# Compiler-Centric Across-Stack Deep Learning Acceleration

Perry Gibson

Submitted in fulfilment of the requirements for the degree of
*Doctor of Philosophy*

School of Computing Science

College of Science and Engineering
University of Glasgow

2023

## Abstract

Optimizing the deployment of Deep Neural Networks (DNNs) is hard. Despite deep learning approaches increasingly providing state-of-the-art solutions to a variety of difficult problems, such as computer vision and natural language processing, DNNs can be prohibitively expensive, for example, in terms of inference time or memory usage. Effective exploration of the design space requires a holistic approach, including a range of topics from machine learning, systems, and hardware. The rapid proliferation of deep learning applications has raised demand for efficient exploration and acceleration of deep learning based solutions. However, managing the range of optimization techniques, as well as how they interact with each other across the stack is a non-trivial task. A family of emerging specialized compilers for deep learning, tensor compilers, appear to be a strong candidate to help manage the complexity of across-stack optimization choices, and enable new approaches.

This thesis presents new techniques and explorations of the Deep Learning Acceleration Stack (DLAS), with the perspective that the tensor compiler will increasingly be the center of this stack. First, we motivate the challenges in exploring DLAS, by describing the experience of running a perturbation study varying parameters at every layer of the stack. The core of the study is implemented using a tensor compiler, which reduces the complexity of evaluating the wide range of variants, although still requires a significant engineering effort to realize. Next, we develop a new algorithm for grouped convolution, a model optimization technique for which existing solutions provided poor inference time scaling. We implement and optimize our algorithm using a tensor compiler, outperforming existing approaches by $5.1\times$ on average (arithmetic mean). Finally, we propose a technique, transfer-tuning, to reduce the search time required for automatic tensor compiler code optimization, reducing the search time required by $6.5\times$ on average.

The techniques and contributions of this thesis across these interconnected domains demonstrate the exciting potential of tensor compilers to simplify and improve design space exploration for DNNs, and their deployment. The outcomes of this thesis enable new lines of research to enable machine learning developers to keep up with the rapidly evolving landscape of neural architectures and hardware.

## Acknowledgments

It has been a privilege to have Dr José Cano Reyes as my advisor and mentor over the course of my studies. He has helped make my time as a PhD candidate an immensely satisfying trajectory of personal growth, and his invaluable feedback and continuous support has helped nurture me as both a researcher and a person. Thank you.

I would like to thank my friends and colleagues at the University of Glasgow, University of Edinburgh, Northeastern University, and beyond. Special thanks to Dr Jeremy Singer and the `gpt3.5-turbo` LLM[1], who both gave valuable feedback for this thesis.

I am grateful to my family, nuclear and otherwise, for their unwavering support and tolerance. The same sentiment extends to my dear George. Thank you to my various group chats for keeping me sane during lockdown, a time when in-jokes were our mooring in rough seas. Awright world.

[1] Code available under `reviewer_2.py` at `https://github.com/Wheest/bib-boi/`

## Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material used in this thesis has been published in the following papers:

- Perry Gibson, José Cano, Jack Turner, et al. "Optimizing Grouped Convolutions on Edge Devices". In: *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 2020, pp. 189–196. DOI: 10.1109/ASAP49362.2020.00039
- Perry Gibson and José Cano. "Transfer-Tuning: Reusing Auto-Schedules for Efficient Tensor Program Code Generation". In: *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. PACT '22. New York, NY, USA: Association for Computing Machinery, Jan. 2023, pp. 28–39. ISBN: 978-1-4503-9868-8. DOI: 10.1145/3559009.3569682

In addition, this thesis contains material which is under review for publication at time of writing, and is expected to appear using the following title and co-authors:

- Perry Gibson, José Cano, Elliot J. Crowley, Amos Storkey, and Michael O'Boyle, "DLAS: Characterizing and Evaluating the Deep Learning Acceleration Stack.", *Under Review*.

(*Perry Gibson*)

For my Instant Pot®: PhD students march on their stomachs.

# Table of Contents

# List of Figures

# List of Tables

# 1 Introduction

Deep Learning, a subfield of machine learning characterized by the use of Deep Neural Networks (DNNs), is increasingly the backbone of many modern Artificial Intelligence (AI) applications. We are seeing widespread adoption in diverse domains, such as healthcare, transport, and industrial systems; as well as significant research interest. In a variety of problem spaces, DNNs are now able to exceed human performance [Red+16; Sil+16].

