Benchmarking, Profiling and White-Box Performance Modeling for DNN Training

by

Hongyu Zhu

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy

Department of Department of Computer Science
University of Toronto

Benchmarking, Profiling and White-Box Performance Modeling for DNN Training

Hongyu Zhu
Doctor of Philosophy

Department of Department of Computer Science
University of Toronto
2022

Abstract

Recent years have witnessed the co-evolution of deep neural network (DNN) algorithms and the underlying hardware and software design. Despite that system researchers proposed a variety of optimization techniques to improve DNN training efficiency, most techniques are under-utilized in practice. The software/hardware deployments that ML programmers use in practice are widely diverse. Differences in algorithms, hardware features, or even software versions could all shift the performance bottlenecks, and hence the effective optimization techniques for the given deployments. Profiling is a technique that could help ML programmers to pinpoint ad-hoc performance bottlenecks, but it involves extensive amount of domain knowledge and engineering workloads (e.g. writing scripts, instrumenting frameworks, reading and analyzing hardware data or diagrams).

In this thesis, we explore the approach of performance modeling, which can effectively provide ML programmers with accurate insights about performance bottlenecks and optimizations. Our key insight is that the entire DNN training workload can be decomposed into atomic small tasks of heterogeneous hardware components, and performance bottlenecks could shift among different components. Hardware traces and counters from each components need to be collected and properly structured to build a complete profile. Using this approach, we enable ML practitioners to identify effective optimizations for any given software/hardware deployments. In summary, this thesis makes the following major contributions.

First, we propose a new benchmark suite for DNN training workloads, called `TBD`, that contains nine mainstream DNN models, covering six major DNN applications. We also propose a set of performance metrics that can directly indicate performance bottlenecks on CPU/GPU, and build a tool chain that exploits hardware profiling tools to extract these metrics, enabling end-to-end profiling for DNN training workloads.

Second, we propose a new performance predictor (*Daydream*), that could accurately estimate the efficacy of various optimization techniques for DNN training workloads. *Daydream* utilizes a dependency graph approach to model the hardware execution, which tracks the task dependencies at kernel level and addresses several unique challenges in the ML context.

Third, we propose two tensor compilers, Sokoban and Roller, that can generate tensor programs in seconds. Both compilers view the underlying architecture as a data processing pipeline, and the computation of a tensor operator as a combination of tiles. Sokoban utilizes an end-to-end cost model based on the tile-based pipeline execution model, which manages to accurately and efficiently estimate the runtime of each tiling schedule. Roller on the other hand, is a construction-based tensor compiler, which utilizes a recursive construction policy to find efficient tiling schedules. The policy is based on heuristics that the tile shapes of optimal kernels should balance the latency between compute and memory, and satisfy the alignment of hardware features such as memory transactions and warp size, which allows Roller to find the shape choices of efficient tiling schedules within one second.

Acknowledgements

First of all, I would like to thank my adviser, Gennady Pekhimenko, who always gives me support on my research throughout my PhD career. Gennady not only provides me with deep insights when I feel lost about how to progress next, but also detailed guidance on my presentation and writing skills. Gennady's empathy also helped me go through the tough time when I was struggling with my depression. Gennady's passion, sense of responsibility, self-discipline, and broad visions set up an example and show me what a true researcher ought to be.

I am grateful to Prof. Bianca Schroeder, who gave me the PhD offer to enter University of Toronto and was my former advisor prior to Gennady. At the start of my second year, I tried to search for a topic that could match our common interests, but Bianca encouraged me to become independent, self-determined. Knowing my interests, Bianca later introduced me to Gennady as our interests greatly match. I am grateful to Bianca who truely cares about the success of her students, and even gave me nice suggestions after I shifted to Gennady.

I am grateful to Amar Phanishayee for being my mentor in Microsoft Research (MSR). Amar not only monitors my research progress by providing insights and guidance, but also cares about my success in my research career even after my internship in MSR. I am also grateful to other group members in MSR, including Deepak Narayanan and Jorgen Thelin who gave me feedback on my presentation, and Shivaram Venkataraman who offered help when I had technical troubles.

I am grateful to Jilong Xue, Lingxiao Ma, and Fan Yang in Microsoft Research Asia (MSRA). They have always been strong supports for the Sokoban and Roller compiler projects, providing smart ideas when I got stuck with difficult issues.

I am grateful to Tom Glinos, who not only always helps us manage our cluster and warmly offer help on technical issues, but also shares with me a lot of interesting ideas.

I am grateful my supervisory committee, which includes both Gennady and Bianca, as well as Angela, for always giving me constructive feedback and warm encouragement on my progress and presentation skills.

I am grateful to EcoSystem members, including James Gleeson, Bojian Zheng, Geoffrey Yu, Pavel Klishin, Pavel Golikov, Andrew Pelegris, Serina Tan, Anand Jayarajan, Shang Wang, Qiongsi Wu, Jiacheng Yang, Hanjie Qiu, as well as new comers to the group for active brainstorming on my research topic, and creating a friendly environment that can tolerate my immatureness. I am grateful to Xin Li, Yubo Gao, Cong Wei for helping me update the TBD benchmarking.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In recent years, deep learning algorithms have achieved state-of-the-art results in many application domains. The success of deep learning is fueled by the ever-increasing amount of data and sizes of deep learning models. As a result, the amount of compute required to train a state-of-the-art deep neural network (DNN) is growing extremely fast. To sustain this trend, system researchers built a complicated software stack to utilize heterogeneous hardware components, including CPUs, accelerators, interconnects, etc. As the DNN models are quickly evolving, system developers built DNN frameworks (e.g., Theano [25], PyTorch [187], Tensorflow [2], MXNet [36], Keras [44], Chainer [230], CNTK [213], Caffe [109], Horovod [214], PaddlePaddle [146], etc) to enable fast and agile development of new DNN models for ML practitioners. Meanwhile, architecture developers are proposing new hardware accelerators for DNN computation (e.g., AMD GPU [14], Google TPU [117], Alibaba Hanguang [115], Baidu Kunlun [181], Graphcore IPU [105], Huawei Kirin [93], Amazon Inferentia [215], Habana Gaudi/Goya [150], etc.). This becomes a major driving force for system researchers to propose automatic tensor compilers.

Despite these advances, the benefits of many proposed optimization techniques are hard to exploit in reality. The software/hardware deployments used by ML practitioners are generally different. Small changes in DNN models, software/hardware configurations could completely alter the performance bottlenecks of the training system, and hence the efficacy of various techniques. Given a practical deployment, it usually requires non-trial amount of effort for profiling to identify the critical performance bottlenecks. Ad-hoc profiling tools/scripts are often not scalable enough against diversities on application and hardware. As a result, the performance improvement of many proposed DNN optimization techniques are greatly limited in practice.

1.1 Focus of this Dissertation: Optimizations for DNN Training Workloads

This dissertation focuses on performance optimization for DNN training workloads in practical deployments. System researchers have proposed various optimization techniques to optimize the DNN training workloads. We observe that the benefits of many proposed techniques are limited due to: (i) a strong bias towards optimizing image classification models; (ii) unstable efficacy due to diverse software/hardware deployments used by ML practitioners; (iii) extremely long compilation time for the tensor compilers.

We first propose a benchmark suite (called `TBD`, **T**raining **B**enchmark for **D**NNs) for DNN training workloads, which reflects the diversity of the DNN models. The `TBD` benchmark suite was proposed prior to the MLPerf [148], currently the most prestigious benchmark for DNN training. We also proposed a set of performance metrics and a toolchain to extract these performance metrics to enable end-to-end performance profiling. The performance bottlenecks can directly indicate potential performance bottlenecks in the workloads.

We identify the usual suspects of performance bottlenecks, and explore the effective techniques that optimize each of them respectively. We then propose a system called Daydream, which leverages a dependency graph analysis to capture the dependencies and parallel execution of CPU, GPU, and communication tasks. Such design enables ML practitioners to quickly explore the efficacy of optimization techniques under their own hardware/software deployments.

We observe that the root cause for extremely long per-operator compilation time is that prior tensor compilers treat the underlying hardware as a black-box kernel executor. We explore two different approaches to overcome this limitation by proposing an abstraction for modern AI accelerators.

We propose an execution model, called RATIONAL, to describe the data movement and compute during execution of tensor kernel programs. Based on RATIONAL, we develop a search-based tensor compiler, Sokoban, equipped with a cost model to accurately estimate the performance of potential kernel schedules. As a result, Sokoban is able to generate high-quality tensor programs within one minute.

Finally, we propose a novel construction-based tensor compiler, Roller, which is also based on dissecting and abstracting both the underlying accelerator architecture and the operator. Roller identifies several key requirements that an efficient DNN operator should satisfy, such as balanced compute and memory latency and alignment to memory transactions. Such design allows Roller to generator efficient tensor programs in only `seconds`.

Notice that ML workloads are essentially statistical. Besides the software/hardware performance, the mathematical semantics of the underlying operators could also greatly affect the model convergence. In this dissertation, we limit our focus of the performance modeling on the software/hardware behaviours, rather than the mathematical semantics and model convergence.

1.1.1 Lack of Benchmark Suite for DNN Training Workloads

Most system and architecture researchers use image classification models as their benchmark application, as it has been the most popular application domain. However, recent advances in the machine learning area demonstrate that DNN algorithms can achieve state-of-the-art results for a wide range of application domains (e.g. machine translation, object detection, speech recognition, etc.). The typical DNN structures for different application domains tend to be different as well. The optimization techniques that are designed for convolutional networks (CNNs) may not improve recurrent neural networks (RNNs).

1.1.2 Why Benefits of Many Proposed Optimization Techniques Are Limited

Training modern DNNs are usually extremely compute-intensive workloads. It usually requires multiple hardware accelerators for the full training time of a modern DNN model to be tolerable. Hence one common optimization goal is to reduce the communication overhead in a distributed configuration. During the execution of training, data needs to be moved across different types of interconnects, for example, PCIe channels when moving data across devices on the same CPU host, or InfiniBand when moving data across CPU hosts. The bandwidths of the communication channels, the amount of computation and transmitted data could all affect the significance of the communication bottlenecks, and hence the efficacy of reducing communication overhead.

Besides communication, the computation of DNN training is performed by launching kernel programs on hardware accelerators. Hence another optimization goal is to directly improve the runtime of specific kernel programs. The efficacy of such optimizations depends on not only the significance of the target kernels, but also the quality of existing kernel implementations. The runtime improvement will hence vary across different DNN models and software/hardware deployments.

We observe that most system optimizations are not publicly available in the mainstream software stack. Besides that the improvement could greatly vary across different deployments, there are several other reasons, as these optimizations could potentially: (i) have conflicts with others and cannot be applied at the same time; (ii) disrupt the abstractions of the existed software stack; (iii) involve hyper-parameters that are critical to the efficacy and require non-trivial effort of tuning; (iv) damage the training accuracy. As a result, ML developers often have to implement and debug specific optimizations themselves, which involves onerous engineering effort if developers want to enjoy the benefits.

1.1.3 Why Existing Tensor Compilers Suffer from Long Compilation Time

To fully utilize the compute power of GPUs, system developers carefully crafted kernel libraries (e.g. cuBLAS, cuDNN, NCCL, etc.), which provide high-performance kernel programs for various types of operators. As architecture developers propose new accelerators for DNN training, manually building and maintaining these libraries is becoming increasingly burdensome. Hence recently, system researchers proposed tensor compilers (e.g. AutoTVM [35], Ansor [265]) that are able to automatically generate kernel programs, quickly providing support for new hardware.

Despite these efforts, generating efficient DNN kernels remains challenging. First, it can easily cost hours to compile an efficient DNN kernel (§6.2). The search space defined by the code template is often very large, containing millions or even billionaire of configuration choices. Machine learning algorithms usually need thousands of search steps, each evaluated in the real accelerator, to find a performant kernel. Second, to make the situation worse, the defined search space usually exponentially increases with the operator size (i.e., tensor size), and hence the compilation time. This makes it hard to scale the ML-based search techniques to large operators, which are used extensively in pre-trained models [59, 31, 140], a clear trend in deep learning. Finally, for a kernel to run efficiently on a different accelerator, the compiler has to repeat the search process from scratch, even if the device is only slightly different in architecture from the previous one. This further limits the scalability of existing ML compilers. The excessively long compile time has become a major obstacle that slows down DNN model development cycles. Our own experience shows that tuning an end-to-end DNN model using state-of-the-art compilers [39, 265, 145] often requires days, if not weeks.

1.2 Related Work

Researchers have proposed various approaches to improve the DNN computation. These techniques target different aspect of the performance bottlenecks with different strategies, including (i) lowering communication overhead, (ii) reducing numerical precision, (iii) improving kernel implementation, (iv) reducing memory footprint, etc. In this section, we summarize these optimizations that are related to our research in this dissertation, as well as prior benchmark suites and profiling work for DNN computation. We also highlight prior profiling works in non-ML contexts which share the high-level insight and mechanisms with our work.

1.2.1 DNN Benchmarks

Prior to our TBD benchmark (Training Benchmark for DNNs), there exists only a handful of open-source benchmarks for DNN workloads [49, 46, 220, 267]. These workloads were usually proposed at relative early stages of DNN development, hence with heavy focuses on convolutional neural networks (CNNs) and the image classification application. Meanwhile, DeepBench [57] was proposed for the performance of individual operators (e.g. matrix multiplication) and low-level kernel programs. We aim to propose an open-source benchmark suite designed for training workloads, covering all the major development of DNN models and applications.

1.2.2 Vendor-Provided Tools for DNN Accelerators

Modern DNN training heavily relies on hardware accelerators (e.g. GPUs [172], TPUs [117], etc.). Hardware vendors provide profiling tools [171, 170, 16] that can expose hardware performance counters, including per-kernel start/end time, core utilization, memory throughput, cache miss rate, along with hundreds of other hardware counters. NVIDIA also provides CUPTI [174] APIs, enabling programmers to extract and manipulate these counters at runtime. Since these tools have no domain knowledge, it often requires expertise from both architecture and application sides to effectively utilize these counters and uncover optimization opportunities.

Recently NVIDIA and Google proposed ML-specific profiling tools (DLProf [56] and Cloud TPU Tools [232]). Beyond the traditional domain-agnostic low-level kernel statistics, these tools reveal extra counters that aim to promote the use of the hardware optimizations including tensor cores [147] and XLAs [227] in practice. Our proposed system exploits these vendor-provided tools, and achieves accurate performance estimation for a wider range of optimizations, that operate at higher-level abstraction of a DNN training system.

1.2.3 Framework Built-in Profilers

Mainstream DNN frameworks usually implement their own profiling tools by mainly inserting timestamps, which are able to reveal the performance hotspots during DNN training. For example, MXNet [36] and PyTorch [187] provide built-in profilers [104, 161] that illustrate per-layer or per-operator traces, as well as traces of network, IO, or host-device memory copies etc. However, without access to vendor-provided profiling tools, these profilers often omit crucial hardware counters, such as CPU runtime, achieved FLOPS and memory throughput for tensor programs, etc. This fundamentally limits the benefits of using these tools when profiling low-level kernel programs.

1.2.4 Optimizations for Distributed DNN Training

The cost of weight exchange in large scale DNN training workloads is a main source of performance bottlenecks. Researchers explores several techniques to mitigate the communication overhead through various strategies such as (i) compressing the exchanged data [243, 137, 10, 26, 223, 124], (ii) improving the overlap of computation and communication [261, 107, 85, 191, 219, 43], (iii) asynchronous/stale gradient descent [94, 132, 238, 77]. The efficacy of these approaches depends on not only the duration, but also the interleaving of communication and computation tasks. Meanwhile, the use of some of these techniques has conflicts with certain optimizations for computation (e.g. fusing weight update kernels) due to data dependencies.

1.2.5 Reducing Numerical Precision

Low precision training (mixed precision or quantization [153, 53, 241, 128, 151]) is one of the most straight-forward strategy to reduce the training time, as DNN training is extremely compute-intensive. The efficacy of these techniques could vary significantly for different software/hardware deployments, depending on how much overhead is caused by CPU host runtime, and how much speedup is brought to each kernel program. Such strategy could potentially damage the model convergence, which is hard to verify without running a substantial amount of training iterations, making it tricky to use in practice.

1.2.6 Tensor Compilers

As architecture researchers and engineers constantly propose new designs of DNN accelerators, system researchers propose tensor compilers, which can automatically compile the high-level domain-specific language (DSL) into low-level tensor programs. Existing tensor compilers [135, 160, 39, 125, 20, 234, 266, 265, 80] generally treat the hardware accelerator as a black-box executor for kernel programs with various schedules, which brings two fundamental limitations. First, the compilation process requires a substantial amount of trials to identify the best kernel schedules or learn a black-box cost model. Second, a generated tensor program may perform badly under slight changes of hardware conditions, as the performance of generated tensor programs are often sensitive to conditions the target hardware. To overcome these limits, we propose a new tensor compiler which exploits a white-box pipeline execution model. The proposed model accurately depicts the behaviours of data movement and computation during kernel execution, and can easily fit to mainstream DNN hardware structures.

1.2.7 What-if Analysis in Non-ML Contexts

Prior works have tried to explore what-if questions in other contexts through analyzing low-level traces. Curtsinger *et al.* (COZ [52]) proposes a technique to achieve the performance exploration by running performance simulation with certain functions being virtually speed-up, without the need to tracking dependencies across small functions. For data analytic frameworks, dependencies across various small tasks are necessary for performance estimation [179, 180]. For the what-if exploration problem in ML context, our design is similar to Ousterhout *et al.* [179] in the sense that we both rely on analysis over a dependency graph on low-level traces, as both systems involve collaboration of heterogeneous hardware components.

1.3 Thesis Statement: Accurate and Efficient Performance Analysis and Modeling for DNN Training Workloads

The ultimate goal of this dissertation is to effectively improve DNN training efficiency for practical deployments, which software/hardware configurations could be widely variant. To this end, our thesis is that:

> ***It is possible to design systematic approaches/abstractions that are both accurate enough and efficient enough to pinpoint the potential performance bottlenecks and depict the hardware behaviours, they can quickly identify effective optimizations for training performance under various software/hardware configurations, and hence could potentially become a vital abstraction to implement for future AI systems.***

This dissertation explores white-box performance modeling techniques and their potentials for DNN training workloads under three different scenarios: i) an end-to-end profiling toolchain that can pinpoint performance bottlenecks; ii) dependency graph analysis that enables what-if explorations under any potential system-level optimizations; iii) a data processing pipeline and a tile-based abstraction that depict the behaviours of memory and compute during the execution of tensor kernel programs.

Our proposed designs are based on accurate insights of potential performance bottlenecks under different abstraction levels: (i) in a high-level scenario where we dissect the execution of end-to-end training, we need to accurately pinpoint bottlenecks among the communication, the accelerator and the host runtime; (ii) in a low-level scenario where we dissect the execution of tensor kernel programs on accelerators, the performance bottlenecks could shift among the computation cores and any levels in the memory hierarchy. For each of the proposed design, we manage to achieve the following goals at the same time:

- First, our designs are *accurate*, meaning that they can precisely capture the hardware behaviours under various configurations.
- Second, our designs are *efficient*, meaning that they can deliver high profiling accuracy with low cost (i.e., no excessive simulation or real execution).

Both of these requirements are vitally important so that our designs can effectively help improve the training performance in practical deployments eventually.

1.4 Contributions

This dissertation makes the following contributions.

- We propose a new benchmark suite for DNN training workloads, called `TBD`, that contains nine mainstream DNN models, covering six major DNN applications. We also propose a set of performance metrics that can directly indicate performance bottlenecks on CPU/GPU, and build a tool chain that exploits hardware profiling tools to extract these metrics, enabling end-to-end profiling for DNN training workloads. We will introduce the details k load details in **Chapter §3**.

- We propose a new performance predictor (*Daydream*), that could accurately estimate the efficacy of various optimization techniques for DNN training workloads. *Daydream* constructs a dependency graph to model the hardware execution, which tracks the task dependencies at kernel level. It correlates the low-level kernel traces with high-level DNN topology knowledge, and provides simple primitives for users to model potential optimization techniques. **Chapter §4** describes the details of the design and implementation of *Daydream*.

- We introduce *Sokoban*, a search-based tensor compiler that can generate fast tensor programs within one minute. We introduce a multi-level tile-based abstraction, RATIONAL, to dissect the behaviors of hardware components such as memory during kernel execution. Sokoban exploits the *RATIONAL* execution model, defines a configuration space consisting of multi-level tiling configurations. It can accurately calculates the performance of a tiling configuration, and greatly reducing the number of trials during compilation. We will explain the details in **Chapter §5**.

- We introduce *Roller*, a construction-based tensor compiler that can generate fast tensor programs in *seconds*. At the core of Roller is rTile, a tile abstraction that encapsulates tensor shapes that align with the key features of the underlying accelerator, thus achieving efficient execution by limiting the shape choices. We will explain the details in **Chapter §6**.

Chapter 2

Challenges of Modeling DNN Training Workloads

A performance model essentially provides users or upstream software with performance statistics or hints about performance bottlenecks, which eventually enables effective improvement to the overall performance. It achieves this goal by gathering and organizing low-level hardware traces and counters which are usually intricate. We illustrate three fundamental requirements for analysis by performance modeling in ML contexts:

- *First*, training DNN workloads usually involves collaboration of heterogeneous hardware components, including compute cores, memory, interconnects, etc. A performance model should be able to inspect various components to capture potential major performance bottlenecks.
- *Second*, there are many state-of-the-art DNN models with distinct graph topology, and various system optimization that operate at different part in the system with the same optimization goal. There are often trade-offs involved, where fitting to all potential models under different deployments could extremely complicate the model design. An intricate performance model might require huge cost for the estimation itself, and overfit to a narrow range of software/hardware deployments, lowering its applicability.
- *Third*, DNN training systems involve a complicated software stack with multiple levels of abstractions (e.g. kernel programs, DNN operators/layers, graph representations, single-node and multi-node parallelism, etc.). The performance model needs to bridge the abstraction gap by organizing intricate low-level hardware counters or traces and rendering domain-specific high-level profiling results at the same time.

These requirements raise non-trivial challenges for performance modeling in DNN training workloads. In this chapter, we explain the details of these challenges and how we could potentially address each of them.

2.1 Complexity of Hardware

Training a modern DNN model requires collaboration of heterogeneous hardware components. $\text{BERT}_{\text{LARGE}}$ [59] for example, is originally trained on a cluster that consists of 16 Cloud TPU boards. Each Cloud TPU board contains four TPU chips, and each TPU chip contains two TPU cores. Data needs to be exchanged among different Cloud TPU boards, and different chips on the same Cloud TPU board, via different types of interconnects. Each TPU core is equipped with scalar, vector and matrix multiplication units, where each type of units performs distinct arithmetic. Besides the communication, runtime bottlenecks could alter between CPU hosts and TPU cores, and among different types of cores and memory bandwidth within one TPU core.

GPU is also the commonly-used power house for training large DNN models, which has a different architecture design compared against TPUs. For example, an NVIDIA V100 GPU is equipped with single-precision cores, hard-precision CUDA cores, special function units, tensor cores, etc. Its memory hierarchy contains a unified global memory, L2 cache and on-chip scratchpad memory (shared memory) and L1 cache. Each of these hardware components could potentially become performance bottlenecks during execution. A performance model should be able to accurately pinpoint when certain hardware component becomes the major bottleneck. Meanwhile, depending on the abstraction level that the profiling operates, the performance model might also need to capture the parallelism and dependencies across different hardware components.

2.2 Diversity of DNN Training Workloads and Optimizations

ML researchers designed various types of DNN models for different application domains. Most of the state-of-the-art DNN models share the same basic iterative back propagation algorithm, but the DNN model structure (i.e. the size and type of layers, depth of the network, etc.) could be widely different. These differences in the DNN algorithm might change the low-level tensor programs, the memory footprint, the ratio between computation and communication, and hence lead to different performance characteristics during runtime. As a result, the experience with one DNN training workload usually cannot be simply migrated to other models.

Another characteristic about DNN training workloads is that they are essentially statistic workloads rather than deterministic ones. In reality, ML practitioners usually need to balance among the training accuracy, the cost of the hardware, the training time, etc. Hence, apart from traditional optimizations that target the usual suspects of performance bottlenecks (e.g. communication, CPU, GPU, memory footprint, etc.), ML practitioners might also adopt some optimizations or DNN model variations that could change the algorithm and affect the accuracy. Given the diversity on both the workloads and the potential optimization techniques, designing a performance model requires a comprehensive understanding about the high-level applications: how can we design a performance model that can fit to various DNN models, and how can we potentially optimize detected performance issues.

2.3 "Abstraction Gap"

Training a modern DNN involves a complicated software stack, that allows ML developers to manipulate hardware without writing low-level programs. This stack involves multiple levels of abstractions, bridging the gap between high-level application programs provided by DNN frameworks (e.g. TensorFlow [2], PyTorch [187]), and the hardware components that is manipulated directly by low-level libraries (e.g. cuDNN [40], cuBLAS [164], NCCL [169]). One of the consequences of this "abstraction gap", is that it requires non-trivial amount of effort to utilize the hardware profiling tools.

Hardware vendors usually provide profiling tools (e.g. nvprof for NVIDIA GPUs, μProf for AMD GPUs, Cloud TPU Tools for Google TPUs etc.) which reveal performance counters for their specific accelerators. These tools can capture performance counters within their own scopes. Some of such counters, such as compute core utilization, memory throughput, are necessary for users to discover potential bottlenecks, and these hardware profiling tools are the only sources that could extract such information. The main limitation of these profiling results is that they are usually in the kernel-level abstraction (i.e. performance counters are per-kernel), making them hard to exploit.

Modern DNN frameworks (e.g. PyTorch [187], MXNet [36], TensorFlow [2]), on the other hand, usually provide their own built-in profiling tools. These tools are often designed to reveal performance details in the layer-wise abstraction (i.e. the runtime statistics for DNN operators and even layers). This layer-wise abstraction are intuitive for programmers to understand the "where time goes" problem, but hides important information about the parallel execution of the CPU functions, GPU kernels, and memory transfers.

In summary, the deep "abstraction gap" between application programs and the hardware, makes it often hard to infer how the changes in the high-level application programs could affect the hardware behaviors during runtime. Hence, a white-box performance model should be able to (i) utilize the low-level hardware counters and deliver accurate profiling results; (ii) generate intuitive insights to pinpoint performance bottlenecks and indicate useful optimizations. Designing the model at the right abstraction is the key to achieve these two goals at the same time, and it requires both knowledge from both the hardware and the application domains.

2.4 Summary of Our Proposal

In this dissertation, our goal is to develop effective performance modeling solutions to meet these requirements under multiple scenarios. All these solutions allow users or upstream software to easily pinpoint the performance bottlenecks and adopt effective optimizations.

First, we propose a benchmark (TBD) that covers major state-of-the-art DNN models. We then propose a set of performance metrics that can directly indicate the potential performance bottlenecks, and build a toolchain that exploits existing hardware profiling tools to perform an end-to-end profiling for DNN training workloads. The toolchain can illustrate the performance bottlenecks on both CPU and GPU, and dissect the memory footprint usage (Chapter 3).

Second, we design and implement a new system (Daydream), that allows ML practitioners to accurately estimate the efficacy of system-level optimizations on their specific hardware/software deployments, without the need for implementing the optimizations. Daydream exploits the dependency graph analysis approach that system researchers used for what-if exploration in non-ML contexts, and addresses several unique challenges to explore what-if questions in the ML context (Chapter 4).

Finally, we propose a tensor compiler (Roller), that adopts a fundamentally different approach comparing against prior arts [39, 265]. Roller is a construction-based policy, which is designed based on three key insights

Chapter 3

Benchmarking and Analyzing Deep Neural Network Training

3.1 Introduction

The availability of large datasets and powerful computing resources has enabled a new type of artificial neural networks—deep neural networks (DNNs [92, 24])—to solve hard problems such as image classification, machine translation, and speech processing [127, 88, 17, 91, 247, 235]. While this recent success of DNN-based learning algorithms has naturally attracted a lot of attention, the primary focus of researchers especially in the systems and computer architecture communities is usually on *inference*—i.e. how to efficiently execute already trained models, and *image classification* (which is used as the primary benchmark to evaluate DNN computation efficiency).

While inference is arguably an important problem, we observe that efficiently *training* new models is becoming equally important as machine learning is applied to an ever growing number of domains, e.g., speech recognition [17, 252], machine translation [21, 144, 224], automobile industry [28, 102], and recommendation systems [50, 90]. But researchers currently lack comprehensive benchmarks and profiling tools for DNN training. In this chapter, we present a new benchmark for DNN training, called `TBD`, that uses a representative set of DNN models covering a broad range of machine learning applications: image classification, machine translation, speech recognition, adversarial networks, reinforcement learning. `TBD` also incorporates an analysis toolchain for performing detailed resource and performance profiling of these models, including the first publicly available tool for profiling memory usage on major DNN frameworks. Using `TBD` we perform a detailed performance analysis on how these different applications behave on three DNN training frameworks (TensorFlow [2], MXNet [37], CNTK [257]) across different hardware configurations (single-GPU, multi-GPU, and multi-machine) gaining some interesting insights.

`TBD`'s benchmark suite and analysis toolchain is driven by the motivation to address three main challenges:

1. Training differs significantly from inference. The algorithmic differences between training and inference lead to many differences in requirements for the underlying systems and hardware architecture. First, *backward pass* and *weight updates*, operations unique to training, need to save/stash a large number of intermediate results in GPU memory, e.g., outputs of the inner layers called

	Image Classification Only	**Broader (include non-CNN workloads)**
Training	[42][54][60][97][118][123][207][222][236]	[2][29][108][133][186][190][249]
Inference	[8][9][12][33][41][60][62][65][118][136][142] [184][203][216][217][218][222][259][262]	[2][61][83][86][116][186]

Table 3.1: The table above shows a categorization of major computer architecture and systems conference papers (SOSP, OSDI, NSDI, MICRO, ISCA, HPCA, ASPLOS) since 2014. These papers are grouped by their focus along two dimensions: Training versus Inference and Algorithmic Breadth. There are more papers which optimize inference over training (25 vs. 16, 4 papers aim for both training and inference). Similarly more papers use image classification as the *only* application for evaluation (26 vs. 11).

feature maps or activations [207]. This puts significant pressure on the memory subsystem of modern DNN accelerators (usually GPUs) – in some cases the model might need tens of gigabytes of main memory [207]. In contrast, the memory footprint of inference is significantly smaller, in the order of tens of megabytes [81], and the major memory consumers are model weights rather than feature maps. Second, training usually proceeds in waves of *mini-batches*, a set of inputs grouped and processed in parallel [70, 255]. Mini-batching helps in avoiding both overfitting and under utilization of GPU's compute parallelism. Thus, throughput is the primary performance metric of concern in training. Compared to training, inference is computationally less taxing and is latency sensitive.

2. Workload diversity. Deep learning has achieved state-of-the-art results in a very broad range of application domains. Yet most existing evaluations of DNN performance remain narrowly focused on just image classification as their benchmark application, and convolutional neural networks (CNNs) remain the most widely-used models for systems/architecture researchers (Table 3.1). As a result, many important non-CNN models have not received much attention, with only a handful of papers evaluating non-CNNs such as recurrent neural networks [2, 116, 86]. Papers that cover unsupervised learning or deep reinforcement learning are extremely rare. The computational characteristics of image classification models are very different from these networks, thus motivating a need for a broader benchmark suite for DNN training. Furthermore, given the rapid pace of innovation across the realms of algorithms, systems, and hardware related to deep learning, such benchmarks risk being quickly obsolete if they don't change with time.

3. Identifying bottlenecks. It is not obvious which hardware resource is the critical bottleneck that typically limits training throughput, as there are multiple plausible candidates. Typical convolutional neural networks (CNNs) are usually computationally intensive, making *computation* one of the primary bottlenecks in single GPU training. Efficiently using modern GPUs (or other hardware accelerators) requires training with large mini-batch sizes. Unfortunately, as we will show later in Section §3.4.2, for some workloads (e.g., RNNs, LSTMs) this requirement can not be satisfied due to capacity limitations of GPU *main memory* (usually 8–16GBs). Training DNNs in a distributed environment with multiple GPUs and machines, brings with it yet another group of potential bottlenecks, *network and interconnect bandwidths*, as training requires fast communication between many CPUs and GPUs (see Section §3.4.5). Even for a specific model, implementation and hardware setup pinpointing whether performance is bounded by computation, memory, or communication is not easy due to limitations of existing profiling tools. Commonly used tools (e.g., vTune [202], nvprof [171], etc.) have no domain-specific knowledge about the algorithm logic, can only capture low-level information within their own scopes, and usually cannot perform analysis on full application executions with huge working set sizes. Furthermore, no tools for memory profiling are currently available for any major DNN framework.

This chapter makes the following contributions.

- **TBD, a new benchmark suite.** We create a new benchmark suite for DNN training that currently covers *six* major application domains and *eight* different state-of-the-art models. The applications in this suite are selected based on extensive conversations with ML developers and users from both industry and academia. For all application domains we select recent models capable of delivering state-of-the-art results. We will open-source our benchmarks suite later this year and intend to continually expand it with new applications and models based on feedback and support from the community.

- **Tools to enable end-to-end performance analysis.** We develop a toolchain for end-to-end analysis of DNN training. To perform such analysis, we perform piecewise profiling by targeting specific parts of training using existing performance analysis tools, and then merge and analyze them using domain-specific knowledge of DNN training. As part of the toolchain we also built new memory profiling tools for the three major DNN frameworks we considered: TensorFlow [2], MXNet [37], and CNTK [257]. Our memory profilers can pinpoint how much memory is consumed by different data structures during training (weights, activations, gradients, workspace etc.), thus enabling developer to make easy data-driven decisions for memory optimizations.

- **Findings and Recommendations.** Using our benchmark suite and analysis tools, we make several important observations and recommendations on where the future research and optimization of DNNs should be focused. We include a few examples here: (1) We find that the training of state-of-the-art RNN models is not as efficient as for image classification models, because GPU utilization for RNN models is 2–3× lower than for most other benchmark models. (2) We find that GPU memory is often not utilized efficiently, the strategy of exhausting GPU memory capacity with large mini-batch provides limited benefits for a wide range of models. (3) We also find that the feature maps, the output of the DNN intermediate layers, consume 70–90% of the total memory footprint for all our benchmark models. This is a significant contrast to inference, where footprint is dominated by the weights. These observations suggest several interesting research directions, including efficient RNN layer implementations and memory footprint reduction optimizations with the focus on feature maps.

The TBD benchmark suite and the accompanying measurement toolchain, and insights derived from them will aid researchers and practitioners in computer systems, computer architecture, and machine learning to determine where to target their optimizations efforts within each level in the DNN training stack: (i) applications and their corresponding models, (ii) currently used libraries (e.g., cuDNN), and (iii) hardware that is used to train these models.

In the rest of this chapter, we first provide some background on DNN training, both single-GPU and distributed training (with multiple GPUs and multiple machines) in Section 2. We then present our methodology, explaining which DNN models we selected to be included in our benchmark suite and why, and describing our measurement framework and tools to analyze the performance of these models (Section §3.3). We use our benchmark and measurement framework to derive observations and insights about these models' performance and resource characteristics in Section §3.4. We then show our update on the benchmarks due to recent development on the DNN algorithm and impact to the community in Section §3.5. We conclude the chapter with a description of related work in Section §3.6 and a summary of our work in Section §3.8.

3.2 Background

3.2.1 Deep Neural Network Training and Inference

A neural network can be seen as a function which takes data samples as inputs, and outputs certain properties of the input samples (Figure 3.1). Neural networks are made up of a series of layers of neurons. Neurons across layers are connected, and layers can be of different types such as fully-connected, convolutional, pooling, recurrent, etc. While the edges connecting neurons across layers are weighted, each layer can be considered to have its own set of *weights*. Each layer applies a mathematical transformation to its input. For example, a fully-connected layer multiplies intermediate results computed by its preceding/upstream layer (input) by its weight matrix, adds a bias vector, and applies a non-linear function (e.g., sigmoid) to the result; this result is then used as the input to its following/downstream layer. The intermediate results generated by each layer are often called *feature maps*. Feature maps closer to the output layer generally represent higher order features of the data samples. This entire layer-wise computation procedure from input data samples to output is called *inference*.

A neural network needs to be trained before it can detect meaningful properties corresponding to input data samples. The goal of *training* is to find proper weight values for each layer so that the network as a whole can produce desired outputs. Training a neural network is an iterative algorithm, where each iteration consists of a *forward* pass and a *backward* pass. The forward pass is computationally similar to inference. For a network that is not fully trained, the inference results might be very different from ground truths labels. A *loss function* measures the difference between the predicted value in the forward pass and the ground truth. Similar to the forward pass, computation in the backward pass also proceeds layer-wise, but in an opposite direction. Each layer uses errors from its downstream layers and feature maps generated in the forward pass to compute not only errors to its upstream layers according to the chain rule [208] but also gradients of its internal weights. The gradients are then used for updating the weights. This process is known as the *gradient descent* algorithm, used widely to train neural networks.

As modern training dataset are extremely large, it is expensive to use the entire set of the training data in each iteration. Instead, a training iteration randomly samples a *mini-batch* from the training data, and uses this mini-batch as input. The randomly sampled mini-batch is a stochastic approximation to the full batch. This algorithm is called *stochastic gradient descent* (SGD) [130]. The size of the mini-batch is a crucial parameter which greatly affects both the training performance and the memory footprint.

3.2.2 GPUs and Distributed Training via Data Parallelism

While the theoretical foundations of neural networks have a long history, it is only relatively recently the people realized the power of deep neural networks. This is because to fully train a neural network

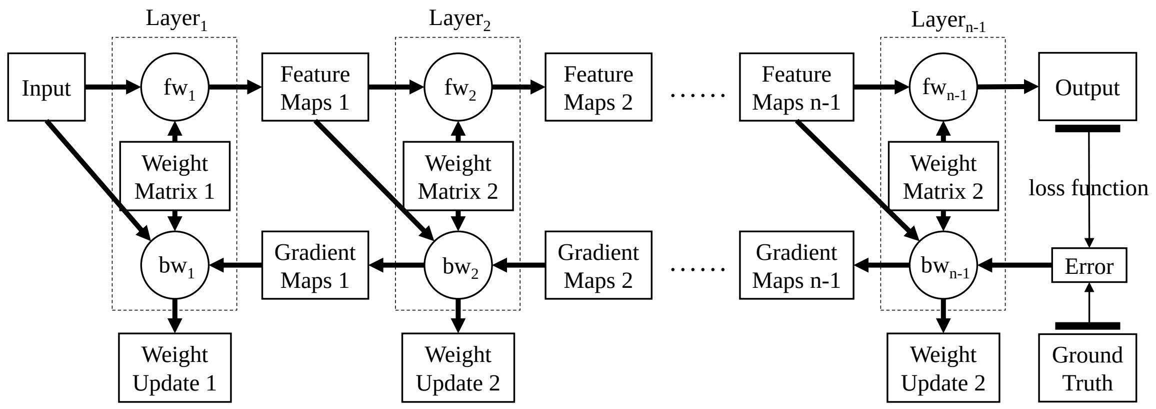


Figure 3.1: Feed-forward and Back-propagation

on a CPU is extremely time-consuming [220]. The first successful deep neural network [127] that beat all competitors in image classification task in 2012, was trained using two GTX 580 GPUs [177] in six days instead of months of training on CPUs. One factor that greatly limits the size of the network is the amount of tolerable training time. Since then, almost all advanced deep learning models are trained using either GPUs or some other type of hardware accelerators [116, 72].

One way to further speed up the neural network training is to parallelize the training procedure and deploy the parallelized procedure in a distributed environment. A simple and effective way to do so is called *data parallelism* [55]. It lets each worker train a single network replica. In an iteration, the input mini-batch is partitioned into n subsets, one for each worker. Each worker then takes this subset of the mini-batch, performs the forward and backward passes respectively, and exchanges weight updates with all other workers.

Another way to parallelize the computation is by using *model parallelism* [240], an approach used when the model's working set is too large to fit in the memory of a single worker. Model parallel training splits the workload of training a complete model across the workers; each worker trains only a part of the network. This approach requires careful workload partitioning to achieve even load-balancing and low communication overheads. The quality of workload partitioning in model parallelism depends highly on DNN architecture. Unlike model parallelism, data parallelism is simpler to get right and is the predominant method of parallel training. In this chapter we limit our attention to data parallel distributed training.

3.2.3 DNN Frameworks and Low-level Libraries

DNN frameworks and low-level libraries are designed to simplify the life of ML programmers and to help them to efficiently utilize existing complex hardware. A DNN framework (e.g., TensorFlow or MXNet) usually provides users with compact numpy/matlab-like matrix APIs to define the computation logic, or a configuration format, that helps ML programmers to specify the topology of their DNNs layer-by-layer. The programming APIs are usually bounded with the popular high-level programming languages such as Python, Scala, and R. A framework transforms the user program or configuration file into an internal intermediate representation (e.g., dataflow graph representation [2, 37, 25]), which is a basis for backend execution including data transfers, memory allocations,

Application	Model	Number of Layers	Dominant Layer	Framework	Dataset
Image classification	ResNet-50 [127] Inception-v3 [225]	50 (152 max) 42	CONV	TF, MXNet, CNTK	ImageNet1K [209]
Machine translation	Seq2Seq [224] Transformer [235]	5 12	LSTM Attention	TF, MXNet TensorFlow	IWSLT15 [32]
Object detection	Faster R-CNN [204]	101[a]	CONV	TF, MXNet	Pascal VOC 2007 [64]
Speech recognition	Deep Speech 2 [17]	9[b]	RNN	MXNet	LibriSpeech [182]
Adversarial learning	WGAN [73]	14+14[c]	CONV	TF	Downsampled ImageNet [45]
Reinforcement learning	A3C [157]	4	CONV	MXNet	Atari 2600

Table 3.2: Overview of Benchmarks, including the models and datasets used, number and major layer types, and frameworks with available implementations.

Dataset	Number of Samples	Size	Special
ImageNet1K	1.2million	3x256x256 per image	N/A
IWSLT15	133k	20-30 words long per sentence	vocabulary size of 17188
Pascal VOC 2007	5011[d]	around 500x350	12608 annotated objects
LibriSpeech	280k	1000 hours[e]	N/A
Downsampled ImageNet	1.2million	3x64x64 per image	N/A
Atari 2600	N/A	4x84x84 per image	N/A

Table 3.3: Training Datasets

and low-level CPU function calls or GPU kernel[1] invocations. The invoked low-level functions are usually provided by libraries such as cuDNN [40], cuBLAS [164], MKL [237], and Eigen [63]. These libraries provide efficient implementations of basic vector and multi-dimension matrix operations (some operations are NN-specific such as convolutions or poolings) in C/C++ (for CPU) or CUDA (for GPU). The performance of these libraries will directly affect the overall training performance.

3.3 Methodology

3.3.1 Application and Model Selection

Based on a careful survey of existing literature and in-depth discussions with machine learning researchers and industry developers at several institutions (Google, Microsoft, and Nvidia) we identified a diverse set of interesting application domains, where deep learning has been emerging as the most promising solution: image classification, object detection, machine translation, speech recognition, generative adversarial nets, and deep reinforcement learning. While this is the set of applications we will include with the first release of our open-source benchmark suite, we expect to continuously expand it based on community feedback and contributions and to keep up with advances of deep learning in new application domains.

[1]A GPU kernel is a routine that is executed by an array of CUDA threads on GPU cores.

