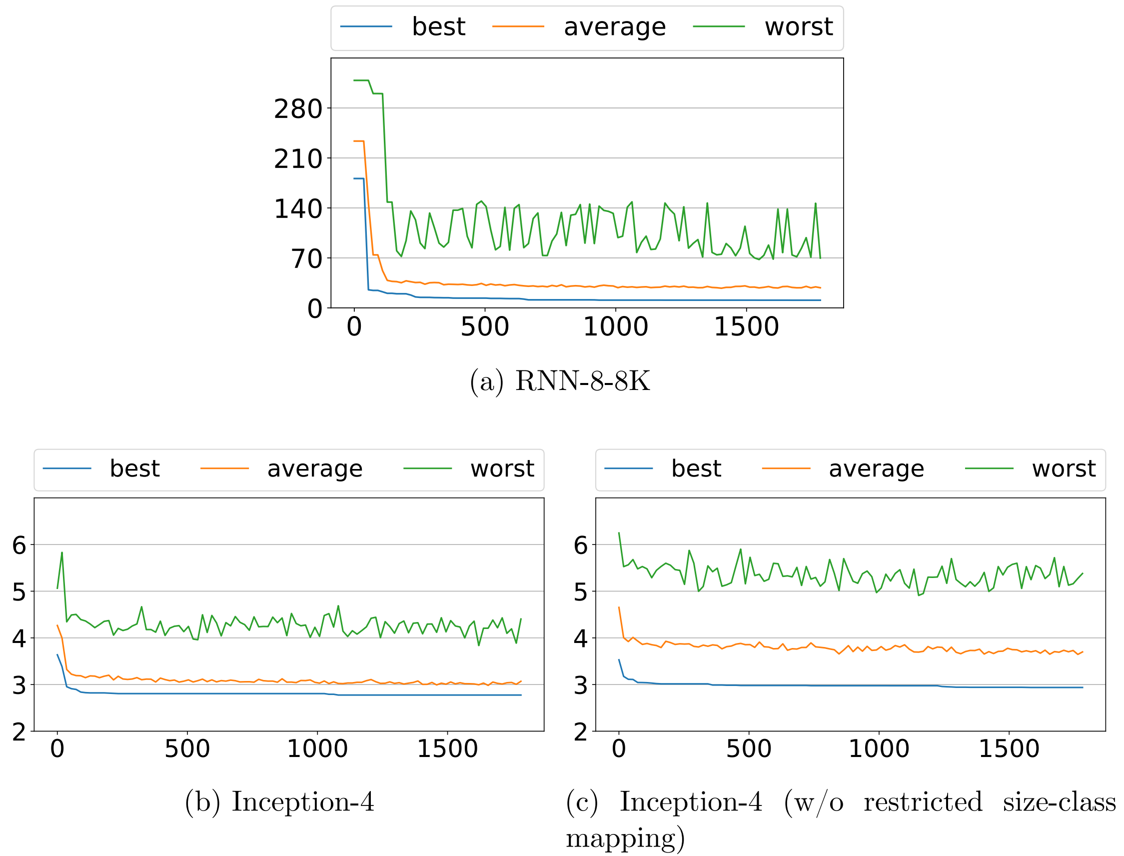


Figure 5.3: 99 percentile latency versus throughput for serving multiple ResNet-152 models.

ResNet-152 with different batch sizes. In the table, each cells is the running time of one inference iteration with the corresponding available GPU memory size. ”N/A” means SwapAdvisor cannot run the inference job with such a small memory capacity. The running time with a bold font means swapping is required to run the job (memory is not enough). The running time with an italic font means that the running time is close to the performance using full 16GB memory



(a) RNN-8-8K

(b) Inception-4

(c) Inception-4 (w/o restricted size-class mapping)

Figure 5.4: The search result versus search time. X axis is the search time and Y axis is the running time (seconds/iteration).

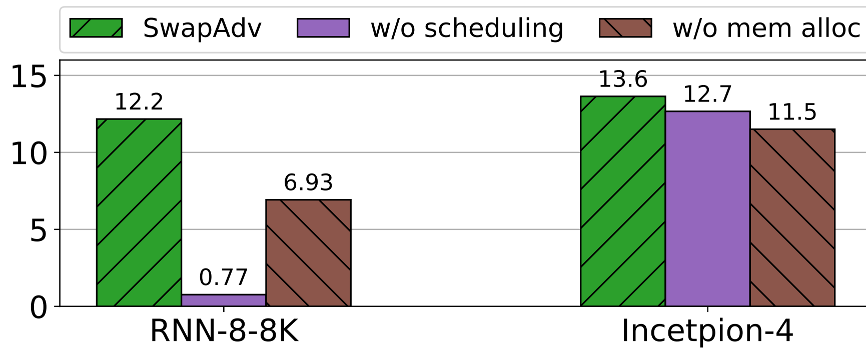


Figure 5.5: Throughput for SwapAdvisor with different search settings for RNN and Inception-V4.

capacity ($< 1\%$ performance difference).