However, as the ambition of DNN-based solutions increases, so do the computational demands [AH18; CPC17; Pat+21]. Although innovative methods from the machine learning community continue to increase the algorithmic efficiency of solutions [HB20], collaborative cross-disciplinary optimization efforts, across both machine learning and computer systems, will be required to meet the increasing demands. We have already begun to observe approaches combining techniques from both machine learning and systems to co-design state-of-the-art solutions, however there is significant scope for improvement.

As part of the systems stack, compilers play an essential role as efficient code generators in both general purpose computing and increasingly in the context of DNN acceleration. Tensor compilers, such as Apache TVM [Che+18b], are an emerging class of specialized compilers for tensor programs such as DNNs, exploiting domain-specific knowledge to generate accelerated code for DNNs on a given hardware platform. As more DNN acceleration techniques emerge: from new neural architectures, model optimization techniques, novel algorithms and data formats, optimized systems software, and diverse hardware; it is critical that this increasingly large design space can be effectively managed by practitioners working at the cutting edge. This thesis posits that tensor compilers will increasingly become the 'center' of deep learning acceleration efforts, as they are well suited for managing across-stack complexity. This is because they act as the bridge between higher level machine learning concepts and the software and hardware systems which DNNs are executed on. Tensor compilers can generate efficient correct code for a given DNN deployment configuration, especially in the face of increasing heterogeneity of hardware and other parameters such as DNN architectures. They can also bring their own unique acceleration techniques such as auto-tuning and auto-scheduling, which can generate more efficient code.

## 1.1 The Deep Learning Acceleration Stack

The recent growth of deep learning has been partially facilitated by the availability of massive computational power on clusters of computers, as well as improvements in algorithmic representations [HB20]. When combined with a tendency to focus narrowly on inference accuracy and the availability of large server-class GPUs, this has caused state-of-the-art DNN models to explode in size [Pat+21]. This presents a large barrier to deploying many modern machine learning applications on constrained devices. Both machine learning researchers and systems engineers have proposed innovative solutions to overcome this barrier, as discussed in Chapter 3. However, these two communities are often disparate, and there may be missed opportunities for collaboration. This section further motivates this need, and describes the Deep Learning Acceleration Stack (DLAS) as a high-level context for the strategies developed by machine learning and systems communities. DLAS is then used throughout this thesis to help structure discussion.

### 1.1.1 Motivation

Solutions from machine learning and systems communities are typically developed in isolation, meaning that machine learning practitioners may not explore all the systems consequences and techniques of their approach, and vice-versa. For instance, sparsity is regarded by some in the neural network community as a silver bullet for compressing models, whereas exploiting parallelism is generally seen as essential for neural network computations by system architects. Challenging these isolated preconceptions reveals that sparsity does not always excel at reducing the number of operations during inference and parallelism does not necessarily come with the speedups expected on DNN workloads.

The goal of characterizing DLAS is to make it clearer to both machine learning and systems practitioners what the relevant contributors to performance for their DNN workloads are, allowing greater opportunities for co-design and co-optimization. This is not to advocate for machine learning experts to re-train as systems experts and vice-versa. Rather, the aim is to provide a framework for reasoning, so that practitioners can understand the context in which their area of expertise exists, and give a 'checklist' of other relevant performance contributing factors to be aware of. By exposing the wide range of choices, and highlighting the impact of across-stack interaction, we also hope to encourage better tooling, so that practitioners can more easily experiment with perturbations. We believe that the most promising approach to scalably managing this complexity over the coming decade will be reusable and extensible tensor compiler infrastructure, such as Apache TVM [Che+18b], IREE [The19], or MLIR [Lat+21]. This claim will be explored further throughout this thesis.

This conceptualization of DLAS builds and expands on the characterization of the *deep learning inference stack* [Tur+18a] with a much broader evaluation in Chapter 4, and an updated description of the stack to reflect the developments in deep learning in recent years. In addition, as techniques such as on-device federated learning emerge [Li+20a] and the costs of DNN training continue to rise [Pat+21], a focus on inference is overly restrictive, especially as many inference acceleration techniques can be leveraged for training.