Table 3.2 summarizes the models and datasets we chose to represent the different application domains. When selecting the models, our emphasis has been on picking the most recent models capable of producing state-of-the-art results (rather than for example classical models of historical significance). The reasons are that these models are the most likely to serve as building blocks or inspiration for the development of future algorithms and also often use new types of layers, with new resource profiles, that are not present in older models. Moreover, the design of models is often constrained by hardware limitations, which will have changed since the introduction of older models.

Image Classification

Image classification is the archetypal deep learning application, as this was the first domain where a deep neural network (AlexNet [127]) proved to be a watershed, beating all prior traditional methods. In our work, we use two very recent models, Inception-v3 [225] and Resnet [88], which follow a structure similar to AlexNet's CNN model, but improve accuracy through novel algorithm techniques that enable extremely deep networks.

Object Detection

Object detection applications, such as face detection, are another popular deep learning application and can be thought of as an extension of image classification, where an algorithm usually first breaks down an image into regions of interest and then applies image classification to each region. We choose to include Faster R-CNN [204], which achieves state-of-the-art results on the Pascal VOC datasets [64]. A training iteration consists of the forward and backward passes of two networks (one for identifying regions and one for classification), weight sharing and local fine-tuning. The convolution stack in a Faster R-CNN network is usually a standard image classification network, in our work a 101-layer ResNet.

In the future, we plan to add YOLO9000 [201], a network recently proposed for the real-time detection of objects, to our benchmark suite. It can perform inference faster than Faster R-CNN, however at the point of writing its accuracy is still lagging and its implementations on the various frameworks is not quite mature enough yet.

Machine Translation

Unlike image processing, machine translation involves the analysis of sequential data and typically relies on RNNs using LSTM cells as its core algorithm. We select *NMT*[247] and *Sockeye*[91], developed by the *TensorFlow* and *Amazon Web Service* teams, respectively, as representative RNN-based models in this area. We also include an implementation of the recently introduced [235] *Transformer* model, which achieves a new state-of-the-art in translation quality using attention layers as an alternative to recurrent layers.

Speech Recognition

Deep Speech 2 [17] is an end-to-end speech recognition model from *Baidu Research*. It is able to accurately recognize both English and Mandarin Chinese, two very distant languages, with a unified model architecture and shows great potential for deployment in industry. The Deep Speech 2 model contains two convolutional layers, plus seven regular recurrent layers or Gate Recurrent

Units (GRUs), different from the RNN models in machine translation included in our benchmark suite, which use LSTM layers.

Generative Adversarial Networks

A generative adversarial network (GAN) trains two networks, one generator network and one discriminator network. The generator is trained to generate data samples that mimic the real samples, and the discriminator is trained to distinguish whether a data sample is genuine or synthesized. GANs are used, for example, to synthetically generate photographs that look at least superficially authentic to human observers.

While GANs are powerful generative models, training a GAN suffers from instability. The WGAN [18] is a milestone as it makes great progress towards stable training. Recently Gulrajani et al. [73] proposes an improvement based on the WGAN to enable stable training on a wide range of GAN architectures. We include this model into our benchmark suite as it is one of the leading DNN algorithms in the unsupervised learning area.

Deep Reinforcement Learning

Deep neural networks are also responsible for recent advances in reinforcement learning, which have contributed to the creation of the first artificial agents to achieve human-level performance across challenging domains, such as the game of Go and various classical computer games. We include the *A3C* algorithm [157] in our benchmark suite, as it has become one of the most popular deep reinforcement learning techniques, surpassing the DQN training algorithms [158], and works in both single and distributed machine settings. A3C relies on asynchronously updated policy and value function networks trained in parallel over several processing threads.

3.3.2 Framework Selection

There are many open-source DNN frameworks, such as TensorFlow [2], Theano [25], MXNet [37], CNTK [257], Caffe [109], Chainer [230], Torch [48], Keras [44], PyTorch [187]. As there is not one single framework that has emerged as the dominant leader in the field and different framework-specific design choices and optimizations might lead to different results, we include several frameworks in our work. In particular, we choose TensorFlow [2], MXNet [37], and CNTK [257], as all three platforms have a large number of active users, are actively evolving, have many of the implementations for the models we were interested in[2], and support hardware acceleration using single and multiple GPUs.

[a]We use the convolution stack of ResNet-101 to be the shared convolution stack between Region Proposal Network and the detection network.

[b]The official Deep Speech 2 model has 2 convolutional layers plus 7 RNN layers. Due to memory issue, we use the default MXNet configuration which has 5 RNN layers instead.

[c]The architecture for both the generator and discriminator of WGAN is a small CNN containing 4 residual blocks.

[d]We use the train+val set of Pascal VOC 2007 dataset.

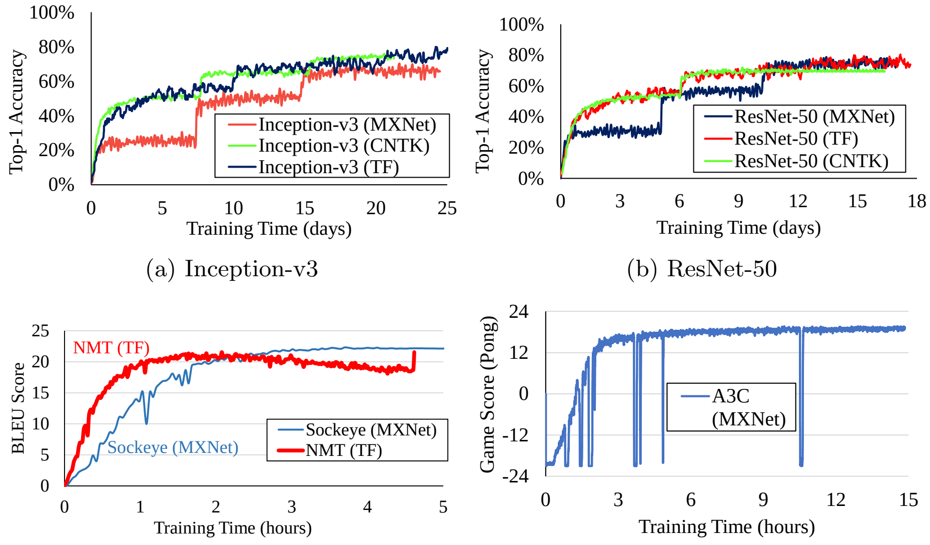
[e]The entire LibriSpeech dataset consists of 3 subsets with 100 hours, 360 hours and 500 hours respectively. By default, the MXNet implementation uses the 100-hour subset as the training dataset.

[2]Note that implementing a model on a new framework from scratch is a highly complex task beyond the scope of our work. Hence in this chapter we use the existing open-source implementations provided by either the framework developers on the official github repository, or third-party implementations when official versions are not available.

3.3.3 Training Benchmark Models

To ensure that the results we obtain from our measurements are representative we need to verify that the training process for each model results in classification accuracy comparable to state of the art results published in the literature. To achieve this, we train the benchmark models in our suite until they converge to some expected accuracy rate (based on results from the literature).

Figure 3.2 shows the classification accuracy observed over time for four representative models in our benchmark suite, *Inception-v3*, *ResNet-50*, *Seq2Seq*, and *A3C*, when trained on the single Quadro P4000 GPU hardware configuration described in Section §3.4. We observe that the training outcome of all models matches results in the literature. For the two image classification models (*Inception-v3* and *ResNet-50*) the Top-1 classification accuracy reaches 75–80% and the the Top-5[3] accuracy is above 90%, both in agreement with previously reported results for these models [88]. The accuracy of the machine translation models is measured using the BLEU score [183] metric, and we trained our model to achieve a BLEU score of around 20. For reinforcement learning, since the models are generally evaluated by Atari games, the accuracy of the A3C model is directly reflected by the score of the corresponding game. The A3C curve we show in this figure is from the Atari Pong game and matches previously reported results for that game (19–20) [157]. The training curve shape for different implementations of the same model on different frameworks can vary, but most of them usually converge to similar accuracy at the end of training.



(a) Inception-v3

(b) ResNet-50

(c) Seq2Seq

(d) A3C

Figure 3.2: The model accuracy during the training for different models.

3.3.4 Performance Analysis Framework and Tools

In this section we describe our analysis toolchain. This toolchain is designed to help us understand for each of the benchmarks, where the training time goes, how well the hardware resources are utilized and how to efficiently improve training performance.

[3]In the Top-5 classification the classifier can select up to 5 top prediction choices, rather than just 1.

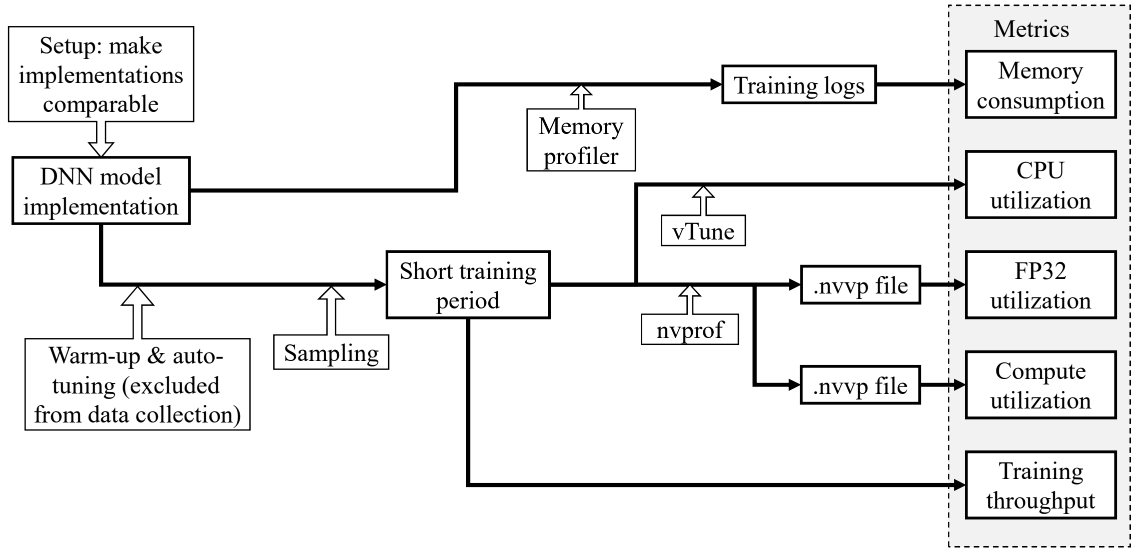


Figure 3.3: Performance Analysis Framework

Making implementations comparable across frameworks

Implementations of the same model on different frameworks might vary in a few aspects that can impact performance profiling results. For example, different implementations might have hard-coded values for key hyper-parameters (e.g., learning rate, momentum, dropout rate, weight decay) in their code. To make sure that benchmarking identifies model-specific performance characteristics, rather than just implementation-specific details, we first adapt implementations of the same model to make them comparable across platforms. Besides making sure that all implementations run using the same model hyper-parameters, we also ensure that they define the same network, i.e. the same types and sizes of corresponding layers and layers are connected in the same way. Moreover, we make sure that the key properties of the training algorithm are the same across implementations. This is important for models, such as Faster R-CNN [204], where there are four different ways in which the training algorithm can share the internal weights.

Accurate and time-efficient profiling via sampling

The training of a deep neural network can take days or even weeks making it impractical to profile the entire training process. Fortunately, as the training process is an iterative algorithm and almost all the iterations follow the same computation logic, we find that accurate results can be obtained via sampling only for a short training period (on the order of minutes) out of the full training run. In our experiments, we sample 50-1000 iterations and collect the metrics of interest based on these iterations.

To obtain representative results, care must be taken when choosing the sample interval to ensure that the training process has reached stable state. Upon startup, a typical training procedure first goes through a warm-up phase (initializing for example the data flow graph, allocating memory and loading data) and then spends some time auto-tuning various parameters (e.g., system hyper-parameters, such as matrix multiplication algorithms, workspace size). Only after that the system enters the stable training phase for the remainder of the execution. While systems do not explicitly indicate when they enter the stable training phase, our experiments show that the warm-up and auto-tuning phase can be easily identified in measurements. We see that throughput stabilizes after several hundred iterations (a few thousand iterations in the case of Faster R-CNN). The sample time interval is then chosen after throughput has stabilized.

Relevant metrics

Below we describe the metrics we collect as part of the profiling process.
• *Throughput:* Advances in deep neural networks have been tightly coupled to the availability of compute resources capable of efficiently processing large training data sets. As such, a key metric when evaluating training efficiency is the number of input data samples that is being processed per second. We refer to this metric as *throughput.* Throughput is particularly relevant in the case of DNN training, since training, unlike inference, is not latency sensitive.

For the speech recognition model we slightly modify our definition of throughput. Due to the large variations in lengths among the audio data samples, we use the total duration of audio files processed per second instead of the number of files. The lengths of data samples also varies for machine translation models, but the throughput of these models is still stable so we use the throughput determined by simple counting for them.
•*GPU Compute Utilization:* The GPU is the workhorse behind DNN training, as it is the unit responsible for executing the key operations involved in DNN training (broken down into basic operations such as vector and matrix operations). Therefore, for optimal throughput, the GPU should be busy all the time. Low utilization indicates that throughput is limited by other resources, such as CPU or data communication, and further improvement can be achieved by overlapping CPU runtime or data communication with GPU execution.

We define GPU Compute Utilization as the fraction of time that the GPU is busy (i.e. at least one of its typically many cores is active):

$$\text{GPU utilization} = \frac{\text{GPU active time} \times 100}{\text{total elapsed time}}\% \tag{3.1}$$

• *FP32 utilization:* We also look at GPU utilization from a different angle, measuring how effectively the GPU's resources are being utilized *while the GPU is active.* More specifically, the training of DNNs is typically performed using single-precision floating point operations (FP32), so a key metric is how well the GPU's compute potential for doing floating point operations is utilized. We compare the number of FP32 instructions the GPU actually executes while it is active to the maximal number of FP32 instructions it can theoretically execute during this time, to determine what percentage of its floating point capacity is utilized. More precisely, if a GPU's theoretical peak capacity across all its cores is $FLOPS_{peak}$ single-precision floating point operations per second, we observe the actual number of floating point operations executed during a period of T seconds that the GPU is active, to compute *FP32 utilization* as follows:

$$\text{FP32 utilization} = \frac{\text{actual flop count during T} \times 100}{FLOPS_{peak} \times T}\% \tag{3.2}$$

The FP32 utilization gives us a way to calculate the theoretical upper bound of performance improvements one could achieve by a better implementation. For example, an FP32 utilization of

50% indicates that we can increase throughput by up to 2x if we manage to increase the FP32 utilization up to 100%.

In addition to looking at the aggregate FP32 utilization across all cores, we also measure the per-core FP32 utilization for individual kernels, to identify the kernels with long duration, but low utilization. These kernels should be optimized with high priority.

• *Memory consumption:* In addition to compute cycles, the amount of available physical memory has become a limiting factor in training large DNNs. In order to optimize memory usage during DNN training, it is important to understand where the memory goes, i.e. what data structures occupy most of the memory. Unfortunately, there are no open-source tools currently available for existing frameworks that can provide this analysis. Hence we build our own memory profilers for three main frameworks (TensorFlow, MXNet, and CNTK).

When building our memory profiler we carefully inspect how the different DNN frameworks in our benchmark allocate their memory and identify the data structures that are the main consumers of memory. We observe that most data structures are allocated before the training iterations start for these three frameworks. Each of the data structures usually belongs to one of the three types: weights, weight gradients and feature maps (similarly to prior works [207]). These data structures are allocated statically. In addition, a framework might allocate some workspace as a temporary container for intermediate results in a kernel function, which gives us another type of data structure. The allocation of workspace can be either static, before the training iterations, or dynamic, during the training iterations. We observe that in MXNet, data structures other than workspace are allocated during the training iterations (usually for the momentum computation) as well. We assign these data structures to a new type called "dynamic". As memory can be allocated and released during the training, we measure the memory consumption by the maximal amount of memory ever allocated for each type.

3.4 Evaluation

In this section, we use the methodology and framework described in the previous section for a detailed performance evaluation and analysis of the models in our `TBD` benchmark suite.

3.4.1 Experimental Setup

We use Ubuntu 16.04 OS, TensorFlow v1.3, MXNet v0.11.0, CNTK v2.0, with CUDA 8 and cuDNN 6. All of our experiments are carried out on a 16-machine cluster, where each node is equipped with a Xeon 28-core CPU and one to four NVidia Quadro P4000 GPUs. Machines are connected with both Ethernet and high speed Infiniband (100 Gb/sec) network cards.

As different GPU models provide a tradeoff between cost, performance, area and power, it is important to understand how different GPUs affect the key metrics in DNN training. We therefore also repeat a subset of our experiments using a second type of GPU, the NVidia TITAN Xp GPU. Table 3.4 compares the technical specifications of the two GPUs in our work. We show the comparative throughput and comparisons of our metrics between TITAN Xp and P4000 in Section 3.4.3.

	Titan Xp	Quadro P4000	Xeon E5-2680
Multiprocessors	30	14	
Core Count	3840	1792	28
Max Clock Rate (MHz)	1582	1480	2900
Memory Size (GB)	12	8	128
LLC Size (MB)	3	2	35
Memory Bus Type	GDDR5X	GDDR5	DDR4
Memory BW (GB/s)	547.6	243	76.8
Bus Interafce	PCIe 3.0	PCIe 3.0	
Memory Speed (MHz)	5705	3802	2400

Table 3.4: Hardware specifications

3.4.2 Performance Analysis

As previously explained, our analysis will focus on a set of key metrics: throughput, GPU and CPU compute utilization, FP32 utilization, as well as a memory consumption breakdown.

Since one of the aspects that makes our work unique is the breadth in application domains, models and frameworks covered by our TBD benchmark suite we will pay particular attention to how the above metrics vary across applications, models and frameworks.

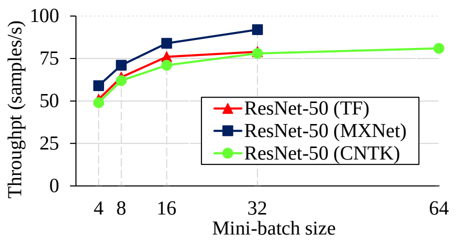
Moreover, we will use our setup to study the effects of a key hyper-parameter, the mini-batch size, on our metrics. It has been shown that to achieve high training throughput with the power of multiple GPUs using data parallelism, one must increase the mini-batch size, and additional work needs to be done on model parameters such as learning rate to preserve the training accuracy [70, 255]. In the single-GPU case, it is often assumed that larger mini-batch size will translate to higher GPU utilization, but the exact effects of varying mini-batch size are not well understood. In this work, we use our setup to quantify in detail how mini-batch size affects key performance metrics.

Throughput

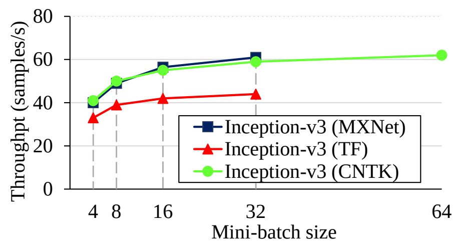
Figure 3.4 shows the average training throughput for different models from the TBD suite when varying the mini-batch size (the maximum mini-batch size is limited by the GPU memory capacity). For Faster R-CNN, the number of images processed per iteration is fixed to be just one on a single GPU, hence we do not present a separate graph for Faster R-CNN. Both TensorFlow and MXNet implementations achieve a throughput of 2.3 images per second for Faster R-CNN. We make the following three observations from this figure.

Observation 1: Performance increases with the mini-batch size for all models. As we expected, the larger the mini-batch size, the higher the throughput for all models we study. We conclude that to achieve high training throughput on a single GPU, one should aim for a reasonably high mini-batch size, especially for non-convolutions models. We explain this behavior as we analyze the GPU and FP32 utilization metrics later in this section.

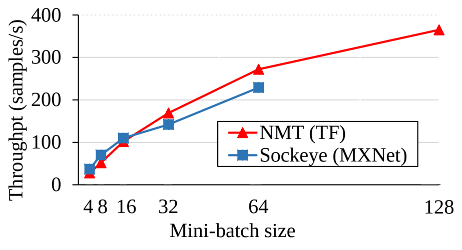
Observation 2: The performance of RNN-based models is not saturated within the GPU's memory constraints. The relative benefit of further increasing the mini-batch size differs a lot between different applications. For example, for the *NMT* model increasing mini-batch size from 64 to 128 increases training throughput by 25%, and the training throughput of Deep Speech 2 scales almost linearly. These two models' throughput (and hence performance) is essentially limited by the GPU memory capacity and we do not see any saturation point for them while increasing the mini-batch size. In contrast, other models also benefit from higher mini-batch size, but after certain *saturation*



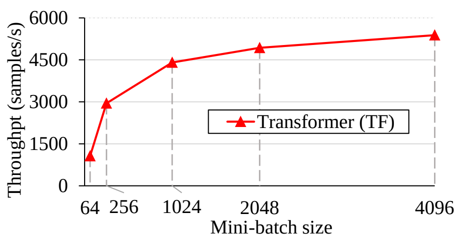
(a) ResNet-50



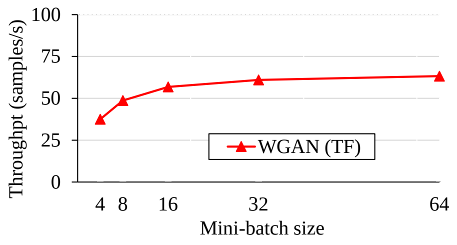
(b) Inception-v3



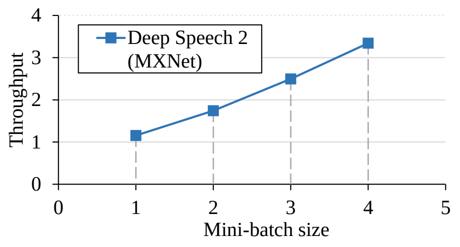
(c) Seq2Seq



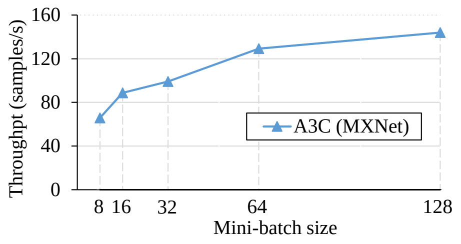
(d) Transformer



(e) WGAN



(f) Deep Speech 2

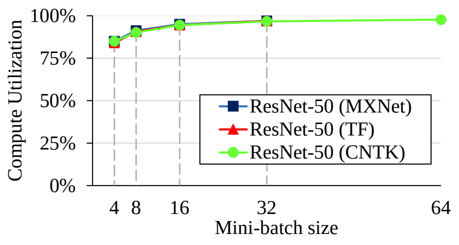


(g) A3C

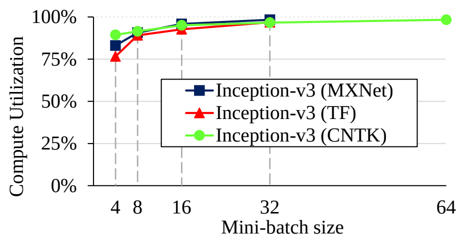
Figure 3.4: DNN training throughput for different models on multiple mini-batch sizes.

point these benefits are limited. For example, for the *Inception-v3* model going from batch size of 16 to 32 has less than 10% in throughput improvement for implementations on all three frameworks.

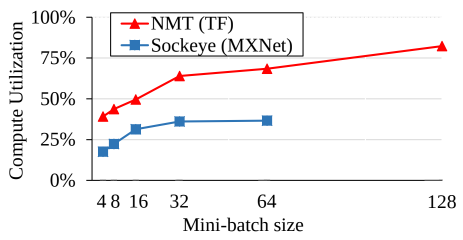
Observation 3: Application diversity is important when comparing performance of different frameworks. We find that the results when comparing performance of models on different frameworks can greatly vary for different applications, and hence using a diverse set of applications in any comparisons of frameworks is important. For example, we observe that for image classification the *MXNet* implementations of both models (*ResNet-50* and *Inception-v3*) perform generally better than the corresponding *TensorFlow* implementations, but at the same time, for machine translation the *TensorFlow* implementation of *Seq2Seq* (*NMT*) performs significalty better than its *MXNet* counterpart (*Sockeye*. TensorFlow also utilizes the GPU memory better than MXNet for Seq2Seq models so that it can be trained with a maximum mini-batch size of 128, while MXNet can only be trained with a



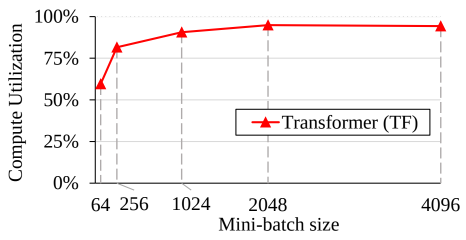
(a) ResNet-50



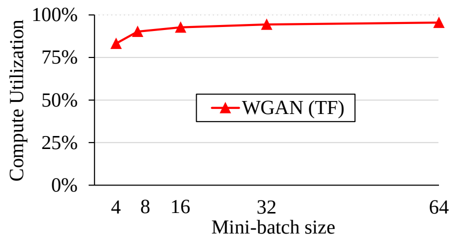
(b) Inception-v3



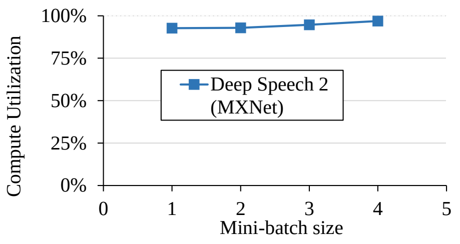
(c) Seq2Seq



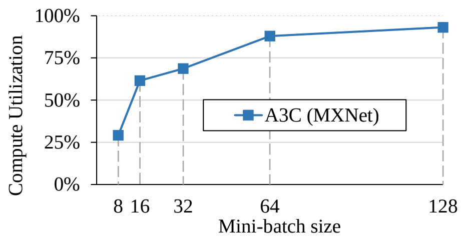
(d) Transformer



(e) WGAN



(f) Deep Speech 2



(g) A3C

Figure 3.5: GPU compute utilization for different models on multiple mini-batch sizes.

maximum mini-batch of 64 (both limited by 8GB GPU memory). For the same memory budget, it allows TensorFlow achieve higher throughput, 365 samples per second, vs. 229 samples per second for MXNet. We conclude that there is indeed a signficant diversity on how different frameworks perform on different models, making it extremely important to study a *diverse* set of applications (and models) as we propose in our benchmark pool.

GPU Compute Utilization

Figure 3.5 shows the GPU compute utilization, the amount of time GPU is busy running some kernels (as formally defined by 3.1 in Section §3.3) for different benchmarks as we change the mini-batch size. Again, for *Faster R-CNN*, only batch of one is possible, and TensorFlow implementation achieves a relatively high compute utilization of 89.4% and the *MXNet* implementation achieves

90.3%. We make the following two observations from this figure.

Observation 4: The mini-batch size should be large enough to keep the GPU busy. Similar to our observation 1 about throughput, the larger the mini-batch size, the longer the duration of individual GPU kernel functions and the better the GPU compute utilization, as the GPU spends more time doing computations rather than invoking and finishing small kernels. While large mini-batch sizes also increase the overhead of data transfers, our results show that this overhead is usually efficiently parallelized with the computation.

Observation 5: The GPU compute utilization is low for LSTM-based models. Non-RNN models and *Deep Speech 2* that uses regular RNN cells (not LSTM) usually reach very high utilization with large batches, around 95% or higher. Unfortunately, LSTM-based models (*NMT*, *Sockeye*) cannot drive up GPU utilization significantly, even with maximim mini-batch sizes. This means that, in general, these models do not utilize the available GPU hardware resources well, and further research should be done in how to optimize LSTM cells on GPUs. Moreover, it is important to notice that the low compute utilization problem is specific to the layer type, but not the application – the *Transformer* model also used in machine translation does not suffer from low compute utilization as it uses different (non-RNN) layer called *Attention*.

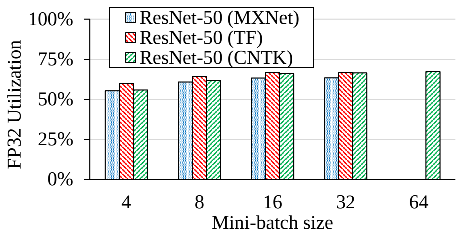
GPU FP32 utilization

Figure 3.6 shows the GPU FP32 utilization (formally defined by 3.2 in Section §3.3) for different benchmarks as we change the mini-batch size (until memory capacity permits). For Faster R-CNN, the MXNet/TensorFlow implementations achieve an average utilization of 70.9 and %/58.9% correspondingly. We make three major observations from this figure.

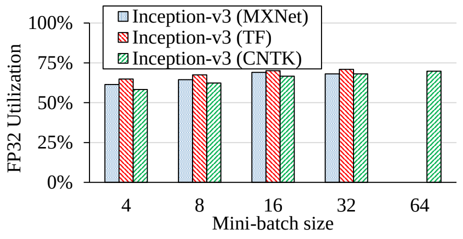
Observation 6: The mini-batch size should be large enough to exploit the FP32 computational power of GPU cores. As expected, we observe that large mini-batch sizes also improve GPU FP32 utilization for all benchmarks we study. We conclude that both the improved FP32 utilization (Observation 6) and GPU utilization (Observation 4) are key contributors to the increases in overall throughput with the mini-batch size (Observation 1).

Observation 7: RNN-based models have low GPU FP32 utilization. Even with the maximum mini-batch size possible (on a single GPU), the GPU FP32 utilization of the two RNN-based models (*Seq2Seq* and *Deep Speech 2*, Figure 3.6c and Figure 3.6f, respectively) are much lower than for other non-RNN models. This clearly indicates the potential of designing more efficient RNN layer implementations used in TensorFlow and MXNet, and we believe further research should be done to understand the sources of these inefficiences. Together with Observation 5 (low GPU utilization for LSTM-based models) this observation explains why in Observation 2 we do not observe throughput saturation for RNN-based models even for very large mini-batches.

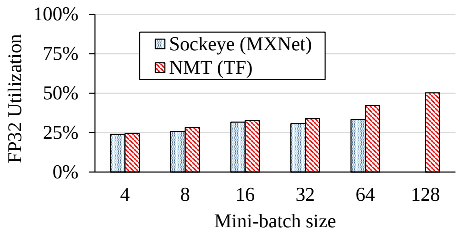
Observation 8: There exists kernels with long duration, but low FP32 utilization even for highly optimized models. The previous observation might have brought up the question why average FP32 utilizations are so low, even for extremely optimized CNN models. In this observation we provide an answer: Different kernels vary greatly in their FP32 utilizations, and even optimized models have long-running kernels with low utilization. Table 3.5 and Table 3.6 show the five most important kernels with the FP32 utilization *lower than average* (for *ResNet-50* model on TensorFlow and MXNet). We observe that the cuDNN batch normalization kernels (have *bn* part in their names) are the major source of inefficiency with FP32 utilizations more than 20% below the average. Note



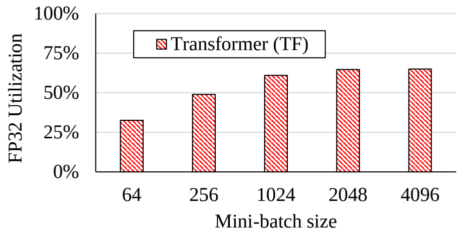
(a) ResNet-50



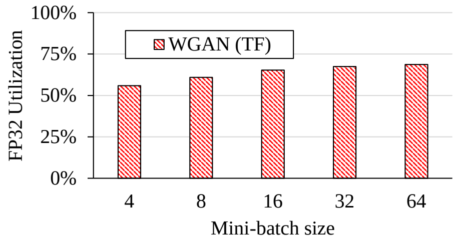
(b) Inception-v3



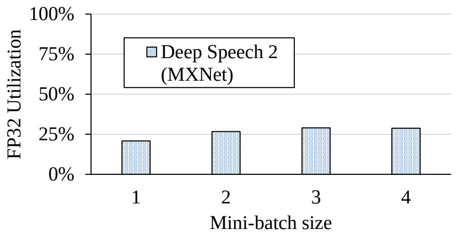
(c) Seq2Seq



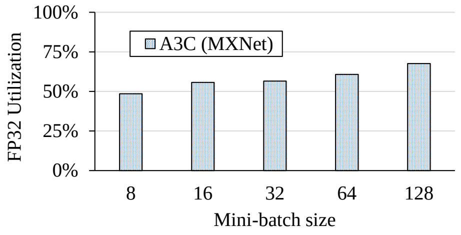
(d) Transformer



(e) WGAN



(f) Deep Speech 2



(g) A3C

Figure 3.6: GPU FP32 utilization for different models on multiple mini-batch sizes.

that this observation is true for implementations on different frameworks. If we want to get further progress in improving DNN training performance on GPUs, these kernels are the top candidates for acceleration.

3.4.3 Hardware Sensitivity

The results presented so far, were based on experiments with the Quadro P4000 GPU. In this section we are interested in seeing how the performance of DNN training will depend on the hardware used. Toward this end we compare the training throughput, GPU utilization, and FP32 utilization for several of our models: *ResNet-50*, *Inception-v3*, and *Seq2Seq* on P4000 GPU and the more powerful Titan Xp GPU. For throughput comparison, we normalized each model result to the throughput of less powerfull P4000 card. We make the following observation from this figure.

Duration	Utilization	Kernel Name
8.36%	30.0%	magma_lds128_sgemm_kernel...
5.53%	42.3%	cudnn::detail::bn_bw_1C11_kernel_new...
4.65%	46.3%	cudnn::detail::bn_fw_tr_1C11_kernel_new...
3.12%	20.0%	Eigen::internal::EigenMetaKernel...
2.48%	40.0%	tensorflow::BiasNHWCKernel...

Table 3.5: Top 5 time-consuming kernels with utilization level below the average (ResNet-50, mini-batch size 32, TensorFlow)

Duration	Utilization	Kernel Name
9.43%	30.0%	cudnn::detail::bn_bw_1C11_kernel_new...
7.96%	42.3%	cudnn::detail::bn_fw_tr_1C11_kernel_new...
5.14%	46.3%	cudnn::detail::activation_bw_4d_kernel...
3.52%	20.0%	cudnn::detail::activation_fw_4d_kernel...
2.85%	40.0%	_ZN5mxnet2op8mxnet_op20mxnet_generic_kernel...

Table 3.6: Top 5 time-consuming kernels with FP32 utilization below the average (ResNet-50, mini-batch size 32, MXNet)

Observation 9: More advanced GPUs should be accompanied by better systems designs and more efficient low-level libraries. Titan Xp usually helps improving the training throughput (except for *Sockeye*), however the computation power of Titan Xp is not well-utilized. Both the GPU and the FP32 utilizations of Titan Xp appear to be worse than those of P4000. Hence we conclude that although Titan Xp is more computationally powerful (more multiprocessors, CUDA cores, and bandwidth, see Table 3.4), the proper utilization of these resources requires a more careful design of existing GPU kernel functions, libraries (e.g., cuDNN), and algorithms that can efficiently exploit these resources.

3.4.4 Memory Profiling

As we have previously shown, the throughput (and hence the performance) of DNN training can be significantly bottlenecked by the available GPU memory. Figure 3.8 shows the result of our analysis where the memory is separated in five categories: weights, gradient weights, feature maps, dynamic, and workspace. Where appropriate, we vary the size of the mini-batch (shown in parentheses). The Faster R-CNN model results are similar to image classification models, but only support one batch size (hence we do not plot them in a separate graph).

Observation 10: Feature maps are the dominant consumers of memory. It turns out that *feature maps* (intermediate layer outpus) are the dominant part of the memory consumption, rather than weights, which are usually the primary focus of memory optimization for inference. The total amount of memory consumed by feature maps ranges from 62% in *Deep Speech 2* to 89% in *ResNet-50* and *Sockeye*. Hence any optimization that wants to reduce the memory footprint of training should, first of all, focus on feature maps. This is an interesting observation also because it expands on the results reported in the only prior work reporting on memory consumption breakdown for DNN training by Rhu et al. [207]. They look at CNN training only and find that weights are only responsible for a very small portion of the total memory footprint. We extend this observation outside of CNNs, but also observe that there are models (e.g., Deep Speech 2) where weights are equally important.

Observation 11: Simply exhausting GPU memory with large mini-batch size might be inefficient. The memory consumption of feature maps scales almost linearly with the mini-batch size. From

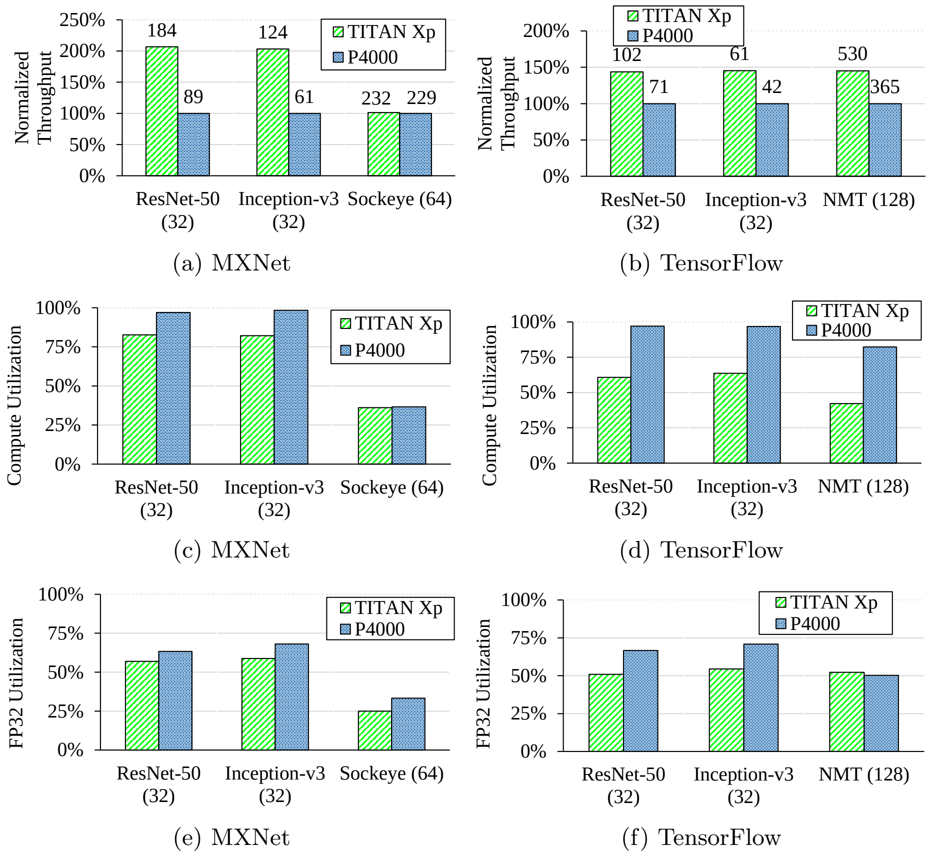


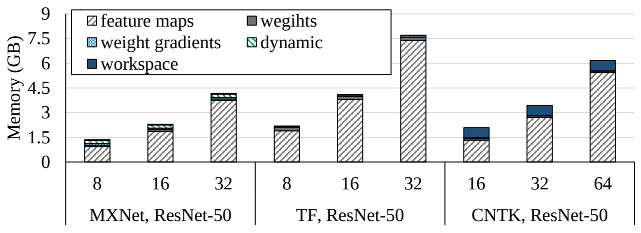
Figure 3.7: Throughput, Compute Utilization, FP32 Utilization comparison between P4000 and Titan Xp for different benchmarks.

observation 11, we know that reducing the mini-batch size can dramatically reduce the overall memory consumption needed for training. Based on observation 1, we also know that the side-effect of throughput loss while reducing the mini-batch size can be acceptable (for non-RNN models) until you do not go below saturation point. One can use the additional GPU memory for larger workspace (can be used for faster implementation of matrix multiplications or convolutions) and deeper models (e.g., ResNet-102 vs. ResNet-50).

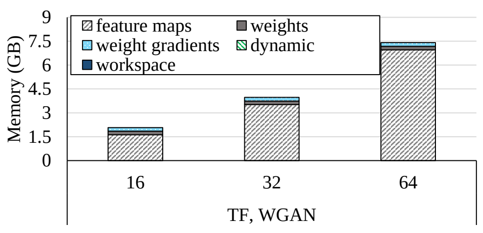
3.4.5 Multi-GPU and Multi-Machine Training

Training large DNNs can be done faster when multiple GPUs and/or multiple machines are used. This is usually achieved by using *data parallelism*, where mini-batches are split between individual GPUs and the results are then merged, for example, using the *parameter server* approach [133]. But in order to realize the computational potential of multiple GPUs the comminication channels between them need to have sufficient bandwidth to exchange proper weight updates. In our work, we analyze the performance scalability of DNN training using multiple GPUs and multiple machines. We use the *ResNet-50* model on MXNet to perform this analysis. Figure 3.9 shows the throughput results of our experiments.

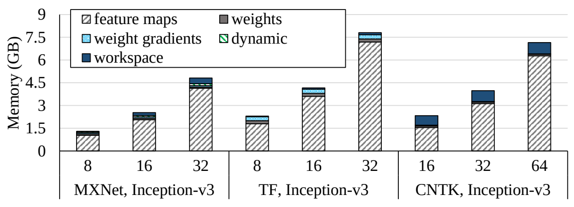
Observation 12: Network bandwidth must be large enough for good scalability. We observe that going from the one machine to the two machine coniguration, the performance degrades significantly



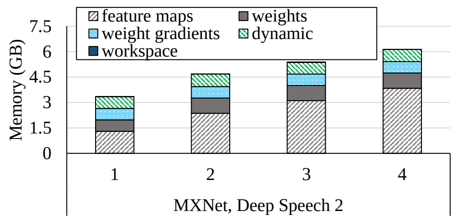
(a) ResNet-50



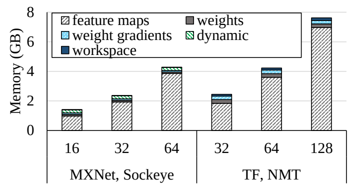
(b) WGAN



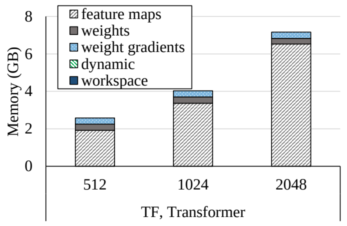
(c) Inception-v3



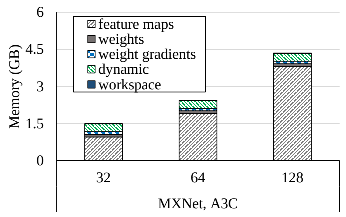
(d) Deep Speech 2



(e) Seq2Seq

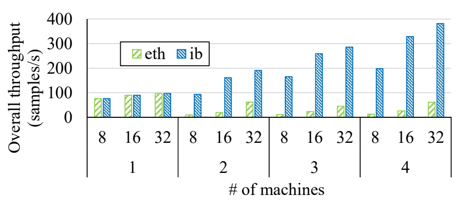


(f) Transformer

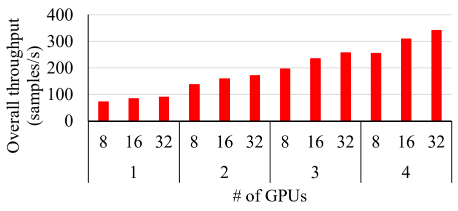


(g) A3C

Figure 3.8: GPU memory usage breakdown for different models on multiple mini-batch sizes.