An interesting experiment is when the batch size is 1. Batch size 1 is rarely used for training or inference on a cluster, but it is not uncommon to be used for inference on a mobile device. Table 5.2 demonstrates SwapAdvisor can reduce the memory usage for batch size 1 to as few as 64MB with 40% running time overhead or to 192MB with only 6% overhead. SwapAdvisor can help to fit a DNN model to a resource-limited GPU. Though the communication speed between GPU and CPU on a mobile device is slower than that on an EC2 GPU instance, V100 GPU is also much faster than a mobile GPU. Consequently, the actual swapping overhead on a mobile device can be different from what Table 5.2 shows. Nevertheless, the result still poses a potential use case for SwapAdvisor

Another use case for SwapAdvisor is to time-share the GPU resource among different DNN inferences. In the setting, a GPU machine time-shares the computation and memory among models. What if we only time-share GPU computation but partition the GPU memory and distribute the GPU memory to the models? Since the split GPU memory capacity may be too small for a DNN model, we apply SwapAdvisor so that all the models can fit on the partitioned memory.

We consider time-sharing GPU memory as the baseline and compare the latency of clients. In the evaluation, there are multiple ResNet-152 on the GPU, each has its own trained parameters. We assume that the client arrival rate follows a Poisson distribution and randomly assign batches of clients to different models. Figure 5.3 shows the 99 percentile latency versus the throughput of the GPU machine. The number after a legend denotes how many models are run on

the GPU. The figure also shows the latency for serving only one model on the GPU, denoted as "not_shared".

We can see that the 99 percentile latency of SwapAdvisor is at most 2× slower than "not_shared" when the throughput is less than 400 for both 4 and 16 models. On the other hand, the latency of "time_shared" is 8x slower than "not_shared" with 16 models when the throughput is 300. It may not be a wise decision to serve several ResNet-152 on a GPU when the throughput is larger 400 as the latency dramatically increases for both SwapAdvisor and "time_shared".

The main benefit of SwapAdvisor is that it overlaps the memory copy with the computation. On the other hand, the baseline has to swap in the parameter tensors for the next model after the current model execution. It is possible for "time_shared" to prefetch the parameters for the next model if the system can predict which model to execute next. With such a task scheduler, the baseline may outperforms SwapAdvisor. However, SwapAdvisor can still be used to mitigate the potential overhead when the task scheduler predicts incorrectly.

5.4 The Effectiveness of SwapAdvisor's Design Choices

The effectiveness of scheduling and memory allocation We would like to see the importance to optimize both scheduling and memory allocation. Figure 5.4 shows that it is very important for a RNN model to search a swap-friendly schedule. Without searching a good schedule, SwapAdvisor can only achieve 7%

of the performance with the full search. On the other hand, Figure 5.5 shows that the memory allocation affects the performance of Inception-V4 more than the scheduling. Without searching schedules, SwapAdvisor can still achieve 93% performance of the full search. Figure 5.5 demonstrates that it is important to optimize both scheduling and memory allocation for swapping as the effectiveness of the two components vary from model to model.

SwapAdvisor's genetic algorithm performance Figure 5.4 shows the search performance of the genetic algorithm. In the figures, there are three lines, representing the best, average, and worst simulated time of all the alive samples (144 samples) at the moment. The first generation of sample is randomly generated. The genetic algorithm can generally find a good solution within 100 seconds, as both the average and the best simulated time converge quickly within the first 100 seconds. However, the difference between the worst (or the average) result and the best result shows that the population still maintains diversities, allowing the genetic algorithm to gradually optimizes the sample in the remaining time.

Both Figure 5.4b and Figure 5.4c show the search for Inception-4, but Figure 5.4c assumes that if the tensor size mapping is unrestricted. We can see that all of the best, average, and worst result in Figure 5.4c are worse than Figure 5.4b, proving that the restricted search space helps SwapAdvisor to find better results. The effectiveness of the restricted search space is universal to all the evaluated models.

Chapter 6

Related Work

6.1 Spartan's Related Work

There is much prior work in the area of distributed array framework design and optimization.