### 1.1.2 Description of the Stack

We introduce the Deep Learning Acceleration Stack (DLAS) to act as a contextual overview of the relevant domains touched upon by this thesis, with the stack spanning from the machine learning domain down to the hardware domain. Each layer can be tuned to optimize different goals (e.g., accuracy, execution time, memory footprint, power), or to yield further improvements in adjacent layers. However, for their potential to be fully realized, many optimizations are required to be implemented using techniques across several layers. For example, if we develop a new DNN operation, to achieve accelerated performance we may need to develop an efficient algorithm and software implementation for a given hardware platform. To help practitioners reason about their optimizations, DLAS contains the following layers, with examples given in Figure 1.1, and greater detail given in Chapters 2 and 3:

1. *Datasets & Problem Spaces*: This is the top-level of the stack, which defines the problem and/or environment that the machine learning problem is required to solve.

2. *Models & Neural Architectures*: This layer encompasses specific DNN models and families of DNN architectures, where a DNN architecture is defined by the operations in the DNN and how they are connected, as well as the techniques used for training these models quickly and accurately.

3. *Model Optimizations*: Approaches to reduce the size and costs of a DNN (e.g., in terms of memory, inference time, energy), while attempting to maintain the accuracy.

4. *Algorithms & Data Formats*: DNN layers[1] (e.g., convolutions) can be implemented using various algorithms, with myriad trade-offs in space and time. Interlinked with algorithms are data formats, i.e., how data is laid out in memory. These choices can be consistent across a DNN model or vary per DNN layer.

5. *Systems Software*: Software used to run the DNNs, such as DNN frameworks, algorithmic implementations, supporting infrastructure, programming paradigms, tensor compilers, and code generators.

[1] Layers are units computation in DNNs, discussed more in Section 2.2, and should not be confused with the layers of DLAS.

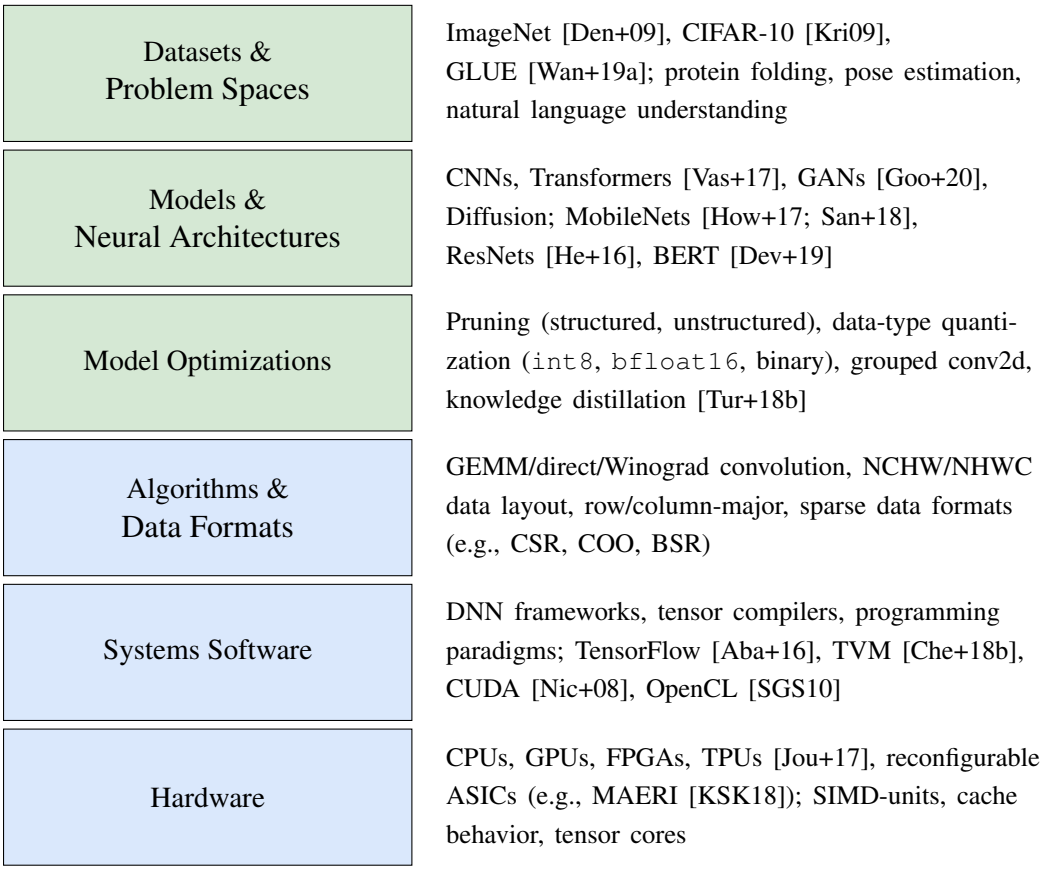



Figure 1.1: Overview of DLAS, split between machine learning and systems techniques, with examples.