(a) Multi-machine training with infiniband (ib) or ethernet (eth)



(b) Multi-GPU training on a single machine

Figure 3.9: ResNet-50 on MXNet with per-GPU mini-batch size of 8,16,32 on multiple GPUs/machines.

if with bandwidth of only 1Gbps. This is because DNN training requires constant synchronization between GPUs in distributed training. Hence faster networking is required to improve the situation (our infiniband configuration has 100Gbps IniniBand Mellanox networking). In contrast, DNN training on a single machine with multiple GPUs scales reasonably well as PCIe 3.0 gives enough bandwidth of 6 GBps. In summary, this suggests that networking bandwidth is critical for performance of distributed training and different techniques (in both software and hardware) should be applied to either reduce the amount of data sent or increase the available bandwidth.

We also applied our toolchain to measure the GPU compute and FP32 utilization for multi-GPU and multi-machine training. Given sufficient bandwidth (100 Gbps infiniband or 6 GBps PCIe 3.0),

Application	**Model**	**Number of Layers**	**Dominant Layer**	**Framework**	**Dataset**
Image classification	ResNet-50 [127] Inception-v3 [225]	50 (152 max) 42	CONV	TF, MXNet	ImageNet1K [209]
Machine translation	Seq2Seq [224] Transformer [235]	5 12	LSTM Attention	TF, MXNet	IWSLT16 [32]
Object detection	Mask R-CNN [89] EfficientDet [226]	101 8	CONV FPN	TF, PyTorch	COCO 2017 [64]
Speech recognition	Deep Speech 2 [17]	9	RNN	PyTorch	LibriSpeech [182]
Language modeling	BERT [59]	12 or 24	BERT block	PyTorch	SQuAD [199]
Reinforcement learning	MiniGo [157]	38	CONV	TF	

Table 3.7: Overview of our updated TBD benchmark suite, including the models and datasets used, number and major layer types, and frameworks with available implementations.

these utilization levels almost resembles with the single-GPU configuration. The GPU memory consumption per GPU remains the same if mini-batch size per GPU is the same. In this case, to improve the overall performance, one needs to focus again on addressing the single-GPU performance bottlenecks.

3.5 Progress

Machine learning is a rapid-developing area, as researchers constantly propose new DNN algorithms, improving the model accuracy and exploring new applications. New design in the algorithm could potentially introduce different configurations of typical layers, new custom implementation of low-level operators, or even new training paradigm, which could drive new system/architecture designs. Hence, a benchmark for DNN computation ought to be frequently updated with new DNN models to reflect state-of-the-art progress on the algorithms. We actively maintain TBD as an active and agile benchmark for system researchers. In the meantime, several new benchmarking works are proposed recently, each with a different focus. In this section we show the recent progress on TBD benchmarking.

3.5.1 Updating TBD Benchmark Models

Our model selection for the benchmark models (shown in Section §3.3.1) was based on the development of deep learning area in 2017. We went through a round of benchmark update in 2020 to include new DNN models with high impact. Table 3.7 shows the updated benchmark suite. Two of the major updates are the inclusion of the BERT [59] and EfficientDet [226] models.

EfficientDet [226] is a state-of-the-art model for the classical object detection problem, which locates and recognizes objects shown in an image. There was a series of state-of-the-art model based on regional CNN (RCNN) since 2014, and these models are widely used in real world applications such as face detection, video surveillance and self-driving cars. EfficientDet features the use of bi-directional feature pyramid network, and a compound scaling method that scales up a backbone model to fit different image resolutions. It has a potential for high impact on this application, as it achieves higher mean average precision with a much smaller number of FLOPS required.

BERT [59] is a recent major breakthrough in the natural language processing (NLP) area. It features a two-phase training paradigm: pre-training and fine-tuning. The pre-training phase is an unsupervised learning procedure which trains deep bidirectional representations. The pre-trained model is then fine-tuned with labeled data based on the downstream tasks. Despite that the pre-training phase is much more expensive than the fine-tuning phase, both phases adopt similar training iterations and their performance characteristics can be directly transferred to each other. It achieved ground-breaking state-of-the-art results on 11 NLP tasks, and its huge impact on the NLP area can be demonstrated by the 25k citations within 3 years. BERT is also now applied by to understand user search inquiries on Google over 70 languages.

3.5.2 Memory Profiler

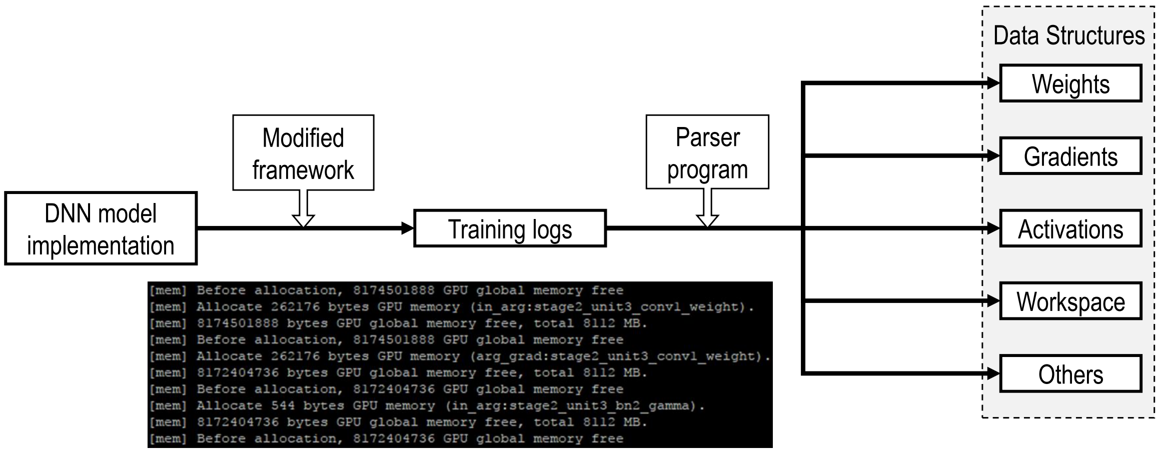
In this subsection we briefly describe the implementation details of our memory profiler, as it has been merged to the main branch of the MXNet framework and now viable to a wide range of ML developers. The goal of a memory profiler is to show the breakdown of the bulk of GPU allocated memory involved in DNN training. We classify allocations according to data structure, including:

- **Weights**: the weights and biases of each layers in the DNN model.
- **Gradients**: the gradients calculated in the backward pass, which often have equal size with weights.
- **Activations** (or feature maps): the intermediate results generated in the forward pass and reused in the backward pass, which is usually the major consumer of GPU memory.
- **Workspace**: the temporary space allocated for detailed implementations of operations (e.g. matrix multiplications, convolutions).

Besides these four categories of memory use, the frameworks might additionally allocate memory for its internal executions. Such memory footprint is usually framework-specific, less related to the model, and hard to track the cause to the memory allocation due to the complexity of the framework codebase. Based on our experience, such part usually takes a few hundreds memory footprint, which is usually not the major consumer. Our profiler will also demonstrate the memory consumption of this part for completeness.

Figure 3.10 illustrates the workflow of our memory profiler. We implement our memory profilers based on instrumentation to the mainstream frameworks by tracking the invocation of each memory allocation. We tag each memory allocation with domain knowledge, including its data structure and layer information based on the high-level call stack that triggers the allocation. The tags are written to separate log files, which are later fed to a log parser to analyze the We successfully implemented our memory profiler on MXNet and PyTorch, two of the mainstream DNN frameworks. In addition, our implementation of memory profiler has been merged to MXNet's main branch, making it much more viable to ML developers as mainstream DNN frameworks are constantly upgraded.

Notice that the memory consumption calculated by our memory profiler is often less then the real memory consumption due to memory paging. Each memory allocation that doesn't fully fit in a page contributes to inflating the actual memory consumption. This means that massive tiny



```
[mem] Before allocation, 8174501888 GPU global memory free
[mem] Allocate 262176 bytes GPU memory (in_arg:stage2_unit3_conv1_weight).
[mem] 8174501888 bytes GPU global memory free, total 8112 MB.
[mem] Before allocation, 8174501888 GPU global memory free
[mem] Allocate 262176 bytes GPU memory (arg_grad:stage2_unit3_conv1_weight).
[mem] 8172404736 bytes GPU global memory free, total 8112 MB.
[mem] Before allocation, 8172404736 GPU global memory free
[mem] Allocate 544 bytes GPU memory (in_arg:stage2_unit3_bn2_gamma).
[mem] 8172404736 bytes GPU global memory free, total 8112 MB.
[mem] Before allocation, 8172404736 GPU global memory free
```

Figure 3.10: Illustration of TBD Memory Profiler

memory allocations could cause significant overhead in memory footprint. We observe approximately hundreds of MBs of extra memory allocation when training our benchmark models on the NVidia 2080ti GPU (which page size is 2MB) due to paging. Such extra memory consumption can further increase if the page size becomes larger.

3.6 Related Work

There are only a handful of existing open-source DNN benchmark projects, each with a focus that is very different from our work. ConvNet [49], CNN-benchmarks [46] and Shaohuai et al. [220] focus exclusively on convolutional network models mainly for image classification based on the ImageNet data, with the only exception being one LSTM network in [220]. In contrast the goal of our work is a benchmark that covers a wide range of models and applications, beyond just CNNs and image classification.

DeepBench DeepBench [57] is an open-source project from Baidu Research, which is targeted at a lower level in the deep learning stack than our work: rather than working with implementations of deep learning models and frameworks, it instead benchmarks the performance of individual lower level operations (e.g. matrix multiplication) as implemented in libraries used by various frameworks and directly executed against the underlying hardware.

The Eyeriss project [23] presents evaluations of a few DNN processors [34, 83] on hardware metrics for several convolutional networks, but their work is focused on inference, while ours targets training.

Fathom Among existing work, Fathom [4] is probably the one closest to our own, as it also focuses on training and more than a single application (machine translation, speech recognition and reinforcement learning). However, their focus is on micro-architectural aspects of execution, breaking down training time into time spent on the various operation types (e.g. matrix multiplication). In contrast, our benchmark pool focuses on *system level* aspects of execution such as throughput, hardware utilization, and memory consumption profiling. Moreover, Fathom is based on only one framework (TensorFlow), does not consider distributed training and uses models that are somewhat out-dated by now.

DawnBench DawnBench [47] is the first work that benchmarks end-to-end training systems. It proposes the time-to-accuracy performance metric, which measures the end-to-end time for the DNN model to converge to a state-of-the-art accuracy level. This metric allows full-stack optimizations, ranging from advancements in DNN algorithms to the low-level architecture. It enables a fair comparison among techniques that involve a speed-quality trade-off: increasing training throughput with lower converging rage per iteration, which widely exhibits in many system/architecture-level optimizations.

Benchmarking ML Inference Systems As mentioned in Section §3.1, ML inference workloads significantly differ from training from various aspects (e.g., memory capacity, compute power, accuracy quality, etc), and are deployed on widely various platforms. AI Benchmark [103] is the first benchmark suite designed for mobile inference systems. It includes nine common ML applications on mobile, which contains mostly image-related tasks, and uses latency as the performance metric. EEMBC MLMark [231] is proposed for benchmarking inference workloads on edge devices. It measures both throughput and latency for the systems under test, but includes only three vision models (ResNet-50, MobileNet, SSD-MobileNet), and does not mandate threshold for inference accuracy.

AIBench AIBench [66] was proposed recently as another comprehensive benchmarking work, aiming to It covers major end-to-end AI applications on Internet service at industry scale. It summarizes sixteen DNN models for end-to-end macro-benchmarks, as well as twelve DNN operators as the micro-benchmarks. AIBench also provides a benchmarking framework that implements query generators, performance profilers, and deployment tools, creating a standard execution environment for easy porting and fair comparisons among different systems under test. While AIBench focuses on AI applications on cloud, Edge AIBench [84] is proposed for benchmarking end-to-end AI applications on edge devices. It identifies four typical application scenarios including ICU patience monitor, surveillance camera, smart house, and automatic vehicle, and six DNN testbeds from these applications. Besides performance metrics, AIBench also includes energy consumption as a standalone performance metric for each testbed.

3.7 Impact

In this section we discuss the major impact that `TBD` brings to the community. The `TBD` project not only helped motivate multiple research projects, but also contributed to the founding of MLPerf [149, 200], which currently is the most prestigious AI benchmark across both industry and academia.

3.7.1 Driving Optimization Innovations for DNN Training Workloads

The benchmarks, performance characterization analysis, and profiling toolchains of `TBD` have motivated multiple research opportunities. For example, DeepRecSys [75] describes a new infrastructure that enables design space explorations for a variety of recommendation models. DarkFPGA [143] addresses unique challenges of DNN training workloads over FPGAs. Echo [264] addresses the memory footprint problem in training RNN models based on observations derived from TBD's memory profiler. Wootz [71] and Radu et al. [197] propose new strategies to optimize CNN training via pruning. For distributed training workloads, BPPSA [242] proposes a new algorithm to overcome the scalability limitation due to backward propagation in training algorithms. P3 [107] uses a network profiling tool to discover new opportunity to improve scalability by improving the overlap between computation and communication. HetPipe [185] improves training efficiency on a cluster of heterogeneous GPUs.

3.7.2 Impact on Performance Analysis for DNN Workloads

TBD's methodology for profiling DNN training workloads has driven multiple performance analysis and characterization works on a diverse range of scenarios to discover new research opportunities for ML workloads. For example, researchers proposed several performance analysis works for DNN training/inference on a variety of hardware, such as mobile and edge devices [138, 251, 250], NVIDIA DGX clusters [159], AWS cloud [78], Google TPUs [248], FPGAs [120]. Meanwhile, several works [11, 131, 121, 152] provide in-depth analysis on GPU behaviours. Skyline [258] presents a new interactive tool that supports in-editor performance profiling and debugging DNN training workloads. Wu et al. [245] analyzes and compares modern ML frameworks over neural architecture implementations, resource usage and data loading.

On the application side, there are also a wide range of performance characterization works on specific and newly-emerged types of ML workloads. Gupta et al. [76] proposes a set of performance metrics for analyzing recommendation workloads. Gupta et al. [76] and Hsia et al. [96] conduct in-depth analysis over several industry-scale recommendation models. Wang et al. [239] presents performance characterizations for the DNN training workloads run on Alibaba's platform of AI. SeqPoint [189] borrows the idea of sampling iterations in TBD, and extends it for profiling sequins-based neural networks. OARF [98] presents characterizations for federated learning workloads. RLScope [68] focuses on reinforcement learning (RL) workloads, presenting a open-source cross-stack tool to support systematic analysis over RL algorithms. Pati et al. [188] focuses on characterizing BERT models and derives several implications for architecture design.

3.7.3 MLPerf

The urge for a standard machine learning benchmark has been widely recognized in system and architecture communities. MLPerf [149, 200] involves a joint effort from a wide range of industry and academic groups, and is currently the most prestigious and widely influential benchmark work. It has a wide scope for the benchmarking goals, which aims to drive machine learning innovations with diverse focuses (i.e., innovations on algorithms/software/hardware for training/inference on cloud/edge devices, etc.). Hence, MLPerf introduces new metrics and rules designed for benchmarking DNN computation workloads for various scenarios.

Performance Metrics MLPerf benchmarks both training and inference workloads. Similar to DawnBench, MLPerf utilizes time-to-accuracy as the performance metric for benchmarking the training workloads. On the other hand, the inference workloads could be deployed on hardware platforms ranging from data center servers, to edge devices such as smartphones, and IoT devices, leading to a wide variety of end-user scenarios. Each of these scenarios requires a corresponding performance metric to reflect the performance in production. MLPerf identifies four representative scenarios, including single-stream, multi-stream, server, and offline, and proposes latency, latency-bound number-of-streams, latency-bound queries-per-second, and throughput as the corresponding metrics for these scenarios respectively.

Model Selection Different ML systems in production come with different constraints and specifications, which drives new model designs and optimization techniques (e.g. quantization, sparsification, etc.). Hence, comparing to TBD and other prior works, MLPerf selects the benchmarking models from a wider spectrum. For example, the MLPerf v0.5 suite includes MobileNet [210] for image classification inference task, and SSD [139] network with MobileNet backbone for object detection inference task on edge device.

Open and Closed Divisions Optimizations that improve the end-to-end time-to-accuracy could operate at any abstraction level in the full stack of ML training systems. The open division of MLPerf aims to encourage co-designs across model architecture, systems and hardware, and hence only puts restrictions on the dataset and quality threshold. On the other hand, MLPerf also has the closed division, which restricts the mathematical model implementations, and allows submitters to modify a small set of hyper-parameters to guarantee model accuracy. This division enables fair comparisons among different system/architecture designs.

TinyMLPerf TinyMLPerf [22] expands the scope of ML benchmarking to AI inference workloads on ultra-low-power devices, which is a recently-emerging area for low-power AI computation. These systems avoid the cost of communication and guarantee data privacy, but come with extremely restricted power limit, and hardware resources. TinyMLPerf identifies four typical inference workloads, each with an extremely small neural structure. Currently TinyMLPerf is under development, while open challenges such as power and memory measurements, heterogeneity in software and hardware are yet to resolve.

Despite the success and wide influence of MLPerf, the evolving of MLPerf benchmarks is relatively slow. For example, BERT [59], a major breakthrough for language modeling tasks in 2018, was not included in MLPerf benchmark suite until 2020. On the contrary, TBD is maintained as a more agile benchmark suite for academic purpose. TBD's analysis overcomes the complexity of the hardware/software stack and reveals the end-to-end training performance, which often differs from the performance claimed by the hardware specifications. As a result, the Vector Institute utilizes the TBD open-source benchmarks to guide their purchase of GPUs for its internal cluster.

3.7.4 Impact on Other ML Benchmarks

Besides MLPerf, there are also a wide range of newly-proposed benchmarks, each with a specific focus. For example, DeepOBS [211] benchmarks DNN optimizers as well as some other hyper-parameters to derive insights for neural architecture design. HPC AI500 [114] and AIPerf [205] present new methodologies for benchmarking HPC AI systems. Huang et al. [101] benchmarks DNN workloads with time series data. QuTiBench [27] targets a broader spectrum of customized and heterogeneous architectures, and presents benchmarking with support of optimizations that operate on multiple software levels. iBench [] benchmarks distributed DNN inference workloads on edge devices, as well as provides insights on the impact of data movement, data arrival patterns and architecture designs.

3.8 Conclusion

In this chapter, we proposed a new benchmark suite for DNN training, called `TBD`, that covers a wide range of machine applications from image classification and machine translation to reinforcement learning. `TBD` consists of eight state-of-the-art DNN models implemented on major deep learning frameworks such as TensorFlow, MXNet, and CNTK. We used these models to perform extensive performance analysis and profiling to shed light on efficiency of DNN training for different hardware configurations (single-/multi-GPU and multi-machine). We developed a new tool chain for end-to-end analysis of DNN training that includes (i) piecewise profiling of specific parts of training using existing performance analysis tools, and (ii) merging and analyzing the results from these tools using the domain-specific knowledge of DNN training. Additionally, we built new memory profiling tools specifically for DNN training for all three major frameworks. These useful tools can precisely characterize where the memory consumption (one of the major bottlenecks in training DNNs) goes and how much memory is consumed by key data structures (weights, activations, gradients, workspace). By using our tools and methodologies, we made several important observations and recommendations on where the future research and optimization of DNN training should be focused. We hope that our `TBD` benchmark suite, tools, methodologies, and observations will be useful for a large number of ML developers and systems designers in making their DNN training process efficient.

Chapter 4

Daydream: Accurately Estimating the Efficacy of Optimizations for DNN Training

4.1 Introduction

Recent years have witnessed the co-evolution of deep neural network (DNN) algorithms and the underlying hardware and software design. ML researchers have developed many important models [59, 89, 87, 235] at a rapid pace, creating a huge demand for computation power [212]. To meet the demand for fast DNN computation, computer architects respond with new, AI-optimized GPUs (e.g., NVidia Turing architecture [172]) and various domain-specific hardware accelerators from FPGAs (e.g., Microsoft Catapult [195]) to ASICs (e.g., Google TPU [117], Amazon Inferentia [215]). However these accelerators might not be effective in improving performance without proper software optimizations across the full systems stack [267]. As a result, systems researchers have proposed many optimizations, targeting different bottlenecks across the system stack – for example, improving memory utilization [206, 106], better overlapping of communication with computation [261, 107, 85], and increasing communication efficiency [43]. Moreover, researchers have also developed workload-centric optimizations to exploit the stochastic nature of DNN computation. For example, precision reduction [53, 74, 153] aims to reduce runtime as well as memory consumption, and gradient compression [141, 137] aims at reducing the communication overhead in distributed training.

Despite these advances, the benefits of many proposed optimizations cannot be fully exploited due to two main reasons. First, the efficacy of many proposed performance optimizations can drastically change when applied to different ML models and deployment configurations. The hardware deployments that practitioners use might be completely different from the hardware configurations used by optimization and model inventors. Differences in DNN models, accelerator type, compute capabilities, available memory, networking capabilities, and software library versions can all shift the major runtime bottlenecks. Second, it is onerous for programmers to implement and evaluate various optimizations to identify the ones that actually work for their models. As a result, it is common for users to ask *what-if* questions such as:

Why did my DNN training workload run slowly? Will optimization X improve the performance of my model? Does GPU memory capacity limit the performance of my model? Would upgrading to a faster network improve training throughput? How will my workload scale with the number of GPUs?

The central focus of this chapter is to answer the following general question for DNN training workloads: *Given a model and a deployment scenario, how can we efficiently explore the efficacy of potential solutions*? Systems researchers have tried to explore the impact of different potential performance bottlenecks (e.g., CPU, network, IO) in many non-ML contexts [154, 5, 194, 52, 179, 180]. The basic approaches to explore the what-if questions are similar: decompose the workloads into atomic tasks, profile runtime statistics for each task, model the what-if question, and use simulation to estimate performance. These systems typically address what-if questions of the form: "How does runtime change if a task T is N times (or even infinitely) faster?" [52, 179]. Such questions can be simply modeled by shrinking task runtime. While this basic approach seems sufficient to address the central question above for ML workloads, the **diversity of DNN optimizations** introduces three key requirements unique to these workloads, thus motivating the need for a novel solution.

First, we need to **track dependencies at a kernel-level abstraction** i.e., one GPU kernel corresponds to one task (the smallest unit of execution in the dependency graph). Such fine-grained abstraction is necessary because optimizations that improve hardware utilization typically target individual compute kernels (e.g., mixed precision [153]). Meanwhile, accurate performance estimation has to consider both CPU and GPU runtime. Certain optimizations, e.g., kernel fusion, require potentially removing existing CPU and GPU tasks from the dependency graph. Existing tools do not provide such dependency tracking. It is therefore important to track kernel-level dependencies among concurrently executing tasks.

Second, we need to **map tasks to DNN layers**. In contrast to prior works that explore what-if questions in non-ML contexts, predicting the performance of DNN optimizations requires domain knowledge about DNNs to properly model them. For example, MetaFlow [112] and TASO [113] fuse DNN layers. Modeling them requires a mapping from tasks to specific DNN layers. However, collecting kernel-level traces on accelerators requires generic vendor-provided tools (e.g., NVProf [171], CUPTI [174]), which have no application specific knowledge. We therefore need to have the ability to map low-level tasks to DNN layers.

Third, we need the **ability to easily model diverse DNN optimizations**. Modeling a DNN optimization might involve not just scaling or shrinking task durations, but also complicated transformations to the dependency graph. For example, TicTac [85] reschedules communication tasks, BlueConnect [43] replaces the communication primitives to utilize parallel network channels, and the optimization proposed by Jung *et al.* [119] restructures the GPU kernel implementations. Manually manipulating the kernel-level dependency graph could be extremely intricate and error-prone. The system should enable users to flexibly and effectively model such diverse optimizations with minimal effort.

We introduce *Daydream*, a new system that fulfills all three requirements described above, and achieves our goal of answering potential what-if questions for DNN workloads. Constructing dependencies among potentially thousands of low-level tasks is not an easy problem: tasks can be spread across multiple execution threads (including both CPU threads and GPU streams), thus even for

Optimization Goal	**Strategy**	**Technique Examples**
Improving Hardware Utilization in Single-Worker Setting	Reducing Memory Footprints	**vDNN** [206], **Gist** [106], Chen *et al.* [38]
	Reducing Precision	***Micikevicius et al.*** [153], Gupta *et al.* [74], Das *et al.* [53]
	Fusing Kernels/Layers	***FusedAdam*** [163], **MetaFlow** [112], Ashari *et al.* [19], TASO [113]
	Improving Low-level Kernel Implementation	***Restructing Batchnorm*** [119], Tensor Comprehensions [234], Kjolstad *et al.* [126], TVM [39]
Lowering Communication Overhead in Distributed Training	Reducing Communication Workloads	**DGC** [137], AdaComm [238], Parallax [124], TernGrad [243], QSGD [10]
	Improving Communication Efficiency/Overlap	***Wait-free Backprop*** [261], ***P3*** [107], **BlueConnect** [43], BytePS [191], Xue *et al.* [253]

Table 4.1: Representative optimizations for DNN training. In this chapter, we show how we can accurately estimate the performance of optimizations (shown in ***italics***), and can effectively model many other optimizations (shown in **bold**).

simple DNN workloads, this results in thousands of tasks to be tracked. The intricacy comes from identifying dependencies across threads. We make a key observation about DNN training workloads: despite the large number of tasks that need to be tracked, the number of concurrently executing threads is surprisingly quite limited. Based on this observation, Daydream constructs the low-level dependency graph, which provides a realistic model of overlapping among CPU, GPU, and communication runtimes in a DNN training workload. It uses a synchronization-free approach to map GPU tasks onto appropriate higher-level DNN layer abstractions. We also introduce a set of graph-transformation rules, allowing programmers to effectively model various performance optimizations. After modeling the optimization, Daydream simulates the execution based on the new dependency graph to predict the overall runtime. In our evaluation, we show that Daydream is able to distinguish effective DNN optimizations from those that will bring limited improvements by accurately predicting their performance speedups.

4.2 DNN Training Optimizations and Tools

DNN training is an iterative algorithm, in which one iteration consists of three phases: (i) *forward*, (ii) *backward*, and (iii) *weight update*. The *forward* phase takes training data samples as input and produces output based on current weights (or parameters). The error between the *forward* output and the input data labels is fed to the *backward* phase, which computes the gradients of weights with respect to the input data. The *weight update* phase then uses the gradients to update weights accordingly. In each iteration, the input data samples are randomly selected [30], forming a *mini-batch* of input.

4.2.1 DNN Training Optimizations

Modern DNNs have millions of parameters [82], resulting in training times of days or even weeks [127]. To improve DNN training performance, researchers have proposed various strategies focusing on different optimization goals. To understand the potential what-if questions and how to design a system to answer them, we study a list of software-level techniques that speedup DNN training from top systems and ML conferences in recent years. Table 4.1 shows our summary.

Exploiting computation power of hardware accelerators. ML programmers often use large mini-batches, within the memory budget, for better hardware utilization and faster convergence. This motivates strategies that reduce the memory footprint of DNN training and hence enables training with larger mini-batch sizes [38, 106, 206]. Researchers have also proposed some generic strategies to increase hardware utilization, including precision reduction [53, 74, 153], kernel/layer fusion [19, 112, 113], and improving low-level kernel implementation [119, 126, 234]. Meanwhile, libraries such as cuDNN [40], cuBLAS [164], MKL [237], Eigen [63], and NCCL [169] are also constantly evolving to provide operations and primitives that can better utilize underlying hardware.

Scalable distributed training. Data parallelism [30] is a simple and effective strategy to improve training performance. Using multiple accelerators significantly reduces DNN training time to hours or even minutes [156]. This success is mainly based on the techniques that guarantee model convergence under extremely large mini-batch size [7, 70, 256]. One of the major performance bottlenecks for distributed training is communication, which can be optimized by compressing traffic [137, 141, 238, 243], increasing network utilization [43, 253], or increasing the overlap between communication and computation [85, 107, 261]. Exploring the efficacy of these optimizations without prediction requires a multi-machine cluster. Our proposed design, Daydream, avoids the potential cost of cluster setup (i.e. extra machines, accelerators, high-speed communication), by predicting distributed training performance with profiles collected from a single-worker environment.

4.2.2 Profiling Tools for DNNs

As the full ML system stack is constantly evolving, profiling tools play a key role in helping programmers identify the performance bottlenecks under different system configurations.

Hardware profiling tools. Modern DNN training heavily relies on hardware accelerators such as GPUs [172] and TPUs [117]. To help programmers develop highly efficient applications, hardware vendors provide profiling tools that can expose hardware performance counters. For example, NVProf [171] provides programmers with information including start/end time, core utilization, memory throughput, cache miss rate, along with hundreds of other hardware counters for every GPU kernel. CUPTI [174] enables programmers to extract and manipulate these counters at runtime. Nsight [170] aims to provide details on the state of more fine-grained counters for recent GPU architectures [172]. Our proposed system, Daydream, relies on CUPTI to collect low-level traces for further analysis.

Framework built-in tools. For more intuitive profiling results, it is often desirable for a profiler to show runtime statistics for framework operations, or even DNN layers. DNN frameworks have built-in tools to achieve this goal by correlating the hardware counters with runtime information collected in frameworks. TensorFlow [2], coupled with the Cloud TPU Tool [69], can provide an execution timeline and runtime statistics for each TensorFlow operation. Similarly, other mainstream

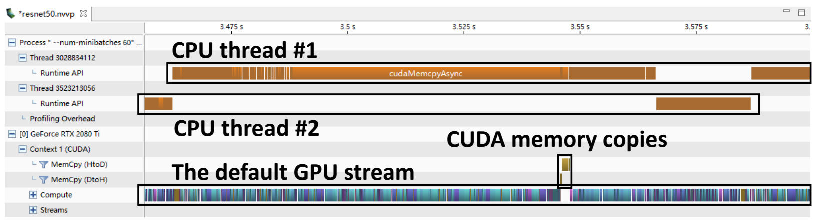


Figure 4.1: NVProf timeline example of training ResNet-50.

frameworks (e.g., MXNet [36] and PyTorch [187]) provide built-in tools that can extract per-layer or per-operation runtime from both the CPU and the GPU. The framework built-in tools render intuitive results for programmers, but omit important details (for example, the CPU runtime). We show in this chapter that such information is crucial in building an accurate runtime predictor.

4.3 Key Ideas

In this section we highlight the key ideas and observations behind the Daydream design.

Constructing kernel-granularity dependency graph. The neural network topology is a natural graph structure in which nodes are DNN operators or layers. Most mainstream DNN frameworks [36, 187] provide built-in tools to record the layer-level runtime profile. The layer-level abstraction is intuitive for programmers to understand the "where time goes" question, but hides important information about the parallel execution of the CPU functions, GPU kernels, and memory transfers. This information is crucial for accurate performance predictions. For example, optimizations that reduce numerical precision will change the duration of GPU kernels while the CPU runtime remains unchanged, and optimizations like vDNN [206] will inject CUDA memory copies, without changing the duration of GPU kernels. It is extremely hard to predict how duration of each layer changes when applying these optimizations if lacking low-level details about CPU and GPU runtime. To accommodate optimizations that target fine granularity tasks (such as GPU kernels), our proposed system, Daydream chooses to model the training workloads using a kernel-level dependency graph (i.e., each GPU kernel has one corresponding task in the graph), incorporating detailed traces of CPU, GPU and communication runtime.

With a large number of kernel-level tasks that are spread across several threads and CUDA streams, the complexity of constructing the dependency graph comes mainly from identifying the inter-thread dependencies [194]. Existing tools do not provide such dependency tracking. We make the following key observations about the DNN training workloads to overcome this general challenge of dependency tracking in concurrent systems. First, for the implementations in the mainstream frameworks [36, 187], once a mini-batch has been prepared by data loading threads, only one or two CPU threads are involved in the control flow of computation.[1] Second, there is a very limited number of concurrent GPU kernels. Such serialization of GPU kernels is due to two main reasons: (i) GPU kernels in the modern cuDNN library achieve high GPU core utilization; (ii) ML frameworks usually invoke only one CUDA stream. Figure 4.1 shows the NVProf profiles of one training iteration of ResNet-50. There are two CPU threads involved, but no CPU tasks run concurrently. The high serialization of low-level traces is not a unique phenomenon for just convolutional networks. We

[1]Our approach can be generalized to frameworks that use more concurrent CPU threads.

observe a similar phenomenon in most DNN training workloads.

Based on these insights, Daydream constructs the kernel-level dependency graph in three major steps. First, Daydream uses CUPTI to extract traces of all GPU kernels, CUDA memory copies, and CUDA APIs. Second, Daydream captures the dependencies between CPU and GPU tasks, caused by CUDA synchronizations and GPU kernel launches. Third, when predicting performance for distributed training, Daydream adds communication tasks to the dependency graph.

Synchronization-free task-to-layer mapping. In distributed training, mainstream frameworks implement the wait-free backpropagation strategy [261] to overlap communication with computation. This strategy immediately transfers gradients once they are computed by corresponding backward layers. To properly add dependencies related to communication tasks, we need the task-to-layer mapping to know when the computation of each layer ends. Meanwhile, accurately modeling DNN optimizations by changing the graph potentially requires this task-to-layer mapping to determine which tasks are involved and how to change them.

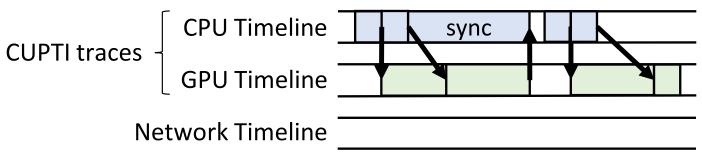
Unfortunately, vendor-provided tools like CUPTI do not have the required knowledge about these applications and building such a mapping requires extra DNN framework instrumentation. A naïve approach to achieve this mapping is to compare the start and stop timestamps of GPU kernels and DNN layers. This requires additional CUDA synchronization calls for each layer since GPU kernels are launched asynchronously. However, such synchronizations might significantly alter the execution runtime by adding additional dependencies from GPU to CPU tasks. Hence, we design a synchronization-free procedure to achieve this mapping by instrumenting timestamps for each layer in the frameworks, and utilizing the correlations between CPU and GPU tasks.

Representing complex optimizations with simple graph-transformation primitives. As shown in Table 4.1, DNN optimizations target a wide range of performance bottlenecks with various approaches. Unlike prior dependency graph analysis in non-ML contexts [52, 179, 180], where users can model most what-if questions by simply shrinking and scaling task runtime, accurately modeling DNN optimizations with the low-level dependency graph might require complicated changes to the dependency graph. Manually changing the kernel-level graph to model optimizations could be both complicated and error-prone, and the programmers might simply opt to rather directly implement the optimizations.

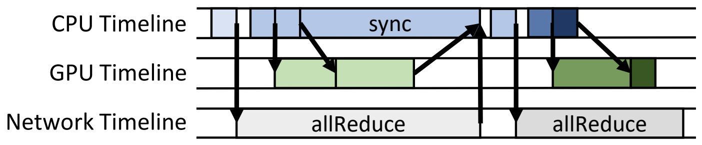
To address this problem, we propose a small set of graph-transformation primitives, so that popular optimization techniques can be effectively represented as a combination of these primitives. These primitives include (i) task insertion/removal, (ii) taskselection and update, and (iii) changing the policy for scheduling tasks. The proposed primitives are simple yet powerful enough to represent many different optimizations as we will show in Section §4.5. They play a key role in realizing our goal of efficiently exploring what-if questions.

4.4 Design

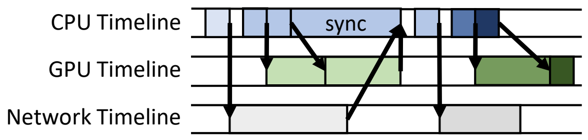
We describe Daydream's design with an emphasis on how to construct Daydream's proposed graph abstraction: the kernel-granularity dependency graph with mappings back to DNN layers. We also describe the primitives for mutating this graph to model different optimizations and how Daydream uses the graph to estimate the efficacy of various DNN optimizations.



(a) Constructing the dependency graph based on CUPTI traces (the black arrows represent task dependencies).



(b) Mapping each task to DNN layers (shown in different colors in the figure), then inserting communication tasks based on mapping and instrumentation.



(c) Predicting ”what if network bandwidth is 2×” by shrinking allReduce duration by 2× and simulating the new dependency graph.

Figure 4.2: An example showing Daydream's overall workflow for predicting runtime assuming network bandwidth doubles.

4.4.1 Overview of Daydream

Figure 4.2 shows the workflow of performance prediction in Daydream. It consists of the following four phases:

Phase 1: Trace collection. Constructing a kernel-level dependency graph requires low-level details for all tasks. These details are extremely massive, differ across ML frameworks, and can be obtained by profiling a baseline workload. Daydream collects low-level profiling data using CUPTI [174], a tool which provides details for all CPU/GPU tasks including name, start time, duration, CUDA stream ID, thread ID, etc. We manually augment three popular frameworks (Caffe, MXNet, PyTorch) for use with CUPTI and modify the layer modules of these frameworks to collect timestamps of each layer, which will be used for task-to-layer mapping, described in Section §4.4.3. Through our instrumentation, we also collect the necessary information (e.g., size of gradients) to construct the dependency graph of distributed training via a profile collected in a single worker setting.

Phase 2: Dependency graph construction. Daydream constructs the dependency graph with details of tasks provided by the first phase. A dependency could be induced by domain knowledge (e.g., a GPU task triggers a communication task), or by hardware/software implementation (e.g., a cudaLaunchKernel API triggers the corresponding GPU task). Based on our analysis, we identify five different types of dependencies (described in Section §4.4.2), which are sufficient for Daydream to accurately simulate baseline execution.

Phase 3: Graph transformation. To estimate the efficacy of a given optimization, Daydream models the optimization by transforming the dependency graph. Daydream provides a set of primitives (e.g. selection, insertion/removal) to represent these transformations. We design these primitives in a way such that they are succinct (easy to use), flexible (able to depict a wide range of optimizations), and accurate (being able to achieve high prediction accuracy).

Algorithm 1: Daydream's Simulation Algorithm

```
Input  : Dependency graph: $G(V, E)$
Output: The start time of each task $u \in V$
1  $F \leftarrow \emptyset$ // initialize the frontier task set
2  $P \leftarrow \{0\}$ // initialize thread progress
3  foreach task $u \in V$ do
4      $u.ref \leftarrow |\{u'sparents\}|$
5      if $u.ref = 0$ then
6          $F \leftarrow F \cup \{u\}$
7  end
8  while $F \neq \emptyset$ do
9      $u \leftarrow schedule(F)$ // pick a task to exec.
10     $t \leftarrow u.ExecutionThread$
11     $F \leftarrow F - \{u\}$
12     $u.start \leftarrow max(P[t], u.start)$
13     $P[t] \leftarrow u.start + u.duration + u.gap$
14     foreach $c \in u.children$ do
15         $c.ref \leftarrow c.ref - 1$
16         $c.start \leftarrow max(c.start, u.start + u.duration + u.gap)$
17         if $c.ref = 0$ then
18             $F \leftarrow F \cup \{c\}$
19         end
20     end
21 end
```

Phase 4: Runtime simulation. Daydream simulates the execution of optimizations to predict runtime based on the dependency graph. Algorithm 1 shows the simulation process, which traverses the dependency graph and puts tasks into execution threads. In each iteration, Daydream picks one task from the execution frontier (i.e. tasks that are ready to execute), dispatches it to its corresponding execution thread, and updates the thread progress. The simulation determines the start time of each task and records the total execution time.

4.4.2 Dependency Graph Construction

Constructing the dependency graph is essential to determine the node (task) set and edge (dependency) set. Daydream's kernel-level dependency graph contains the following four types of tasks:

GPU tasks. Each GPU task in the graph corresponds to one GPU kernel. Daydream also views CUDA memory copies as GPU tasks, because each memory copy is associated with a specific CUDA stream, and therefore has dependencies with other GPU kernels. The runtime of all these tasks can be collected using CUPTI.

CPU tasks. To model the concurrency and dependencies between CPU runtime and the GPU runtime, Daydream generates CPU tasks based on CPU traces collected by CUPTI. One of the limitations of CUPTI is that it can only expose CUDA-related traces. Instead of adding massive instrumentation to the framework, Daydream captures the non-CUDA runtime by recording the lengths of gaps between consecutive CPU tasks (shown in line 13 of Algorithm 1).

Data loading tasks. One data loading task corresponds to loading one mini-batch from disk/flash to CPU memory. We include data loading tasks for completeness, even though data loading in most DNN training workloads is not a performance bottleneck. In Daydream's implementation, we treat all data loading tasks as CPU tasks.

Communication tasks. A communication task corresponds to one communication primitive, e.g., a push/pull operation in parameter-server based frameworks [133], or an all-reduce operation in decentralized frameworks. When predicting distributed training performance, Daydream automatically adds communication tasks to the dependency graph based on a single-worker profile. We notice that in PyTorch, gradients from multiple layers can be grouped and sent with a single allReduce primitive [196]. Thus, properly adding communication tasks to a PyTorch profile requires additional instrumentation to extract knowledge about gradients grouping.

Given the types of tasks in the graph, Daydream collects and maintains the following information for each task, which is later used in what-if analysis and simulation:

ExecutionThread. Depending on the type of a task, its execution thread can be on of the following: (i) a CPU process, (ii) a GPU stream, and (iii) a communication channel. A data loading task is executed in a CPU process. A CPU process has a process ID, a GPU stream has a stream ID, and a communication channel could be send/receive when using parameter server primitives, or a unified one when using collective primitives. This field is used in line 10 of Algorithm 1.

Duration. This field specifies how long a task takes to execute. The duration of a CPU/GPU task is collected by CUPTI. The runtime of data loading tasks is measured by injecting timestamps to the framework. Daydream aims to predict distributed training performance based on profiling in a single-GPU configuration. Hence we calculate the duration of all communication task based on the size of gradients, the communication type (push/pull/all-reduce), and the network bandwidth. These numbers can be obtained based on knowledge of the DNN model and framework implementation.

Gap. The duration of low-level CUDA APIs (e.g., `cudaMalloc`) might be only tens of microseconds, which is of the same magnitude as the runtime of their non-CUDA equivalent C functions (e.g., `malloc`), or the runtime of the call stack from Python front-end to C back-end. NVidia-provided tools cannot expose non-CUDA traces, but they are indispensable to simulation accuracy. The non-CUDA CPU runtime is usually not a target for optimization in DNN models, hence, we do not need to define and measure corresponding tasks. Instead, for each CPU task in our current definition, we measure the gap between its end and the start of the next task in the same execution thread, and simulate these gaps in Algorithm 1.

Layer. This field refers to which DNN layer a task belongs to, which is necessary information for programmers to transform the graph and model optimizations. Daydream uses a synchronization-free approach to map a task to DNN layers. We will describe the details of this approach in Section §4.4.3.

Based on our discussion in Section §4.3, we identify the following five types of dependencies for accurate simulations.

Sequential order of CPU tasks in the same thread. CPU tasks in the same thread are serialized. The order that CPU tasks are executed in is determined by the framework and does not change in two separate executions. We add a dependency between each two consecutive CPU tasks in the same thread.

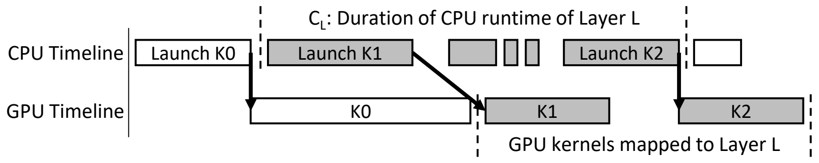

C_L: Duration of CPU runtime of Layer L
CPU Timeline
Launch K0
Launch K1
Launch K2
GPU Timeline
K0
K1
K2
GPU kernels mapped to Layer L

Figure 4.3: The mapping of GPU kernels to a layer. CUPTI provides correlations between CUDA launches and GPU kernels.