Compiler-assisted data distribution. Prior work in this space proposes static, compile-time techniques for analysis. The first set of techniques focuses on partitioning [33] and the latter set on data co-location [49, 50, 51]. Prior work also has examined nested loops with affine array subscript patterns, using different structures (vector [33], matrix [52] or reference [53]) to model memory access patterns or polyhedral model [54] to perform localization analysis. Since static analysis deals poorly with ambiguities in source code [55], recent work proposes profile-guided methods [56] and memory-tracing [57] to capture memory access patterns. Simpler approaches focus on examining stencil code [57, 58, 59, 60, 61]. Spartan simplifies analysis significantly since high-level operator access patterns

are well-defined. For example, the *map* operator accesses elements in the same position across array pairs, together.

Access patterns can be used to find a distribution of data that minimizes communication cost [33, 62, 63, 64, 65]. All approaches construct a weighted graph that captures possible layouts. Although searching the optimal solution is NP-Complete [66, 67, 68, 69], heuristics perform well in practice [69, 50]. Spartan also constructs a weight graph where finding a solution is NP-Complete (appendix A). Our heuristic is able to almost always find an optimal solution in practice. Moreover, prior work presents language-specific solutions that rely on compile-time analysis. Spartan avoids the analysis by introducing high-level operators with known tiling costs. Spartan uses lazy evaluation to obtain runtime information.

Parallel vector languages. ZPL [70], SISAL [71], NESL [72] and MatLab*P [73] share a common goal with Spartan. These languages expose distributed arrays and vector primitives and some provide a few core operators for parallel operations. Unlike Spartan, ZPL does not allow arbitrary indexing of distributed arrays and does not allow parallelization of indexable arrays. NESL relies on a *PRAM* model which assumes that a shared, distributed region of memory can be accessed with low latency. Spartan makes no such assumption. SISAL provides an explicit tiled model for arrays [74], however does not consider tiling strategies.

Distributed programming frameworks. Most distributed frameworks target primitives for key-value collections (e.g. MapReduce [3], Dryad [13], Piccolo [14], Spark [4], Ciel [75], Dandelion [15] and Naiad [16]). Some provide graph-centric primitives (e.g. GraphLab [76] and Pregel [77]). While one can encode arrays

as key-value collections or graphs, doing so is much less efficient than Spartan's tile-based backend. Nevertheless, Spartan borrows many ideas from these systems, such as dataflow graphs, fault tolerance and load balancing. It is possible to implement Spartan's backend by augmenting an in-memory framework, such as Spark or Piccolo. However, we built our own prototype to allow for better integration with NumPy.

FlumeJava [31] provides programmers with a set of high-level operators. Its operators are transformed into MapReduce's [3] dataflow functions. FlumeJava is targeted at key-value collections instead of arrays. FlumeJava's operators look similar to Spartan's, but their underlying semantics are specific to key-value collections instead of arrays. Moreover, FlumeJava does not explicitly optimize for data locality because it is not designed for in-memory computation.

Relational queries are a natural layer on top of key-value centric distributed execution frameworks, as seen in systems like DryadLINQ [19], Shark [78], Dandelion [15] and Dremel [79]. Several efforts attempt to build an array interfaces on these. MadLINQ [18] adds support for distributed arrays and array-style computation to the dataflow model of DryadLINQ [19]. SciHadoop [80] is a plug-in for Hadoop to process array-formatted data. Google's R extensions [81], Presto [17] and SparkR [82] extend the R language to support distributed arrays. Julia [83] is a newly developed dynamic language designed for high performance and scientific computing. Julia provides primitives for users to parallel loops and distribute arrays. These extensions and languages rely on users to specify a tiling for each array, which burdens users with making non-trivial optimization that

require deep familiarity which each operation and its data.

Distributed array libraries. Optimized, distributed linear algebra libraries, like LAPACK [35], ScaLAPACK [84], Elemental [85] Global Arrays Toolkit [86] and Petsc [87, 88] expose APIs specifically designed for large matrix operations. They focus on providing highly optimized implementations of specific operations. However, their speed depends on correct partitioning of arrays and their programming model is difficult to extend.

Global Address Spaces. Systems such as Unified Parallel C [89] and co-array Fortran [90] provide a global distributed address space for sharing arrays. They can be used to implement the backend for distributed array libraries. They do not directly provide a fully functional distributed array language.

Specialized application frameworks. There are a number of frameworks specifically targeted for distributed machine learning (e.g. MLBase [5], Apache Mahout [6], and Theano [91], for GPUs). Unlike these, Spartan targets a much wider audience and thus must address the complete set of challenges, including support for a number built-ins, minimizing the number of temporary copies and optimizing for locality.

Array Databases and Query Languages SciDB [92] and RasDaMan [93] are distributed databases with n-dimensional data storage and an array query language inspired by SQL. These represent the database community's answer to big numerical computation. The query language is flexible, but as the designers of SciDB have seen, application programmers often prefer expressing problems in more comprehensive array languages. SciDB-R is an attempt to win over R

programmers by letting R scripts access data in SciDB and use SciDB to execute some R commands. SciDB's partition strategy is optimized for disk utilization. In contrast, Spartan focuses on in-memory data.