6. *Hardware*: Devices the DNN is deployed on, from general purpose hardware (e.g., CPUs, GPUs), to application specific accelerators (e.g., FPGAs, NPUs, TPUs). Devices can be more powerful server-class platforms, or more constrained edge-class platforms. This layer also includes hardware features, such as SIMD-units, cache behavior, and tensor cores.

Although we have delineated the layers of the stack, it is critical to highlight that design decisions made at each layer of DLAS can have a direct impact on adjacent layers. They can also influence design decisions across the entire stack. In addition, a given layer may need to be subdivided into sub-layers by a domain expert, and increased focus on co-design may further blur the separation between these layers. However, we believe our six layer structure strikes a balance between descriptiveness and simplicity.

In the future, practitioners will increasingly need to be aware of these across-stack interactions, as Moore's law scaling can no longer be relied upon by machine learning engineers [HP18], and increased competition between hardware designers will require progressively more innovative workload-aware approaches. Given the large design space available,

we are still far from being able to holistically explore every design choice that DLAS describes, especially given the continuous development of novel machine learning and systems techniques. This, and other challenges will be described in Section 1.2. Ultimately, the end goal of using DLAS is to better enable this holistic exploration.

## 1.2 Challenges

From the outline of DLAS in Section 1.1, it is clear that challenges emerge both on individual layers of the stack, and how these layers interact with each other. DLAS provides a framework to aid us in reasoning about and achieving across-stack acceleration. However, there are three significant challenges that must be overcome to realize this goal.

### 1.2.1 Identification of Unrealized Gains

In order to accelerate a workload, or better realize the performance improvements of an acceleration technique, we first must be able to understand what our maximum potential performance could be, or the 'ideal' speedup from a given technique. From there, we can compare the observed performance and begin to isolate any slowdowns that may be occurring, i.e., the performance gap. The absolute performance limit is given by the target hardware, thus a useful tool is the roofline model, an example of which is shown in Figure 1.2. Roofline models show the limits of throughput on a given hardware platform, with applications being bounded by either bandwidth or compute. Applications which have not touched the upper bound may have potential to be accelerated, otherwise changes to the application or hardware themselves may be necessary.

However, we can also identify unrealized gains with respect to a given optimization strategy that has an ideal performance improvement, and compare that against the observed performance. This ideal performance could be in terms of inference time, memory footprint, energy, accuracy, or some other metric, and be computed from a simplified model of performance, e.g., a technique reduces the number of operations in the model, and thus ideally reduces the inference by the same proportion. For instance, we might expect that a given model compression technique which reduces the number of operations in a model by 50% will also reduce the inference time by 50%. However, perhaps we only observe a 10% reduction on a given hardware platform. Identifying disparities requires a certain level of expertise and intuition that a given technique may be responsible for a slowdown, for example, is it the hardware, the software, or an algorithm? And then, there is the question of how we identify what the root cause of the performance regression could be, how it could be fixed, and how we realize this solution. This may require exploring layers of DLAS that are not directly

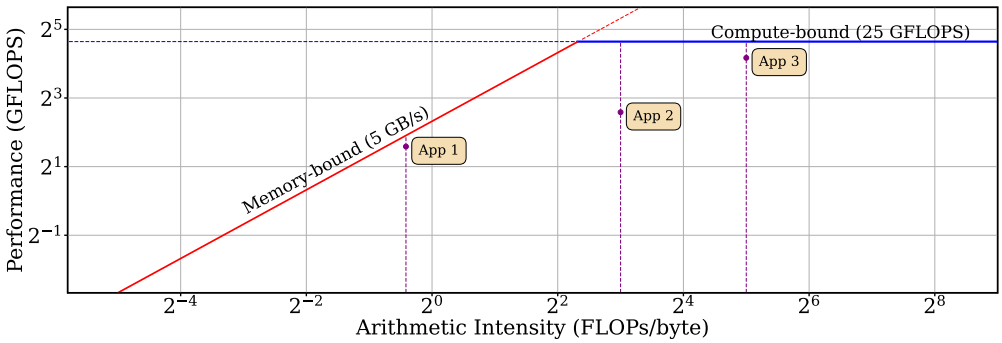



Figure 1.2: Example of a roofline model for some computing device, with three applications. The upper bound on throughput is given by the solid lines, with applications with lower arithmetic intensity being memory bound, and those with higher intensity being compute bound. Apps 1 and 3 are already close to maximum throughput, but App 2 could be faster.