Sequential order of GPU tasks in the same CUDA stream. GPU kernels belonging to the same CUDA stream are executed sequentially. Similar to CPU tasks, the order of GPU tasks in the same stream does not change between executions. Hence, two consecutive GPU tasks in the same CUDA stream have a dependency between them.

Correlation from CUDA APIs to GPU kernels. Each GPU kernel or CUDA memory copy has a corresponding CPU-sided CUDA API (`cudaLaunch`, `cudaMemcpy`, or `cudaMemcpyAsync`) that triggers the GPU task. CUPTI provides a correlation ID for every CUDA API and GPU kernel. A GPU kernel is dependant on a CUDA API if they share the same correlation ID.

CUDA Synchronization. A CUDA synchronization API (e.g., `cudaDeviceSynchronize`) is invoked on CPU, and returns after GPU kernels (or CUDA memory copies) that are launched before this synchronization complete. A CUDA synchronization therefore generates dependency from a GPU task to a CPU task. Similar to CUDA synchronizations, even though a `cudaMemcpyAsyncDtoH` call returns before a memory copy completes, we found it still blocks the CPU until all previous GPU kernels on the same stream are completed.

Communication. Mainstream frameworks including PyTorch and MXNet implement the wait-free backpropagation strategy [261] to schedule gradient communication. Here, a communication primitive is launched as soon as the weight gradients are ready, thus overlapping communication with the backward phases of subsequent layers. Hence, we need to know the runtime of DNN layers (not just kernels) to determine which tasks trigger communication.

4.4.3 Mapping Tasks to Layers

The task-to-layer mapping enables Daydream to construct the dependency graph for distributed training, and provides necessary domain knowledge for Daydream to model DNN optimizations. Figure 4.3 shows how Daydream determines which tasks belong to a certain layer. Let L be the forward phase of a DNN layer. Daydream collects the CPU and GPU runtime information using CUPTI [174], as well as timestamps before and after the forward, backward, and weight update phases for each layer. The start and end timestamps of L will determine the CPU runtime of L (denoted by C_L). To determine the GPU runtime of L, Daydream gathers all CUDA launch calls invoked during C_L. With CUPTI providing the correlations between CUDA launch calls and corresponding GPU kernels, Daydream can identify all the GPU kernels launched during C_L, and map these kernels to L. This process can also be applied to the backward or weight update phases of any layers, and can be further generalized to any code region of interest in the framework or user-level programs.

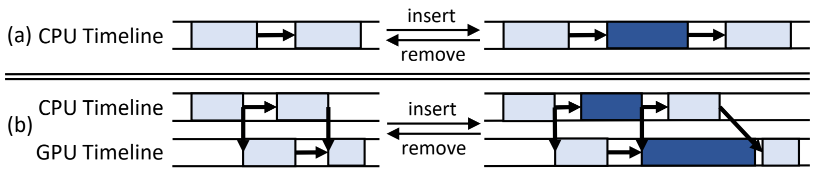


Figure 4.4: Insert/Remove a (a) CPU task; (b) GPU task.

4.4.4 Graph Transformation

What-if analysis by transforming the graph and simulating the execution requires input about the optimizations from programmers. Daydream provides a set of primitives for programmers to model DNN optimizations by modifying the graph. Like most what-if analysis in non-ML contexts, modeling DNN optimizations requires potentially shrinking or scaling the duration of tasks (the `shrink/scale` primitives). We carefully study common DNN optimization techniques and identify the following primitives (besides the `shrink/scale` primitives), which are sufficient for programmers to describe those optimizations.

Insert/Remove a task. Inserting a task to an execution thread just involves an appending of a node to a linked list. Figure 4.4 shows how this process works. When inserting a GPU task, we need to insert the corresponding CPU tasks that launch it. Which CPU tasks to insert and their duration depend on the framework implementation, and can be inferred based on collected traces.

Select. This operation allows users to select tasks of interest for further operations. One potentially useful selection criterion is select-by-layer, as many optimizations are depicted based on DNN layers. Another potentially useful criterion is to select by keywords in task names, based on knowledge of the software library (e.g., cuDNN [40]). For example, kernels with keywords such as `elementwise` or `PointwiseApply` in the names are element-wise arithmetic operations. These kernels are typically *not* compute-bound, and could be much shorter than their corresponding CUDA launch calls. Similarly, kernels with `sgemm` string in names are compute-bound matrix-multiplications.

Schedule. The `schedule` function picks one task from a set of frontier tasks that are ready to execute (line 9 in Algorithm 1). By default, it picks the task with the earliest start. Programmers can override this function and implement any custom scheduling policy. which is useful to model optimizations that increase computation-communication overlap.

4.5 Modeling Optimizations

To demonstrate that Daydream is able to estimate the performance of the most common optimizations in DNN training, we select ten techniques from Table §4.1 with different optimization goals. We show that we can easily model these optimizations using the primitives Daydream provides.

4.5.1 Optimizations for Evaluation

We select the following five DNN optimizations, which we are able to acquire the implementations, to evaluate Daydream's prediction accuracy. We use implementations from the authors of these optimizations in cases where they were not readily available.

Automatic Mixed Precision (AMP). We aim to predict the efficacy of the AMP optimization [153], implemented using NVidia's Apex package [162]. We expect that AMP will improve memory-bounded GPU kernels by 2× because the number of transferred bits is halved. With Tensor Cores in the Volta and Turing architectures, AMP empirically yields up to 3× speedup on the most compute-intensive workloads [175]. To predict AMP performance, we simply `select` all the compute-intensive (e.g., `sgemm`, `conv`) kernels and memory-bounded (e.g., `elementwise`, `batchnorm`, `RELU`) kernels, and `shrink` their duration by 3× and 2× respectively.

FusedAdam Optimizer. We use the FusedAdam optimizer [163] implemented in NVidia's Apex package [162] as an example for the kernel fusion optimization. This optimizer fuses all kernels in one weight update phase into one unified kernel. It is applicable to the models that use the Adam optimizer (e.g., GNMT, BERT). Daydream uses the kernel-to-layer mapping to identify the CPU/GPU tasks that belong to a weight update phase. We `remove` all these tasks, then `insert` a new GPU task whose duration is roughly estimated by the sum of all removed compute-intensive kernels.

Reconstructing Batchnorm. Recently Jung et al. [119] proposed a technique that optimizes non-convolutional layers in state-of-the-art CNNs. It first splits each batch normalization layer into two sub-layers, then fuses the first sub-layer with the previous convolutional layer, and the second sub-layer with the following activation and convolutional layers. We `remove` the affected activation kernels when estimating performance, since they are memory-bound kernels now fused with compute-intensive convolutional kernels. For the batch nomalization layers, we estimate that the GPU kernels will be improved by 2× since this optimization halves the amount of input data that these layers load from GPU memory.

Distributed Training. Using Daydream we can accurately predict distributed training performance with the profile based on the single-GPU environment. We evaluate Daydream's prediction based on PyTorch, which uses collective communication primitives from the NCCL library [169]. PyTorch groups gradients from multiple layers into buckets before transferring them. Hence, to predict distributed training performance, we need to `insert` one allReduce task for every bucket. The dependencies of the inserted tasks are determined based on the layer-to-bucket mapping (which requires additional instrumentation to the PyTorch framework).

Priority-Based Parameter Propagation (P3). P3 [107] is a technique that optimizes communication overhead by slicing and prioritizing. We evaluate Daydream's prediction of P3 based on MXNet, which uses the parameter-server mechanism [133]. In order to model parameter slicing, we `insert` multiple push task and pull tasks between the backward and the forward GPU tasks for each layer. The duration of the push/pull task is calculated from the slice size and the network bandwidth. To model the priority scheduling, we override the `schedule` function with a priority queue.

4.5.2 Modeling Additional Optimizations

In addition to the above optimizations, we show that Daydream is capable of modeling an additional set of diverse DNN optimizations.

BlueConnect. BlueConnect [43] optimizes communication by decomposing the allReduce primitives into a series of reduce-scatter and all-gather primitives. These primitives run concurrently as they use parallel communication channels. To predict the performance of BlueConnect, instead of `insert`ing regular allReduce or push/pull tasks, we need to `insert` reduce-scatter and all-gather tasks, and assign them to corresponding network channels (the duration can be estimated according to formulas shown in [173]).

MetaFlow. MetaFlow [112] is a layer-fusion technique to optimize DNN training by fusing DNN layers to simplify the DNN topology. We `select` the GPU kernels of substituted layers, `remove` them, and `insert` GPU kernels of new layers to predict the performance of MetaFlow in Daydream. The new layers are mostly existing layers with different dimensions; their GPU kernel durations can be inferred by profiling.

vDNN. Virtualized DNN [206] reduces GPU memory consumption by temporarily offloading intermediate data from GPU memory to CPU memory. The offloaded data needs to be prefetched back to GPU to perform execution, which causes potential performance overhead due to PCIe traffic or late prefetching. To predict the performance overhead using Daydream, we only need to `insert` additional CUDA memory copies, and override the `schedule` function to implement a custom prefetching policy.

Gist. Gist [106] reduces GPU memory consumption by storing encoded intermediate data and decoding before the data is used. The encoding and decoding introduces performance overhead. We `insert` extra encoding and decoding GPU kernels (along with `cudaLaunchKernel` calls in CPU) to estimate the performance overhead in Daydream. The duration of the inserted encoding/decoding kernels can be estimated using existing element-wise kernels.

Deep Gradient Compression (DGC). DGC [137] is a technique that reduces communication overhead by compressing the gradients. To estimate performance, we: (i) `scale` the duration of communication; (ii) `insert` the GPU tasks of compression and decompression. The duration of inserted GPU tasks can be estimated according to the compression rate and duration of existing element-wise GPU kernels.

4.6 Evaluation

4.6.1 Methodology

We implement Daydream based on three mainstream DNN frameworks: PyTorch [187], MXNet [36], and Caffe [109]. We add CUPTI [174] support to each framework to obtain traces of CUDA APIs and GPU kernels. We also add instrumentation to the frameworks to acquire layer-wise timestamps for the kernel-to-layer mapping process, and communication information such as the size of each allReduce call and their dependencies with other layer-wise computation.

Infrastructure. We evaluate Daydream's runtime prediction on a cluster of four machines. Each machine contains one AMD EPYC 7601 16-core processor [13], and four 2080Ti GPUs [168] with 11GB GDDR6 memory each, connected through PCIe 3.0 [6]. Our experiments are based on Ubuntu 16.04, CUDA v10.0 [165], cuDNN v7.4.1 [167], and NCCL v2.4.2 [169]. Our software implementation is based on PyTorch v1.0, MXNet v1.1, and Caffe v1.0.

Models. Table 4.2 shows the DNN models and datasets we use to evaluate Daydream. We select five DNN models from three different applications, covering a diverse set of DNN models. For the BERT model, we evaluate both ”base” and ”large” versions. The difference between these versions is that the ”base” version contains 12 ”Transformer blocks” (the main layer type in BERT) where as the ”large” version contains 24.

Application	Model	Dataset
Image Classification	VGG19 [221]	ImageNet [58]
	DenseNet-121 [99]	
	ResNet-50 [87]	
Machine Translation	GNMT [246]	WMT16 [3]
Language Modeling	BERT [59]	SQuAD [199]

Table 4.2: The models and datasets we use in this paper.

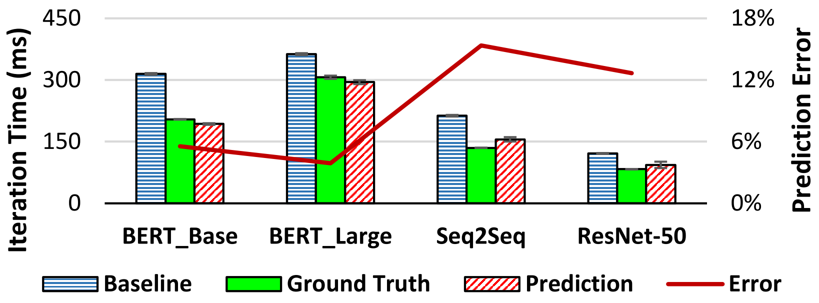


Figure 4.5: AMP – comparing baseline (FP32), ground truth with mixed precision, and predictions by Daydream.

4.6.2 Automatic Mixed Precision (AMP)

We evaluate Daydream's prediction accuracy of AMP [153], which is implemented in NVidia's Apex package [162] based on the PyTorch framework. Figure 4.5 shows the performance of using AMP and the corresponding performance prediction given by Daydream. Our predictions have errors below 13% for all the models we evaluate.

Our experiments show that using AMP brings speedups generally less than 2× – much less than the theoretical boost of using AMP for individual kernels (e.g., 3×). To understand how AMP improves performance, we break down the overall runtime into the following three components:

CPU-only runtime. This component refers to the runtime when the CPU is busy, but the GPU is not executing any kernels. It is straightforward to calculate this runtime by simply subtracting all GPU kernel runtime from the total runtime.

GPU-only runtime. This component refers to the runtime when the CPU is waiting for the GPU kernels to complete. It includes not only the duration of CUDA synchronization APIs, but also the `cudaMemcpyAsync` calls of all the device-to-host CUDA memory copies.

CPU+GPU parallel runtime. This component refers to the runtime when both CPU and GPU are busy. We calculate this part of runtime by deducting the CPU-only and GPU-only parts

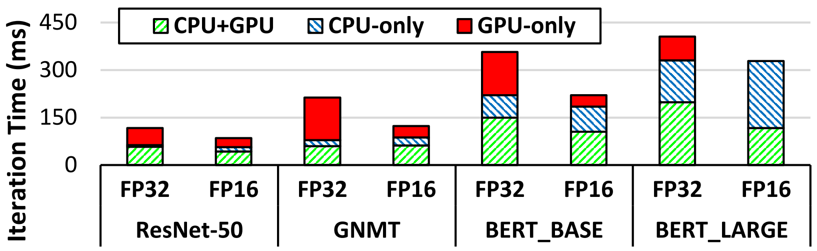


Figure 4.6: Runtime breakdown of the baseline (FP32) and mixed precision (FP16).

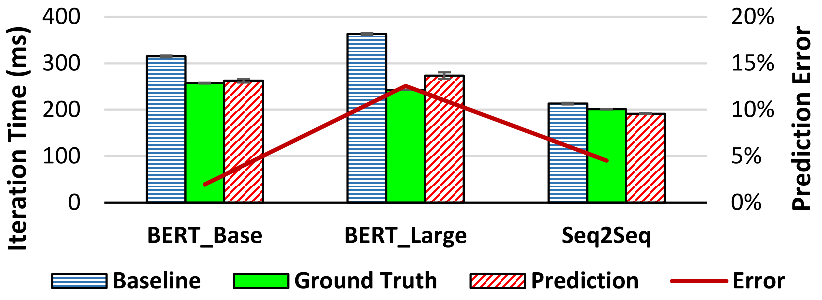


Figure 4.7: FusedAdam - comparing baseline (FP32), ground truth with FusedAdam, and predictions by Daydream.

from the total runtime.

Figure 4.6 shows the runtime breakdown of the models we evaluated. CPU runtime generally becomes the new performance bottleneck in the models that incur limited speedups (e.g., BERT$_{\text{LARGE}}$). When applying AMP, the CPU bottleneck increases, because the GPU runtime becomes shorter and part of the CPU+GPU parallel runtime is shifted to the CPU-only runtime. The overall runtime improvement comes mostly from the reduction of GPU-only runtime while CPU runtime barely changes. This demonstrates the necessity of the kernel-level abstraction when predicting performance.

4.6.3 FusedAdam Optimizer

We apply the FusedAdam optimization to the BERT and GNMT models as they use the Adam optimizer. Figure 4.7 shows the performance of using the FusedAdam optimizer. Our predictions are within 13% of the ground truth runtime.

There are two reasons why the FusedAdam optimizer substantially improves the performance of BERT models. First, unlike most DNN training workloads, the weight update phase is a significant proportion of a BERT model's iteration runtime (around 30% for BERT$_{\text{BASE}}$ and 45% for BERT$_{\text{LARGE}}$). Second, the weight update phase consists of very many element-wise GPU kernels (2633 for BERT$_{\text{BASE}}$, 5164 for BERT$_{\text{LARGE}}$). Thus, the CUDA launch calls on the CPU become the main bottleneck. The FusedAdam optimizer almost eliminates all CPU kernel launch overhead in the weight update phase by fusing all GPU kernels into one single GPU kernel. Compared to BERT models, the GNMT model spends less than 10% of its iteration time on the weight update phase, explaining the lower speedup improvements.

4.6.4 Reconstructing Batchnorm

We evaluate our performance prediction for the optimization of reconstructing batch normalization [119] based on the Caffe implementation of DenseNet-121 [99]. Using Daydream, we predict that reconstructing batchnorm will yield a moderate performance improvement of 12.7% compared to the baseline. This suggests that reconstructing batchnorm in our configuration is less promising than the paper claims (17.5% speedup). We verify this conclusion by testing the ground truth implementation of reconstructing batchnorm, and find out that this optimization yields even lower 7% speedup.

We notice that there are two main reasons for the difference between our prediction and the ground truth. First, the ground truth uses a completely new implementation of the batchnorm layers, and it is hard to precisely predict the runtime of newly implemented kernels. Second, the ground truth implementation introduces new CUDA memory copies and allocations, which add performance overhead. Obtaining a very precise estimate would require us to understand not just the high-level idea from the paper, but also the detailed implementation of the user-level programs and the Caffe framework.

4.6.5 Distributed Training

Next we evaluate distributed training using PyTorch with the NCCL [169] library. Figure 4.8 shows the comparisons between runtimes predicted by Daydream and the measured ground truth runtimes, for each DNN model under different system configurations. We evaluate the prediction accuracy for Ethernet and InfiniBand connecting multi-machine systems under different network bandwidths (10, 20, 40 Gbps). In most of the configurations, Daydream predicts distributed runtime with at most 10% prediction error, with a few exceptions for the 20Gbps and 40Gbps configurations.

The prediction errors of the overall iteration times are mainly due to inaccurate estimates of individual NCCL primitives. Figure 4.9 shows the comparisons of NCCL allReduce calls between the ground truths and predictions. The ground truths are on average 34% higher than the theoretical values.

An NCCL primitive is both a communication primitive and a GPU kernel, suggesting that it could be bottlenecked by two types of hardware resources: (i) the network bandwidth, and (ii) GPU resources (e.g., memory bandwidth, streaming multiprocessors). Figure 4.9 shows that the predicted values are very close to the runtimes measured when running NCCL primitives exclusively. This suggests that the ground truth is slower because they compete for GPU resources with other GPU kernels. Based on this insight, we try to reduce this interference by adding CUDA synchronizations before invoking NCCL primitives. As shown in Figure 4.9, adding synchronizations improve the NCCL primitives by 22.8% on average when compared to the baseline.

We also verify the impact to the overall iteration time when adding synchronizations before NCCL primitives. We run the experiments on all the configurations shown in Figure 4.8. We find that this simple approach does not lead to performance degradation in any configuration. Instead, it could bring an improvement of up to 22%.

4.6.6 Priority-Based Parameter Propagation

We evaluate Daydream's prediction accuracy of applying Priority-Based Parameter Propagation (P3) to VGG-19 and ResNet-50. To reproduce the performance speedups of P3, we use a cluster of four machines with one P4000 GPU per machine (which is consistent with the evaluation setup of the P3 paper [107]). We use MXNet v1.1, and have one worker process and one parameter server process on each machine.

Figure 4.10 shows the iteration time of the baseline, ground truth, and prediction using Daydream under different bandwidths. Our prediction faithfully reflects the trend of P3 speedups when the network bandwidth increases. The prediction error is at most 16.2% among all the configurations we tested, and lower in most of the configurations.

We overestimate the speedup of P3, especially when training VGG-19 with a 15 or 20 Gbps network bandwidth. The reason is similar to our previous insight about NCCL primitives: when bandwidth is higher, a communication task is increasingly bottlenecked by non-network resources. In the case of MXNet, this overhead could be caused by the server processes, or the control flow of the worker processes.

4.7 Related Work

To help programmers understand the performance of the hardware accelerators and develop highly efficient applications, hardware vendors provide profiling tools (e.g., NVProf [171], Nsight [170], and vTune [202]) that can reveal low-level performance counters (e.g., cache hit rate, memory speed or clock rate). These tools are usually designed with general applications in mind, and expose hundreds of low-level performance counters. The fundamental limitation of all these tools is that they do not utilize application-specific knowledge.

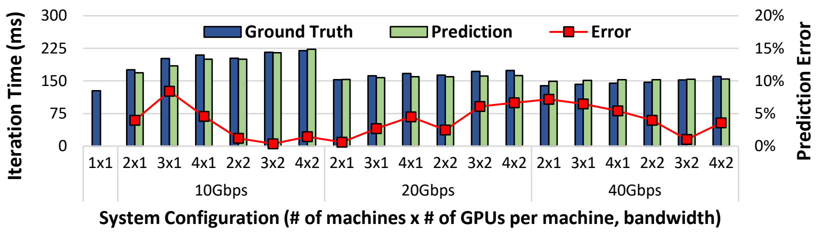
The new generation of profiling tools feature the *application-aware* property, enabling them to deliver domain-specific (e.g., ML-specific) insights about performance to programmers. The Cloud TPU Tool [69] is an example of such a profiling tool. It correlates low-level TPU metrics with the DNN structure, and shows the performance for each DNN layer. Similarly, MXNet [36] and PyTorch [187] also have their own built-in profiling tools. These domain-specific tools can highlight performance hotspots, but are less efficient in finding optimization opportunities. In contrast, Daydream is not only *application-aware*, but also *optimization-aware*, enabling Daydream to quantitatively estimate the efficacy of different optimizations without fully implementing them.

Prior works have tried to explore what-if questions in other contexts by using low-level traces. Curtsinger *et al.* proposed a causal profiler (COZ [52]) to identify potentially unknown optimization opportunities by running performance simulation with certain functions being virtually speed-up. Unlike Daydream, COZ does not require dependencies among functions because it does not consider the cases where functions can be added or deleted (which is the case for many ML optimizations). Pourghassemi *et al.* uses the idea of COZ to analyze the performance for web browser applications [193]. For data analytic frameworks, such as Spark [260], Ousterhout *et al.* use dependency analysis to understand the overhead caused by I/O, network, and stragglers [179, 180]. Daydream is designed to address a more diversified set of what-if questions, and hence requires more powerful modeling.

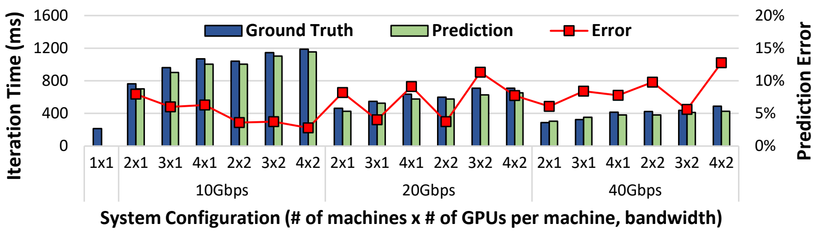
Prior works address what-if questions of the form ”What if we can speedup task T by N times (or infinity)?”, but they do not study whether existing optimizations can deliver this speedup. In the ML context, given an optimization, accurately predicting the performance of individual tasks in the dependency graph, is still an open problem. It requires additional knowledge about the kernel implementation and the architecture design. Currently Daydream can not automatically estimate the runtime of new GPU kernels. However, as we show in Section §4.6, even with rough estimates of per-kernel duration based on domain knowledge and reasonable assumptions, we can still achieve high overall prediction accuracy.

4.8 Summary

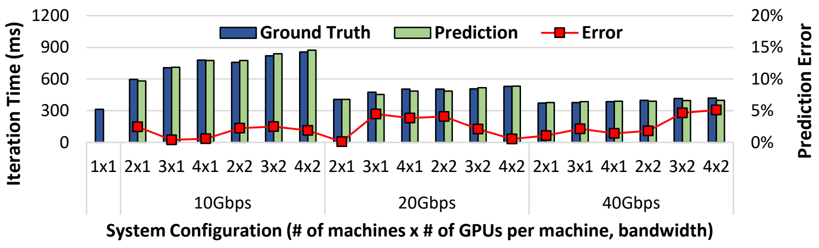
The efficacy of DNN optimizations can vary largely across different DNN models and deployments. Daydream is a new profiler to effectively explore the efficacy of a diverse set of DNN optimizations. Daydream achieves this goal by using three key ideas: (i) constructing a kernel-level dependency graph by utilizing vendor-provided profiling tools, while tracking dependencies among concurrently executing tasks; (ii) mapping low-level traces to DNN layers in a synchronization-free manner; (iii) introducing a set of rules for programmers to effectively describe and model different optimizations. Our evaluation shows that using Daydream, we can effectively model (i.e. predict runtime) the most common DNN optimizations, and accurately identify both optimizations that result in significant performance improvements as well as those that provide limited benefits or even slowdowns.



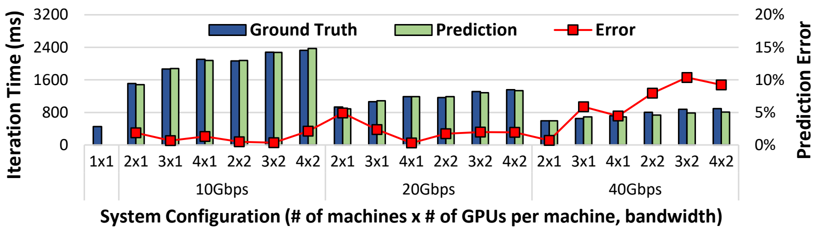
(a) Runtime predictions for ResNet-50.



(b) Runtime predictions for GNMT.



(c) Runtime predictions for BERT$_{\text{BASE}}$.



(d) Runtime predictions for BERT$_{\text{LARGE}}$.

Figure 4.8: The error between Daydream's runtime predictions and the baseline with synchronization before each allReduce under various system configurations.

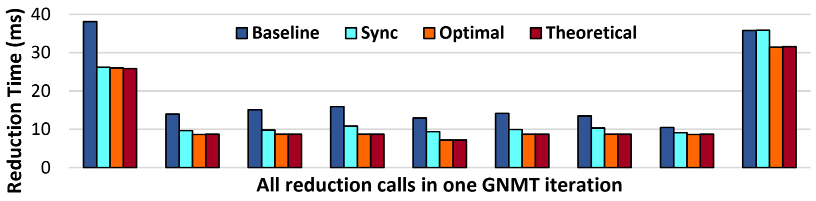
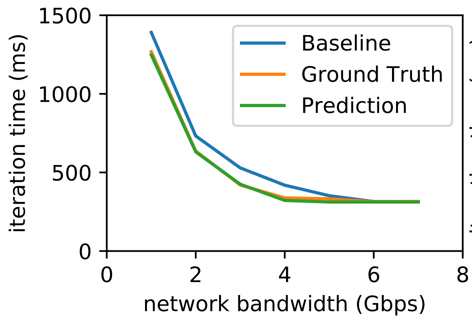
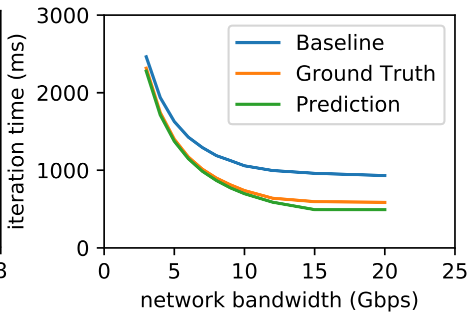


Figure 4.9: Comparison of all individual reduction runtimes in one training iteration of GNMT. **Baseline**: runtime measured in regular training; **Sync**: runtime measured with an additional CUDA synchronization before each reduction; **Optimal**: runtime measured when executing exclusively; **Theoretical**: runtime calculated using the formula [173].



(a) ResNet-50.



(b) VGG-19.

Figure 4.10: Daydream's prediction for how the P3 optimization will help under different network bandwidths.

Chapter 5

Sokoban: White-Box Fast Tensor Compilation

5.1 Introduction

Deep neural networks (DNN) are widely used in a wide range of tasks, including image classification and natural language processing. A DNN model is usually modeled as a data flow graph, where each node in the graph is an operator with input and output tensors. A wide range of hardware accelerators, such as GPU, TPU [117], and GraphCore IPU [105], for different use cases (e.g., server, phone, edge devices) are quickly evolving to meet the growing computing demand of DNN models. To support high performance DNN computations, it is important to use a kernel implementation for a set of DNN operators that can run on the accelerators efficiently. Such kernel implementations comprise of the basic building blocks in existing DNN frameworks and compilers [39, 265, 227].

However, generating efficient kernel implementation for thousands of types of DNN operators on a growing list of hardware accelerators remains as a challenge, both to hardware vendors and to DNN compilers. Hardware vendors, such as Nvidia, offer libraries like cuDNN [40] and cuBLAS [164] for efficient implementations of the most popular operators. However, offering accelerator-specific kernels requires significant engineering efforts and is vendor specific. Moreover, it is hard for the hardware vendors to keep up with the rapid pace of development of new DNN operators and techniques.

To overcome the limitation of vendor specific approach, DNN compilers offer a general-purpose way to generate kernel implementations of different operators, including custom ones, for a target accelerator [39, 266, 265]. DNN compilers typically transform each operator's computation into a handcrafted code template that defines a program optimization space, and leverages a black-box machine learning algorithm to search for a good kernel implementation based on the feedback on each search point evaluated on the target device. Although existing compilers can produce good kernel implementations for the majority of DNN operators, their approach also has some fundamental limitations:

1) Long compilation time. The search space defined by the code template is often very large. Hence, the compilation of a single operator usually requires hundreds to thousands of evaluation steps, all of which must be run on the target device and could cost hours of compilation time. We conduct an experiment (see §5.2.1). Such an approach is especially challenging for DNN models with frequently-changing configurations, for example AutoML generated DNN structures and models with dynamic input shapes, where a small change of the configuration may necessitate hours of re-tuning to achieve reasonable performance.

2) Performance sensitive to resource changes. The machine learning based approach frequently over-fits the kernel to the hardware resource conditions that exist at compilation time. As a result, if the hardware resources shift dynamically during kernel execution (e.g., the memory cache could be polluted by other concurrent kernels), the performance can significantly degrade. We conduct a small experiment (see §5.2.2). The performance degradation under dynamic environments also makes it difficult to reuse the compiled kernel on another device, even if these devices are only slightly different, e.g., two GPUs with a different number of streaming multi-processors.

Implicitly, prior machine learning compilers assume that the computational patterns of DNN models are inherently complex and the optimal parameters of different software and hardware settings for a specific operator can only be "learned" by black-box machine learning algorithms. In contrast, we make the observation that DNN operators can be modeled analytically, and that they typically follow a *regular* computation pattern and their behaviors on DNN accelerators are largely *deterministic*. We propose RATIONAL, a deterministic element-wise data computation and movement model that views a DNN operator as multiple parallel and homogeneous computation tasks. Each one of the tasks loads a specified data tile through the memory hierarchy of the hardware accelerator, performs computation in a parallel computation unit in the accelerator, and stores the result back through the memory hierarchy.

We design SOKOBAN a compiler that utilizes the RATIONAL model to compile operators to hardware, achieving both low compilation time and high performance. Following RATIONAL, SOKOBAN can implicitly adjust the hardware resource usage like register, memory bandwidth, and GPU cores by changing the size of the data tile in each parallel task. Therefore, given a hardware specification like computation capacity (in FLOPS), memory bandwidth (in b/s), plus the tile size of a given operator, SOKOBAN can use RATIONAL to calculate it is memory bound or computation bound, following the well-known roofline model [244]. As a result, SOKOBAN can not only generate high-performance kernel implementations *within seconds*, but also uncover the behavior of an operator in the target accelerator, identifying which part of the data computation and data movement is bottlenecking on which component of the hardware.

In contrast to prior black-box approaches, SOKOBAN's RATIONAL model is not coupled with a specific hardware vendor and a specific operator. SOKOBAN can quickly adopt the model to new hardware and operators by just changing a few parameters (§5.3) and still produce high-performance kernels within seconds, thus remaining robust to changes in the hardware environment. This is particularly valuable to new players in the computing accelerator market, who usually need to spend significant engineering efforts to provide efficient kernel implementations.

Our evaluation on 20+ typical DNN operators and 4 different kind of end-to-end DNN models shows that SOKOBAN can generate operator code with comparable performance to both vendor-provided DNN libraries and state-of-the-art tensor compilers. SOKOBAN can significantly reduce the compilation time from hours to seconds or less. This grants SOKOBAN the ability to support AutoML scenarios where operator definitions are frequently changing. More importantly, SOKOBAN's flexibility allows it to adapt to different accelerators like AMD GPU and Graphcore IPU, and provides information on whether an operator is approaching the theoretic limit on a particular accelerator.

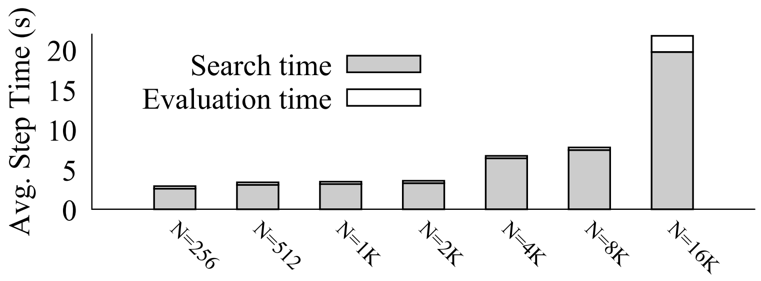


Figure 5.1: The average searching time (in hours) and evaluation time of each step in Ansor when compiling matrix multiplication operator with varying sizes.

Therefore, RATIONAL and Sokoban could potentially be used to guide the development of new hardware accelerators, indicating whether a particular hardware component (computation core vs. memory bandwidth) should be enhanced or can be reduced (to save costs) for a particular DNN workload.

5.2 Background

In this section we highlight some results to illustrate the limitation of existing search-based tensor compilation approach. Without loss of generality, we experiment with Ansor [265], the state-of-the-art DNN compilers, on an NVIDIA V100 GPU.

5.2.1 Long Compilation Time

Prior tensor compilers treat the underlying hardware as a black box executor of kernel programs with various schedules, leading to the following two limitations.

First, **the schedule space is huge**. There are many factors that could potentially affect the kernel program performance (e.g. loop tiling, unrolling, reordering, etc.). These factors are mostly mutually orthogonal, and are in general all included in the schedule space, so that the compiler is able to generate tensor programs with optimal performance. As a result, the size of the schedule space of a DNN operator is often huge (e.g., greater than than 10^{11} [266]), requiring long time for the further search progress (e.g. genetic algorithm or simulated annealing) to find the optimal schedules.

Second, **the cost model needs to be based on learning**. An accurate cost model is essential to quickly estimate the performance of a schedule during the compilation process. Both TVM [39] and Ansor [265] adopt learning-based cost models. The learning process itself requires substantial amount of training data, which are generated by compiling and executing kernel programs on hardware (trials), with different arithmetic on massive amount of schedules. Moreover, as the kernel performance is sensitive to both the arithmetic and hardware, the models learned based are often on different hardware cannot be directly transferred to each other, making it harder to reduce the number of trials during the compilation. The huge schedule space increases the complexity of the cost model itself, making it more difficult to train (i.e. requiring more trials for training data).

The direct consequence of these limitations is that such approaches usually require a long compilation time. Our experiments of compiling a set of common DNN operators shows that the average

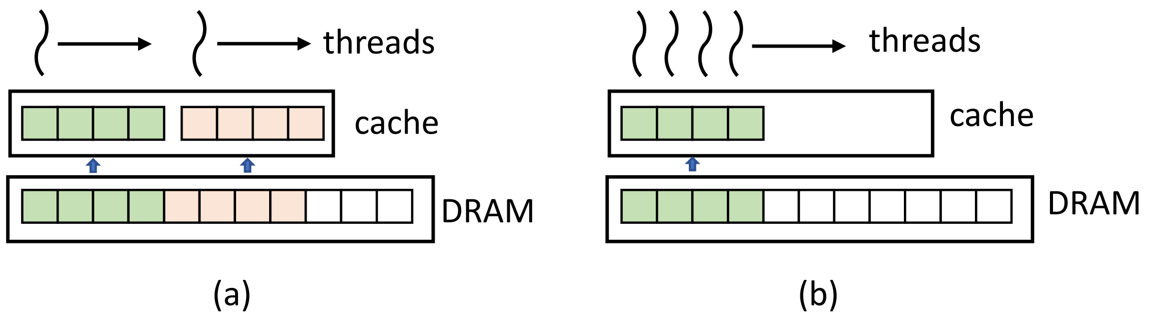


Figure 5.2: Illustration of different implementations of an element-wise operator: (a) each thread computes a contiguous range of elements, and (b) a group of threads compute a contiguous range of elements at a time.

time of compiling a single convolution operator with Ansor is 1.6 hours. Moreover, the search space is usually increased with the increasing of operator size, which is a common trend to use larger DNN models recently, e.g., BERT [59], GPT [31], etc. Our experiments on compiling several $[N, N] \times [N, N]$ matrix multiplication operators with different size of N, show that the searching time of each evaluate step is also significantly increased with the matrix size, due to the increased search space, as shown in Figure 5.1. This leads to adopting such compilations in the ever-growing large models quite challenging.

5.2.2 Resource-Sensitive Performance

The machine learning based approach could frequently overfit the kernel to the hardware resource conditions that exist at compilation time. Taking a simple element-wise operator as an example, there are two kernel implementation candidates. In the first one, each thread processes K contiguous elements. Since memory is usually accessed at a transaction granularity, each concurrent thread will try to cache its data of a translation into the fast memory and then processes them iteratively, shown in Figure 5.2(a). Contrarily, the second implementation processes a range of contiguous elements by a group of threads, with each processing one at a time, shown in Figure 5.2(b). These two implementations are both in the search space of Ansor by searching the schedule of K. When the input tensor size is small, i.e., all the translation data accessed by the concurrent threads can be cached in fast memory, there is no evident performance difference for this two implementations. Even the first candidate may be a better choice, as it will lead to a higher cache hit rate, which is usually a good indicator for a ML-based model. However, if we transform this kernel to a larger input tensor, note that this is equal to running multiple kernels on small tensors concurrently, the second kernel implementation will significantly outperform the first one. This is mainly because the cache size is unable to hold all data and some of them are evicted and re-loaded for multiple times.

Figure 5.3 shows the performance comparison of the two kernel candidates, the first one is tuned by Ansor with input size of 8K, where ”K=8” stands for each thread processing 8 contiguous elements. The second kernel is manually implemented with K=1. When applying the two kernels on input tensor with 8K size, the Ansor searched result is slightly better than our manually implemented one, i.e., 3.7% latency reduction. However, when scaling to 512K, the second kernel significantly outperforms the first one by 2.9×. This experiment demonstrates that kernels tuned by ML-based

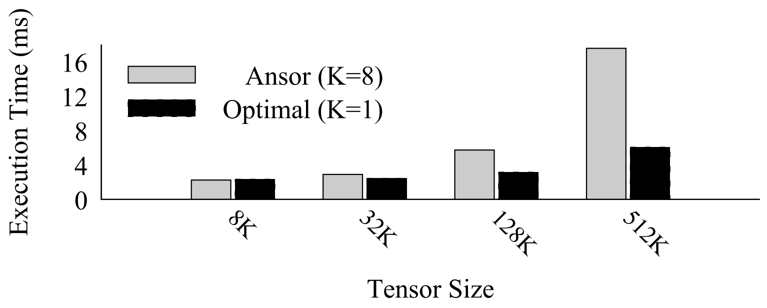


Figure 5.3: Performance of element-wise operators on different input tensor sizes. (a) Ansor tuned kernel on 8K input size, and (b) manually implemented kernel through considering hardware behavior.

approach could easily lead to bad performance with some environment changes.

5.3 RATIONAL Computation Model

The fundamental reason led to above limitations is that ML-based compilers assume that the computational patterns of DNN models are inherently complex. In contrast, we observe that DNN operators typically follow a *regular* computation pattern and their behaviors on DNN accelerators are largely *deterministic*. This section first discusses the general pattern of DNN operators, revisits the commonly-used *tiling* concept, and gives the abstraction for device in RATIONAL. We then introduces the RATIONAL model, and shows how it can be mapped to different memory architecture.

5.3.1 Abstraction for DNN Operators: Tile-Based Computation

DNN operators are usually composed of some numerically-intensive computation (e.g., matrix multiplication) over tensors. Prior work [39, 265, 266] has shown that common DNN operators or their combinations can be expressed as index-based expressions (such as the TVM IR [39] or Halide IR [198]), i.e., each indexed element in the output tensor can be calculated by taking corresponding indexed elements in input tensors. This implies the following properties of common DNN operators:

- **Data Parallelism**. No data dependencies exist between any two output elements. This suggests that the computation of two different output elements can be performance independently and concurrently, allowing massive parallelism in computing the entire output tensor.
- **Deterministic Computation**. The amount of data loaded and compute for one output element do not change for different addresses or values of input elements. Similarly, given a size of an output subtensor, the compute and memory workloads remain identical regardless of the addresses and input values.

Such properties allows the computation of DNN operators to be partitioned as many parallel and homogeneous tiles, each of which moves a data tile through the memory hierarchy of the hardware accelerator, performs computation in a parallel computation unit in the accelerator, and stores back the result through the memory hierarchy. As the computation of different tiles can be performed independently and concurrently, these tiles can be easily mapped to the massive-parallel architecture in modern DNN accelerators. We can also define a tiling schedule by just the shape of a subtensor of the output, each tile computes the output elements within one subtensor.

In reality, there are two motivations for partitioning the computation of a tensor into tiles. First, smaller tiles require less memory footprint, which is necessary as high-level memory usually has low memory capacity when mapped to real hardware. Second, it usually requires a sufficient amount of data parallelism to fully utilize the massive parallel hardware resources in a DNN accelerator. On the other hand, small tiles usually have low computational intensity, which could negatively affect the performance as data movement more likely becomes the performance bottleneck.

5.3.2 Abstraction for Hardware Accelerators: Data Movement Pipeline

Data copies across the memory hierarchy has always been a major source for the performance bottlenecks. To efficiently model the data movement pattern in hardware devices, RATIONAL abstracts hardware accelerators as an unified *pipeline device*, which is composed of multiple parallel computing units (CUs). Each CU is associated with multiple memory layers (e.g., the register file, shared memory, L2 cache, and DRAM in GPUs). During the runtime of a kernel program, input data is always fetched from the low-level memory (e.g. global memory in GPUs), to high-level registers through intermediate memory levels (e.g. cache). Each memory level has its own bandwidth and capacity. Data in registers is later consumed by compute cores. The latency of data copies between any two adjacent memory levels could all potentially become the performance bottleneck of a kernel program. Such pipeline model allows SOKOBAN to easily reason about the memory bottleneck.

5.3.3 Multi-Level Tile-Based Pipeline Model

Based on the properties of the DNN operators and the data movement pattern across the memory hierarchy, our proposed model is a multi-level tile-based pipeline execution model that describes the data movement and computation during the kernel execution. Figure 5.4 illustrates the basic idea of RATIONAL in a two-level pipeline device. The input data is initially stashed in the last-level memory. The computation of the output tensor is then partitioned into tiles, and the execution of computing each tile is completely independent. During the execution of the tensor program, the kernel first fetches the input data of an output subtensor from last-level memory to the intermediate-level memory one level higher. The fetched data is then consumed by an iterative procedure, which further partitions the output subtensor into high-level tiles, and fetches the input data of each high-level tile to highest level memory (which represents registers in real hardware). The data in highest level memory is fed to compute cores, generating results and write back to lowest-level memory through the entire memory hierarchy. The execution of tiles at different memory levels are pipelined, and the performance is bottlenecked by either the computation or the data movement between two adjacent memory levels.