6.2 SwapAdvisor's Related Work

Swapping for DNN Existing swapping systems rely on manual insights to determine what to swap. vDNN [24] swaps out all activation tensors or swap all convolution tensors only. TFLMS [26] also only swaps activation. [27] uses the length of the critical path for an activation tensor and its loss-function node as the heuristic to decide what activation tensors to swap out. [30] is an on-demand swapping mechanism for TensorFlow. Its heuristic is to swap out tensors from the previous iterations to the host memory, a strategy that only works for RNNs.

None of the work above swaps parameters, hence cannot support a large model. SuperNeurons [25] adopts a different approach; it combines swapping with recomputation. However, SuperNeurons restricts the swapping to convolution operators. The decision forbids SuperNeurons from supporting an RNN model with large parameters. By constrast, SwapAdvisor can support various kinds of deeper and wider DNN models.

Alternative approaches to overcome GPU memory limit There exist approaches that do not rely on swapping to reduce memory consumption. The first direction includes computing with lower-precision floating-point numbers [94, 95], quantization, and parameters compression [96, 97, 98, 99, 100, 101]. [102] ob-

serves the similarities among the activation tensors for CNN inferences and proposes to reuse the activation tensors to speedup the performance and to reduce memory consumption. These techniques either affect the model accuracy or require heavily hyper-parameter tuning while swapping does affect the results.

Another approach is recomputation. Recomputation utilizes the fact that an activation tensor can be recomputed. As a result, [103, 104, 105] deallocate activation tensors after their last usage in the forward-propagation and later recompute the activation tensors when they are needed. Although recomputation can be used for deeper models and large input data, it fails to support wider models where large parameter tensors occupy the memory and cannot be recomputed.

Finally, training DNN models with multiple GPUs is an active research. The most popular way to parallelize a DNN model is data parallelism [106, 107, 108]. With data parallelism, each GPU gets a portion of the input data and full parameters of the model. Thus, the input tensor and activation tensors are sliced and distributed to GPUs, effectively reducing the memory consumption of each GPU. While easy to use, data parallelism duplicates the full parameters on each GPU, limiting the largest parameter size the model can have. Contrary to data parallelism, model parallelism partitions both activation tensors and parameter tensors [109, 110]. However, applying model parallelism for a model requires significant engineering work. [111, 112, 113] propose to automate the model parallelism and reduce the communication with dataflow graph analysis.

Multiple DNN inferences on a GPU TensorRT [114] leverages GPU streams to run multiple model inferences concurrently. NVIDIA MPS [115] also supports concurrent GPU tasks, but the tasks are not limited to DNN inference. Both TensorRT and MPS requires users to partition the GPU memory for tasks. SwapAdvisor can help both systems to alleviate the memory pressure. Salus [116] aims to support fine-grained GPU sharing among DNN tasks. Salus allocates a shared memory space to store activation tensors and scratch space for all the models as these memory consumption can be dropped directly after the last usage in an iteration. It assumes parameter tensors are in the GPU memory, and thus can borrow SwapAdvisor's technique to support even more(and larger) models on a GPU.

Genetic algorithm for computer systems Genetic algorithm has been used to schedule tasks on parallel or distributed systems [117, 118, 43, 119, 120, 121]. SwapAdvisor borrows several ideas from the existing work, e.g., how to cross over schedules. However, the setting of SwapAdvisor is different. The existing work is designed for a multi-cores or multi-machines system where a task can be scheduled on a different core or machine. On the other hand, all of the computation tasks in SwapAdvisor are executed on the same GPU. Consequently, only the execution order matters for SwapAdvisor, resulting in a different crossover and mutation.

Some research discuss applying genetic algorithm to allocate data objects in a heterogeneous memory system [122, 123]. The work uses the genetic algorithm to decide how to allocate data objects on different memory types (e.g., SRAM

and DRAM). On the other hand, the memory allocation of SwapAdvisor also decides how many memory pools before allocating memory objects for a pool.

Chapter 7

Conclusion

In this thesis, we demonstrated two systems to help program large-scale machine learning and deep learning programs. Both designs leverage the information exposed by the application's dataflow graphs to control how the program to execute in order to improve the overall performance.

Spartan is a distributed array framework designed for traditional machine learning applications with a smart tiling algorithm to partition distributed arrays effectively. A set of carefully chosen high-level operators expose well-defined communication cost and simplify the tiling process. User array code is captured by the frontend and turned into an expression graph whose nodes correspond to these high-level operators. With the expression graph, our smart tiling can estimate the communication cost across expressions and find good tilings for all the expressions.