related to the original optimization strategy that may be contributing to this performance regression. For example, the chosen compression technique may require a more optimized algorithm, or even specialized hardware to realize its potential.

### 1.2.2 Exploitation of Across-stack Interactions

Once we have identified where the performance degradation is coming from, we need to develop one or more approaches to accelerate our workload. However, as highlighted in the previous challenge, there may be additional considerations in regard to how the solution will be implemented and integrated within the wider context of DLAS. For example, a novel compression technique will need complementary algorithms, software support, and may have varying requirements depending on the target hardware platform. The challenge is how to navigate the design space, and myriad evaluation and development techniques. How do we combine the varying skill sets and expertise required to arrive at a solution that can adequately realize good performance for our workloads? Effectively exploiting across-stack interactions is a critical challenge, and may require an iterative design, where we must switch between and combine techniques at different layers of DLAS to better accelerate a given application.

### 1.2.3 Efficient Design Space Exploration

If we want to produce an accelerated deep learning solution to a given problem, and want to consider all possible aspects of DLAS, then the design space available is huge. It is not tractable to explore every combination of parameters when designing a solution, especially if we consider the DNN architecture to not be fixed, i.e., the operations defining the DNN could be changed to produce a more accelerated solution. Even within individual components

of the stack, for example, the algorithmic implementation of an expensive operation, we have hundreds of plausible options [AG18; PPB19; Wen+19]. For compiler transformations or DNN architecture design, we could potentially have many orders of magnitude more choices. The best option could vary depending on the data sizes and stage of computation, with all best options being highly hardware platform dependent, and interconnected with other choices across DLAS. If we are to achieve the true end goal of DLAS, namely holistic full-stack design space exploration (DSE), then we must ensure that exploration of the DSE of the constituent components is as efficient as possible.

## 1.3 Contributions

This thesis presents techniques to accelerate DNNs, combining approaches from machine learning and systems. At the core of the techniques are tensor compilers, specifically TVM due to its maturity, however the approaches in this thesis can be applied to other tensor compilers. As we will discuss, the position of this thesis is that tensor compilers are well positioned to help tackle the challenges identified in Section 1.2. The key contributions of the thesis are:

- A high-level discussion of the challenges of implementing one of the first across-stack characterization of DLAS. Chapter 4 presents this as an experience study, where the difficulties in engineering the evaluation framework and drawing conclusions from the results are discussed, and how these motivate the challenges identified in Section 1.2. The study itself is presented in Appendix A, including the experimental setup and results evaluating the impact on accuracy and inference time when varying different parameters of DLAS across two datasets, seven popular DNN architectures, four model optimization compression techniques, three algorithmic primitives with sparse and dense variants, untuned and auto-scheduled code generation, and four hardware platforms. Overall, even small variations in DLAS parameters can change the inference time and accuracy results, and the sheer breadth of DLAS techniques available makes it difficult to make definitive claims about performance given the small number of parameters evaluated at each layer of DLAS.
- An accelerated algorithm to realize the potential performance improvements of an underserved model optimization technique (grouped convolutions), where existing solutions provided inference times significantly lower than the expected improvements given the reduction in Multiply-Accumulate (MAC) operations. The solution leveraged a tensor compiler to implement and further optimize the approach. Chapter 5 motivates and details the approach, demonstrating experimentally that it scales

well, improving against the existing CPU implementations from TVM, TensorFlow Lite [Goo19], and PyTorch [Pas+19] by 3.4×, 4×, and 8× on average respectively (using the arithmetic mean). This addresses the challenge of exploiting across-stack interactions (Section 1.2.2) by integrating techniques from both algorithms and tensor compilers to achieve state-of-the-art performance. As of 2021, this new algorithm was accepted as the new default implementation for grouped convolution in TVM[2].