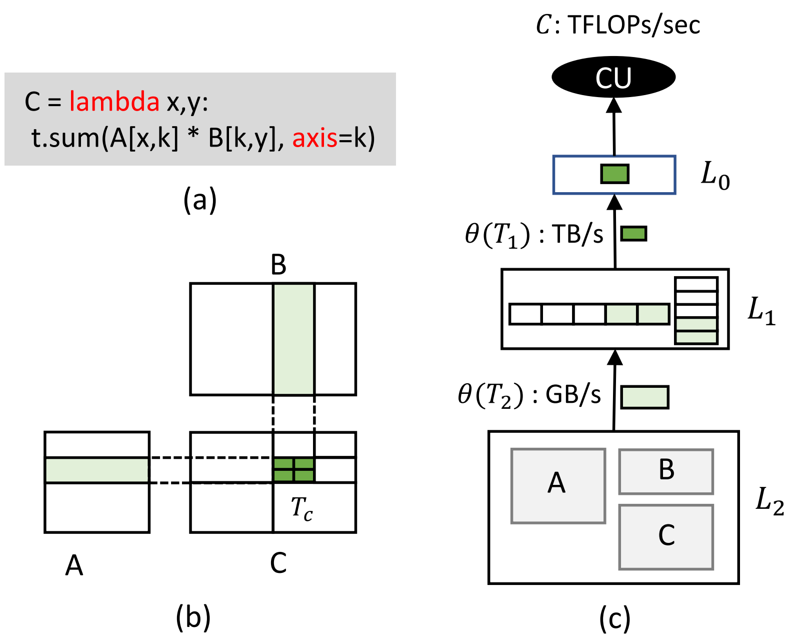


Figure 5.4: Illustration of RATIONAL computation model. (a) An example TVM IR for matrix multiplication; (b) An example of tile of a matrix multiplication operator; (c) The computation pipeline of tasks to hardware device. (CU: Computing Unit)

5.4 Sokoban Compiler

5.4.1 System Overview

Based on the RATIONAL model, we propose SOKOBAN, a tensor compiler for DNN operators that deterministically finds efficient tiling configurations and generates device code. Figure 5.5 shows the workflow of SOKOBAN. First, the input to SOKOBAN is a DNN model in the format of data flow graph of DNN operators. Each DNN operator is defined by its lambda formula, plus the shapes of input and output tensors. The *operator module* parses the arithmetic information, provides functions that are necessary for the *cost model* to accurately estimate the memory latency for each memory level, and the compute latency. The *cost model* takes both arithmetic and hardware information as input, and calculates the theoretical performance for any given tiling schedule. SOKOBAN later employs a simple search algorithm, which explores the schedule space and identify the schedule with optimal theoretical performance according to the *cost model*. Finally, the schedule is given to the underlying *code generation* module, which generates kernel programs that can be directly deployed to hardware devices. We implement the *code generation* module of our prototype system based on an augmented version of TVM.

SOKOBAN's tensor compilation is based on the RATIONAL computation model, which enables white-box performance analysis for tensor programs. An operator fed to SOKOBAN's compilation process is defined by its lambda expression and the sizes of input and output tensors. SOKOBAN then employs a general tile-based iterative code template to implement RATIONAL's abstractions for the given operator and the hardware, which is different from the loop-based code template in prior works (TVM, FlexTensor), allowing SOKOBAN to analytically estimate the performance of

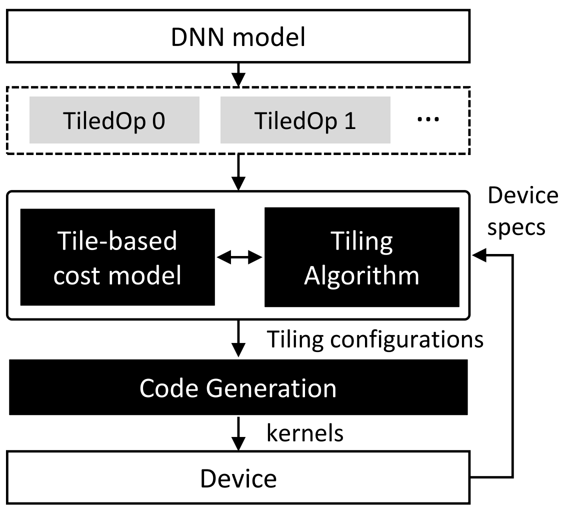


Figure 5.5: System overview of Sokoban

each schedule. With an accurate analytical cost model based on the RATIONAL model, Sokoban uses a naïve search policy to find the optimal schedule, and then generates the device code for the target hardware. We define a set of hardware specifications that are critical for accurate white-box estimation of tensor performance, so that Sokoban can generate high-performance device code for a wide range of accelerators.

5.4.2 Multi-Level Tile-Based Compilation

Based on the proposed RATIONAL model, Sokoban views the execution process of an operator as processing individual tiles, which move corresponding data subtensors throughout the pipelined device. We derive a simple tile-based iterative code template to implement this tile-based compilation paradigm, and then specify the critical hyper-parameters that are in general critical to kernel performance, regardless of the underlying hardware.

Figure 5.6 demonstrates Sokoban's iterative pseudo code for computing a binary operator on a two-level pipeline device. The main body of the template is essentially a series of nested iteration steps, which implements the execution on a pipeline device with a multi-level memory hierarchy. In each iteration step, the kernel program copies input data from its corresponding memory level to the adjacent upper-level memory level, computes and accumulates the output subtensor. The outputs accumulated at the highest-level memory (i.e. registers) are directly written back to the lowest-level memory.

Besides the partitions along the spatial axes, Sokoban also partitions the reduction axes when processing each subtensor for each iteration step. This partition can effectively reduce the memory footprints, allowing larger tile size within the memory capacity of real DNN accelerators. In summary, Sokoban's multi-level tile-based schedule consists of the **tiling sizes** and the **reduction step sizes** for each memory level. We then discuss how these schedule choices affect the memory and compute latency respectively, and hence the overall kernel performance.

```
// on each compute unit:
for ti in Output.tiles:
  // partition input tiles into chunks along the reduction axis
  for ki in T2_reduce_ids:
    // load tiles from last level memory to L1
    A_L1 = load_tile(ti.A[ki], L1, L2)
    B_L1 = load_tile(ti.B[ki], L1, L2)
    // T1_C = compute\_tile(A_L1, B_L1, L1)
    for tj in T1_C.tiles:
      for kj in T1_reduce_ids:
        // load tiles from L1 to registers
        REG_A = load_tile_reg(tj.A[kj], L0, L1)
        REG_B = load_tile_reg(tj.B[kj], L0, L1)
        // compute a tile
        T0_C = compute\_tile(REG_A, REG_B, L0)
        WriteBack(Tile_C)
```

Figure 5.6: The main loop body of SOKOBAN's template for computing an operator with one reduction axis on a two-level pipeline device. The outer iteration step is shown from Line X to Line Y, and the inner iteration step is denoted by Line X to Line Y.

5.4.3 Performance Analysis with Tile-Based Schedules

The hyper-parameters in SOKOBAN's multi-level tile-based schedules essentially summarize the arithmetic information that SOKOBAN utilizes to model performance. In this subsection we explain how the **tiling sizes** and the **reduction step sizes** affect the tensor program performance. The tiling sizes affect the performance from two aspects: *first*, the sizes determine the dimensions of input subtensors for each output tile, which decides the memory traffic inside the memory hierarchy; *second*, the sizes determine the number of parallel executed tiles, which needs to be high enough to fully exploit the massive parallel compute and memory resources in the hardware. We then use a $M \times K \times N$ matrix multiplication as an example to illustrate the impact of tiling sizes on the memory traffic:

$$C(M,N) = A(M,K) * B(K,N)$$

Let x_i, y_i be the tile size on the output tensor for memory level i in the pipeline device model, and W_i be the total number of input elements loaded for the corresponding memory level i. Computing an output tile of $C(x_i, y_i)$ requires loading input data from input subtensors $A(x_i, K)$ and $B(K, y_i)$. Hence, we can estimate W_i with the following formula

$$W_i(x_i, y_i) = (\frac{M}{x_i}\frac{N}{y_i})(Kx_i + Ky_i) = MNK(\frac{1}{x_i} + \frac{1}{y_i})$$

This formula illustrates that larger tiling sizes in general lead to less amount of memory traffic, as well as higher arithmetic intensity since the amount of FMAs (fused multiply-add) for computing the whole output tensor stays unchanged. A proper tiling configuration needs to balance the memory and compute workloads.

The actual memory traffic and computational throughput has to consider the hardware costs. SOKOBAN explicitly models the most critical factors, including unaligned memory accessing, bank conflict, unaligned thread mapping, etc. For example, loading a tile $[x, y]$ from a tensor $[m, n]$ on memory with transaction size of k, the actual memory traffic should be calculated by the number of aligned transaction accesses considering the accessing address offset, instead of the theoretical traffic of $xy \times sizeof(T)$. Similarly, mapping a tile with N elements to a CU with computing vector size of k, the maximum throughput can be achieved should be less than $\frac{N}{(\lfloor \frac{N-1}{k} \rfloor + 1)k}$ of the peak performance. We also model the cost of memory bank conflict as a penalty to the memory accessing throughput. For example, if a tensor program always has 1/2 threads accessing conflict memory banks with the rest, the actual memory throughput will be also halved.

5.4.4 Finding Efficient Schedules

We analyze the performance of the Sokoban's tensor programs under RATIONAL's pipeline device abstraction. To maximize the overall computing throughput P, we need to guarantee the slowest layer in the pipeline device, L_s, achieves its maximum throughput, by varying the tile size under the memory capacity constraint of L_s. Thus, our optimization goal is,

$$\begin{aligned} \max \;\; & P_s = \min\{P_1, ..., P_n\} \\ \text{s.t. } & P_i = \min\{\frac{W_i(T_i)}{Q_i(T_i)})\rho_i, C_{max}\} \quad Mem(T_i) \leq Cap_i \;\; \forall i \in 1, .., n \end{aligned}$$

where $Mem(T_i)$ is the total memory footprint required by T_i and Cap_i is the capacity of memory layer L_i. Assuming the overall maximum computing throughput is P^{max}, we define an *efficient task configuration* for a CU as a configuration list $\{T_1, T_2, ..., T_{n-1}\}$, where for each T_i at memory layer L_i, the corresponding peak computing throughput P_i satisfies $P^{max} \leq P_i \leq P^{max} + \epsilon$. Such a configuration would guarantee that the maximum computing throughput is achieved with a minimum memory capacity and bandwidth usage at each layer, which determines the most efficient resource configuration on a single CU. Thus, an optimal program for an operator can be obtained through partitioning into efficient tasks and then uniformly assigning them to all CUs.

5.4.5 Tiling Algorithm

Sokoban reduces the search space of a tensor program by the following three steps: 1) by modeling tensor computation with RATIONAL model, the tensor program is reduced into a specific pattern of data processing pipeline; 2) reducing the large tiling configuration space to only resource-efficient ones by modeling the efficient tiling configuration; and 3) pruning the obviously sub-optimal configurations by the cost modeling. The other configuration candidates are theoretically optimal ones proposed by Sokoban. For each configuration, we calculate the overall computing throughput P_{min} by calculating throughput of the slowest layer, and pass theoretically optimal schedules to code generation.

Notice that Sokoban's performance analysis for individual schedules is extremely cheap, which involves only calculating a series of memory/compute traffic formulas, and does not require any trials or ML-based cost models. Even without performance analysis, the amount of tiling schedules is also extremely limited (only around 10^3 for matrix multiplications on V100). This allows Sokoban's compilation to exhaustively enumerate all possible schedules and then identify the effective ones using our performance analysis, with time cost of few seconds.

5.4.6 Code Generation

Given that tensor computation in Sokoban is limited into a fixed tile-processing pipeline, this allows Sokoban to adopt an unified template to generate specific program for each operator, as illustrated by the pseudo code in Figure 5.6. Moreover, Sokoban's tiling algorithm can analytically propose a set of efficient task configurations. Combined with the code template, we can directly generate a set of (usually tens of) theoretically-optimal kernel candidates for a specific device.

In practice, not all of these candidates could perform optimal, since there could be some device-compiler and hardware related hidden factors that are not modeled in Sokoban. For example, on CUDA GPUs, Sokoban generates kernels in CUDA code and relies on nvcc [51] to compile it into machine code. However, nvcc itself will conduct some code optimizations, e.g., register allocation, which might affect our desired program execution behaviors. Sokoban addresses this challenge by introducing a *parallel kernel profiler* to quickly evaluate several candidates with the best theoretical performance and select the optimal one. Note that this evaluation has fundamental differences with the ones in a search-based compiler. First, the search-based approach usually requires hundreds even thousands of sequential evaluation steps, while Sokoban just needs a profiling on tens of candidates, thanks to the reduced code space by our computation model and the analytic-based tiling algorithm. More importantly, these evaluation candidates can run in parallel in $O(1)$ time if there are sufficient computing resources. Second, Sokoban's cost modeling is extensible to add more factors as long as hardware vendors could provide more performance counters. Moreover, if the device has interface to allow Sokoban to directly generate low-level code, e.g., assembly code, it could also largely avoid the non-deterministic from device compiler.

5.5 Implementation

To estimate the performance of tiling configurations and eventually identify an efficient tiling configurations, Sokoban has to rely on performance-critical arithmetic functions (e.g., the amount of data loaded and FMAs under a given tile configuration) and hardware counters (i.e., peak FLOPS, memory bandwidth, memory transaction size). In this section, we will explain in details how we implement the abstractions for both the operators and the hardware, and demonstrates how Sokoban can fit to diverse architecture like GPUs and IPUs. We will also explain how we utilize the arithmetic and hardware information to build an accurate cost model.

5.5.1 Operator and Hardware Modules

We implement the *Operator* and *Hardware* modules to capture the RATIONAL's abstractions for computation and device. The *Operator* module extracts arithmetic information from end-user programs and provides functions for the cost model to evaluate compute and memory workloads. For the device abstraction, it defines a set of hardware specifications that allows users to implement the mapping from the RATIONAL model to the device, and provide the cost model with necessary hardware counters for accurate performance estimation.

Operator Interfaces As stated in Section 5.4, Sokoban's compilation process takes the arithmetic information from the end-user programs, which specifies the dimensions of input and output tensors, as well as the lambda expression for the operator. The lambda expression defines the data dependencies between input and output elements. This enables the *Operator* module to provide the following three functions, which are necessary for the cost model to accurately estimate schedule performance, and the tiling algorithm to explore the schedule space.

- **Compute_Workload(tiling_shape)** Based on the provided lambda expression and tensor sizes, this function calculates the theoretical amount of FMAs required to compute a output tile. The amount of FMAs for a tile grows proportionally as the tile size. The FMAs required for a single output element, which can be infered by parsing the reduction computation from the lambda expression for each output element.
- **Subtensor_Sizes(tiling_shape)** This function returns the dimensions of the input subtensors required to compute a given tile, which can be inferred from the dependencies between input and output elements, provided by the lambda expression. With cooperative fetching implementation, we can accurately deduce the co-fetched input data addresses, which is necessary to accurately estimating the cost of data movement, considering the memory "transaction" utilization and the bank conflicts.
- **Memory_Footprint(tiling_shape, reduction_size)** In order to explore in a search space that contains only schedules that fit in memory, this function calculates the memory footprint given the tiling and reduction sizes. It utilizes the input/output dependency information, similar to the **Subtensor_Sizes** function, and also considers partitions along the reduction axes. It can also be used to calculate the register usage when fed with the configuration on the highest memory level.

Hardware Interfaces SOKOBAN defines a set of static hardware specifications, allowing users to easily map the hardware architecture to the RATIONAL model. The mapping the architecture of an accelerator to the RATIONAL model follows the data movement flow of input tensors from low-level memory to registers during the kernel execution. To determine the number of levels in the RATIONAL model, the generated device programs must be able to explicitly manage the intermediate levels of physical memory architecture (e.g. the shared memory in GPUs) that are mapped to the RATIONAL pipeline. For each level in the RATIONAL model, SOKOBAN provides basic hardware specifications for accurate estimation of theoretical latency in each stage of RATIONAL's pipeline. These specification include the number of CUs, the size of compute vectors (e.g., the warp size in NVIDIA GPUs), as well as memory information for each memory level in RATIONAL's pipeline abstraction. The memory counters include the memory capacity, peak achievable throughput, size of memory transaction, and the bank size for each memory level, which are critical to data movement cost, and their impact can be analytically estimated by a white-box cost model. With SOKOBAN's hardware interfaces, users can easily migrate the compilation to new devices, without significant effort for the compiler to learn a cost model from scratch.

5.5.2 Cost Model

Accurately estimating memory throughput relies on both arithmetic and hardware information. A naive estimate is to calculate the amount of bytes loaded and divide this number by the bandwidth of the corresponding memory level. This implementation often greatly underestimates the memory latency, which causes the tiling algorithm to emit inefficient tiling schedules. We identify two major causes that lead to under-estimation of the memory latency: non-aligned memory transactions and bank conflicts.

Underutilized Memory Transactions A full transaction of data is loaded regardless of how much data inside one memory transaction is explictly requested by a tile. If the elements within one memory transaction are divided across different tiles, there is a high risk that one memory transaction could be loaded multiple times by different tiles, leading to redundant memory traffic. Our cost model needs to know the addresses of all dependant input elements for each tile in the output tensor. The size of the memory transaction depends on the design of the architecture, which is a key hardware counter in SOKOBAN's RATIONAL model.

Bank Conflict To boost parallelism, modern hardware accelerators usually execute threads in groups, threads in one group are executed concurrently. However, if more than one concurrent memory load instructions access different addresses in the same memory bank, these instructions will be serialized. This could dramatically reduce the actual memory bandwidth that the kernel program can achieve in a sub-linear scale [111]. On V100 GPU, we empirically found that this effect could significantly decrease the achieved memory bandwidth in runtime by up to $4\times$.

The amount of bank conflicts is determined by the memory access pattern, i.e., which memory addresses are accessed by co-run memory instructions. A naïve simulating implementation is to enumerate each group of co-run threads, calculating the addresses of all accessed input elements based on their output element indices, and finally count the number of collisions in each memory bank. This implementation is expensive and could lead to very long compilation time. We notice that in most DNN operators (e.g. convolution), the dependant input elements of two different output elements are the simple translation of each other. In this case, all groups of co-run threads share extremely similar memory access patterns, which allows us to accurately approximate the overall bank conflict by examining only one thread group. The memory bank size is used to determine which addresses belong to the same memory bank, hence a crucial hardware counter for accurate estimating the amount of bank conflicts.

5.5.3 Mapping Accelerators to RATIONAL

RATIONAL Model on NVIDIA V100 GPU NVIDIA GPUs employ a centralized memory architecture. Data in a CUDA program is stashed in the global memory that can be directly accessed by all the *streaming multi-processors (SMs)*. On V100 GPU [111], there is also a unified L2-cache, and 80 SMs each with its own local scratchpad memory (i.e. shared memory). When executing a kernel program, data is moved from the global memory to the local memory on each SM through the on-chip L2 cache, and then fetched to registers for computation units.

Similar to the global memory, the L2 cache can be accessed by all SMs, hence we view L2 cache and the global memory as a unified memory layer in our abstraction. The V100 local memory consists of scratchpad shared memory and L1 cache. We ignore L1 cache because it cannot be controlled by user programs. Hence, using our RATIONAL model, SOKOBAN will perform a two-level tiling: the size of a bottom-level task controls the traffic between the global memory and shared memory, and the size of a upper-level task controls the traffic between the shared memory and registers. The size of the shared memory on a GPU provides a upper bound for the size of a bottom-level task. We use memory bandwidths of each levels based on benchmarking [111], which are a bit lower than the V100 specifications.

A V100 thread allows one thread to use no more than 255 registers. Exceeding this limit will lead to register swap-out, causing significant performance decay. This raises an upper limit to the size of a upper-level task. We notice that the *nvcc* compiler will implicitly declare more registers (for loop variables or other purposes). Such behaviour is extremely hard to predict. Therefore, we reduce the register limit threshold to only 96 registers per thread to avoid unexpected performance decay. We employ TVM's code generator APIs to generate kernel programs based on the schedules emitted by SOKOBAN.

RATIONAL Model on AMD MI50 GPU The AMD MI50 GPU [14] is a second generation of AMD's Vega series. It shares similar memory architecture as the V100 GPU. There is a centralized global memory that can be accessed by all *compute units (CUs)*. Like SMs in NVIDIA GPU, each CU has its own scratchpad memory, registers and computation cores. The data movement during the runtime of a ROCm [15] kernel program is also similar to V100 GPU. Our RATION model for MI50 GPU is similar to V100: a two-level tiling, in which the task sizes are adjusted to control the traffic between the global memory and the on-chip scratchpad memory, as well as the traffic between the scratchpad memory and registers separately. As TVM does not support ROCm, we use a modified version of TVM to generate ROCm kernel programs.

Unlike NVIDIA GPUs, there are a few hardware counters that AMD did not make them public available. One of such counters is the bandwidth of the on-chip scratchpad memory on each CU. We use an approximate value of 8 TBps, which is higher than Vega 64 (a first generation of Vega series), but lower than a newer 5700 XT GPU (9.76 TBps).

RATIONAL Model on Graphcore IPU The Graphcore IPU [110] is a massive parallel MIMD chip with 1216 parallel processing cores. Distinct from NVIDIA and AMD GPUs, an IPU employs a distributed memory architecture. There is only 256KB on-chip local memory attached per core, and no unified global memory. During the runtime of a kernel program, a thread first fetches data from remote cores to its local memory, then load the data to registers for processing. The data movement route defines a two-level tiling model, and SOKOBAN's tiling algorithm is responsible to maintain the balance among the computation throughput, the throughput between local memory and the registers, and the throughput between local memory and all the remote memory on chip.

By default, the initial data of a kernel program is stashed in the on-chip local memory and evenly distributed across the nodes. For a fair comparison, we use the same data placement policy for both the baseline and SOKOBAN's generated programs. We take the advantage of prior benchmarking works [110], which have successfully probed the measured memory bandwidths and computation throughput. The size of the register files per core is not publicly available. Considering that we have no prediction for behaviours of the IPU program compiler, we allow each upper-level task to use only 10 registers, which safely guarantee that the tiling algorithm does not emit invalid tiling configurations.

5.6 Evaluation

In this section, we evaluate SOKOBAN on both DNN operator benchmarks and end-to-end models through comparing with other state-of-the-art DNN compilers and frameworks.

Operator	**Configuration**	**Source**	**Note**
MatMul	(512, 1024), (1024, 1024)	BERT	MM0
MatMul	(512, 4096), (4096, 1024)	BERT	MM1
MatMul	(512, 1024), (1024, 4096)	BERT	MM2
MatMul	(128, 256), (256, 256)	LSTM	MM3
Conv2D	(64, 168, 42, 42) (168, 168, 1, 1)	NASNet	CV0
Conv2D	(64, 336, 21, 21) (336, 336, 1, 1)	NASNet	CV1
Conv2D	(64, 672, 11, 11) (672, 672, 1, 1)	NASNet	CV2
Conv2D	(64, 3, 230, 230), (64, 3, 7, 7)	Resnet50	CV3
Conv2D	(64, 64, 56, 56), (64, 64, 3, 3)	Resnet50	CV4
Conv2D	(64, 128, 28, 28), (128, 128, 3, 3)	Resnet50	CV5
DepthwiseConv	(64, 168, 42, 42) (168, 1, 3, 3)	NASNet	DC0
DepthwiseConv	(64, 336, 21, 21) (336, 1, 3, 3)	NASNet	DC1
DepthwiseConv	(64, 336, 21, 21) (336, 1, 5, 5)	NASNet	DC2
BiasAdd	(512, 1024), (, 1024)	BERT	EW0
Add	(64, 256), (64, 256)	LSTM	EW1
Sigmoid	(64, 256)	LSTM	EW2
Multiply	(64, 256), (64, 256)	LSTM	EW3
Tanh	(64, 256)	LSTM	EW4

Table 5.1: Operator configurations in our benchmark.

5.6.1 Experimental Setup

Sokoban is evaluated on three types of servers with different accelerators equipped. The CUDA GPU evaluations use an Azure NC24s_v3 VM equipped with Intel Xeon E5-2690v4 CPUs and 4 NVIDIA Tesla V100 (16GB)GPUs, with Ubuntu 16.04, CUDA 10.2 and cuDNN 7.6.5. The AMD ROCm GPU evaluations use a server equipped with Intel Xeon CPU E5-2640 v4 CPU and 4 AMD Radeon Instinct MI50 (16GB) GPUs, installed with Ubuntu 18.04 and ROCm 3.5.0 [1]. The IPU evaluations use an Azure ND40s_v3 preview VM equipped with Intel Xeon Platinum 8168 CPUs and 16 IPUs with Poplar-sdk 1.0.

We compare the operator performance with TVM [39] and Ansor [265], which represents the state-of-the-art tensor compilers. We also compare with vendor-specific libraries including cuDNN, cuBlas, TensorRT for CUDA GPUs, rocblas for ROCm GPUs, and POPLAR library for Graphcore IPU.

Benchmarks Our evaluation benchmark uses four representative DNN model types including CNN model (ResNet-50 [87]), RNN model (LSTM [95]), state-of-the-art CNN model obtained by the neural architecture search (NasNet [268]), and transformer-based model (BERT [59]). From each model, we choose the most-frequently used operators to construct our operator benchmark. Table 5.1 lists the full set of operators we used as well as their configurations and model sources. The last column lists the corresponding abbreviation of each operator.

5.6.2 Evaluation of Operator Benchmark

This section presents the detailed evaluation of Sokoban on a set of operator benchmark listed in Table 5.1.

Evaluation on CUDA GPUs

Operator performance We first demonstrate the performance of Sokoban generated kernels through comparing with TVM (with XGBoost tunning algorithm), Ansor, cuBlas (for matrix multi-

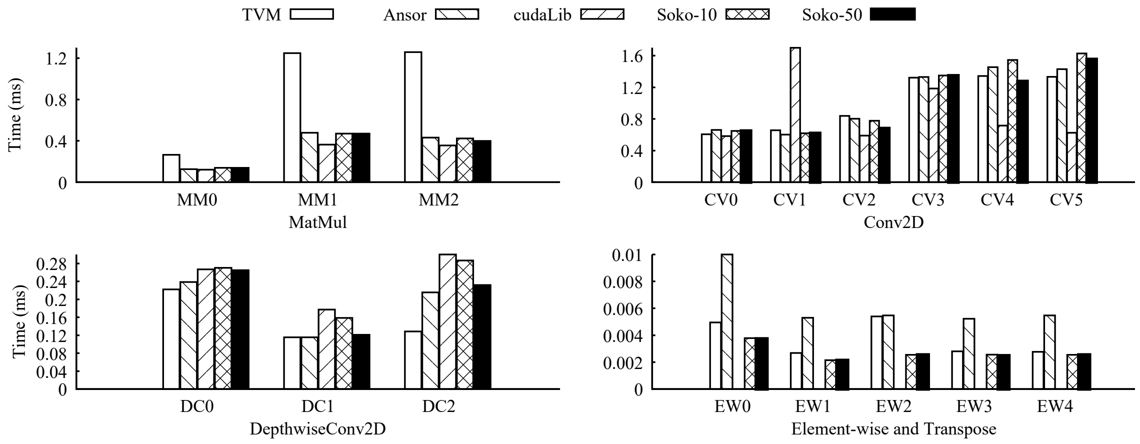


Figure 5.7: Operator performance on V100 GPUs.

plication operators) and cuDNN (for convolution operators). We set the tuning steps for TVM and Ansor as 2,000 according to their paper [265] and report the best results. Since Sokoban can return a set of kernel candidates for each operator, we sort them based on their theoretical performance predicted by our RATIONAL cost model, and select the top K candidates to conduct real profiling on hardware to get the optimal one. We evaluate three versions of Sokoban with $K = 10$ and 50 (denoted as Soko-10 and Soko-50) respectively.

Figure 5.7 shows the performance comparison of the operator benchmark on a V100 GPU. For MatMul and Conv2D operators, Soko-10 can have a comparable performance with Ansor with only 2.8% performance gap on average. However, compared with cuBlas and cuDNN libraries, Soko-10 has about 23.2% performance gap on average. This is mainly because these libraries have some in-house optimizations and are well tuned by their CUDA experts. For DepthwiseConv operator, the performance gaps between Soko-10 and Ansor are 27.8% on average, while Soko-10 can still outperform cuDNN by 28.7%. Finally, for the element-wise operators, Soko-10 can significantly outperform Ansor by 2.8× on average. If we increase the size of generated kernel candidates to 50, the Soko-50 could further increase the kernel performance by 5.8% on average than Soko-10. This means if we care the extreme performance of an operator, we can spent a longer compilation time to evaluate a larger candidate set.

Evaluation on ROCm GPUs

We evaluate the operator performance on AMD MI50 [14] GPU to demonstrate the applicability of Sokoban on various hardware. Figure 5.8 demonstrates the runtime comparisons among AMD's rocLib library, Ansor, and Sokoban for each ROCm operator program. For matrix multiplication and element wise operators, Sokoban is able to generate tensor programs that are much faster than the manually-coded ROClib, and comparable results against Ansor. We notice that Sokoban's results for convolution programs are worse than Ansor and rocLib for a few operator configurations. The reason for the less optimal performance on convolution operators is mainly due to the inaccuracy of our cost model, as the performance-critical hardware counters like shared memory bandwidth are missing. We use some approximate values for these counters, which could be largely different from

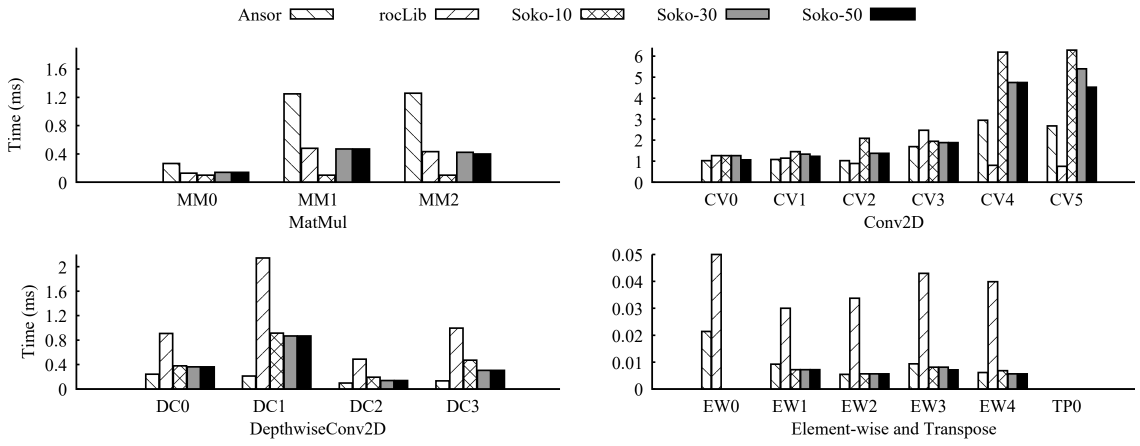


Figure 5.8: Operator performance on MI50 GPUs.

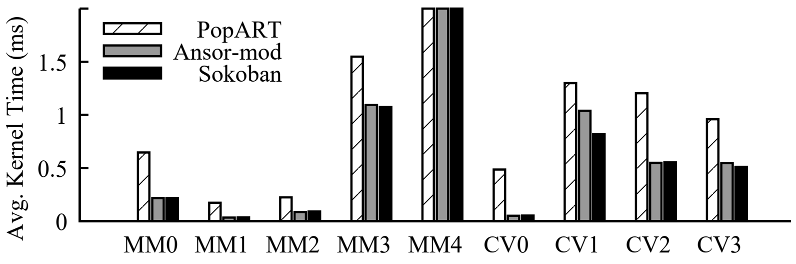


Figure 5.9: Operator performance on Graphcore IPU.

the ground truths. This could significantly decrease the accuracy of our cost model, and hence affects the quality of generated tensor programs. In the future, we aim to address this issue by building a standard benchmarking programs to probe the performance-critical counters when they are unknown.

Evaluation on Graphcore IPU

We use the Graphcore IPU to evaluate SOKOBAN's performance on the distributed memory architecture. Due to the limited on-chip memory capacity, we use operators with relatively small sizes. We tested matrix multiplication and convolution operators with different sizes. Figure 5.9 shows the runtime of each operator program generated by SOKOBAN, compared against the manually-coded Poplar-sdk library provided by Graphcore, as well as a modified version of Ansor which support code generation for IPU. SOKOBAN delivers faster kernel programs comparing to the Poplar-sdk library in almost each operator size. For matrix multiplications, SOKOBAN is able to render up to 4.7x speedups comparing to the Poplar-sdk baseline. For convolutions, SOKOBAN outperforms Poplar-sdk up to 9.2x speedups. For all of these operators, SOKOBAN is able to generate the optimal kernel programs by evaluating less than 10 candidates, and achieves equal or better performance compared against Ansor's programs for all tested operators.

5.7 Related Work

Hand-tuned libraries. Tensor computation usually dominates the runtime of DNN workloads. High-performance tensor programs are critical to fully exploit the power of modern hardware accelerators. Today most DNN frameworks [2, 187, 36] still rely on hand-tuned vendor libraries to achieve high performance. On NVIDIA GPUs, cuBlas [164] and cuDNN [40] provide high-performance kernels for linear algebra and DNN operators. Similar to CUDA [166], AMD released ROCm [15] platform for developing high-performance programs, and open-source libraries including rocBLAS [1] optimized for linear algebra operators, and MIOpen [122] optimized for machine learning operators. On IPU [110], Graphcore released the open-source framework called Poplar [192], which implements basic operators such as convolutions and matrix multiplications for DNN computations. While these libraries provide high performance, they also require significant engineering efforts to build and maintain for new operators and hardwares. In contrast, SOKOBAN can quickly generate competitive kernel programs on various types of hardware, avoiding high engineering costs.

Auto-tuners for tensor compilation. In recent years researchers have proposed various accelerators for DNN computation. The drawbacks of the manually-coded libraries motivated the creation a series of automatic tensor compilers [198, 35, 266, 20]. Halide [198] introduces a schedule language to describe loop primitives, and is designed for image processing applications. TVM [39] utilizes a scheduling language that allows users to generate device code across different hardware platforms. It also includes AutoTVM [35], a search framework that can automatically generate kernel programs based on manually written templates and predefined schedule search space. FlexTensor [266] can automatically explore the schedule space. Ansor [265] generates high-performance programs by exploring an automatically generated schedule search space with a novel search strategy, without manually-written templates. These search-based approaches rely on a huge search space to raise the chances of generating kernel programs with optimal performance. This results in long compilation time due to extensive search within a huge space. Our solution on the other hand, can quickly generate schedule candidates and evaluate them concurrently, which reduces the compilation time to just a few minutes and even seconds, vs. hours in those prior works.

Factors that affect the tensor program performance. While the search-based auto-schedulers are able to find fast kernel programs, they are all black-box approaches that usually try to include all factors that could affect the performance into the search space. It is hard to understand why a generated kernel program is inefficient. Researchers also proposed techniques that optimize a limited set of factors. NeuroVectorizer [79] for example, uses a deep reinforcement learning approach to optimize the loop vectorization. AutoPhase [100] addresses the challenge of phase ordering. Recently, Interstellar [254] analyzes the design space of the DNN accelerators, claiming that loop blocking is essential to energy efficiency and good performance, while other factors like dataflow are inferior. SOKOBAN defines a systematically approach to identify good loop blocking strategies.

5.8 Summary

In this chapter, we propose a new tensor compiler, called SOKOBAN, which can generate high-performance kernel programs on various hardware devices. We first propose the multi-level tile-based

RATIONAL model, which is essentially a white-box performance model for modern DNN accelerators. The proposed RATIONAL accurately dissects the hardware behaviours during execution of tensor programs. We then implement the prototype SOKOBAN system based on i) configurations consisting of performance-critical hyper-parameters, ii) the multi-level tile-based code template; and iii) a white-box cost model that accurately estimate the per-level compute and memory performance in RATIONAL's pipeline abstraction. We also propose a set of hardware counters so that SOKOBAN can easily fit to multiple hardware devices with distinct architecture designs. Our evaluation shows that SOKOBAN is able to generate tensor programs with high performance on various hardware platforms, and greatly reduce the compilation time at the same time.

Chapter 6

Roller: Construction-Based Tensor Compiler

6.1 Introduction

Deep neural networks (DNN) have been used extensively in intelligent tasks like computer vision and natural language understanding. As DNN computation is known for its complexity, the compute intensive sub-tasks (e.g., matrix multiplication) in a DNN model are abstracted as operators and implemented as kernels, executed on modern accelerators (e.g., GPUs, TPUs) to speed up the computation. DNN compilers play an important role in producing high-performance kernels for the development of DNN models. It reduces the burden of (often hand-crafted) library-based kernel development (e.g., cuDNN [40] and cuBLAS [164]) and provides a flexible way to cover the fast-growing number of custom operators, which libraries struggle to catch up with and optimize, a growing pain especially for new hardware vendors.

DNN compilers treat a DNN operator as tensor computation, which is then translated into nested multi-level loops iterated over the computation on each tensor element along different axes (dimensions). Compiler optimization techniques like loop partitioning/fusion/reordering are applied to nested loops. Due to the inherent complexity of loop rearrangement, it is a combinatorial optimization problem to find a good solution among a large search space, often with millions of choices. Therefore, advanced compilers [39, 266, 265] propose to adopt machine learning algorithms to search for a good solution. This usually takes thousands of search steps, each evaluated in a real accelerator, to find a reasonable solution. Our own experience shows that tuning an end-to-end DNN model using state-of-the-art compilers [39, 265] often requires days, if not weeks. The tuning time may be even longer if the DNN model runs on less mature accelerators (e.g., AMD GPU or GraphCore IPU [105]) (§6.2).

In this chapter, we propose Roller, a deep learning tensor compiler that addresses the problem in a radically different way. Roller is built on the following insights. First, instead of multi-level nested loops, Roller treats the computation in a DNN operator as a *data processing pipeline*, where data tiles (a fraction of a tensor) are moved and processed in an abstracted hardware with parallel execution units and multi-layer memory hierarchy. The goal of generating efficient kernel programs then becomes that of improving the throughput of the pipeline.

Second, for an accelerator to execute efficiently, the shape of a data tile should *align* with the hardware characteristics, including memory bank, memory transaction length, and minimum schedulable unit (e.g., warp size in GPUs). To achieve the full alignment across multiple hardware features, the available tile shapes are limited. More importantly, with alignment as a constraint, to maximize the throughput of a pipeline, one only needs to *construct* an aligned tile shape that saturates the execution unit of the accelerator. This is a significantly more efficient process than solving the originally unconstrained combinatorial optimization problem.

Third, the performance of an aligned pipeline is highly predictable. Key performance metrics under the aligned pipeline (e.g., memory throughput) can be derived from the hardware specification (or through micro-benchmarking). This greatly simplifies performance evaluation under variant aligned configurations, eliminating the need of a complex cost model and/or expensive hardware-based evaluation on each aligned configuration.

With these insights, ROLLER proposes rTile, a new abstraction that encapsulates data tile shapes that *align* with the key features of the hardware accelerator and the input tensor shapes (§6.3.1). A data processing pipeline can then be described as an rTile-based program (a.k.a. rProgram) composed by three interfaces: `Load`, `Store`, and `Compute`, acted against rTile. To construct an efficient rProgram, ROLLER follows a *scale-up-then-scale-out* approach. It adopts a recursive rTile-based construction algorithm (Figure 2) to gradually increase the size of the rTile shape to saturate a single execution unit of the accelerator (e.g., a streaming multi-processor (SM) in a NVIDIA GPU). It then replicates the resulting rProgram to other parallel execution units, thanks to the homogeneity of both the computation pattern of deep learning and the parallel execution units in an accelerator.

ROLLER can evaluate the performance of different rTiles without significant overheads. The peak (saturate) compute throughput can simply be measured *once per operator type*. And due to the alignment, other key performance factors like memory pressure of an rTile can be derived analytically from hardware specifications. This leads to an efficient micro-performance model, avoiding the expensive online profiling on each configuration required by existing DNN compilers, thereby significantly speeding up the compilation process. In addition, due to the strict alignment requirements, the recursive construction process can produce a few desired rTiles (and rProgram) quickly. Combined, ROLLER can generate efficient kernels *in seconds*.

We have implemented ROLLER on top of TVM [39] and Rammer [145], and plan to open source the code. Our evaluation on 6 types and 119 popular DNN operators from several mainstream DNN models shows that ROLLER can generate highly-optimized kernels in *seconds*, especially for large expensive custom operators. This achieves *three orders of magnitude improvement* on compilation time. The performance of ROLLER-generated kernels is comparable to and often better than the state-of-the-art tensor compilers and even vendor-provided DNN libraries. With the three rTile-based interfaces (`Load`, `Compute`, `Store`) describing an rProgram, ROLLER can easily adapt to different accelerators like AMD GPU and Graphcore IPU. ROLLER has been used to develop custom DNN kernels internally and shown to significantly speed up our development cycle. It offers potentially disruptive opportunities to new players in the compute accelerator market, who previously have to spend significant engineering efforts on efficient kernels.

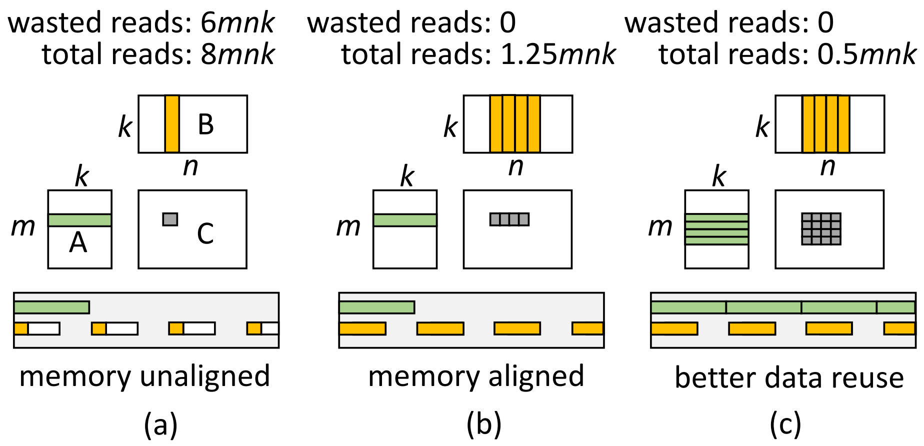


Figure 6.1: Access pattern of different tile shape. Matrix multiplication, $C_{m,n} = A_{m,k} \times B_{k,n}$.

6.2 Motivation and Key Observations

Excessive compilation time. Our own experience in a set of DNN operators (detailed setting in §6.5) shows that the average compile time for a single operator using Ansor [265], a state-of-the-art tensor compiler, is 0.65 hours. Among them, one convolution operator in ResNet model takes 2.17 hours. A DNN model may contain hundreds of operators, thus it easily takes days to compile the model. For example, to compile a NASNet model (§6.5), we reach only 32% of the overall searching progress after tuning for 41.8 hours. Our experience also shows the compilation speed is even worse on less mature devices, the compiler takes much longer time for a kernel.