SwapAdvisor is designed to enable DNN training and inference with limited GPU memory size. SwapAdvisor achieves the good performance via optimiz-

ing three dimensions of a program, scheduling, memory allocation, and swap planning. To simultaneously optimize scheduling and memory allocation, SwapAdvisor adopts the genetic algorithm to search a good combination. For a given schedule and memory allocation, SwapAdvisor's swap planner is able to determine what and when tensors to swap to maximize the overlapping of the computation and communication.

Appendix A

NP-Completeness Proof of Tiling Optimizaion

This chapter proves that the general tiling optimization is an NP-Complete problem.

Problem Definition: To simplify the proof, we first, consider only *newarray*, *map* and *swapaxis* operators. The general case is discussed in the last part of the proof. This problem contains several operators in a program and each one can be the input of others. The first step is to build an expression graph for this problem as shown in Section 2.3.3. Next is to convert the expression graph to the tiling graph. We define a *tiling graph* as following:

1. A node group represents an operator and contains several partition nodes.

2. If an operator A is an input of an operator B in the expression graph, there

are some edges between node group A and group B in the tiling graph. How node group A connects to node group B depends on the type of operator B.

3. The cost of an edge $A.tiling_I \rightarrow B.tiling_K$ is the network transmission cost to do operator B when A is tiled as $tiling_I$ and B is tiled as $tiling_K$.

Figure A.1 shows three operators that will be used in the proof. There are two kinds of tilings, row and column, for each operator. There is no input for a *newarray*. As for *map*, there is at least one input array. The tiling nodes of an input node group are fully connected to the tiling nodes of *map*. If two tiling nodes represent the same tilings, there is no cost for the edge between them. Otherwise, the cost is the size of the array, N. The last operator is *swapaxis*. There is one input array for *swapaxis* and each tiling node of the input array connects to the tiling node of *swapaxis* representing the swapped tiling. The cost for both edges are zero.

The problem is to choose a unique tiling node for each node group without conflict and achieve the minimum overall cost (summation of cost of all edges adjacent to two chosen tiling nodes). Conflict means that if there are edges between node group A and node group B, the chosen nodes must bear the same relationship. For example, if the chosen tiling node for the input of *swapaxis* means row tiling, the chosen tiling node for *swapaxis* can only be column tiling to avoid conflict.

Instead of directly proving the problem, we prove the corresponding verify problem which is to find out if there is a choice with the cost less than or equal

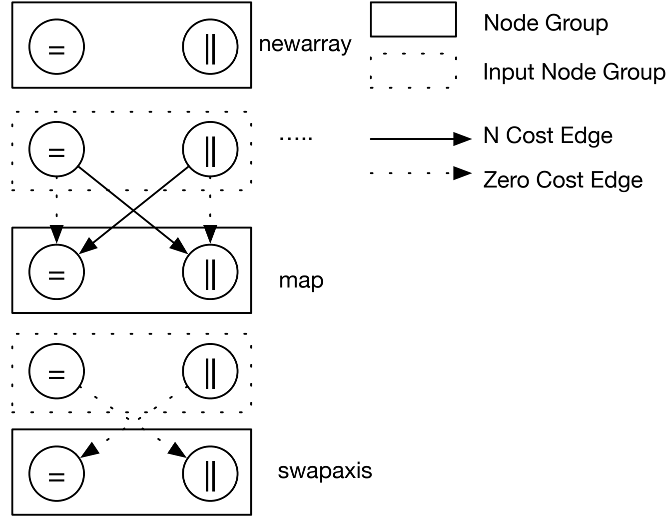


Figure A.1: Three node groups and edge relationship with their input(s).

to K where K is an integer. We denote the verify problem as $TILING(K)$.

NP Proof: To show that $TILING$ is in NP, we need to prove that a given choice can be verified in polynomial time. Suppose N is the number of node groups. Given a solution, we can verify the solution by adding up the cost for all edges connected to each chosen tiling node. There are at most $N-1$ edges connected to a tiling node and N chosen tiling nodes, we can get the total cost in $O(n^2)$. Therefore, $TILING(K)$ is in NP.

NP-Completeness Proof: To show $TILING(K)$ is NP-Complete, we prove that $NAE-3SAT(N)$ can be reduced to $TILING(K)$. $NAE-3SAT$ is similar to $3SAT$ except that each clause must have at least one $true$ and one $false$. Therefore, it rules out TTT and FFF while $3SAT$ only excludes FFF.

Assume that there are N literals and M clauses in the given question. M is

polynomial to N. We prove that $NAE{-}3SAT(N)$ can be reduced to $TILING(K)$ where $K = M * 2$.