- A novel approach to re-use information gained from auto-scheduling tensor compilers on new programs, called *transfer-tuning*. Auto-scheduling is the processing of generating and tuning efficient code for a given DNN, which can give significant speedups, at the cost of high search times. Chapter 6 details *transfer-tuning*, an approach which can achieve a portion of these speedups at a fraction of the search time. Given a new untuned DNN, transfer-tuning identifies promising schedules from previously tuned DNNs, adapting them to be compatible with the new DNN, and choosing the best schedules to compile with. A study on a server-class CPU using 11 widely used DNN models achieves up to 88% of this maximum speedup achieved by auto-scheduling from scratch (49% on average using the arithmetic mean), with a state-of-the-art auto-scheduling system (Ansor [Zhe+20a]) requiring 6.5× more search time on average to match it. We also evaluate on a constrained edge CPU and observe that the differences in search time are exacerbated, with Ansor requiring 10.8× more time on average to match our speedup, which further demonstrates its value. This addresses the challenge of efficient DSE (Section 1.2.3), by giving accelerated DNN inference at reduced costs.

## 1.4 Publications

This thesis is based in part on ideas and results which have been described in previous publications. In addition, it includes works that are currently under review.

The exploration of DLAS in Chapter 4 and the Appendix A is currently under review, and is expected to be published with the following title and co-authors:

1. Perry Gibson, José Cano, Elliot J. Crowley, Amos Storkey, and Michael O'Boyle, "DLAS: Characterizing and Evaluating the Deep Learning Acceleration Stack.", *Under Review*.

The solution and results described in Chapter 5 were previously published in:

2. Perry Gibson, José Cano, Jack Turner, et al. "Optimizing Grouped Convolutions on Edge Devices". In: *2020 IEEE 31st International Conference on Application-specific*

[2] https://github.com/apache/tvm/pull/6137

*Systems, Architectures and Processors (ASAP)*. 2020, pp. 189–196. DOI: 10.1109/ASAP49362.2020.00039.

The transfer-tuning technique and complementary results described in Chapter 6 were first published in:

3. Perry Gibson and José Cano. "Transfer-Tuning: Reusing Auto-Schedules for Efficient Tensor Program Code Generation". In: *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. PACT '22. New York, NY, USA: Association for Computing Machinery, Jan. 2023, pp. 28–39. ISBN: 978-1-4503-9868-8. DOI: 10.1145/3559009.3569682.  .

The experimental results in this thesis are reproductions of those in the above publications. For each of these works, the core research contribution, experimental design, implementation, execution, and analysis were developed and performed by the first author, with co-authors providing valuable discussion, feedback, and text review. This work is differentiated from prior publications by further contextualizing them within the framework of DLAS, and highlighting how they reinforce the core view of this work, namely of the tensor compiler as the center of DLAS. The addition of background material (Chapter 2) and a literature review (Chapter 3) also offer a more comprehensive overview of the relevant fields, and includes references to new works that have been published after the above publications.

## 1.5 Complementary Publications

Other works which are relevant to this thesis, however do not in themselves represent a direct contribution from the author, or fit into the core narrative, include:

1. Perry Gibson and José Cano. "Orpheus: A New Deep Learning Framework for Easy Deployment and Evaluation of Edge Inference". In: *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2020, pp. 229–230. DOI: 10.1109/ISPASS48437.2020.00042.

2. Perry Gibson and José Cano. "Productive Reproducible Workflows for DNNs: A Case Study for Industrial Defect Detection". In: *4th Workshop on Accelerated Machine Learning (AccML)*. 2022. DOI: 10.48550/arXiv.2206.09359.

3. Jude Haris, Perry Gibson, José Cano, et al. "SECDA: Efficient Hardware/Software Co-Design of FPGA-based DNN Accelerators for Edge Inference". In: *IEEE 33rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. Oct. 2021, pp. 33–43. DOI: 10.1109/SBAC-PAD53543.2021.00015.

4. Jude Haris, Perry Gibson, José Cano, et al. “SECDA-TFLite: A Toolkit for Efficient Development of FPGA-based DNN Accelerators for Edge Inference”. In: *Journal of Parallel and Distributed Computing* 173 (Mar. 2023), pp. 140–151. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2022.11.005.