Observation and insights. We observe that there exists a different view to the computation of an DNN operator. Taking matrix multiplication (MatMul), $C_{m,n} = A_{m,k} \times B_{k,n}$, as an example to illustrate our observation. Unlike existing compilers that treat MatMul as a 3-level loop iterated over each axis m, k, n, the computation process is also a data processing pipeline. One can `Load` each sub-matrix (i.e., a tile) from A and B, `Compute` the two tiles, and `Store` the resulting tile of C to memory. Thus, the performance of the computation depends on how fast one can move the data tiles in the `Load-Compute-Store` pipeline.

The key factor affecting the performance in all steps in the pipeline is the *shape* of tiles and the corresponding layout in the one-dimension memory space. Figure 6.1(a) illustrates the computation of one element in C (in the top part) and the memory accessing pattern (in the bottom part). Assuming all matrices stored in a row-major layout, loading a column from B causes strided accesses in length of 1. Suppose the memory transaction length is 4, there will be 3/4 of total redundant data reads. Thus, the data tile shape should align with the memory transaction length for efficient memory access. In Figure 6.1(b), when computing B in the granularity of 1×4 tile, there will be no memory bandwidth waste. Besides memory alignment, the tile shape should also align with the hardware execution unit, e.g., the parallel threads number, to avoid waste in computing cycles. Moreover, the tile shape also affects data reuse opportunities due to caching, a common feature in modern accelerators. For example, Figure 6.1(a) needs $2mnk$ data reads when computing a 1×1 tile each time. However, in Figure 6.1(b), only $1.25mnk$ reads are required, as one read from A can be reused 4 times. If setting the tile size along M dimension to 4×4, as shown in Figure 6.1(c), the total reads can be reduced to $0.5mnk$. A $16\times$ improvement over Figure 6.1(a).

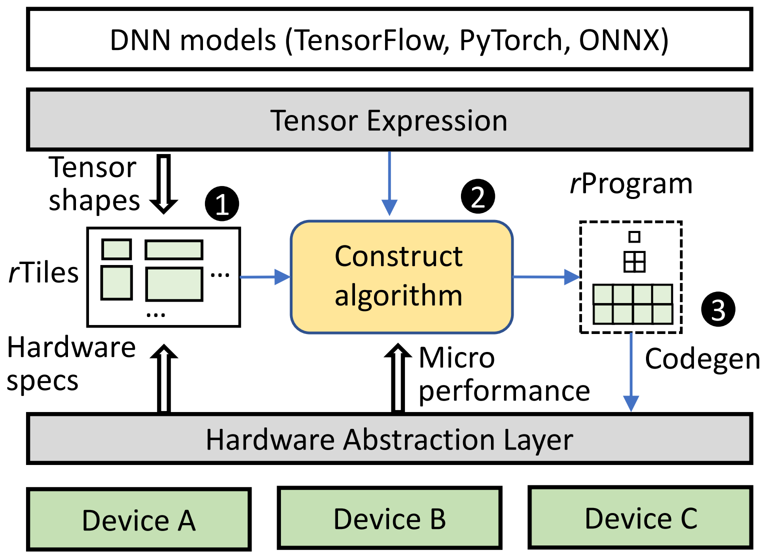


Figure 6.2: System overview of ROLLER.

These observations motivate ROLLER, a system that identifies the aligned tile shapes and constructs an efficient tile processing pipeline to improve the end-to-end throughput.

6.3 System Design

Figure 6.2 shows the system overview. ROLLER takes an operator described as a tensor expression (§6.3.1). The expression is generated by users or from a graph-level DNN compiler [39, 145, 265], which might further fuse multiple operators into a single expression. ROLLER extracts the tensor shapes from the tensor expression and leverage hardware specifications to construct rTiles, i.e., a hardware-aligned building block (§6.3.1). Based on rTiles, ROLLER proposes a *scale-up-then-scale-out* recursive construction algorithm to generate efficient tensor programs (named rProgram) that describes the data processing pipeline (§6.3.2). When generating rProgram, the construction algorithm identifies good rTile configurations by evaluating the performance of a constructed rProgram through a micro-performance model. It is built on top a device described through a hardware abstraction layer exposing only rTile-related interfaces: `Load`, `Compute`, and `Store` (§6.3.3). The constructed rProgram is finally realized through a code generator to emit the final kernel code corresponding to the specific device.

6.3.1 Tensor Expression and rTile

ROLLER takes input of a tensor computation as an index-based lambda expression, i.e., tensor expression [39, 198]. It describes how each element in the output tensor is computed based on the corresponding elements in the input tensors. For example, a MatMul operator with output tensor C of the shape $M \times N$ can be expressed as,

```
C = compute((M,N), lambda i,j:sum(A[i,k]*B[k,j])),
```

where the element indexed by (i, j) in C is computed by a sum reduction over the elements in row i of A and column j of B, and k is the reduction axis. Such an expression can cover the majority

```
class rTile {
  TensorExpr expr;
  TileShape shape;
  TileShape storage_padding;
  vector<TileShape> GetInputDataTiles();
  vector<TileShape> GetOutputDataTiles();
};
```

Figure 6.3: The data structure of rTile.

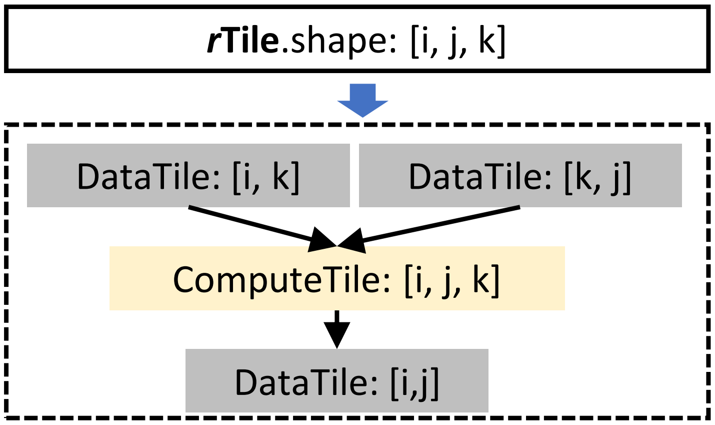


Figure 6.4: The data tiles and computing tile inferred by an rTile for MatMul expression.

of operators in DNN models and is widely used in existing DNN compilers including TVM [39], Ansor [265], and FlexTensor [266].

Roller introduces *RollingTile* (rTile for short) as the basic computing unit to compose a tensor computation. As shown in Figure 6.3, an rTile encapsulates a multi-dimensional tile `shape` defined along each loop axis of a given tensor expression `expr`. Given `shape` and `expr`, an rTile can statically infer the involved input and output data tiles. For example, a tile shape $[4, 4, 2]$ along axes i, j, k denotes an rTile for the above MatMul expression, where each rTile loads a 4×2 data tile from A and a 2×4 tile from B, conducts total $4 \times 4 \times 2$ multiply-add computations, and stores a 4×4 data tile to C, as illustrated in Figure 6.4.

A unique property of an rTile is that it must align with both the underlying hardware features and the tensor shapes in a given tensor expression. All these alignments are controlled by the rTile *shape* and the *storage_padding* fields in Figure 6.3, which represent the logical form and the physical layout of an rTile, respectively. We elaborate the detailed requirements of alignment next.

Alignment with the hardware execution unit. First, the shape of an rTile must align with the parallelism of the execution unit it runs on. For example, if running on a warp of threads in a GPU, the size of `shape` in the rTile should be a multiple of the warp size, e.g., 32, for maximal computing efficiency. When using TensorCore in NVIDIA GPUs, the rTile shape should be a multiple of $16 \times 16 \times 16$. Similarly, an rTile executed on a streaming multi-processor (SM) should align its size as a factor of execution unit number on the SM.

Alignment with memory transaction. Second, a data tile's shape should align with the length of memory transaction for optimal memory access. Specifically, for each data tile of an rTile, we should guarantee that its leading dimension (e.g., the inner-most dimension in a row-major tensor)

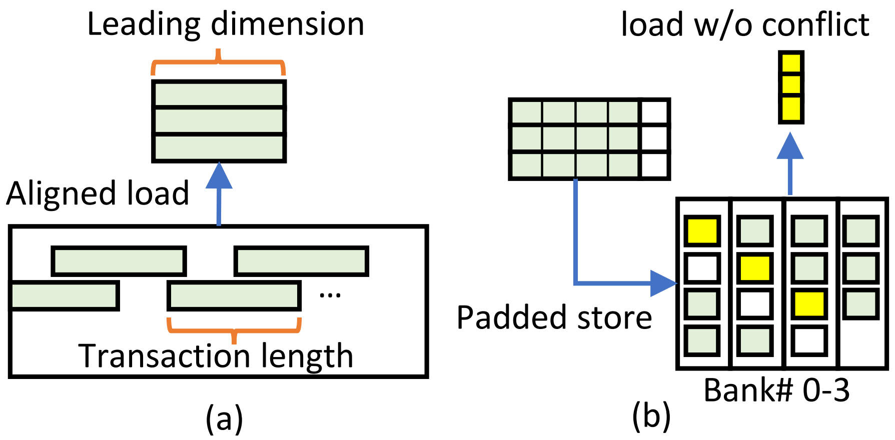


Figure 6.5: Illustration of (a) transaction aligned memory load and (b) bank conflict-free padding.

is a multiple of the memory transaction length, as illustrated in Figure 6.5(a). In Roller, tensors are allocated in a cache-aligned way. Thus, an rTile can avoid any wasted transaction read, as its shape is aligned with the memory transaction.

Alignment with memory bank. Third, the memory layout of a data tile should align its stride with the memory bank to avoid read conflicts. For example, a $[3,4]$ data tile is kept in the memory across 4 banks and is read by an upper-memory-layer tile with a shape of $[3,1]$, as shown in Figure 6.5(b). A naive approach that stores all the $[3,1]$ values in the same bank will result in conflicted loading. rTile avoids such inefficiency by padding a data tile. Given a data tile with a leading dimension of size N, which is read by another tile with a leading dimension of size n, we add a padding size of $(BL - N\%(BL) + L\lceil n/L \rceil)\%(BL)$ along N when storing this tile, where B and L are the bank number and the bank width, respectively. The padding sizes along each axis are calculated and stored in the `storage_padding` field in Figure 6.3. For the case in Figure 6.5(b), by a padding size of 1, all the $[3,1]$ values are distributed in different banks and can be read efficiently.

Alignment with tensor shape. Finally, an rTile's shape should align with the tensor shape of an input tensor expression. Otherwise, the computation cannot be evenly partitioned by the rTile, wasting compute resources or incurring heavy boundary checking overheads. A simple solution is to add a padding size P_i along a tensor dimension i with size of N_i, which makes $N_i + P_i$ a multiple of the rTile shape's dimension size at axis i. However, a large padding might waste computation. Roller therefore restricts tensor padding under a range ϵ, where an rTile's shape dimension size S_i has to satisfy that $\frac{S_i - N_i\%S_i}{N_i} \leq \epsilon$, where N_i is the tensor size at dimension i. This ensures the wasted percentage of computation is bounded by ϵ. With this restriction, we can enumerate all the valid rTile shapes that satisfy this condition.

Deriving all rTiles. Given the above alignment requirements, for a specific tensor expression and hardware device, Roller incrementally derives all the conforming rTiles using the following interface:

```
vector<int> GetNextAlignedAxisSize(rTile T, Dev d),
```

which returns the next aligned size for each axis in the `shape` of rTile T given the specific device specification d. This is calculated by gradually increasing the dimension size along each axis until it satisfies all the alignment requirements. Note that the alignment requirements can be easily extended by adding rules to restrict the rTile shape in this interface.

```
for L1_iter in L2_rtile.split(L1_rtile):
  L1_input_tiles = Load(L1_iter); //L2 to L1
  for L0_iter in L1_rtile.split(L0_rtile):
    L0_input_tiles = Load(L0_iter) //L1 to L0
    L0_out_tile = Compute(L0_input_tiles);
    Store(L0_out_tile, L2_out_tile);//L0 to L2
```

Figure 6.6: The pseudo code of an rProgram on a device with a 3-layer memory hierarchy (Bottom-up: layer `L2` to layer `L0`).

Calculating data reuse score. An interesting property of rTile is that we can implicitly control the memory traffic by adjusting its `shape`. Increasing the rTile size usually brings more *data reuse* opportunities at the cost of occupying more memory space. Given an rTile T and its next aligned size in each axis, we can calculate the data reuse score S_i for axis i by $S_i = \frac{Q(T)-Q(T'_i)}{F(T'_i)-F(T)}$, where T'_i is a newly enlarged rTile by replacing the dimension size at axis i with the next aligned size from `GetNextAlignedAxisSize`. Functions $Q(T)$ and $F(T)$ calculate the memory traffic and memory footprint when the computation is executed in the granularity of T, which can be directly inferred based on the given tensor expression and hardware memory specification (§6.3.3). A larger S_i means better cost-efficiency, i.e., more memory traffic can be saved with the same memory usage. The memory reuse score plays a critical role in constructing an efficient rProgram (using rTiles), as shown in the next subsection.

6.3.2 Tensor Program Construction

***r*Tile program.** Given rTile and the hierarchical memory structure of modern accelerators, a tensor computation can be naturally treated as a streaming data processing pipeline. The computation loads data tiles (specified in rTile) from the lowest memory layer through the memory hierarchy to the highest layer, performs rTile computation on the execution units of the accelerator, and stores the resulting data tiles back to the lowest memory. For each memory layer, a specific rTile is defined to align with the characteristics of this memory layer. Thus, ROLLER describes tensor computation as a data processing pipeline with a hierarchical rTile configuration, which is called an rTile program (i.e., rProgram).

Figure 6.6 shows an rProgram on a device with three memory layers (L0, L1 and L2). The rProgram is described by the rTile at each layer and the rTile instructions (i.e., `Load`, `Store`, and `Compute`) at each memory layer. Figure 6.7(a) shows a MatMul rProgram illustrated in Figure 6.7(b). Figure 6.7(c) illustrates how the rProgram is mapped to each memory layer of a device. Specifically, each time it loads a $[4, 4]$ data tile in A and a $[4, 8]$ tile in B from memory L2 to L1; and then it loads the data tiles from memory L1 to memory L0 (i.e., registers) in shapes of $[2, 1]$ and $[1, 2]$. After each `Compute`, the resulting $[2, 2]$ tile will be directly stored from L0 to L2.

Given a data processing pipeline, the optimization goal of the corresponding rProgram is to maximize the throughput of the pipeline. The goal can be translated into three conditions: 1) the computation and memory movement should fully leverage the hardware features; 2) the throughput should saturate the bottleneck stage; and 3) there needs to be sufficient parallelism to leverage all the parallel execution units. Thus, ROLLER proposes the following rProgram construction policy: first scale-up on one core by saturating its hardware utilization and then scale-out to leverage the

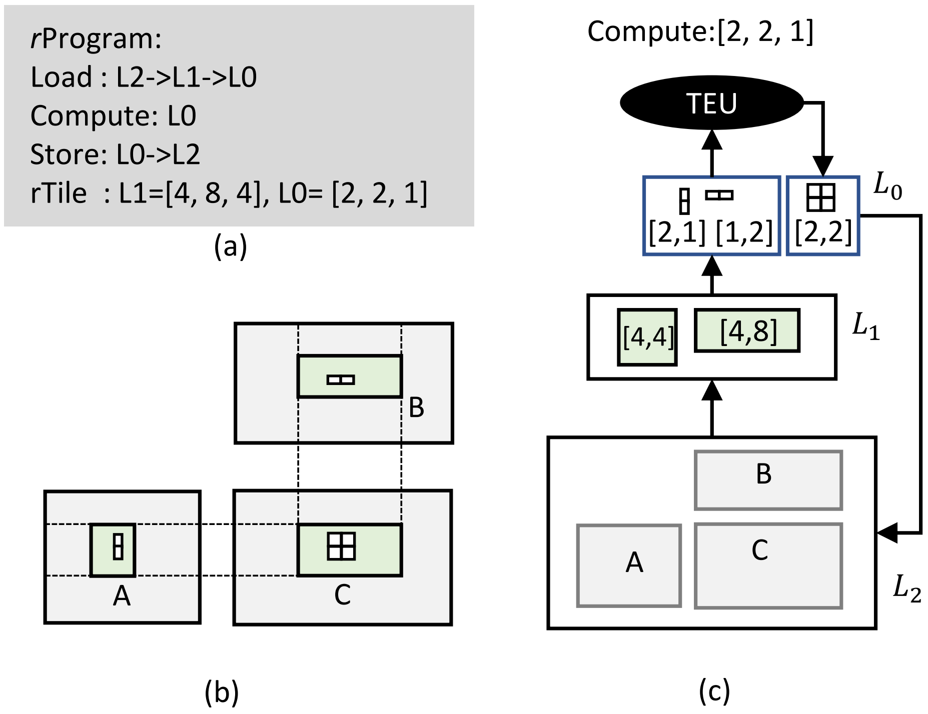


Figure 6.7: Roller computation model. (a) An rTile program; (b) rTiles on matrix multiplication; (c) Execution of the rTile program on a hardware memory hierarchy.

multi-core parallelism.

Scaling up an *r*Program. Since the alignment properties of rTile ensure hardware efficiency, Roller can just focus on maximizing the throughput at each memory layer by constructing the right rTile shape. By leveraging the data reuse score defined in §6.3.1, the rProgram construction algorithm starts from an initial rTile and gradually enlarges it towards the most cost-effective axis in the rTile (i.e., with the largest data reuse score). During the process, the memory performance improves until it hits the computational bound or the maximal memory capacity. The above process repeats for each memory layer from top to bottom, until a desired rProgram is constructed. Note that if the data reuse score remains constant for some tensor expressions, e.g., element-wise operators, Roller will just construct rTiles for the top layer and loads them directly from the bottom layer memory.

Figure 2 shows the detailed construction algorithm. Given a tensor expression `expr` and a target device `dev`, the algorithm constructs an initial rTile T at the top memory layer and enlarges T recursively (`EnlargeTile` in line 4). At each step, it gets the next larger rTile T' that improves the data reuse score (`GetNextRTileShape` in line 10). If T' hits the memory capacity (line 11) or the data tile loading throughput $MemPerf(T')$ exceeds the peak computing throughput $MaxComputePerf(T')$ (line 14), the algorithm records the current rTile T and goes on to `EnlargeTile` at the next memory layer. Otherwise, it continues to enlarge T' at the current layer (line 18). The construction finishes at the lowest memory layer (line 6), producing one result and repeating, until it obtains K (e.g., 5-20) rPrograms (to tolerate the hidden factors affected by the device compiler). Note that $MemPerf(T')$ and $MaxComputePerf(T')$ are derived based on `dev`, based on the micro-performance model (§6.3.3).

Algorithm 2: ROLLER's rProgram constructing algorithm.

```
Func ConstructProg(expr:TensorExpr, dev:Device):
    $T$ = rTile($expr$);
    Results = [];
    EnlargeTile($T$, $dev$.MemLayer(0), rProg());

Func EnlargeTile(T:rTile, mem:MemLayer, P:rProg):
    if mem.IsLowestLayer()
        Results.append(P);
        if (Results.Size() ¿ TopK) Exit();
        Return();
    $T'$ = GetNextRTileShape($T$, $mem$);
    if MemFootprint($T'$) ¿ mem.Capacity()
        $P$.Add($mem$, $T$);
        EnlargeTile($T$, $mem$.Next(), $P$);
    else if MemPerf($T'$) ¿ MaxComputePerf($T'$.expr)
        $P$.Add($mem$, $T'$);
        EnlargeTile($T'$, $mem$.Next(), $P$);
    else
        EnlargeTile($T'$, $mem$, $P$);

Func GetNextRTileShape(T:rTile, mem:MemLayer)
    $alignedSizes$ = GetNextAlignedAxisSize($T$, $mem$);
    $S_{max} = 0$, $T_{max} = T$;
    for $d$ : T.Dimensions() do
        $T'$ = $T$.Replace($d$, $alignedSizes$[axis]);
        if DataReuseScore($T'$) ¿ $S_{max}$
            $S_{max}$ = DataReuseScore($T'$); $T_{max} = T'$;
    Return $T_{max}$;
```

Scaling out an rProgram. Given the homogeneity of both the computation pattern of DNN operators and the parallel execution units in an accelerator, ROLLER simply replicates the rProgram constructed on one execution unit to other units, by partitioning the computation with the lowest layer rTile. We achieve this by distributing all the partitions evenly to all execution units. Note that ROLLER prefers to assign the partitions split along a reduction axis on the same execution unit, as they can share the reduction results in the higher memory layers.

Small operator and irregular tensor shape. The scale-out algorithm works well for large operators that have sufficient parallelism, e.g., where the partition number is significantly larger than the number of execution units. For a small operator, the overall performance of the algorithm could suffer from the low utilization of parallel execution units. In general, this can be addressed by co-scheduling with other operators in compilers like Rammer [145], if there exists sufficient inter-operator parallelism. Otherwise, for each rProgram, ROLLER will try to shrink its rTiles along the axis that has the smallest data reuse score to achieve sufficient parallelism.

In addition, a large operator may contain irregular tensor shapes with small dimensions, whereas ROLLER might not generate a sufficient number of rPrograms due to the alignment requirements. To address this issue, ROLLER transforms a tensor expression into a canonical form by an axis fusion pass. Specifically, for all the involved tensors, if there exist two adjacent axes in one tensor, which are either both existing and still adjacent or both missing in all other tensors, ROLLER can safely

```
// compute interface
int Load(T* src, rTile st, T* dst, rTile dt);
int Store(T* dst, rTile dt, T* src, rTile st);
int Compute(TensorExpr e, rTile t, T** args);

Spec GetDeviceSpec(); // Spec qurey interface

// interfaces of the micro-performance model
size_t MemFootprint(rTile t);
size_t MemTraffic(rTile t);
double MaxComputePerf(TensorExpr expr);
double MemPerf(rTile t);
```

Figure 6.8: The interface of ROLLER's hardware abstraction

merge these two axes. For example, an element-wise operator with the tensor shape $[17, 11, 3]$ in both input and output tensors, ROLLER will transform it into the tensor shape $[561](17 \times 11 \times 3)$ by fusing the three axes. Besides axis fusion, ROLLER will also try to greedily increase the parameter ϵ in the tensor padding mechanism (§6.3.1) until K rPrograms have been constructed.

6.3.3 Efficient Evaluation of an *r*Program

In the construction algorithm, ROLLER needs to evaluate the performance of rProgram. Instead of evaluating the end-to-end rProgram in a real hardware device, ROLLER only needs to evaluate the performance of the corresponding rTile, e.g., `MemPerf` and `MaxComputePerf` in Figure 2.

To this end, ROLLER builds a micro-performance model against a device described in a hardware abstraction layer (HAL). The HAL models an accelerator as multiple parallel execution units with a hierarchical memory layer. The HAL exposes three rTile-based interfaces: `Load`, `Compute`, and `Store` (Figure 6.8). An execution unit is abstracted as an *rTile Execution Unit (TEU)*, which computes the data tiles through the `Compute` interface. Multiple TEUs can be organized as a group, which `Load` and `Store` tiles cooperatively. The HAL treats different memory layers, e.g., register, shared memory, DRAM, as an unified type exposing the hardware specifications that affect the performance of tile movement. The specifications include memory capacity, transaction lengths, cache line size, and number of memory banks, which can be obtained by the `GetDeviceSpec` interface in Figure 6.8.

Micro performance model. With the hardware abstraction layer, ROLLER can easily derive the performance of a rTile (and hence the rProgram). First, given an rTile, the incurred memory footprint (including padding) and the memory traffic volume across different layer can be statically inferred from the rTile's tensor expression `expr` and the `shape`, i.e., the `MemFootprint` and `MemTraffic` interfaces in Figure 6.8. They are used to calculate the data reuse scores and check if an rTile exceeds the memory capacity. Second, to calculate `MaxComputePerf` of an rTile, ROLLER conducts a one-time profiling to measure the peak compute throughput by aggressively enlarging the compute tiles (e.g., multiple of warp size in an SM) to saturate the TEU. This performance data is cached in ROLLER for future query in the construction algorithm. Finally, for a given rTile, ROLLER also estimates `MemPerf`, the performance on loading data tiles from a memory layer to a higher layer. Given the aligned memory access in rTile, the latency of loading a regular chunk of data can be simply modeled by the division of the total traffic to the memory bandwidth. For the memory layer shared by all TEUs, we split the bandwidth evenly. For the smaller accessing sizes, Roller also conducts a one-time offline profiling for each device type and cache the results.

6.4 Implementation

Our implementation of Roller is based on TVM [39] and Rammer [145], two open-source DNN compilers. Roller's core mechanisms, including expression optimization, construction algorithm, micro performance model, etc., are implemented with 8K lines of code. Roller's input is an ONNX graph [178] or a TensorFlow frozen graph [2]. It leverages Rammer to implement graph level optimization (e.g., inter- and intra-operator co-scheduling). Next Roller lowers the TVM tensor expression extracted from the optimized graph, constructs rProgram, and performs code generation.

Code generation. Given the fixed code structure in an rProgram (in Figure 6.6), Roller generates the kernel code through a predefined template, implemented as a TVM schedule with its built-in scheduling primitives. Loading and storing data tiles at each memory layer are implemented by TVM's `cache_read` and `cache_write` primitives. Partitioning on rTile is done through `split` and `fuse`. Some primitive rTile computation is implemented with TVM's intrinsic API. With the template, a given rProgram can be directly generated into device codes, e.g., CUDA kernels.

Tensor padding. Roller relies on tensor padding to align rTiles with tensor shape. In practice, most tensors in the lowest memory (e.g., DRAM) are allocated by external program (e.g., DNN framework), thus we just apply padding in the upper layer memory (e.g., shared memory). Our tensor padding currently requires the input tensor expression to specify whether it allows to pad, as well as the default padding value (e.g., 0 for MatMul operator). For the storage padding for memory bank alignment, we leverage TVM's `storage_align` primitive to add padding.

Performance profiling. Roller implements two profilers: a *micro-performance profiler* and a *kernel profiler*. The former generates device specifications, e.g., memory bandwidth, computing throughput, etc., through a set of micro-benchmarks. The latter profiles the fastest kernels among the top K rPrograms. In practice, the performance of a specific kernel code is also slightly affected by some device-compiler and hardware related hidden factors. For example, on NVIDIA GPUs, Roller relies on nvcc [51] to compile the generated CUDA codes into machine code. However, nvcc itself may conduct code optimizations, e.g., register allocation, which might affect desired program execution behaviors. Thus, Roller leverages the kernel profiler to quickly evaluate top performing rPrograms and select the best one. Note that Roller's kernel profiler differs from the evaluation process driven by a machine learning algorithm in previous compilers [39, 265, 266]. The ML-based approach usually requires hundreds even thousands of *sequential* evaluation steps, while Roller only profiles tens of candidates *in parallel*. In future, we plan to implement assembly-level code generation to alleviate the hidden issues in a high-level device compiler.

Roller's HAL allows us to support different accelerators easily. Next, we share our experiences in implementing the HAL on several popular DNN accelerators, including NVIDIA GPUs, AMD GPUs and Graphcore IPU.

Roller on NVIDIA CUDA GPUs. An NVIDIA GPU usually employs a centralized memory architecture. We implement Roller on V100 and K80, two CUDA GPUs with different architectures on the *streaming multi-processors (SMs)*. Their memory architecture contains global memory, L2 cache, L1 cache, shared memory, and register. In Roller's HAL, we abstract them into 3 memory layers: L2 layer for global memory and L2 cache, L1 layer for only the shared memory, and the L0 layer for register. We ignore L1 cache because it shares the space with shared memory and cannot be controlled by user programs. The memory bandwidths of all levels are measured by our micro-benchmarks. The transaction length at the global memory layer is set to 32 Bytes, i.e., 8 float elements, for both GPUs. For V100 GPUs, the bank number and the bank length of the shared memory is 32 and 4 Bytes respectively. For K80 GPUs, the bank length is 8 Bytes. The shared memory capacities are set as 48KB for both GPUs (based on `deviceQuery`).

We implement the TEU on CUDA GPUs as a warp of 32 threads, which is also the basic unit to execute the TensorCore WMMA instructions. The size of a TEU Group on a HAL (e.g., a SM) is set to the warp scheduler number, which is 4 for both GPUs. The SM number is 80 for V100 [111] and 13 for K80. On CUDA GPUs, each thread has a limited register capacity, e.g., 255 registers for V100. Exceeding this limit will lead to register spilling, causing significant performance degradation. This sets a limit to the size of an rTile at register layer. We notice that the *nvcc* compiler will implicitly declare more registers (for loop variables or other purposes). Given that this behaviour is hard to predict, we reduce the register limit empirically to only 96 registers for V100 and 64 for K80 per thread to avoid unexpected performance impacts.

Roller on AMD ROCm GPUs. We also implement Roller on MI50 [14], AMD's second generation Vega series GPU. MI50 shares a similar memory architecture as V100: the centralized global memory can be accessed by all *compute units (CUs).* Like SMs in NVIDIA GPU, each CU has its own scratchpad memory, registers and computation cores. The data movement of a ROCm [15] kernel program is also similar. The memory transaction size for the global memory is set as 64 Bytes. The memory bank number is 32 and bank length is also 4 Bytes. We also implement the TEU as a warp of threads, which is 64 threads on MI50 GPUs. The maximal register size is also empirically limited to 96 registers per thread. All other specifications such as the memory bandwidths at each layer, peak computing throughput, etc., are measured with our micro-benchmark.

Roller on GraphCore IPUs The Graphcore IPU [110] is a massive parallel MIMD processor with 1216 parallel processing cores. Distinct from NVIDIA and AMD GPUs, an IPU employs a distributed memory architecture. There is only 256KB on-chip local memory attached per core, and no unified global memory. When the local memory is unable to hold all the input data, by default, the initial data of a kernel program is stashed in the on-chip local memory and evenly distributed across the nodes. Thus, Roller's HAL for IPUs also abstracts three memory layers: L2 for all the remote memories across all cores, L2 for the local memory on each core, and L0 for the register. We take advantage of prior benchmarking work [110], which has successfully measured peak memory bandwidth and computation throughput. The size of the register files per IPU core is not publicly available. Considering that we have no prediction for behaviours of the IPU program compiler, we allow each upper-level rTile to use only 10 registers, which safely guarantee that the tiling algorithm does not emit invalid tiling configurations.

6.5 Evaluation

We evaluate Roller on both DNN operator benchmarks and end-to-end models by comparing with state-of-the-art DNN compilers and frameworks. We first summarize our findings: 1) Roller achieves *three orders of magnitude speedup* on compilation time, compared to TVM and Ansor. On

Operator	**Configuration**	**Note**
MatMul	M=65536,K=2,N=1024	M0
MatMul	M=128,K=4032,N=1000	M1
MatMul	M=65536,K=1024,N=4096	M2
Conv2D	D=(128,128,28,28), K=(128,128,3,3),S=1	C0
Conv2D	D=(128,128,58,58), K=(128,128,3,3),S=2	C1
Conv2D	D=(128,256,30,30), K=(256,256,3,3),S=2	C2
DepthwiseConv	D=(128,84,83,83), K=(84,84,5,5),S=2	D0
DepthwiseConv	D=(128,42,83,83), K=(42,42,5,5),S=1	D1
DepthwiseConv	D=(128,84,21,21), K=(336,336,1,1),S=1	D2
Element(Relu)	I=(128,1008,42,42)	E0
Element(Relu)	I=(128,256,14,14)	E1
Element(Relu)	I=(128,1024,14,14)	E2
Avgpool	D=(128,168,83,83),K=1,S=2,VALID	P0
Avgpool	D=(128,617,21,21),K=3,S=2,SAME	P1
Avgpool	D=(128,42,83,83),K=3,S=1,SAME	P2
ReduceMean	I=(128, 512, 1024), axis=[2]	R0
ReduceMean	I=(65536, 1024),axis=[1]	R1
ReduceMead	I=(128, 4032, 11, 11), axis=[2,3]	R2

Table 6.1: A subset of operator configurations in our benchmark.

V100 GPU, the most expensive operator takes 43 seconds, while all other operators take only around 13 seconds to compile. 2) Roller matches the state-of-the-art performance of vendor libraries and other compilers on a wide range of operators. It even outperforms others for more than 50% of operators. 3) For operators with smaller sizes and irregular shapes, Roller's results are sub-optimal because of the difficulty in aligning with the hardware. However, their kernel execution time is usually small (around or below 1ms). 4) We have conducted the most extensive evaluations (119 ops in total) covering different operator types over different accelerators.

Experimental setup. Roller is evaluated on four types of servers equipped with different accelerators. The CUDA GPU evaluations use two types of servers: an Azure NC24s_v3 VM equipped with Intel Xeon E5-2690v4 CPUs and 4 NVIDIA Tesla V100 (16GB) GPUs and an Azure NC24_v1 VM with 24 Intel(R) Xeon(R) CPU E5-2690v3 CPUs and 4 NVIDIA Tesla K80 GPUs. Both running on Ubuntu 16.04 with CUDA 10.2 and cuDNN 7.6.5. The AMD ROCm GPU evaluations use a server equipped with Intel Xeon CPU E5-2640 v4 CPU and 4 AMD Radeon Instinct MI50 (16GB) GPUs, installed with Ubuntu 18.04 and ROCm 4.0.1 [15]. The IPU evaluations use an Azure ND40s_v3 VM equipped with Intel Xeon Platinum 8168 CPUs and 16 IPUs with Poplar-sdk 1.0.

We compare Roller against other tensor compilers, vendor libraries and DNN frameworks, including TVM [39] (v0.8) and Ansor [265] (v0.8), two state-of-the-art tensor compilers; cuDNN, cuBLAS, rocBLAS (ROCm GPUs), POPLAR library (Graphcore IPU), which are vendor libraries; TensorFlow (v1.15), a state-of-the-art DNN framework; TensorFlow-XLA a state-of-the-art DNN full-model compilers; and TensorRT (v7.0) (with TensorFlow integration version), a vendor-specific inference library for NVIDIA GPUs. We validate our compilation results by comparing them against Ansor's.

Benchmarks. Our evaluation benchmark uses four typical DNN models, including ResNet-50 [87] (CNN), LSTM [95] (RNN), NASNet [268] (a state-of-the-art CNN model obtained by the neural architecture search), and BERT-Large [59] (transformer-based). We set the default batch size of each model to 128. From each model, we choose the most-frequently used operators to construct our operator benchmark. It contains 6 classes of operator type with total 119 operator instances with

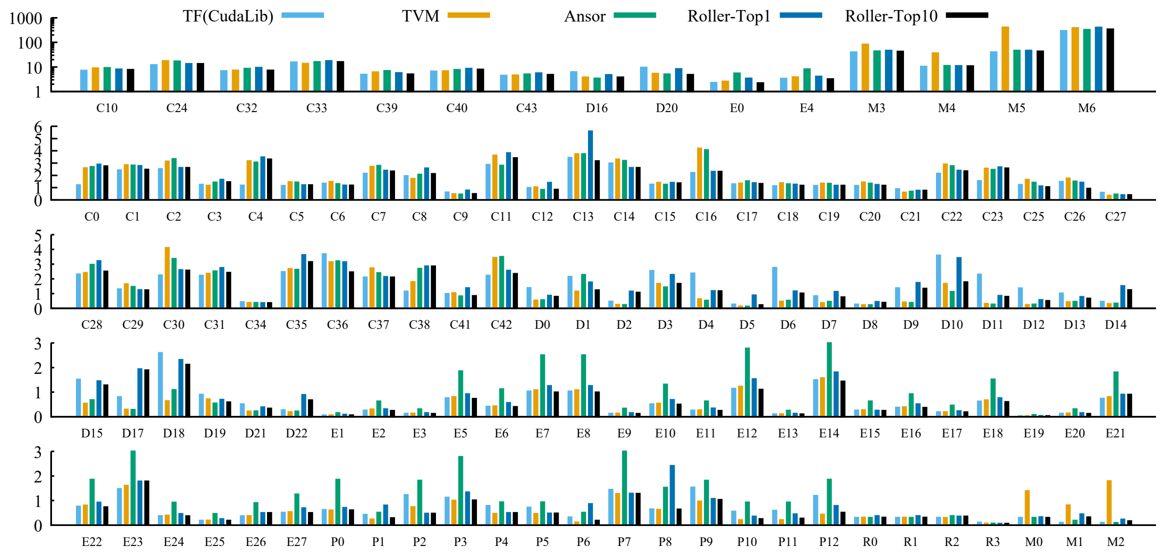


Figure 6.9: Operator performance on V100 GPUs (y-axis: average kernel execution time in ms).

different configurations (7 MatMul operators, 44 Conv2D operators, 23 DepthwiseConv operators, 28 element-wise operators, 13 pooling operators, and 4 reduction operators). Table 6.1 lists a representative subset of operators as well as their configurations. The last column lists the corresponding abbreviation of each operator. The full list of the operator configurations is omitted due to page limit.

6.5.1 Evaluation on NVIDIA GPUs

This section first evaluates Roller's operator performance, compilation time, and scalability on large operators by comparing against the state-of-the-art tensor compilers and vendor libraries. We also evaluate the performance of Roller on TensorCore. Finally, we show the end-to-end model performance compared to existing DNN compilers and framework.

Operator performance. We first evaluate the performance of Roller generated kernels by comparing against TVM (i.e., AutoTVM with XGBoost tuning algorithm [35]), Ansor, cuBLAS (for matrix multiplication operators) and cuDNN (for convolution operators). Vendor libraries like cuBLAS and cuDNN are wrapped in TensorFlow to evaluate the performance. For the rest of operators (e.g., element-wise, reduce), we use TensorFlow's built-in kernel implementations. To amortize the overhead of data feeds/fetches in TensorFlow's session, we repeat the kernel running for 1,000 times in each session and calculate the average. We set the tuning steps for TVM and Ansor to 1,000 for each operator, same as Ansor's evaluation setup [265], and report the best results. We compare both the top-1 and the best from the top-10 kernels constructed by Roller, the latter can tolerate some hidden performance impacts from device compilers.

Figure 6.9 plots the average kernel performance for all the 119 operators in our benchmark, ordered by the operator type and ID. We plot the large operators (e.g., kernel time is larger than 5ms) in the top sub-figure in a log-scale for y-axis, and the other medium and small operators in the bottom 4 sub-figures. First, compared to CUDA libraries (CudaLib), Roller could get comparable performance (i.e., within 10% performance) for 81.5% of the total operators, and can

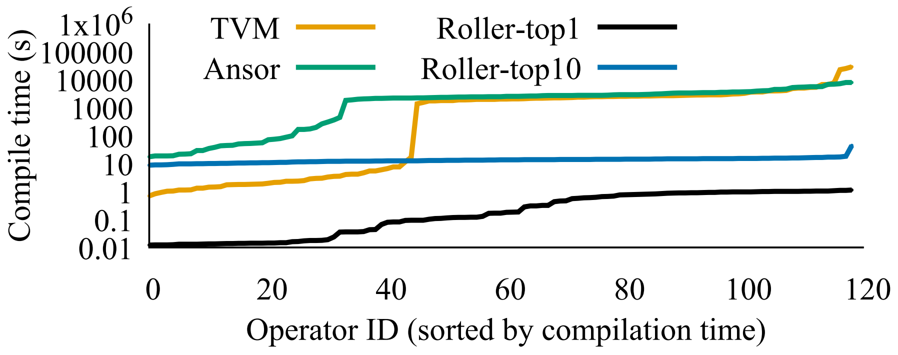


Figure 6.10: Compilation time for each operator.

be even faster for 59.7% of them. We observe that the majority of operators that Roller performs worse are convolution operators with 3×3 or larger filters, which are usually implemented with a more efficient numerical algorithm (e.g., Winograd [129]) in cuDNN and hard to be expressed by the tensor expression. This is the reason Ansor and TVM are also slower than CudaLib in these cases. Second, compared to TVM and Ansor, Roller could also get comparable performance for 72.3% and 80.7% of the total operators respectively. The rest 27.7% and 19.3% of them are mainly small operators or with irregular tensor shapes, which are by natural hard to align with the hardware. However, these operators usually have relatively short kernel time, e.g., only 1.65ms and 1.16ms on average. Among 54.6% and 65.5% of the total operators, Roller can even produce faster kernels than TVM and Ansor, respectively. We observe that the majority of these operators are large and time-consuming ones. As it shows in the top sub-figure where operators are larger than 5ms (up to 343ms), Roller could achieve better performance for most of these operators, e.g., by 1.85× and 1.27× speedup over TVM and Ansor on average.

Compilation time. Given the comparable kernel performance, the major advantage of Roller is its fast compilation. Figure 6.10 compares Roller's compilation time against TVM and Ansor for all the operators. The operator ID is sorted by the compilation time for each line. The average operator compilation time for TVM is 0.65 hours and up to 7.89 hours. For the first 40 operators, which are mainly the element-wise, reduction, and pooling operators, TVM's compilation takes less than 10 seconds. This is because TVM's manually-written code templates for these operators can directly emit code without searching. However, Ansor generates search spaces for all the operators. Its compilation time takes 0.66 hours on average and up to 2.17 hours. In contrast, Roller's top-1 kernel results can be generated in 1 second for most operators and in 0.43s on average, which is *more than three orders of magnitude* faster. The major time is spent on the recursive constructing algorithm, which increases slightly with the growth of operator size, but quickly stabilizes as the recursive depth (to enlarge the rTiles) is bounded by the limited memory capacity. To get the optimal kernels from the top-10 candidates, Roller's average compilation time is only 13.3 seconds. The major cost comes from the kernel code compilation with the device compiler and the evaluation on target devices.

Scale-out with operator size. We evaluate the scalability of Roller on larger operators by comparing with both CUDA libraries, TVM, and Ansor. We select a MatMul operator from the BERT model and a Conv2D operator from the ResNet mode, and scale them by setting different

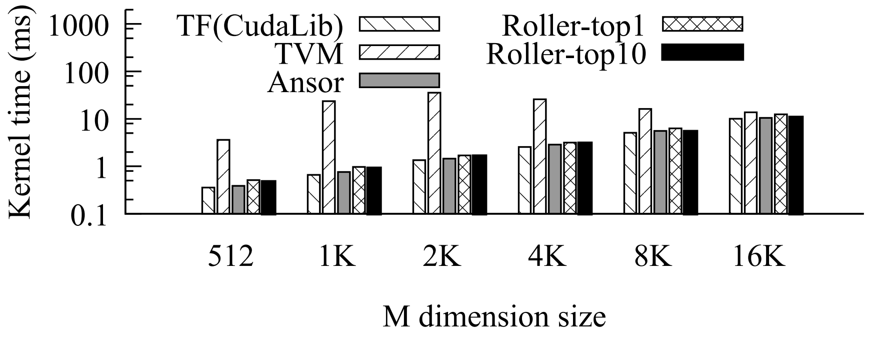


Figure 6.11: Kernel time for MatMul operator with different sizes of M in BERT-Large model, K=1024, N=4096.