1. **Construction Function,** $C(I)$:

   a) For $C(I)$, $True$ is viewed as row tiling and $false$ is viewed as column tiling.

   b) Each literal in $NAE-3SAT$ is an $array$ in $TILING(K)$. A negation literal is viewed as a $swapaxis$ of the original $array$.

   c) For each clause $c_i = (L_1 \vee L_2 \vee L_3)$, $C(I)$ creates six expressions:

$$E_1 = map(swapaxis(L_1, 0, 1), L_2)$$
$$E_2 = map(swapaxis(L_1, 0, 1), L_3)$$
$$E_3 = map(swapaxis(L_2, 0, 1), L_1)$$
$$E_4 = map(swapaxis(L_2, 0, 1), L_3)$$
$$E_5 = map(swapaxis(L_3, 0, 1), L_1)$$
$$E_6 = map(swapaxis(L_3, 0, 1), L_2)$$

   For a negation literal, L, $swapaxis(L)$ represent the original $array$.

For example, $C(I)$ creates six expressions for $c_j = (\neg L_1 \vee L_2 \vee L_3)$:

$$E_1 = map(L_1, L_2)$$

$$E_2 = map(L_1, L_3)$$

$$E_3 = map(swapaxis(L_2, 0, 1), swapaxis(L_1, 0, 1))$$

$$E_4 = map(swapaxis(L_2, 0, 1), L_3)$$

$$E_5 = map(swapaxis(L_3, 0, 1), swapaxis(L_1, 0, 1))$$

$$E_6 = map(swapaxis(L_3, 0, 1), L_2)$$

For explanation purpose, we call the six expressions created by $C(I)$ a clause group.

d) After converting all clauses to clause groups, $C(I)$ create a cost graph according to the definition. Without loss of generality, we assume that the array size is 1. Therefore, the cost for an edge is either 0 or 1.

For a clause group, if three literal have the same symbols, *true* or *false*, the minimum cost is 6. For example, if three literals are all *true* or all *false* for $c_i = (L_1 \vee L_2 \vee L_3)$, the two inputs for each *map* of the clause group must have different tilings because of *swapaxis*. Thus the cost for *map* node group can only be 1. Since there are six *map*s for a clause group, the minimum cost is 6.

For other cases, the minimum cost of a clause group is 2. For example, if L_1 is the only *true* for $c_i = (L_1 \vee L_2 \vee L_3)$, only the input tilings of *map*s for $E4$ and $E6$ are different. Since all *map*s are not referenced by other

operators, we can freely choose their tilings based only on the input tilings. Thus the cost for this case is 2. Other combinations are just symmetries of the above case and have the same cost.

The time complexity for $C(I)$ is $O(N^2)$.

2. $C(B)$ **belongs to** $TILING(K)$ **if** B **belongs to** $TILING(K)$:

   If S is a solution for B, every clause in S has at least one $true$ and one $false$. This implies that at least one row tiling input and column tiling input for each clause group of $C(S)$. Therefore, the cost for $C(S)$ is $M * 2$ which is equal to K.

3. B **belongs to** $NAE-3AT$ **if** $C(B)$ **belongs to** $TILING(K)$:

   If S is a solution for $C(B)$, there are at least one row tiling and one column tiling for each clause group. In other words, if one clause group has all row tiling inputs or all column tiling inputs, the total cost for the tiling graph will be at least $2*(M-1)+6 > K$. As a result, no clause group has all row tiling or column tiling input. Therefore, S is a solution for B.

Step 2 and step 3 prove that $NAE-3SAT$ can be reduced to $TILING(K)$.

General Graph: The previous proof only considers the tiling graph with $Array$, map and $swapaxis$. However, we argue that even though the tiling graph contains more different operators, it is still an NP-Complete problem to find out the solution. For any $TILING(K)$ which contains only the three operators, we add some other operators and expression which are independent from the original ones. Thus the new tiling graph contains two sub tiling graphs, the original tiling

graph and the tiling graph representing the newly added operators. Moreover, two sub tiling graphs are not connected. Thus, to solve new $TILING(K')$ must first solve the $TILING(K)$ which is NP-Complete. Thus, we can also reduce $TILING(K)$ to the general graph.

Bibliography

[1] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

[2] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553), 2015.

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

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

[5] Evan R. Sparks, Ameet Talwalkar, Virginia Smith, Jey Kottalama, Xinghao Pan, Joseph Gonzaleza, Michael Franklin, Michael Jordana, and Tim Kraskab. MLI: An API for distributed machine learning. In *arXiv:1310.5426*, 2013.

[6] Mahout: Scalable machine learning and data mining, 2012. http://mahout.apache.org.