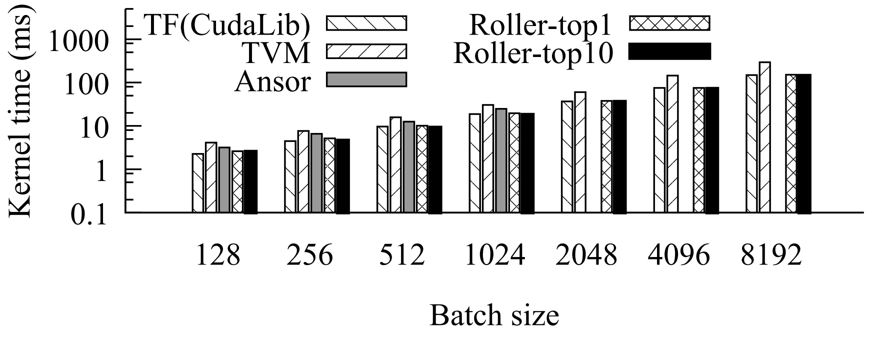


Figure 6.12: Kernel time for Conv2d operator with different batch sizes of N, where C=1024, H=14, F=2048, K=1, S=2.

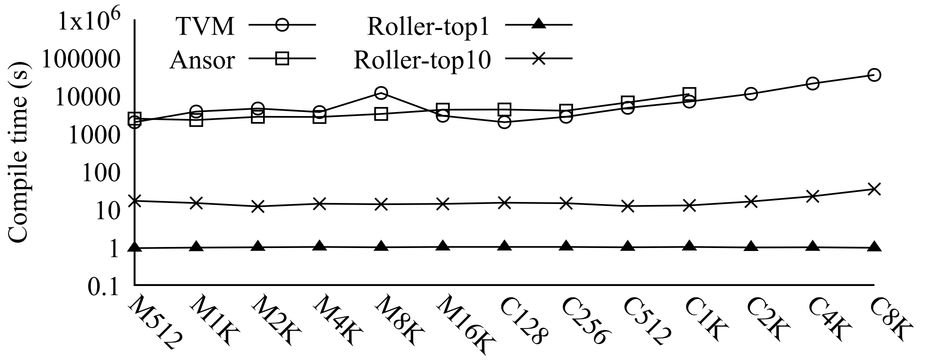


Figure 6.13: Compilation time for both MatMul and Conv2d operator with different batch sizes.

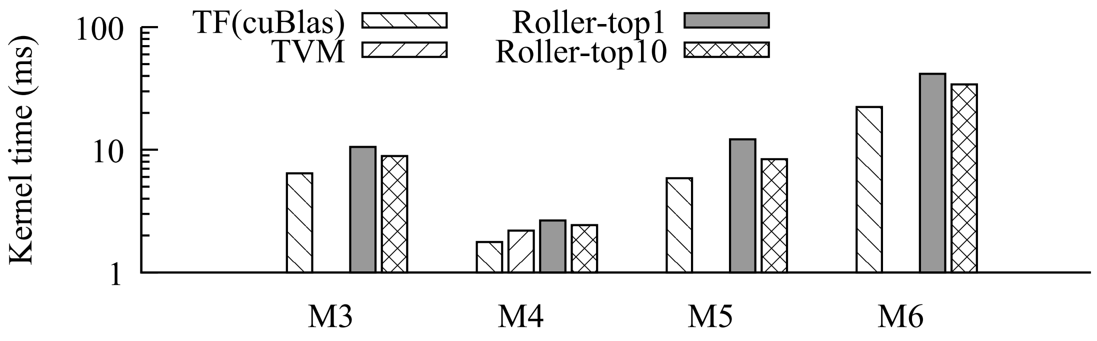


Figure 6.14: Matmul kernel time on TensorCore.

batch sizes. Figure 6.11 and Figure 6.12 show the performance comparisons. For the MatMul operator, both Ansor and ROLLER have a linear scalability over the batch sizes and comparable performance with CudaLib (i.e., cuBLAS). However, TVM's performance is relatively non-stable. For example, ROLLER can outperform TVM by average 11.2× and up to 36.1× for the batch size of 1024. For Conv2D operators, ROLLER can still achieve linear scalability over the batch size, and get slightly better performance than Ansor and TVM (by 1.25× and 1.54× on average). Note that Anosr is unable to search for a valid kernel for the batch size over 2048 using its default configurations. TVM can generate valid kernels, but the performance is scaled sub-linearly for the larger batch sizes, e.g., ROLLER can achieve more than 1.9 × speedup for batch sizes greater than 2048.

Finally, Figure 6.13 compares the compilation time for the two operators with different batch sizes. The average compilation time of TVM and Ansor is 2.36 (up to 9.55) hours and 1.19 (up to 3.0) hours respectively. Moreover, their compilation time grows constantly with the growing of batch size. This is because that they are both based on ML-based search approach, whose search space usually increases exponentially with the operator size. In contrast, ROLLER produces the top-1 kernel in 1 second, and 16 seconds (up to 34 seconds) on average for the top-10 kernel.

Compile on TensorCore. ROLLER could easily support hardware tensor ISAs (e.g., TensorCore) by aligning the rTile shape with the hardware instruction shape. We use the 16×16×16 WMMA instruction in ROLLER. We remove Ansor in this experiment as it does not support TensorCore to our best knowledge. We select 4 large MatMul operators that are friendly to TensorCore in this experiment. Figure 6.14 shows the performance comparisons. As it shows, by constructing from the aligned rTile shape, ROLLER can quickly produce good kernels on TensorCores, e.g., within a 43% performance gap to cuBLAS. Note that cuBLAS is highly optimized with a lot of hand-crafted optimizations on TensorCore. As a comparison, TVM fails to generate valid kernels for 3 of the 4 total operators with the default configurations. We try to increase the tuning steps from 1,000 to 10,000, it is still unable to find a legitimated kernel due to its poorly-defined search space.

Small operators and irregular tensor shape. ROLLER optimizes performance for small operators by shrinking the rTile when there is insufficient parallelism. We demonstrate the performance of this optimization for the two small MatMul operators. Figure 6.15 compares the performance of the original rTile configuration without sufficient parallelism (Roller-O), and the shrunken rTile configuration (Roller-S) which matches the SM parallelism. As it shows, shrinking rTile could significantly improve performance than the original kernel, e.g., by 2.3× on average. However, ROLLER

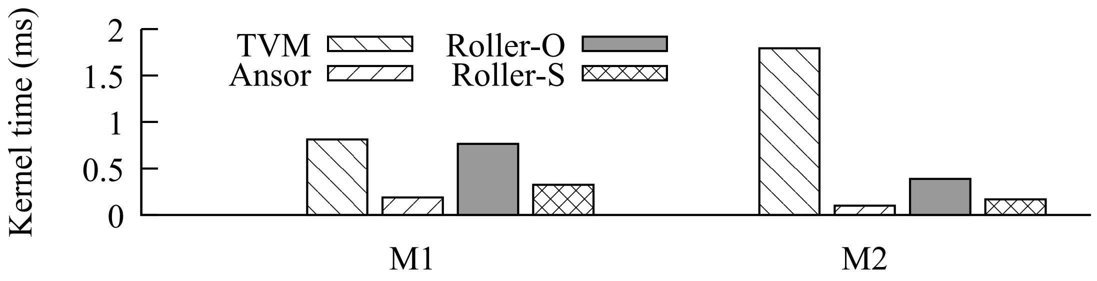


Figure 6.15: Performance for small operators.

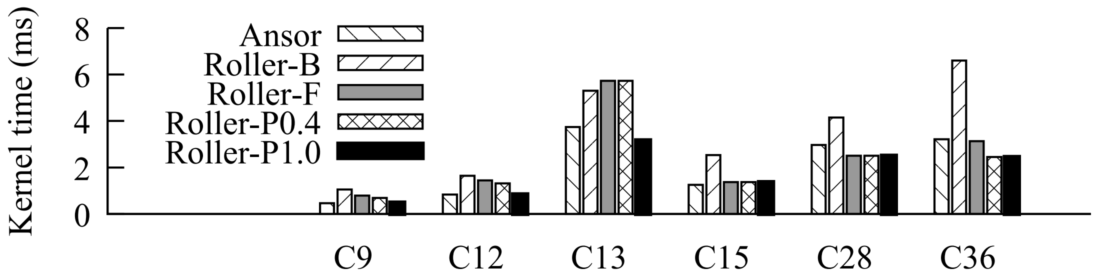


Figure 6.16: Performance for operators with irregular shapes.

is still slower than Ansor, e.g., by 50% on average, on small operators, even it is significantly faster than TVM by 6.6×. For such operators, we can further leverage search-based approach to fine-tune the configurations to obtain a better performance.

Roller compiles operators with irregular tensor shapes with two optimizations: i.e., *axis fusion* and *tensor padding* with bound parameter ϵ. We demonstrate their benefits on a representative set of irregular convolution operators, as shown in Figure 6.16. We compare the performance of Roller without any optimizations (Roller-B), with axis fusion (Roller-F), and further with tensor padding of ϵ from 0.4 to 1.0 (Roller-P0.4 and Roller-P1.0). All Roller's performances are the best one selected from the top-10 candidates. First, with axis fusion optimization, Roller is able to have more rTiles that aligns with the tensor shapes, which improves the kernel performance by 1.5× on average. Moreover, with the tensor padding optimizations (e.g., at ϵ of 1.0), Roller can further improve performance than Roller-F by 1.4×. This is mainly because the number of legitimated kernels is very limited with smaller ϵ for irregular shapes. Increasing the ϵ allows Roller to have chance to select from more candidate kernels.

	BERT-Large	ResNet	NASNet	LSTM
TF	5,186	131	1,041	141
TF-XLA	OOM	112	OOM	98
TF-TRT	N/A	137	883	31
Ansor	46,847 (TVM)	122	927	84
Rammer+TVM	17,730	143	1,168	43
Rammer+Ansor	5466	137	1036	48
Rammer+Roller	4,850	142	1,005	20
Ansor compile-time	30.9h (TVM)	33.4 h	41.8h	11.3 h
Roller compile-time	371s	352s	668s	298s

Table 6.2: End-to-end model execution time (in milliseconds) and compilation time on V100 GPUs.

	TF(CudaLib)	TVM	Ansor
Better Performance	54.6%	36.1%	68.1%
Perf. within 5%	54.6%	45.4%	70.6%
Perf. within 10%	64.7%	50.4%	71.4%
Perf. within 50%	99.2%	85.7%	89.9%
Perf. within 90%	100%	99.2%	100%

Table 6.3: The percentage of better and comparable performant operators on NVIDIA K80 GPUs.

End-to-end model performance. We evaluate the end-to-end model performance of ROLLER by comparing against TensorFlow (TF), TensorFlow-XLA (TF-XLA), TensorRT (TF-TRT), and Ansor, which represent the state-of-the-art DNN framework, graph-level compiler, vendor-provided DNN engine, and DNN compiler with tensor compilation, respectively. We omit TVM in this experiment as it usually requires an order of magnitude longer compilation time on tuning end-to-end models than Ansor [265]. ROLLER's end-to-end model compilation is implemented in Rammer (i.e., Rammer+Roller) by feeding the generated kernels into it. To create a fair baseline, we manually feed both the TVM and Ansor generated kernels for the same set of operators into Rammer, which are denoted as Rammer+TVM and Rammer+Ansor.

Table 6.2 lists the model execution time for each model compiled or executed by each compiler and framework. Note that TF-XLA fails to compile the BERT-Large and NASNet model (out-of-memory). TF-TRT also fails to run the BERT-Large model due to exceeding the maximum protobuf size limit (2GB) in its graph loading stage. For Ansor, we set the total tuning steps as 1,000 multiplied with the number of sub-graphs for each model. However, Ansor also fails to produce a legitimate program for BERT-Large models. Thus, for this case, we use TVM to compile the model. Note that, the performance of TVM for BERT-Large is about 2.6× slower than Rammer+TVM, as the default layout of the dense operator in TVM (i.e., NT) is different from that in Rammer (i.e., NN). First, for the ResNet and NASNet models, ROLLER can only achieve comparable and mostly slower performance than TF, TF-XLA, and TF-TRT (up to 26.7% slower compared to TF-XLA for ResNet). This major overhead in ROLLER is caused by the less efficient convolution kernels compared to cuDNN as explained before. However, for the BERT-Large and LSTM models, ROLLER can outperform all other frameworks and compilers, e.g., by 1.07× and 1.55× faster than the state-of-the-arts, i.e., TF for BERT-Large and TensorRT for LSTM. This mainly due to ROLLER's kernel construction favors large and regular operator shape, which are heavily used in the BERT-Large model. For both the BERT and LSTM models, since ROLLER can control to generate resource-efficient kernels by the scaling-up policy, it provides more opportunities for Rammer to co-schedule parallel kernels on the parallel SMs on GPUs. They together produce an efficient end-to-end program, which can even outperform TF-TRT by 1.55× for LSTM. Among all the implementations, Ansor can also produce very efficient programs for all the rest 3 models except for the BERT. However, it requires a long compilation time (29.3 hours on average). For the NASNet model, it reaches only 32% of the overall searching progress after tuning for 41.8 hours. In contrast, ROLLER only takes 422s on average to compile these models. This includes the graph-level optimization and the full-model compilation time in Rammer, which occupies about 41% of the total time on average.

Operator performance on K80 GPUs. We also evaluate ROLLER on the K80 GPUs. Table 6.3 shows the percentage of better or comparable performing operators (e.g., within 10% differences or 1.1× slow down) ROLLER generates for our operator benchmarks. Compared to CUDA libraries,

	TF(RocLib)	TVM	Ansor
Better Performance	73.1%	58.8%	70.6%
Perf. within 5%	79.0%	62.2%	72.3%
Perf. within 10%	81.5%	62.2%	73.9%
Perf. within 50%	94.1%	84.0%	86.6%
Perf. within 90%	100%	100%	100%

Table 6.4: The percentage of better and comparable performant operators on AMD ROCm MI50 GPUs.

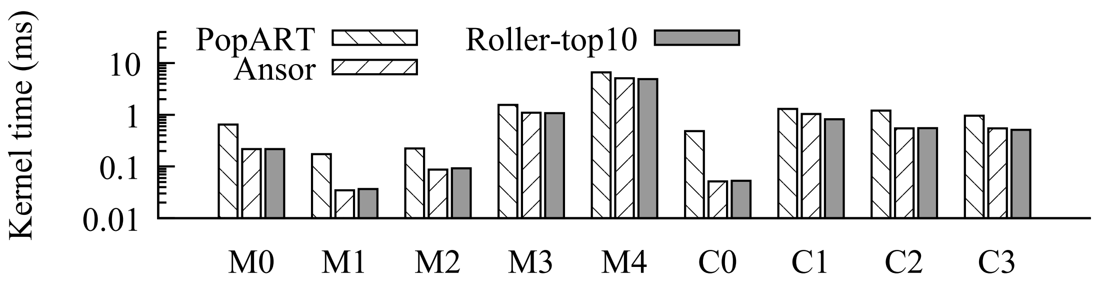


Figure 6.17: Operator performance on GraphCore IPU (y-axis in log-scale).

TVM, and Ansor, Roller produces 54.6%, 36.1% and 72.3% better kernels for the whole operator benchmark. The percentage is relatively low for TVM mainly because the manual-crafted element-wise kernel templates in TVM are already highly-optimized. Finally, the average compilation time for all operators is 0.84 hours for TVM and 1.2 hours for Ansor respectively. In contrast, Roller's average compilation time is only 0.26 seconds for top-1 kernel and 10.26 seconds for top-10 kernel.

6.5.2 Evaluation on Other Accelerators.

Operator performance on AMD ROCm GPUs. We evaluate Roller on AMD ROCm GPUs by comparing it against ROCm libraries, TVM, and Ansor. Table 6.4 shows the percentage of operators that Roller can produce better or comparable performance (e.g., within 5% and 10% differences) in our operator benchmarks. Compared to the ROCm libraries (e.g., rocBlas), 73.1% of the total operators Roller can produce better kernels. This percentage is much higher than that on CUDA GPUs (59.7% and 54.6% for V100 and K80 GPUs). This is mainly because the libraries on CUDA GPUs are more mature than the ROCm GPUs, where Roller can help significantly. Compared to TVM and Ansor, Roller can also produce 58.8% and 70.6% better kernels. Similar to CUDA GPUs, the kernels that are slower by more than 10% are mostly small operator and those with irregular tensor shapes: the average execution time of these kernels are only 1.69ms and 1.57ms for TVM and Ansor, respectively. Finally, the average compilation time for all operators is 0.85 (up to 4.2) hours for TVM and 0.99 (up to 3.4) hours for Ansor, respectively. In contrast, Roller's average compilation time is 0.24 (up to 0.63) seconds for top-1 kernel and 7.69 (up to 49.0) seconds for top-10 kernel.

Operator performance on Graphcore IPU. We evaluate Roller on GraphCore IPUs. Due to the limited on-chip memory capacity, we only evaluate a set of small MatMul and Conv2D operators with different configurations. Figure 6.17 shows the average kernel time of each operator in log-scale, comparing against the Poplar-sdk library (i.e., PopART) provided by Graphcore and Ansor.

Since TVM and Ansor do not have Graphcore backends, we use a modified version of Ansor in this experiment. As it shows, Roller can generate faster kernels than PopART for all operators, with an average of 3.1× and up to 9.2× speedup. Even comparing to Ansor, Roller can still construct comparable or even better kernels in most of operators, i.e., 2.9% average improvement. Note that Ansor still requires hours of tuning for each operator, as the device compiler on IPUs could take up to minutes to compile a program. However, Roller usually produce good kernels from the top-10 constructed candidates in several minutes. This time is mainly bottle-necked by the less-matured device compiler. It also brings more challenges to adopt the ML-based tensor compilers on these devices.

6.6 Comparison with Sokoban

In this thesis we propose two white-box tensor compilers, Sokoban (§5) and Roller (§6). Both compilers adopt a multi-level tile-based compilation paradigm, which achieves fast compilation by dissecting and abstracting the behaviours of data movement during tensor kernel execution. Despite that this feature provides both compilers the ability to reduce the per-kernel compilation time down to minutes and even seconds, there are fundamental differences between these two approaches. Based on RATIONAL, Sokoban is essentially a search-based tensor compiler, which identifies the optimal kernel schedules with an end-to-end cost model. This leads to two fundamental limitations of Sokoban.

First, the schedule space of Sokoban is combinatorial, which size is exponential to the dimension of the output tensor. The size of this space is eventually constrained by two factors: i) the arithmetic of DNN operator and tensor shapes, and ii) the capacity of hardware resources (e.g., the numbers of registers, the size of scratchpad memory, etc.). With the introduce of more advanced accelerators and new DNN operators, the exhaustive enumeration could become increasingly expensive, resulting in longer compilation time.

Second and more critically, we need to balance between the accuracy and the generalization of the end-to-end cost model. During our exploration, we noticed several intricate details that could potentially affect the kernel performance on V100: i) the density of the compute (FMA) instructions in the compiled device assembly; ii) the use of non-user-controllable caches (e.g., L2 in NVIDIA GPUs); iii) the achieved memory bandwidth under insufficient parallelism. These features are often hardware-specific and hence hard to be included in RATIONAL's abstraction. An cost model that is a hundred percent accurate would have to consider these details, and hence easily overfit to certain hardware and lose the generalization. This could eventually lead to non-trivial amount of work when fitting Sokoban to potentially new accelerators, as they could introduce new hardware-specific features that are critical to kernel performance.

On the contrary, Roller's construction compilation process is more robust in terms of the compilation time, as it does not need to perform exhaustive search. The feature of hardware alignment requirements in Roller allows it to be free from intricate modeling under performance-critical factors such as bank conflicts (§5.5.2). As a result, Roller involves only straight-forward micro performance modeling, which improves Roller's generalization, allowing Roller to easily fit to potentially more accelerators.

A drawback of Roller's hardware alignment requirements is that Roller could produce sub-optimal over small operators, or operators with irregular tensor shapes. One of such examples is the multiplication between a 128×256 and a 256×256 matrices, the major operator in the LSTM model [95]. On V100, comparing against SOKOBAN's generated kernel, which performance matches Ansor's result, ROLLER's result is 6.74× slower without the adjustment for small operators (described in §6.3.2), and 1.41× slower with the adjustment. The reason behind ROLLER's slowdown on small operators is that the optimal tiling schedule requires higher parallelism with the sacrifice of data reuse.

Despite this performance decay, our evaluation (§6.5) shows that on new DNN models with large operators, ROLLER is able to produce competitive kernels for almost all operators. We believe that as the DNN models growing larger, they tend to utilize greater operators which can fully exploit the hardware resources. Moreover, when coupled with higher-level optimizations (e.g., concurrent kernels), the performance of end-to-end training with ROLLER's kernels can easily surpass that with Ansor's generated kernels (shown in Table 6.2), despite ROLLER's performance on a single small operator being slower.

6.7 Related Work

Most tensor compilers treat DNN operators as nested multi-level loop computation, which essentially defines a large space with a combinatorial complexity. TVM [39] inherits the insight from Halide [198] and describes DNN operators as loop optimization schedule primitives. Later, AutoTVM [35] extends TVM to apply an ML-method to search for the best configurations from manually written code templates. FlexTensor [266] proposes to automatically explore the space without manual templates. Ansor [265] further advances such automation. It generates an even larger search space considering a hierarchical code structure and adopts an evolution algorithm to find performant kernels. Compilers like Tiramisu [20], AKG [263], and Tensor Comprehensions [233] apply polyhedral-based techniques to loop optimization, which transforms the loop into an integer programming problem and finds a good configuration with a solver. All these approaches rely on a huge search space to provide good kernel, which leads to long compilation/solving time. ROLLER explores a different approach to construct rTiles that align with hardware features.

Tensor Processing Primitives (TPPs) [67] define a set of 2D-tensor operators to compose complex operators on high-dimensional tensors, providing limited expressiveness. In contrast, ROLLER does not limit the dimension of tile shape and can be applied to general tensor expressions. Triton [229] proposes a new intermediate representation (IR) to describe computation with tiles in regular shapes (e.g., power of 2). ROLLER instead decides the rTile shape according to both hardware features and tensor shapes. MLIR [155] and Tensor IR [228] plan to support block-level (i.e., tile) computation representation in their IRs. ROLLER's rTile abstraction and the rProgram construction are compatible with these initiatives.

Graph-level DNN compilers like XLA [227], TVM [39], and Rammer [145] focus on cross-operator optimizations, e.g., operator fusion/co-scheduling. ROLLER's kernel generation is compatible with these compilers. ROLLER's rTile abstraction complements the rTask concept in Rammer [145] as it provides an efficient way to construct an rTask.

Finally, some works focus on operator-specific optimizations. CUTLASS [176] is a template for implementing matrix-multiplication. An analytical model [134] is proposed to find the best loop-level optimization configuration only for convolution operators on multi-core CPUs. Roller's optimization approach is general for DNN operators on various devices.

6.8 Conclusion

Roller takes an unconventional approach to deep learning compiler. Instead of relying on costly machine learning algorithms to find a good solution in a large search space, Roller generates efficient kernels using a recursive construction-based algorithm that leverages the new rTile abstraction with much fewer shapes that align with multiple hardware features. The constructed program can be evaluated by a micro performance model, without running on a real device every time. As a result, Roller can compile high-performance kernels in seconds, even in less mature accelerators. Roller offers a unique opportunity to significantly speed up DNN kernel development cycles, especially for new hardware vendors.

Chapter 7

Conclusions and Future Work

DNN training workloads are extremely time consuming. In practice, the performance characteristics could vary significantly with respect to the trained models and the software/hardware deployments that ML practitioners use. Unfortunately, analyzing performance bottlenecks of DNN training workloads in practical deployments is difficult due to several reasons. In Chapter 2, we described three key challenges for profiling DNN training workloads to promote training performance: the diversity of DNN optimizations, the complexity of the hardware components in a training system, and the abstraction gap between high-level applications and low-level hardware details.

In this dissertation, we first propose the `TBD` benchmark suite for DNN training workloads. `TBD` covers *six* major DNN application domains with *nine* state-of-the-art representative models. Each of these models has high impact on the development of other DNN models. We then explore profiling techniques that enables effective performance improvement for DNN training workloads under three different scenarios.

First, we propose a set of performance metrics to illustrate potential performance bottlenecks, and built an end-to-end profiling toolchain to extract these metrics by utilizing low-level hardware traces and counters. Our proposed performance metrics can demonstrate the effective solutions to the workloads in practice, and our profiling results emphasize the significance of CPU runtime being the potential bottlenecks. Meanwhile, our memory profiling tool is merged to the main branch of MXNet framework, which can serve a wider range of the community.

Second, we propose a dependency graph analysis approach to accurately estimate the efficacy of a wide range of system-level optimizations for DNN training workloads. To address the unique challenges in what-if explorations in ML context, we implement a prototype system called Daydream with three key features: i) constructing a kernel-granularity graph and tracking dependencies among low-level traces; ii) mapping the low-level traces to high-level DNN operators/layers in a synchronization manner; iii) representing complex optimizations with simple graph-transformation primitives. Our evaluation shows that Daydream can accurately predict the performance improvement for a wide range of optimizations.

Third, we propose a novel approach to generate tensor programs with a construction-based policy, which is fundamentally different from prior search-based works. Our proposed tensor compiler, Roller, manages to reduce the compilation time for generating high-performance kernel programs to the scale of seconds with the follow three key insights: i) leveraging an abstraction for the underlying hardware as a data processing pipeline, in which the key performance bottleneck lies in the pipeline stages; ii) viewing the tensor computation as a combination of tiles, which shapes determine the ratio between compute and memory; iii) utilizing a scale-up-scale-out construction policy, which traverses over tile shapes that satisfy the alignment to key hardware features. Our evaluation demonstrate that Roller is able to generate tensor programs with optimal performance within only seconds, for a wide range of operators and accelerators.

7.1 Future Work Directions

This dissertation illustrates the potential of using profiling techniques and white-box performance model to optimize DNN training workloads in practice, as well as completely new problems and opportunities. We conclude our dissertation with three promising research directions.

7.1.1 Overhead Introduced by Profilers

In Chapter §4 we illustrate the necessity of contracting dependency graph at kernel-level granularity. Our initial empirical studies show that existing hardware profilers might introduce non-negligible overhead to the low-level traces, and the overhead could be significantly different for traces on different hardware. For example, the duration of an individual CUDA API measured by CUPTI [174] could be much higher than the duration of the same API measured by nvprof [171], while the duration of a GPU kernel measured by both tools is almost identical. While within one training iteration, there are usually thousands of CUDA APIs invoked, small overhead for individual CUDA APIs accumulated might lead to significant error when estimating overhead on CPU runtime using CUPTI [174]. As a result, a performance profiling based on such tools will very likely lead to inaccurate insights about performance bottlenecks.

In order to produce accurate profile for DNN training workloads, a comprehensive study towards the overhead introduced by hardware profilers is necessary. We aim to answer the following questions: (i) how the existing profilers alter the duration or counters for each type of low-level traces (including traces on both CPUs and accelerators). (ii) how can we deduct accurate insights about performance based on inaccurate hardware profilers.

7.1.2 Automatic Optimizations

While the profiling techniques proposed in this dissertation can reduce the effort of exploration to identify effective optimizations, it usually still requires a substantial amount of human expertise to (i) identify key hyper-parameters induced by the optimizations for specific software/hardware deployments (e.g. how to partition a layer in model parallelism) that are critical to performance, (ii) implement these optimizations. One of the general challenges that makes it almost impossible for automatic optimizations, is that effective optimizations could be extremely arbitrary. For example, to reduce the runtime on accelerators, one can try to improve the implementation of tensor programs, or directly reduce the numerical precision. It is extremely hardware to estimate and automatically implement these techniques without knowledge of significant amount of implementation details.

Despite this challenge, prior works [198, 39, 266, 265] have shown that it is possible to automatically generate tensor programs with high performance. These tensor compilers address this challenge by identifying a limit amount of implementation choices that are critical to kernel performance, and represents individual kernel implementation using a configuration of these implementation choices. This idea can be potentially migrated to optimizations that operate at higher-level abstractions (e.g. the operator fusion based on an operator graph, the parallelism configuration for data/model parallelism). Once optimization scope and the configuration space for potential implementations are well defined, we can leverage a white-box performance model (e.g. a model based on kernel-level dependency graph) to quickly estimate the performance of each configuration, as well as heuristic search algorithm to automatically identify the implementation with optimal performance.

7.1.3 Extension from Roller's Tensor Compilation: A New Hardware Contract

Roller's construction-based technique is one step towards breaking the barriers among the arithmetic, the software stack and the architecture. We believe Roller represents a future trend by enabling much simpler and efficient software-hardware co-design. In order to allow Roller to be utilized by potentially more AI accelerators and have a greater impact, we tend to extend it in the following two aspects:

- **Standard micro benchmarks and performance-critical hardware specifications** Roller's micro performance estimation is based on hardware counters such as memory or cache bandwidths, compute FLOPS, and how these hardware resources are shared among different compute units. These counters however, are not always publicly available (e.g. the bandwidth of shared memory in AMD MI50 GPU). In our current implementation, we design a set of micro benchmark programs to probe the required counters when they are missing. These programs are carefully crafted to deliver accurate measurement. For example, when estimating the shared memory bandwidth, we need to make sure that the measured programs are not bottlenecked by either global memory or compute. We intend to propose a standard for the performance-critical hardware counters based on compiler's perspective, and how to probe them with micro benchmark programs so that we can achieve accurate and fast performance estimation on newly-proposed accelerators.

- **An extension of RISC ISA with tile-based instructions** Roller employs a code generator module which takes in a multi-level tiling schedule and generates deployable device code. Our current implementation of Roller code generator is based on TVM's loop-based IR. The mismatch between the loop-based IR and our tile-based code template extremely complicates our implementation and could potentially introduce unexpected performance bugs. Meanwhile, current software stack requires the hardware vendors to propose a supportive software chain, ranged from low-level implementation of Assembly-level instructions (e.g. RISC ISA), to a programming platform with dedicated APIs for users to compose device code (e.g. CUDA for NVIDIA GPUs), as well as a code compiler (e.g. nvcc for CUDA) that transfers user-level code to low-level ISA. We aim to propose an extension to the existing RISC ISA with tile-based instructions, so that it would be easier for newly-proposed accelerators to benefit from Roller. Noticeably, we saw that tile-based IR was already been proposed at the abstraction of device code level [67], and we believe such IR will have greater impact for the full software/hardware stack in the future.

Appendix A

Daydream's Code Samples

As shown in Table 4.1, there are a wide range of DNN optimizations, which would introduce various impacts on the training runtime. One of such impacts is that duration of tasks in will scale/shrink. For example, using AMP will shrink the duration of GPU kernels. Using Daydream, such impact is easy to model with the help of the `Select` operator to pick tasks of interests.

DNN optimizations might alter the network topology (e.g. kernel fusion [19], MetaFlow [112]), TASO [113], introduce new operators (e.g. Gist [106], vDNN [206], Deep Gradient Compression [137]), or restructuring the communication scheme (e.g., P3 [107], BlueConnect [43]). These optimizations will eventually alter the low-level dependency graph, adding or removing GPU kernels and communication primitives. Daydream provides `Insert/Remove` operators for programmers to model these transformations. Programmers need to locate where tasks are inserted/removed with the help of the `Select` operator. As we will show later, this locating varies across different optimizations, but is generally not complicated.

Rescheduling tasks is another transformation that needs to be supported in Daydream. This operator does not change the dependency graph topology or the task duration. Instead, it manipulates the execution order of the tasks, and aims at higher parallelism among the tasks. One example of such transformation is the prioritization scheme in P3 [107]. Modeling this scheme involves just overriding the `Scheduling` function in the simulation process 1. Programmers might need to attach additional attributes to the tasks to implement a custom scheduling policy. In the optimizations we show below, modeling P3 [107] and vDNN [206] require overriding the `Scheduling` function.

A.1 Automatic Mixed Precision (AMP)

To model AMP, we shrink the duration of GPU kernels by 2×. If TensorCore is available on the GPU, compute intensive kernels such as `sgemm` are expected to speed up by 3× [175]. We show the pseudo code in Algorithm 3.

A.2 Fused Adam Optimizer

The Fused Adam optimizer fuses all the kernels in the weight update phase. To model this optimizer, we remove all but one kernels in the weight update phase, and scale the duration of the remaining kernels with the sum of all fused ones. We show the pseudo code in Algorithm 4.

Algorithm 3: What_If_AMP

```
Input  : Dependency graph: $G(V, E)$
Output: An updated graph $G(V, E)$ to model AMP
1 $GPUTasks \leftarrow \{G.Select(funcPtr(IsOnGPU))\}$
2 foreach $u \in GPUTasks$ do
3     if $"sgemm"$ $in$ $u.Name$ $or$ $"scudnn"$ $in$ $u.Name$ then
4         $u.duration \leftarrow u.duration/3$
5     else
6         $u.duration \leftarrow u.duration/2$
7     end
8 end
```

Algorithm 4: What_If_Fused_Adam

```
Input  : Dependency graph: $G(V, E)$
Output: Am updated graph $G(V, E)$ to model the Fused_Adam optimizer
1 $GPUTasks \leftarrow \{G.Select(funcPtr(IsOnGPU))\}$
2 $WUTasks \leftarrow GPUTasks.Select(funcPtr(IsWeightUpdate))$
3 $WUSum \leftarrow 0$
4 foreach $u \in WUTasks$ do
5     $WUSum \leftarrow WUSum + u.duration$
6 end
7 $First \leftarrow True$
8 foreach $u \in WUTasks$ do
9     if $First$ then
10        $u.duration \leftarrow WUSum$
11        $First \leftarrow False$
12    else
13        $G.Remove(u)$
14 end
```

A.3 Reconstructing Batchnorm

Reconstructing Batchnorm [119] improves the performance of training CNNs by splitting batch normalization layers and fusing memory-intensive kernels with compute-intensive kernels. We show the pseudo-code of using Daydream to model this optimization. We show the pseudo code in Algorithm 5.

Algorithm 5: What_If_Restructuring_Batchnorm

```
Input  : Dependency graph: $G(V, E)$
Output: An updated graph $G(V, E)$ to model Restructuring_Batchnorm
1 $GPUTasks \leftarrow \{G.Select(funcPtr(IsOnGPU))\}$
2 foreach $u \in GPUTasks$ do
3     if $u.layer$ is $ReLU$ then
4         $G.Remove(u)$
5     end
6     if $u.layer$ is $Batchnorm$ then
7         $u.duration \leftarrow u.duration/2$
8     end
9 end
```

A.4 Distributed Training

We show how to use Daydream to model distributed training in PyTorch's decentralized architecture with the NCCL backend, based on runtime on a single GPU. When invoking NCCL all-reduce primitives, PyTorch groups small gradient tensors together to better utilize the bandwidth. Such grouping information can be collected by instrumentation from the PyTorch framework. In our code example, we use `layer_bucket_id` to represent the mapping from layers to communication buckets. Each bucket corresponds to one communication call. We show the pseudo code in Algorithm 6.

A.5 Priority-based Parameter Propagation (P3)

P3 [107] splits each gradient tensor into small slices and reschedules the communication based on the order in which gradient tensors are generated. We show how to model P3 based on MXNet's parameter server architecture (with push/pull communication primitives). To model P3 with Daydream, we insert parallel push/pull primitives for each gradient slice, tag each slice with priority based on the generation order, and override the `Schedule` function to model the prioritization scheme.

A.6 BlueConnect

BlueConnect [43] optimizes the bandwidth usage by decomposing the synchronous all-reduce operations into a series of reduce-scatter and all-reduce operations. The decomposition helps better utilize the heterogeneous intra-node and inter-node bandwidths. The decomposition of all-reduce operations is based on a factorization of the number of GPUs. We show the pseudo code in Algorithm 8.

Algorithm 6: What_If_Distributed_Training

```
Input  : Dependency graph: $G(V, E)$, Gradient Grouping: layer_bucket_id[]
Output: An updated graph $G(V, E)$ to model distributed training
$GPUTasks \leftarrow \{G.Select(funcPtr(IsOnGPU))\}$
$Bucket\_Task \leftarrow []$
$WU \leftarrow$ the earliest node in the weight update phase
foreach $b \in [1..\#\_of\_bucket]$ do
    $AllReduceTask = newNode("AllReduce", ...)$
    $AllReduceTask.size \leftarrow 0$
    $G.AddDependencies(AllReduceTask \rightarrow WU)$
    $Bucket\_Task[n] \leftarrow AllReduceTask$
end
foreach $u \in GPUTasks$ do
    if $u$ is $FF\_layer$ then
        $bucket\_id \leftarrow layer\_bucket\_i[u]$
        $T \leftarrow Bucket\_Task[bucket\_id]$
        $T.size \leftarrow T.size + u.gradient\_size$
        $G.AddDependencies(u \rightarrow T)$
    end
end
```

A.7 MetaFlow

MetaFlow [112] is a relaxed graph substitution optimizer. It simplifies the layer representation of a DNN topology by using operations like enlarging convolution kernel dimensions and layer fusion. The policy to transform the layer-wise topology is determined by a backtracking search algorithm. Daydream does not provide extra support that automatically determines the policy, as this is a duplicated work.

A transformation policy of MetaFlow will eventually remove or scale the dimension of existing layers. Given a policy, Daydream can estimate its performance by modeling layer-wise removal/scaling operations, with the help of layer mapping (described in Section §4.4.3). We show Daydream's pseudo code of implementing these two operations in Algorithm 9.

MetaFlow's search algorithm uses a cost model to evaluate the performance of a given policy. Daydream can be used as a more precise cost model for the search algorithm.

A.8 Virtualized DNN (vDNN)

Virtualized DNN [206] optimizes the memory footprint in CNN training by offloading feature maps from GPU memory to CPU memory. To model vDNN with Daydream, we only need to insert the corresponding cudaMemcpy calls, and implement prefetching strategy by using the overriding `Schedule` function. The custom `Schedule` function delays the execution of the prefetching operation. We demonstrate how to model the $\texttt{vDNN}_{conv}$ policy, which only offloads the feature maps of all convolutional layers. We tag each layer with an ID (a layer with higher ID means closer to the output layer), and use the `findPrefetchLayer` function defined in the original vDNN paper [206]. We show the pseudo code in Algorithm 10.

Algorithm 7: What_If_P3

Input : Dependency graph: $G(V, E)$, slice_size
Output: An updated graph $G(V, E)$ to model P3

```
// Select GPU tasks in BP and FF
$GPUTasks \leftarrow \{G.Select(funcPtr(IsOnGPU))\}$
foreach $u \in GPUTasks$ do
    $v \leftarrow u$'s corresponding BP layer
    $g \leftarrow$ —$u.layer$'s gradients—
    while $g > 0$ do
        $s \leftarrow \min(g,\ slice\_size)$
        $push \leftarrow$ newNode("push v.layer", s, ...)
        $pull \leftarrow$ newNode("pull v.layer", s, ...)
        $push.priority \leftarrow$ -(distance to output)
        $push.ExecutionThread \leftarrow comm.send$
        if this slice is stored on the first server then
            $pull.ExecutionThread \leftarrow comm.send$
        else
            $pull.ExecutionThread \leftarrow comm.receive$
        $G.AddDependencies(u \rightarrow push \rightarrow pull \rightarrow v)$
        $g \leftarrow g - slice\_size$
    end
end
Function Schedule(TaskQueue: Q):
    $earliest \leftarrow Q.first()$
    $thread \leftarrow earliest.ExecutionThread$
    $time \leftarrow max(P[thread], earliest.start)$
    foreach $task \in Q$ do
        $this\_thread \leftarrow task.ExecutionThread$
        $this\_time \leftarrow max(P[this\_thread], task.start)$
        if $this\_time < time$ then
            $time \leftarrow this\_time$
            $earliest \leftarrow task$
        end
        if $this\_time = time \wedge task$ is push/pull $\wedge$ $earliest$ is
          push/pull $\wedge$ $task.priority > earliest.priority$ then
            $earliest \leftarrow task$
        end
    end
    return $earliest$
End Function
```

Algorithm 8: What_If_BlueConnect

Input : Dependency graph of distributed training: $G(V, E)$, decomposition factorization: $p_1 p_2 ... p_k$

Output: Am updated graph $G(V, E)$ to model BlueConnect

```
1  $ReduceTasks \leftarrow \{G.Select(funcPtr(IsAllReduce))\}$
2  foreach $u \in ReduceTasks$ do
3      $s \leftarrow u.prevNodes$
4      $t \leftarrow u.postNodes$
5      $G.Remove(u)$
6      foreach $i \leftarrow 1..k$ do
7          $RSNode \leftarrow new(Reduce\_Scatter\_Node(p_i))$
8          $G.Insert(s, RSNode, t)$
9          $s \leftarrow RSNode$
10     end
11     foreach $i \leftarrow k..1$ do
12         $AGNode \leftarrow new(All\_Gather\_Node(p_i))$
13         $G.Insert(s, AGNode, t)$
14         $s \leftarrow AGNode$
15     end
16 end
```

Algorithm 9: What_If_MetaFlow

Input : Dependency graph: $G(V, E)$

Output: An updated graph $G(V, E)$ to model MetaFlow

```
1  Function Remove_layer(Dependency Graph: G(V, E), Layer: l):
2      $GPUTasks \leftarrow \{G.Select(funcPtr(IsOnGPU))\}$
3      foreach $u \in GPUTasks$ do
4          if $u.layer$ is $l$ then
5              $G.Remove(u)$
6          end
7      end
8  End Function
9  Function Scale_layer(Dependency Graph: G(V, E), Layer: l):
10     $GPUTasks \leftarrow \{G.Select(funcPtr(IsOnGPU))\}$
11     foreach $u \in GPUTasks$ do
12         if $u.layer$ is $l$ then
13             $u.duration \leftarrow u.duration \times s$
14         end
15     end
16 End Function
```

Algorithm 10: What_If_vDNN

Input : Dependency graph: $G(V, E)$

Output: An updated graph $G(V, E)$ to model vDNN

```
1  $GPUTasks \leftarrow \{G.Select(funcPtr(IsOnGPU))\}$
2  $ID2PrefetchTask \leftarrow \{\}$
3  **foreach** $u \in GPUTasks$ **do**
4      **if** $u.layer\ is\ not\ CONV\_FF$ **then**
5          $continue$
6      **end**
7      $v \leftarrow u$'s corresponding BP layer
8      $t1 \leftarrow newCPUNode(\text{"}cudaMemcpyLaunch\text{"}, ...)$
9      $t2 \leftarrow newGPUNode(\text{"}cudaMemcpyH2D\text{"}, ...)$
10     $t3 \leftarrow newCPUNode(\text{"}cudaFree\_vDNN\text{"}, ...)$
11     $t4 \leftarrow newCPUNode(\text{"}cudaMalloc\_vDNN\text{"}, ...)$
12     $ID2PrefetchTask[u.ID] \leftarrow t4$
13     $t5 \leftarrow newCPUNode(\text{"}cudaMemcpyLaunch\text{"}, ...)$
14     $t6 \leftarrow newGPUNode(\text{"}cudaMemcpyD2H\text{"}, ...)$
15     $G.addDependencies(u \rightarrow t1 \rightarrow t2 \rightarrow t3 \rightarrow t4 \rightarrow t5 \rightarrow t6 \rightarrow v)$
16 **end**
17 **Function** Schedule($TaskQueue$: $Q$):
18     $GPUTasks \leftarrow \{G.Select(funcPtr(IsOnGPU))\}$
19     $next \leftarrow Q.last()$
20     **if** $next.layer\ is\ BP$ **then**
21         $l \leftarrow findPrefetchLayer(next.ID)$
22         **if** $l \neq -1$ **then**
23             **return** $next$
24         **else**
25             **return** $ID2PrefetchTask[l]$
26         **end**
27     **end**
28 **End Function**
```

A.9 Gist

Gist [106] is an technique that optimizes the memory footprint when training CNNs. It reduces the memory consumption of the intermediate feature maps by adding encoding/decoding operations to the training iterations. Gist provides both lossless and lossy compression strategies. We can use Daydream to estimate the performance overhead of Gist, by inserting the encoding/decoding kernels. When estimating the lossless compression, we need to insert GPU kernels that are either element-wise kernels (including clamping, pooling-mapping, bit-wise kernels, etc.), or cuSPARSE kernels. When estimating the lossy compression, we need to additionally insert the GPU kernels that perform Delayed Precision Reduction (DPR) scheme.