[7] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In

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

[8] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

[9] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[10] Apache hadoop. http://hadoop.apache.org.

[11] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, 2003.

[12] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST'10, Washington, DC, USA, 2010. IEEE Computer Society.

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

[14] Russell Power and Jinyang Li. Piccolo: Building fast, distributed programs with partitioned tables. In *Symposium on Operating System Design and Implementation (OSDI)*, 2010.

[15] Christopher J Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: a compiler and runtime for heterogeneous

systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[16] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (OSDI)*, 2013.

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

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

[19] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Ulfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Symposium on Operating System Design and Implementation (OSDI)*, 2008.

[20] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2), March 2008.

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

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

[23] T Oliphant et al. NumPy, a Python library for numerical computations.

[24] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. vdnn: Virtualized deep neural networks for scalable,

memory-efficient neural network design. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 2016.

[25] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: Dynamic gpu memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, 2018.

[26] Tung D Le, Haruki Imai, Yasushi Negishi, and Kiyokuni Kawachiya. Tflms: Large model support in tensorflow by graph rewriting. *arXiv preprint arXiv:1807.02037*, 2018.

[27] Chen Meng, Minmin Sun, Jun Yang, Minghui Qiu, and Yang Gu. Training deeper models by gpu memory optimization on tensorflow. In *Machine Learning Systems Workshop (LearningSys) in NIPS*, 2017.

[28] PCI-SIG. Pci express base specification revision 5.0, 2019.

[29] NVIDIA. Nvidia nvlink high-speed interconnect, 2019.

[30] Yuan Yu, Martín Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, et al. Dynamic control flow in large-scale machine learning. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 2018.

[31] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. Flumejava: Easy, efficient data-parallel pipelines. In *PLDI - ACM SIGPLAN 2010*, 2010.

[32] Dataflow program graphs. *IEEE Computer*, 15, 1982.

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

[34] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3), 1979.

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

[36] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[37] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018.

[38] Lawrence Davis. Handbook of genetic algorithms. 1991.

[39] David E Goldberg and John H Holland. Genetic algorithms and machine learning. *Machine learning*, 3(2), 1988.

[40] John Henry Holland et al. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[41] Kim-Fung Man, Kit Sang Tang, and Sam Kwong. *Genetic algorithms: concepts and designs*. Springer Science & Business Media, 2001.

[42] Colin Reeves and Jonathan E Rowe. *Genetic algorithms: principles and perspectives: a guide to GA theory*, volume 20. Springer Science & Business Media, 2002.

[43] Oliver Sinnen. *Task scheduling for parallel systems*, volume 60. John Wiley & Sons, 2007.

[44] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.*, 5(2), June 1966.

[45] Eric W. Weisstein. Bell number. From MathWorld–A Wolfram Web Resource.

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

[47] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8), 1997.

[48] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11), 1997.

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

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

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

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

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

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

[55] Paul Anderson. Software engineering technology the use and limitations of static-analysis tools to improve software quality, 2008.

[56] Michael Chu, Rajiv Ravindran, and Scott Mahlke. Data access partitioning for fine-grain parallelism on multicore architectures. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*. IEEE, 2007.

[57] Eunjung Park, Christos Kartsaklis, Tomislav Janjusic, and John Cavazos. Trace-driven memory access pattern recognition in computational kernels. In *Proceedings of the Second Workshop on Optimizing Stencil Computations*. ACM, 2014.

[58] Jiahua He, Allan E Snavely, Rob F Van der Wijngaart, and Michael A Frumkin. Automatic recognition of performance idioms in scientific applications. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 2011.

[59] Christos Kartsaklis Oscar Hernandez. Open64-based regular stencil shape recognition in hercules. 2013.

[60] Christoph W Kessler. Pattern-driven automatic parallelization. *Scientific Programming*, 5(3), 1996.

[61] Tom Henretty, Kevin Stock, Louis-Noël Pouchet, Franz Franchetti, J Ramanujam, and P Sadayappan. Data layout transformation for stencil computations on short-vector simd architectures. In *Compiler Construction*. Springer, 2011.

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

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

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

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

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

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

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

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

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

[71] J McGraw, S Skedzielewski, S Allan, Rob Oldehoeft, John Glauert, C Kirkham, B Noyce, and R Thomas. *SISAL: streams and iteration in a single assignment language. Language Reference Manual.* 1985.

[72] Guy E Blelloch. NESL: A nested data-parallel language.(version 3.1). Technical report, DTIC Document, 1995.

[73] Ron Choy, Alan Edelman, and Cleve Moler Of. Parallel matlab: Doing it right. *Proceedings of the IEEE*, 93, 2005.