Note that estimating the duration of these kernels is crucial to the prediction accuracy. The duration of these kernels can be either inferred based on existing kernels, or profiled separately (the latter is outside of Daydream's focus and should be resolved using other techniques). We show the pseudo code in Algorithm 11.

Algorithm 11: What_If_Gist

```
Input  : Dependency graph: $G(V, E)$
Output: An updated graph $G(V, E)$ to model Gist
1  $GPUTasks \leftarrow \{G.Select(funcPtr(IsOnGPU))\}$
2  foreach $u \in GPUTasks$ do
3      $v \leftarrow u.postNode$
4      $w \leftarrow v.postNode$
5      if $u.layer\ is\ RELU\_FF \wedge v.layer\ is\ POOL\_FF \wedge w.layer\ is\ CONV\_FF$ then
6          $SSDC\_kernels \leftarrow newNode(...)$
7          $G.insert(v, SSDC, w)$
8      end
9      if $u.layer\ is\ RELU\_FF \wedge v.layer\ is\ POOL\_FF$ then
10         $Binarize \leftarrow newNode(...)$
11         $G.insert(v, Binarize, w)$
12     end
13 end
14 if $LOSSY\_COMPRESSION$ then
15     foreach $u \in GPUTasks$ do
16         if $u\ is\ not\ RELU$ then
17             $v \leftarrow u.postNode$
18             $DPR \leftarrow newNode(...)$
19             $G.insert(u, DPR, v)$
20         end
21     end
22 end
   // Add decode kernels to the backward pass
23 ...
```

A.10 Deep Gradient Compression (DGC)

DGC [137] reduces communication overhead by compressing the gradients before transmission and decompressing the gradients before weight update phase. When using Daydream to estimate the performance overhead of DGC, we need to insert the compression/decompression kernels before/after the communication primitives. Similar to Gist, the prediction accuracy mainly depends on the estimation of the inserted kernels. We show the pseudo code in Algorithm 12.

Algorithm 12: What_If_DGC

Input : Dependency graph: $G(V, E)$
Output: An updated graph $G(V, E)$ to model Deep Gradient Compression

```
1  $ReduceTasks \leftarrow \{G.Select(funcPtr(IsAllReduce))\}$
2  foreach $r \in ReduceTasks$ do
3      $s \leftarrow r.prevNodes()$
4      $t \leftarrow r.postNodes()$
       // Initialize compression kernels
5      $quantize\_op \leftarrow newNode(...)$
6      $sparse\_op \leftarrow newNode(...)$
7      ...
8      $G.Insert(s, quantize\_op, r)$
9      $G.Insert(quantize\_op, sparse\_op, r)$
10     ...
       // Initialize decompression kernels
11     $d\_kernels \leftarrow ...$
12     $G.Insert(r, d\_kernels, t)$
13 end
```

Bibliography

[1] *A BLAS implementation on top of ROCm*. https://rocmdocs.amd.com/en/latest/ROCm_Tools/rocblas.html.

[2] Martın Abadi et al. "Tensorflow: A system for large-scale machine learning". In: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 2016, pp. 265–283.

[3] ACL. *Shared Task: Machine Translation of News*. http://www.statmt.org/wmt16/translation-task.html. 2016.

[4] Robert Adolf et al. "Fathom: reference workloads for modern deep learning methods". In: *Workload Characterization (IISWC), 2016 IEEE International Symposium on*. IEEE. 2016, pp. 1–10.

[5] Marcos K Aguilera et al. "Performance debugging for distributed systems of black boxes". In: *ACM SIGOPS Operating Systems Review*. Vol. 37. 5. ACM. 2003, pp. 74–89.

[6] Jasmin Ajanovic. "PCI Express*(PCIe*) 3.0 Accelerator Features". In: *Intel Corporation* 10 (2008).

[7] Takuya Akiba, Shuji Suzuki, and Keisuke Fukuda. "Extremely large minibatch SGD: training resnet-50 on imagenet in 15 minutes". In: *arXiv preprint arXiv:1711.04325* (2017).

[8] Jorge Albericio et al. "Bit-pragmatic deep neural network computing". In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM. 2017, pp. 382–394.

[9] Jorge Albericio et al. "Cnvlutin: ineffectual-neuron-free deep neural network computing". In: *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE. 2016, pp. 1–13.

[10] Dan Alistarh et al. "QSGD: Communication-efficient SGD via gradient quantization and encoding". In: *Advances in Neural Information Processing Systems*. 2017, pp. 1709–1720.

[11] Johnathan Alsop et al. "Optimizing GPU cache policies for MI workloads". In: *2019 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2019, pp. 243–248.

[12] Manoj Alwani et al. "Fused-layer CNN accelerators". In: *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE. 2016, pp. 1–12.

[13] AMD. *AMD EPYC™ 7601*. https://www.amd.com/en/products/cpu/amd-epyc-7601. 2019.

[14] *AMD Radeon Instinct™ MI50 Accelerator*. https://www.amd.com/en/products/professional-graphics/instinct-mi50.

[15] *AMD ROCm Platform*. https://github.com/RadeonOpenCompute/ROCm.

[16] *AMD Prof*. https://developer.amd.com/amd-uprof/.

[17] Dario Amodei et al. "Deep speech 2: End-to-end speech recognition in English and Mandarin". In: *International conference on machine learning*. 2016, pp. 173–182.

[18] Martin Arjovsky, Soumith Chintala, and Léon Bottou. "Wasserstein gan". In: *arXiv preprint arXiv:1701.07875* (2017).

[19] Arash Ashari et al. "On optimizing machine learning workloads via kernel fusion". In: *ACM SIGPLAN Notices*. Vol. 50. 8. ACM. 2015, pp. 173–182.

[20] Riyadh Baghdadi et al. "Tiramisu: A polyhedral compiler for expressing fast and portable code". In: *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2019, pp. 193–205.

[21] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural machine translation by jointly learning to align and translate". In: *arXiv preprint arXiv:1409.0473* (2014).

[22] Colby R Banbury et al. "Benchmarking TinyML systems: Challenges and direction". In: *arXiv preprint arXiv:2003.04821* (2020).

[23] *Benchmarking DNN Processors*. http://eyeriss.mit.edu/benchmarking.html.

[24] Yoshua Bengio et al. "Greedy layer-wise training of deep networks". In: *Advances in neural information processing systems*. 2007, pp. 153–160.

[25] James Bergstra et al. "Theano: a CPU and GPU math expression compiler". In: *Proceedings of the Python for scientific computing conference (SciPy)*. Vol. 4. 3. Austin, TX. 2010.

[26] Jeremy Bernstein et al. "signSGD: Compressed optimisation for non-convex problems". In: *International Conference on Machine Learning*. PMLR. 2018, pp. 560–569.

[27] Michaela Blott et al. "QuTiBench: Benchmarking neural networks on heterogeneous hardware". In: *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 15.4 (2019), pp. 1–38.

[28] Mariusz Bojarski et al. "End to end learning for self-driving cars". In: *arXiv preprint arXiv:1604.07316* (2016).

[29] Mahdi Nazm Bojnordi and Engin Ipek. "Memristive boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning". In: *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*. IEEE. 2016, pp. 1–13.

[30] Léon Bottou. "Large-scale machine learning with stochastic gradient descent". In: *Proceedings of COMPSTAT'2010*. Springer, 2010, pp. 177–186.

[31] Tom Brown et al. "Language Models are Few-Shot Learners". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901. URL: https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

[32] Mauro Cettolo et al. "The IWSLT 2015 evaluation campaign". In: *IWSLT 2015, International Workshop on Spoken Language Translation*. 2015.

[33] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks". In: *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE. 2016, pp. 367–379.

[34] Yu-Hsin Chen et al. "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks". In: *IEEE Journal of Solid-State Circuits* 52.1 (2017), pp. 127–138.

[35] Tianqi Chen et al. "Learning to optimize tensor programs". In: *arXiv preprint arXiv:1805.08166* (2018).

[36] Tianqi Chen et al. "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems". In: *CoRR* abs/1512.01274 (2015). arXiv: 1512.01274. URL: http://arxiv.org/abs/1512.01274.

[37] Tianqi Chen et al. "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems". In: *arXiv preprint arXiv:1512.01274* (2015).

[38] Tianqi Chen et al. "Training deep nets with sublinear memory cost". In: *arXiv preprint arXiv:1604.06174* (2016).

[39] Tianqi Chen et al. "TVM: End-to-End Optimization Stack for Deep Learning". In: *CoRR* abs/1802.04799 (2018). arXiv: 1802.04799. URL: http://arxiv.org/abs/1802.04799.

[40] Sharan Chetlur et al. "cudnn: Efficient primitives for deep learning". In: *arXiv preprint arXiv:1410.0759* (2014).

[41] Ping Chi et al. "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory". In: *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press. 2016, pp. 27–39.

[42] Trishul M Chilimbi et al. "Project Adam: Building an Efficient and Scalable Deep Learning Training System." In: *OSDI*. Vol. 14. 2014, pp. 571–582.

[43] Minsik Cho et al. "BlueConnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy". In: *IBM Journal of Research and Development* 63.6 (2019), pp. 1–1.

[44] François Chollet et al. *Keras*. https://github.com/fchollet/keras. 2015.

[45] Patryk Chrabaszcz, Ilya Loshchilov, and Frank Hutter. "A Downsampled Variant of ImageNet as an Alternative to the CIFAR datasets". In: *arXiv preprint arXiv:1707.08819* (2017).

[46] *cnn-benchmarks*. https://github.com/jcjohnson/cnn-benchmarks.

[47] Cody Coleman et al. "Dawnbench: An end-to-end deep learning benchmark and competition". In: *Training* 100.101 (2017), p. 102.

[48] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. "Torch7: A matlab-like environment for machine learning". In: *BigLearn, NIPS workshop*. EPFL-CONF-192376. 2011.

[49] *convnet-benchmarks*. https://github.com/soumith/convnet-benchmarks.

[50] Paul Covington, Jay Adams, and Emre Sargin. "Deep neural networks for youtube recommendations". In: *Proceedings of the 10th ACM Conference on Recommender Systems*. ACM. 2016, pp. 191–198.

[51] *CUDA NVCC*. https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/.

[52] Charlie Curtsinger and Emery D Berger. "C oz: finding code that counts with causal profiling". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM. 2015, pp. 184–197.

[53] Dipankar Das et al. "Mixed precision training of convolutional neural networks using integer operations". In: *arXiv preprint arXiv:1802.00930* (2018).

[54] Christopher De Sa et al. "Understanding and optimizing asynchronous low-precision stochastic gradient descent". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM. 2017, pp. 561–574.

[55] Jeffrey Dean et al. "Large scale distributed deep networks". In: *Advances in neural information processing systems*. 2012, pp. 1223–1231.

[56] *Deep Learning Profiler*. https://docs.nvidia.com/deeplearning/frameworks/dlprof-user-guide/.

[57] *DeepBench*. https://github.com/baidu-research/DeepBench.

[58] Jia Deng et al. "Imagenet: A large-scale hierarchical image database". In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255.

[59] Jacob Devlin et al. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805* (2018).

[60] Caiwen Ding et al. "C ir CNN: accelerating and compressing deep neural networks using block-circulant weight matrices". In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM. 2017, pp. 395–408.

[61] Zidong Du et al. "Neuromorphic accelerators: A comparison between neuroscience and machine-learning approaches". In: *Proceedings of the 48th International Symposium on Microarchitecture*. ACM. 2015, pp. 494–507.

[62] Zidong Du et al. "ShiDianNao: Shifting vision processing closer to the sensor". In: *ACM SIGARCH Computer Architecture News*. Vol. 43. 3. ACM. 2015, pp. 92–104.

[63] *Eigen: A C++ linear algebra library*. http://eigen.tuxfamily.org/index.php?title=Main_Page.

[64] Mark Everingham et al. "The pascal visual object classes (voc) challenge". In: *International journal of computer vision* 88.2 (2010), pp. 303–338.

[65] Mingyu Gao et al. "TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory". In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM. 2017, pp. 751–764.

[66] Wanling Gao et al. "AIBench: an industry standard internet service AI benchmark suite". In: *arXiv preprint arXiv:1908.08998* (2019).

[67] Evangelos Georganas et al. "Tensor Processing Primitives: A Programming Abstraction for Efficiency and Portability in Deep Learning Workloads". In: *CoRR* abs/2104.05755 (2021). arXiv: 2104.05755. URL: https://arxiv.org/abs/2104.05755.

[68] James Gleeson et al. “RL-Scope: Cross-stack Profiling for Deep Reinforcement Learning Workloads”. In: *Proceedings of Machine Learning and Systems* 3 (2021).

[69] Google. *Cloud TPU Tools*. https://cloud.google.com/tpu/docs/cloud-tpu-tools. 2018.

[70] Priya Goyal et al. “Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour”. In: *arXiv preprint arXiv:1706.02677* (2017).

[71] Hui Guan, Xipeng Shen, and Seung-Hwan Lim. “Wootz: A compiler-based framework for fast CNN pruning via composability”. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2019, pp. 717–730.

[72] Yijin Guan et al. “FPGA-based accelerator for long short-term memory recurrent neural networks”. In: *Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*. IEEE. 2017, pp. 629–634.

[73] Ishaan Gulrajani et al. “Improved training of wasserstein gans”. In: *Advances in Neural Information Processing Systems*. 2017, pp. 5769–5779.

[74] Suyog Gupta et al. “Deep learning with limited numerical precision”. In: *International Conference on Machine Learning*. 2015, pp. 1737–1746.

[75] Udit Gupta et al. “Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference”. In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2020, pp. 982–995.

[76] Udit Gupta et al. “The architectural implications of facebook’s dnn-based personalized recommendation”. In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2020, pp. 488–501.

[77] Farzin Haddadpour et al. “Local sgd with periodic averaging: Tighter analysis and adaptive synchronization”. In: *arXiv preprint arXiv:1910.13598* (2019).

[78] Ubaid Ullah Hafeez and Anshul Gandhi. “Empirical Analysis and Modeling of Compute Times of CNN Operations on AWS Cloud”. In: *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2020, pp. 181–192.

[79] Ameer Haj-Ali et al. “NeuroVectorizer: End-to-end vectorization with deep reinforcement learning”. In: *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 2020, pp. 242–255.

[80] Ameer Haj-Ali et al. “ProTuner: tuning programs with Monte Carlo tree search”. In: *arXiv preprint arXiv:2005.13685* (2020).

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

[82] Song Han, Huizi Mao, and William J. Dally. “Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding”. In: *International Conference on Learning Representations (ICLR 2016)* (2016).

[83] Song Han et al. “EIE: efficient inference engine on compressed deep neural network”. In: *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press. 2016, pp. 243–254.

[84] Tianshu Hao et al. “Edge AIBench: towards comprehensive end-to-end edge computing benchmarking”. In: *International Symposium on Benchmarking, Measuring and Optimization*. Springer. 2018, pp. 23–30.

[85] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H Campbell. “TicTac: Accelerating distributed deep learning with communication scheduling”. In: *arXiv preprint arXiv:1803.03288* (2018).

[86] Johann Hauswald et al. “DjiNN and Tonic: DNN as a service and its implications for future warehouse scale computers”. In: *ACM SIGARCH Computer Architecture News*. Vol. 43. 3. ACM. 2015, pp. 27–40.

[87] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[88] Kaiming He et al. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[89] Kaiming He et al. “Mask r-cnn”. In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 2961–2969.

[90] Xiangnan He et al. “Neural collaborative filtering”. In: *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee. 2017, pp. 173–182.

[91] Felix Hieber et al. “Sockeye: A Toolkit for Neural Machine Translation”. In: *arXiv preprint arXiv:1712.05690* (2017).

[92] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. “A fast learning algorithm for deep belief nets”. In: *Neural computation* 18.7 (2006), pp. 1527–1554.

[93] *HiSilicon Kirin Chipsets*. https://www.hisilicon.com/en/products/Kirin.

[94] Qirong Ho et al. “More effective distributed ml via a stale synchronous parallel parameter server”. In: *Advances in neural information processing systems*. 2013, pp. 1223–1231.

[95] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Comput.* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: http://dx.doi.org/10.1162/neco.1997.9.8.1735.

[96] Samuel Hsia et al. “Cross-stack workload characterization of deep recommendation systems”. In: *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2020, pp. 157–168.

[97] Kevin Hsieh et al. “Gaia: Geo-Distributed Machine Learning Approaching LAN Speeds.” In: *NSDI*. 2017, pp. 629–647.

[98] Sixu Hu et al. “The oarf benchmark suite: Characterization and implications for federated learning systems”. In: *arXiv preprint arXiv:2006.07856* (2020).

[99] Gao Huang et al. “Densely connected convolutional networks”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 4700–4708.

[100] Qijing Huang et al. “Autophase: Compiler phase-ordering for hls with deep reinforcement learning”. In: *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2019, pp. 308–308.

[101] Xinyuan Huang et al. “Benchmarking deep learning for time series: Challenges and directions”. In: *2019 IEEE International Conference on Big Data (Big Data)*. IEEE. 2019, pp. 5679–5682.

[102] Brody Huval et al. “An empirical evaluation of deep learning on highway driving”. In: *arXiv preprint arXiv:1504.01716* (2015).

[103] Andrey Ignatov et al. “Ai benchmark: Running deep neural networks on android smartphones”. In: *Proceedings of the European Conference on Computer Vision (ECCV) Workshops*. 2018.

[104] *Introducing PyTorch Profiler*. `https://pytorch.org/blog/introducing-pytorch-profiler-the-new-and-improved-performance-tool/`.

[105] *IPU PROGRAMMER’S GUIDE*. `https://www.graphcore.ai/docs/ipu-programmers-guide`.

[106] Animesh Jain et al. “Gist: Efficient Data Encoding for Deep Neural Network Training”. In: *Proceeding of the 45st Annual International Symposium on Computer Architecture*. ISCA 2018. 2018, pp. 776–789. DOI: `10.1109/ISCA.2018.00070`. URL: `https://doi.org/10.1109/ISCA.2018.00070`.

[107] Anand Jayarajan et al. “Priority-based Parameter Propagation for Distributed DNN Training”. In: *Proceedings of Machine Learning and Systems 2019*. 2019, pp. 132–145.

[108] Yu Ji et al. “NEUTRAMS: Neural network transformation and co-design under neuromorphic hardware constraints”. In: *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE. 2016, pp. 1–13.

[109] Yangqing Jia et al. “Caffe: Convolutional architecture for fast feature embedding”. In: *Proceedings of the 22nd ACM international conference on Multimedia*. ACM. 2014, pp. 675–678.

[110] Zhe Jia et al. “Dissecting the graphcore ipu architecture via microbenchmarking”. In: *arXiv preprint arXiv:1912.03413* (2019).

[111] Zhe Jia et al. “Dissecting the NVIDIA volta GPU architecture via microbenchmarking”. In: *arXiv preprint arXiv:1804.06826* (2018).

[112] Zhihao Jia et al. “Optimizing DNN computation with relaxed graph substitutions”. In: *Proc. Conference on Systems and Machine Learning, SysML*. Vol. 19. 2019.

[113] Zhihao Jia et al. “TASO: optimizing deep learning computation with automatic generation of graph substitutions”. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. ACM. 2019, pp. 47–62.

[114] Zihan Jiang et al. “Hpc ai500 v2. 0: The methodology, tools, and metrics for benchmarking hpc ai systems”. In: *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE. 2021, pp. 47–58.

[115] Yang Jiao, Liang Han, and Xin Long. “Hanguang 800 NPU–The Ultimate AI Inference Solution for Data Centers”. In: *2020 IEEE Hot Chips 32 Symposium (HCS)*. IEEE Computer Society. 2020, pp. 1–29.

[116] Norman P Jouppi et al. “In-datacenter performance analysis of a tensor processing unit”. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM. 2017, pp. 1–12.

[117] Norman P. Jouppi et al. “In-Datacenter Performance Analysis of a Tensor Processing Unit”. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA 2017. Toronto, ON, Canada: ACM, 2017, pp. 1–12. ISBN: 978-1-4503-4892-8. DOI: 10.1145/3079856.3080246. URL: http://doi.acm.org/10.1145/3079856.3080246.

[118] Patrick Judd et al. “Stripes: Bit-serial deep neural network computing”. In: *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE. 2016, pp. 1–12.

[119] Wonkyung Jung et al. “Restructuring batch normalization to accelerate CNN training”. In: *arXiv preprint arXiv:1807.01702* (2018).

[120] Aajna Karki et al. “Detailed characterization of deep neural networks on gpus and fpgas”. In: *Proceedings of the 12th Workshop on General Purpose Processing Using GPUs*. 2019, pp. 12–21.

[121] Mahmoud Khairy et al. “Accel-Sim: An extensible simulation framework for validated GPU modeling”. In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2020, pp. 473–486.

[122] Jehandad Khan et al. *MIOpen: An Open Source Library For Deep Learning Primitives*. 2019. arXiv: 1910.00078 [cs.LG].

[123] Duckhwan Kim et al. “Neurocube: A programmable digital neuromorphic architecture with high-density 3D memory”. In: *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE. 2016, pp. 380–392.

[124] Soojeong Kim et al. “Parallax: Sparsity-aware Data Parallel Training of Deep Neural Networks”. In: *Proceedings of the Fourteenth EuroSys Conference 2019*. ACM. 2019, p. 43.

[125] Fredrik Kjolstad et al. “Taco: A tool to generate tensor algebra kernels”. In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2017, pp. 943–948.

[126] Fredrik Kjolstad et al. “The tensor algebra compiler”. In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (2017), p. 77.

[127] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[128] Oleksii Kuchaiev et al. “Openseq2seq: extensible toolkit for distributed and mixed precision training of sequence-to-sequence models”. In: *Proceedings of Workshop for NLP Open Source Software (NLP-OSS)*. 2018, pp. 41–46.

[129] Andrew Lavin and Scott Gray. “Fast Algorithms for Convolutional Neural Networks”. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 4013–4021. DOI: 10.1109/CVPR.2016.435.

[130] Yann LeCun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[131] Jonathan Lew et al. “Analyzing machine learning workloads using a detailed GPU simulator”. In: *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2019, pp. 151–152.

[132] Mu Li et al. “Communication efficient distributed machine learning with the parameter server”. In: *Advances in Neural Information Processing Systems* 27 (2014), pp. 19–27.

[133] Mu Li et al. “Scaling distributed machine learning with the parameter server”. In: *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 2014, pp. 583–598.

[134] Rui Li et al. “Analytical Characterization and Design Space Exploration for Optimization of CNNs”. In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS 2021. Virtual, USA: Association for Computing Machinery, 2021, pp. 928–942. ISBN: 9781450383172. DOI: 10.1145/3445814.3446759. URL: https://doi.org/10.1145/3445814.3446759.

[135] Tzu-Mao Li et al. “Differentiable programming for image processing and deep learning in Halide”. In: *ACM Transactions on Graphics (TOG)* 37.4 (2018), pp. 1–13.

[136] Robert LiKamWa et al. “RedEye: analog ConvNet image sensor architecture for continuous mobile vision”. In: *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press. 2016, pp. 255–266.

[137] Yujun Lin et al. “Deep gradient compression: Reducing the communication bandwidth for distributed training”. In: *arXiv preprint arXiv:1712.01887* (2017).

[138] Jie Liu et al. “Performance analysis and characterization of training deep learning models on mobile device”. In: *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE. 2019, pp. 506–515.

[139] Wei Liu et al. “Ssd: Single shot multibox detector”. In: *European conference on computer vision*. Springer. 2016, pp. 21–37.

[140] Ze Liu et al. “Swin Transformer: Hierarchical Vision Transformer using Shifted Windows”. In: *CoRR* abs/2103.14030 (2021). arXiv: 2103.14030. URL: https://arxiv.org/abs/2103.14030.

[141] Qu Lu et al. “Multi-stage Gradient Compression: Overcoming the Communication Bottleneck in Distributed Deep Learning”. In: *International Conference on Neural Information Processing*. Springer. 2018, pp. 107–119.

[142] Wenyan Lu et al. “FlexFlow: A Flexible Dataflow Accelerator Architecture for Convolutional Neural Networks”. In: *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE. 2017, pp. 553–564.

[143] Cheng Luo et al. “Towards efficient deep neural network training by FPGA-based batch-level parallelism”. In: *Journal of Semiconductors* 41.2 (2020), p. 022403.

[144] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. “Effective approaches to attention-based neural machine translation”. In: *arXiv preprint arXiv:1508.04025* (2015).

[145] Lingxiao Ma et al. "Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 881–897. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/ma.

[146] Yanjun Ma et al. "PaddlePaddle: An open-source deep learning platform from industrial practice". In: *Frontiers of Data and Domputing* 1.1 (2019), pp. 105–115.

[147] Stefano Markidis et al. "Nvidia tensor core programmability, performance & precision". In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2018, pp. 522–531.

[148] Peter Mattson et al. "MLPerf Training Benchmark". In: *Proceedings of Machine Learning and Systems*. Ed. by I. Dhillon, D. Papailiopoulos, and V. Sze. Vol. 2. 2020, pp. 336–349. URL: https://proceedings.mlsys.org/paper/2020/file/02522a2b2726fb0a03bb19f2d8d9524d-Paper.pdf.

[149] Peter Mattson et al. "Mlperf training benchmark". In: *arXiv preprint arXiv:1910.01500* (2019).

[150] Eitan Medina and Eran Dagan. "Habana Labs purpose-built AI inference and training processor architectures: Scaling AI training systems using standard ethernet with Gaudi processor". In: *IEEE Micro* 40.2 (2020), pp. 17–24.

[151] Naveen Mellempudi et al. "Mixed precision training with 8-bit floating point". In: *arXiv preprint arXiv:1905.12334* (2019).

[152] Scott Michael et al. "Performance characteristics of virtualized gpus for deep learning". In: *2020 IEEE/ACM International Workshop on Interoperability of Supercomputing and Cloud Technologies (SuperCompCloud)*. IEEE. 2020, pp. 14–20.

[153] Paulius Micikevicius et al. "Mixed precision training". In: *arXiv preprint arXiv:1710.03740* (2017).

[154] Barton P Miller and Cui-Qing Yang. "IPS: An Interactive and Automatic Performance Measurement Tool for Parallel and Distributed Programs." In: *ICDCS*. 1987, pp. 482–489.

[155] *MLIR*. https://mlir.llvm.org/.

[156] MLPerf. *MLPerf Training Results v0.6*. https://mlperf.org/training-results-0-6. 2019.

[157] Volodymyr Mnih et al. "Asynchronous methods for deep reinforcement learning". In: *International Conference on Machine Learning*. 2016, pp. 1928–1937.

[158] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), p. 529.

[159] Saiful A Mojumder et al. "Profiling dnn workloads on a volta-based dgx-1 system". In: *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2018, pp. 122–133.

[160] Ravi Teja Mullapudi et al. "Automatically scheduling halide image processing pipelines". In: *ACM Transactions on Graphics (TOG)* 35.4 (2016), pp. 1–11.

[161] *MXNet Profiler*. https://mxnet.apache.org/versions/1.4.1/tutorials/python/profiler.html.

[162] NVIDIA. *A PyTorch Extension: Tools for easy mixed precision and distributed training in Pytorch*. https://github.com/NVIDIA/apex. 2018.

[163] NVIDIA. *API Documentation of NVidia's Apex optimizers*. https://nvidia.github.io/apex/optimizers.html. 2018.

[164] NVIDIA. *CUDA implementation of the standard basic linear algebra subroutines (BLAS)*. http://docs.nvidia.com/cuda/cublas/index.html.

[165] NVIDIA. *CUDA Toolkit Documentation v10.0*. https://docs.nvidia.com/cuda/. 2018.

[166] NVIDIA. *CUDA Toolkit Documentation v8.0*. 2017. URL: https://docs.nvidia.com/cuda/archive/8.0/.

[167] NVIDIA. "cuDNN Library Developer Guide v 7.4.1". In: (2018).

[168] NVIDIA. *GEFORCE® RTX 2080 Ti*. https://www.nvidia.com/en-us/geforce/graphics-cards/rtx-2080-ti. 2018.

[169] NVIDIA. *NVIDIA Collective Communications Library (NCCL)*. https://developer.nvidia.com/nccl.

[170] NVIDIA. *NVIDIA Nsight*. https://developer.nvidia.com/tools-overview.

[171] NVIDIA. *NVIDIA Profiler*. docs.nvidia.com/cuda/profiler-users-guide/index.html.

[172] NVIDIA. *NVIDIA Turing GPU architecture*. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf. 2018.

[173] NVIDIA. *Performance reported by NCCL tests*. https://github.com/NVIDIA/nccl-tests/blob/master/doc/PERFORMANCE.md. 2018.

[174] NVIDIA. *The CUDA Profiling Tools Interface (CUPTI)*. https://docs.nvidia.com/cuda/cupti/index.html.

[175] NVIDIA. *Training With Mixed Precision: Deep Learning SDK Documentation*. https://docs.nvidia.com/deeplearning/sdk/mixed-precision-training/index.html. 2019.

[176] *NVIDIA cutlass*. https://github.com/NVIDIA/cutlass.

[177] *NVIDIA GeForce GTX 580*. https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-580.

[178] *ONNX*. https://onnx.ai/.

[179] Kay Ousterhout et al. "Making sense of performance in data analytics frameworks". In: *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*. 2015, pp. 293–307.

[180] Kay Ousterhout et al. "Monotasks: Architecting for performance clarity in data analytics frameworks". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM. 2017, pp. 184–200.

[181] Jian Ouyang et al. "Baidu Kunlun An AI processor for diversified workloads". In: *2020 IEEE Hot Chips 32 Symposium (HCS)*. IEEE Computer Society. 2020, pp. 1–18.

[182] Vassil Panayotov et al. “Librispeech: an ASR corpus based on public domain audio books”. In: *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2015, pp. 5206–5210.

[183] Kishore Papineni et al. “BLEU: a method for automatic evaluation of machine translation”. In: *Proceedings of the 40th annual meeting on association for computational linguistics*. Association for Computational Linguistics. 2002, pp. 311–318.

[184] Angshuman Parashar et al. “SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks”. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM. 2017, pp. 27–40.

[185] Jay H Park et al. “Hetpipe: Enabling large {DNN} training on (whimpy) heterogeneous {GPU} clusters through integration of pipelined model parallelism and data parallelism”. In: *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*. 2020, pp. 307–321.

[186] Jongse Park et al. “Scale-out acceleration for machine learning”. In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM. 2017, pp. 367–381.

[187] Adam Paszke et al. “Pytorch: An imperative style, high-performance deep learning library”. In: *Advances in neural information processing systems* 32 (2019), pp. 8026–8037.

[188] Suchita Pati et al. “Demystifying BERT: Implications for Accelerator Design”. In: *arXiv preprint arXiv:2104.08335* (2021).

[189] Suchita Pati et al. “SeqPoint: Identifying Representative Iterations of Sequence-Based Neural Networks”. In: *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2020, pp. 69–80.

[190] Kexin Pei et al. “Deepxplore: Automated whitebox testing of deep learning systems”. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM. 2017, pp. 1–18.

[191] Yanghua Peng et al. “A generic communication scheduler for distributed DNN training acceleration”. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. ACM. 2019, pp. 16–29.

[192] *Poplar Graph Framework Software*. https://www.graphcore.ai/products/poplar.

[193] Behnam Pourghassemi, Ardalan Amiri Sani, and Aparna Chandramowlishwaran. “What-If Analysis of Page Load Time in Web Browsers Using Causal Profiling”. In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 3.2 (2019), p. 27.

[194] Christoph von Praun, Rajesh Bordawekar, and Calin Cascaval. “Modeling optimistic concurrency using quantitative dependence analysis”. In: *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM. 2008, pp. 185–196.

[195] Andrew Putnam et al. “A reconfigurable fabric for accelerating large-scale datacenter services”. In: *ACM SIGARCH Computer Architecture News* 42.3 (2014), pp. 13–24.

[196] *PyTorch Documentation*. https://pytorch.org/docs/stable/index.html. 2019.

[197] Valentin Radu et al. “Performance aware convolutional neural network channel pruning for embedded GPUs”. In: *2019 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2019, pp. 24–34.

[198] Jonathan Ragan-Kelley et al. “Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines”. In: *Acm Sigplan Notices* 48.6 (2013), pp. 519–530.

[199] Pranav Rajpurkar et al. “Squad: 100,000+ questions for machine comprehension of text”. In: *arXiv preprint arXiv:1606.05250* (2016).

[200] Vijay Janapa Reddi et al. “Mlperf inference benchmark”. In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2020, pp. 446–459.

[201] Joseph Redmon and Ali Farhadi. “YOLO9000: better, faster, stronger”. In: *arXiv preprint arXiv:1612.08242* (2016).

[202] James Reinders. “VTune performance analyzer essentials”. In: *Intel Press* (2005).

[203] Ao Ren et al. “Sc-dcnn: Highly-scalable deep convolutional neural network using stochastic computing”. In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM. 2017, pp. 405–418.

[204] Shaoqing Ren et al. “Faster R-CNN: Towards real-time object detection with region proposal networks”. In: *Advances in neural information processing systems*. 2015, pp. 91–99.

[205] Zhixiang Ren et al. “AIPerf: Automated machine learning as an AI-HPC benchmark”. In: *Big Data Mining and Analytics* 4.3 (2021), pp. 208–220.

[206] Minsoo Rhu et al. “vDNN: Virtualized Deep Neural Networks for Scalable, Memory-efficient Neural Network Design”. In: *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-49. Taipei, Taiwan: IEEE Press, 2016, 18:1–18:13. URL: http://dl.acm.org/citation.cfm?id=3195638.3195660.

[207] Minsoo Rhu et al. “vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design”. In: *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE. 2016, pp. 1–13.

[208] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. “Learning representations by back-propagating errors”. In: *nature* 323.6088 (1986), p. 533.

[209] Olga Russakovsky et al. “ImageNet Large Scale Visual Recognition Challenge”. In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: 10.1007/s11263-015-0816-y.

[210] Mark Sandler et al. “Mobilenetv2: Inverted residuals and linear bottlenecks”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 4510–4520.

[211] Frank Schneider, Lukas Balles, and Philipp Hennig. “DeepOBS: A deep learning optimizer benchmark suite”. In: *arXiv preprint arXiv:1903.05499* (2019).

[212] Roy Schwartz et al. “Green AI”. In: *arXiv preprint arXiv:1907.10597* (2019).

[213] Frank Seide and Amit Agarwal. “CNTK: Microsoft’s open-source deep-learning toolkit”. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2016, pp. 2135–2135.

[214] Alexander Sergeev and Mike Del Balso. “Horovod: fast and easy distributed deep learning in TensorFlow”. In: *arXiv preprint arXiv:1802.05799* (2018).

[215] Amazon Web Services. *AWS Inferentia*. https://aws.amazon.com/machine-learning/inferentia.

[216] Ali Shafiee et al. “ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars”. In: *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press. 2016, pp. 14–26.

[217] Hardik Sharma et al. “From high-level deep neural models to FPGAs”. In: *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE. 2016, pp. 1–12.

[218] Yongming Shen, Michael Ferdman, and Peter Milder. “Maximizing CNN accelerator efficiency through resource partitioning”. In: *arXiv preprint arXiv:1607.00064* (2016).

[219] Shaohuai Shi, Xiaowen Chu, and Bo Li. “MG-WFBP: Efficient data communication for distributed synchronous SGD algorithms”. In: *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE. 2019, pp. 172–180.

[220] Shaohuai Shi et al. “Benchmarking state-of-the-art deep learning software tools”. In: *Cloud Computing and Big Data (CCBD), 2016 7th International Conference on*. IEEE. 2016, pp. 99–104.

[221] Karen Simonyan and Andrew Zisserman. “Very deep convolutional networks for large-scale image recognition”. In: *arXiv preprint arXiv:1409.1556* (2014).

[222] Linghao Song et al. “PipeLayer: A pipelined ReRAM-based accelerator for deep learning”. In: *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE. 2017, pp. 541–552.

[223] Sebastian U Stich, Jean-Baptiste Cordonnier, and Martin Jaggi. “Sparsified SGD with memory”. In: *arXiv preprint arXiv:1809.07599* (2018).

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

[225] Christian Szegedy et al. “Rethinking the inception architecture for computer vision”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 2818–2826.

[226] Mingxing Tan, Ruoming Pang, and Quoc V Le. “Efficientdet: Scalable and efficient object detection”. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2020, pp. 10781–10790.

[227] Tensorflow. “XLA Overview”. In: (2018). URL: https://www.tensorflow.org/performance/xla/.

[228] *TensorIR*. https://discuss.tvm.apache.org/t/rfc-tensorir-a-schedulable-ir-for-tvm/7872.

[229] Philippe Tillet, H. T. Kung, and David Cox. “Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations”. In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 10–19. ISBN: 9781450367196. URL: https://doi.org/10.1145/3315508.3329973.

[230] Seiya Tokui et al. “Chainer: a next-generation open source framework for deep learning”. In: *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*. Vol. 5. 2015, pp. 1–6.

[231] Peter Torelli and Mohit Bangale. “Measuring inference performance of machine-learning frameworks on edge-class devices with the mlmark benchmark”. In: *Techincal Report. Available online: https://www. eembc. org/techlit/articles/MLMARK-WHITEPAPERFINAL-1. pdf (accessed on 5 April 2021)* (2019).

[232] *Using Cloud TPU Tools*. https://cloud.google.com/tpu/docs/cloud-tpu-tools.

[233] Nicolas Vasilache et al. “Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions”. In: *CoRR* abs/1802.04730 (2018). arXiv: 1802.04730. URL: http://arxiv.org/abs/1802.04730.

[234] Nicolas Vasilache et al. “Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions”. In: *arXiv preprint arXiv:1802.04730* (2018).

[235] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems*. 2017, pp. 5998–6008.

[236] Swagath Venkataramani et al. “ScaleDeep: A Scalable Compute Architecture for Learning and Evaluating Deep Networks”. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM. 2017, pp. 13–26.

[237] Endong Wang et al. “Intel math kernel library”. In: *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 2014, pp. 167–188.

[238] Jianyu Wang and Gauri Joshi. “Adaptive communication strategies to achieve the best error-runtime trade-off in local-update SGD”. In: *arXiv preprint arXiv:1810.08313* (2018).

[239] Mengdi Wang et al. “Characterizing deep learning training workloads on alibaba-pai”. In: *2019 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2019, pp. 189–202.

[240] Minjie Wang et al. “Minerva: A scalable and highly efficient training platform for deep learning”. In: *NIPS Workshop, Distributed Machine Learning and Matrix Computations*. 2014.

[241] Naigang Wang et al. “Training deep neural networks with 8-bit floating point numbers”. In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. 2018, pp. 7686–7695.

[242] Shang Wang, Yifan Bai, and Gennady Pekhimenko. “BPPSA: Scaling back-propagation by parallel scan algorithm”. In: *Proceedings of Machine Learning and Systems* 2 (2020), pp. 451–469.

[243] Wei Wen et al. “Terngrad: Ternary gradients to reduce communication in distributed deep learning”. In: *Advances in neural information processing systems*. 2017, pp. 1509–1519.

[244] Samuel Williams, Andrew Waterman, and David Patterson. “Roofline: an insightful visual performance model for multicore architectures”. In: *Communications of the ACM* 52.4 (2009), pp. 65–76.

[245] Yanzhao Wu et al. "Experimental characterizations and analysis of deep learning frameworks". In: *2018 IEEE International Conference on Big Data (Big Data)*. IEEE. 2018, pp. 372–377.

[246] Yonghui Wu et al. "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation". In: *CoRR* abs/1609.08144 (2016). URL: http://arxiv.org/abs/1609.08144.

[247] Yonghui Wu et al. "Google's neural machine translation system: Bridging the gap between human and machine translation". In: *arXiv preprint arXiv:1609.08144* (2016).

[248] Abenezer Wudenhe and Hung-Wei Tseng. "TPUPoint: Automatic Characterization of Hardware-Accelerated Machine-Learning Behavior for Cloud Computing". In: *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2021, pp. 254–264.

[249] Wencong Xiao et al. "Tux2: Distributed Graph Computation for Machine Learning." In: *NSDI*. 2017, pp. 669–682.

[250] Zhujun Xiao. "Heterogeneity and Scalability in Machine Learning Based Sensing and Monitoring Systems". PhD thesis. The University of Chicago, 2021.

[251] Zhujun Xiao et al. "Towards Performance Clarity of Edge Video Analytics". In: *arXiv preprint arXiv:2105.08694* (2021).

[252] Wayne Xiong et al. "The Microsoft 2016 conversational speech recognition system". In: *Acoustics, Speech and Signal Processing (ICASSP), 2017 IEEE International Conference on*. IEEE. 2017, pp. 5255–5259.

[253] Jilong Xue et al. "Fast Distributed Deep Learning over RDMA". In: *Proceedings of the Fourteenth EuroSys Conference 2019*. ACM. 2019, p. 44.

[254] Xuan Yang et al. "Interstellar: Using halide's scheduling language to analyze dnn accelerators". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, pp. 369–383.

[255] Yang You et al. "ImageNet training in minutes". In: *CoRR, abs/1709.05011* (2017).

[256] Yang You et al. "Imagenet training in minutes". In: *Proceedings of the 47th International Conference on Parallel Processing*. ACM. 2018, p. 1.

[257] Dong Yu et al. "An introduction to computational networks and the computational network toolkit". In: *Microsoft Technical Report MSR-TR-2014–112* (2014).

[258] Geoffrey X Yu, Tovi Grossman, and Gennady Pekhimenko. "Skyline: Interactive In-Editor Computational Performance Profiling for Deep Neural Network Training". In: *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. 2020, pp. 126–139.

[259] Jiecao Yu et al. "Scalpel: Customizing dnn pruning to the underlying hardware parallelism". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM. 2017, pp. 548–560.

[260] Matei Zaharia et al. "Spark: Cluster computing with working sets." In: *HotCloud* 10.10-10 (2010), p. 95.

[261] Hao Zhang et al. "Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters". In: *2017 {USENIX} Annual Technical Conference ({USENIX} {ATC} 17)*. 2017, pp. 181–193.

[262] Shijin Zhang et al. "Cambricon-X: An accelerator for sparse neural networks". In: *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE. 2016, pp. 1–12.

[263] Jie Zhao et al. "AKG: Automatic Kernel Generation for Neural Processing Units Using Polyhedral Transformations". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021, pp. 1233–1248. ISBN: 9781450383912. DOI: 10.1145/3453483.3454106. URL: https://doi.org/10.1145/3453483.3454106.

[264] Bojian Zheng, Nandita Vijaykumar, and Gennady Pekhimenko. "Echo: Compiler-based GPU memory footprint reduction for LSTM RNN training". In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2020, pp. 1089–1102.

[265] Lianmin Zheng et al. "Ansor: Generating High-Performance Tensor Programs for Deep Learning". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 863–879. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/zheng.

[266] Size Zheng et al. "FlexTensor: An Automatic Schedule Exploration and Optimization Framework for Tensor Computation on Heterogeneous System". In: Mar. 2020, pp. 859–873. DOI: 10.1145/3373376.3378508.

[267] Hongyu Zhu et al. "Benchmarking and analyzing deep neural network training". In: *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2018, pp. 88–100.

[268] Barret Zoph et al. "Learning transferable architectures for scalable image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 8697–8710.