[74] J-L Gaudiot, Wim Bohm, Walid Najjar, Tom DeBoni, John Feo, and Patrick Miller. The sisal model of functional programming and its implementation. In *Parallel Algorithms/Architecture Synthesis, 1997. Proceedings. Second Aizu International Symposium*. IEEE, 1997.

[75] Derek G Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. Ciel: a universal execution engine for distributed data-flow computing. NSDI, 2011.

[76] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2012.

[77] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD '10: Proceedings of the 2010 international conference on Management of data*, New York, NY, USA, 2010.

[78] Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: Sql and rich analytics at scale. In *SIGMOD*, 2013.

[79] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive analysis of web-scale datasets. In *VLDB*, 2010.

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

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

[82] Sparkr: R frontend for spark. http://amplab-extras.github.io/SparkR-pkg.

[83] Julia language. http://julialang.org.

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

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

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

[87] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Karl Rupp, Barry F. Smith, and Hong Zhang. PETSc users manual. Technical report, Argonne National Laboratory, 2014.

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

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

[90] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17, 1998.

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

[92] Michael Stonebraker, Paul Brown, Jacek Becla, and D Zhang. Scidb: A new dbms for science and other applications with complex analytics. 2013.

[93] Peter Baumann, Andreas Dehmel, Paula Furtado, Roland Ritsch, and Norbert Widmann. The multidimensional database system RasDaMan. In *ACM SIGMOD Record*, volume 27. ACM, 1998.

[94] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *International Conference on Machine Learning*, 2015.

[95] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Proteus: Exploiting numerical precision variability in deep neural networks. In *Proceedings of the 2016 International Conference on Supercomputing*. ACM, 2016.

[96] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems*, 2015.

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

[98] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: efficient inference engine on compressed deep neural network. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016.

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

[100] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems*, 2015.

[101] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Advances in Neural Information Processing Systems*, 2016.

[102] Lin Ning and Xipeng Shen. Deep reuse: streamline cnn inference on the fly via coarse-grained computation reuse. In *Proceedings of the ACM International Conference on Supercomputing*. ACM, 2019.

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

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

[105] James Martens and Ilya Sutskever. Training deep and recurrent networks with hessian-free optimization. In *Neural Networks: Tricks of the Trade*. Springer, 2012.

[106] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R Ganger, Phillip B Gibbons, et al. Exploiting bounded staleness to speed up big data analytics. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014.

[107] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

[108] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R Ganger, Phillip B Gibbons, Garth A Gibson, and Eric P Xing. Managed communication and consistency for fast data-parallel iterative analytics. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 2015.

[109] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, 2012.

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

[111] Zhihao Jia, Sina Lin, Charles R Qi, and Alex Aiken. Exploring hidden dimensions in parallelizing convolutional neural networks. In *International Conference on Machine Learning*, 2018.

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

[113] Minjie Wang, Chien-chin Huang, and Jinyang Li. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*. ACM, 2019.

[114] NVIDIA. Nvidia tensorrt, 2018.

[115] NVIDIA. Cuda multi-process service, 2019.

[116] Peifeng Yu and Mosharaf Chowdhury. Salus: Fine-grained gpu sharing primitives for deep learning applications. *arXiv preprint arXiv:1902.04610*, 2019.

[117] Edwin SH Hou, Nirwan Ansari, and Hong Ren. A genetic algorithm for multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 5(2), 1994.

[118] Harmel Singh and Abdou Youssef. Mapping and scheduling heterogeneous task graphs using genetic algorithms. In *5th IEEE Heterogeneous Computing Workshop (HCW'96)*, 1996.

[119] Lee Wang, Howard Jay Siegel, Vwani P Roychowdhury, and Anthony A Maciejewski. Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. *Journal of Parallel and Distributed Computing*, 47(1), 1997.

[120] Sung-Ho Woo, Sung-Bong Yang, Shin-Dug Kim, and Tack-Don Han. Task scheduling in distributed computing systems with a genetic algorithm. In *Proceedings High Performance Computing on the Information Superhighway. HPC Asia'97*. IEEE, 1997.

[121] Annie S Wu, Han Yu, Shiyuan Jin, K-C Lin, and Guy Schiavone. An incremental genetic algorithm approach to multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 15(9), 2004.

[122] Keke Gai, Meikang Qiu, and Hui Zhao. Cost-aware multimedia data allocation for heterogeneous memory using genetic algorithm in cloud computing. *IEEE Transactions on Cloud Computing*, 2016.

[123] Meikang Qiu, Zhi Chen, Jianwei Niu, Ziliang Zong, Gang Quan, Xiao Qin, and Laurence T Yang. Data allocation for hybrid memory with genetic

algorithm. *IEEE Transactions on Emerging Topics in Computing*, 3(4), 2015.