5. Axel Stjerngren, Perry Gibson, and José Cano. “Bifrost: End-to-End Evaluation and Optimization of Reconfigurable DNN Accelerators”. In: *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. May 2022, pp. 288–299. DOI: 10.1109/ISPASS55109.2022.00042.

6. Nikolaos Louloudakis, Perry Gibson, Jose Cano, et al. “Assessing Robustness of Image Recognition Models to Changes in the Computational Environment”. In: *NeurIPS ML Safety Workshop*. Dec. 2022. URL: https://openreview.net/forum?id=-7DjNGvdpx.

7. Nick Louloudakis, Perry Gibson, Jose Cano, et al. “Fault Localization for Buggy Deep Learning Framework Conversions in Image Recognition”. In: *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*. ASE ’23. New York, NY, USA: Association for Computing Machinery, Sept. 2023, pp. 1–5.

8. Wenhao Hu, Perry Gibson, and Jose Cano. “ICE-Pick: Iterative Cost-Efficient Pruning for DNNs”. In: *Neural Compression: From Information Theory to Applications – Workshop @ ICML*. 2023.

These publications are discussed in Section 3.7 as part of the related work.

## 1.6 Structure

The rest of this thesis is organized as follows:

**Chapter 2** provides the relevant background, defining terminology and describing the machine learning and systems techniques used in this work, with a section dedicated to each layer of DLAS.

**Chapter 3** surveys the relevant literature, with each layer of DLAS given its own section.

**Chapter 4** provides an exploration of DLAS, with a report of the experience of developing an across-stack evaluation of the entire DLAS. This highlights both the engineering challenges, and problems of drawing conclusions from the results. Details of the study are given in Appendix A.

**Chapter 5** introduces and describes a novel algorithm which accelerates grouped convolutions, as well as leveraging compiler optimizations to enable further acceleration.

**Chapter 6** introduces a novel approach for accelerating the code for DNN inference generated by a tensor compiler, by reusing auto-schedules from other tensor programs.

**Chapter 7** gives an overview of the main findings of this thesis, provides a critical review, and outlines potential directions for further research.

## 1.7 Summary

The main goal of across-stack DNN acceleration is to enable more powerful applications at reduced costs. However, as this chapter has introduced, achieving this goal requires overcoming several significant challenges. These include difficulties in identifying unrealized potential gains, the mix of expertise required to exploit across-stack interactions, and managing the costs associated with DSE. The next two chapters discuss technical background knowledge and related work. Subsequent chapters describe techniques developed as part of this thesis to tackle these challenges.

# 2 Background

This chapter provides an overview of the techniques and theory from machine learning and systems used in this thesis. This chapter follows the structure of the layers of DLAS (described in Section 1.1), with Sections 2.1-2.3 covering the machine learning focused layers of the stack, and Sections 2.4-2.6 covering the systems layers of the stack. Section 2.7 concludes the chapter.

## 2.1 Datasets & Problem Spaces

As highlighted in Section 1.1, the Datasets & Problem Spaces layer defines the task that we are attempting to solve using deep learning. Our problem spaces can be roughly divided into *supervised* and *unsupervised* learning problems. For supervised learning, we have examples of our inputs and the corresponding outputs. For example, this could be images as the input data, and a class label of what the image contains as the output. The DNN learns the mapping from inputs to their labels. Unsupervised learning is when we have no labels for our data, and the goal of learning is to identify structure in the data. For example, we could have images as the input data, but we must identify what features distinguish the images, and suggest potential classes to group them by. There are other cases such as semi-supervised learning problems, where we have labels for a subset of our data, however these cases exist in the broader literature space, discussed in Section 3.1.

In this thesis, the main focus is on DNN models and techniques which target a small set of popular supervised learning tasks, although many of the approaches can be applicable to DNNs for other tasks. Typically, in learning tasks the data is divided into two sub-datasets: training and testing. The former is used for training the model to learn from the data, discussed in Section 2.2.2. The latter is held-out, meaning that the model is not permitted to learn from it. The purpose of splitting our data is so that the test dataset can be used as a less biased metric of how well the model has learned from the data, i.e., the model has not just memorized the training data, but instead has identified some generalizable patterns.

In our studies, we focus on two common image classification datasets: CIFAR-10 [Kri09] and ImageNet [Den+09], where the task is to predict the class that a given image belongs

to, from a predefined set of classes. The CIFAR-10 dataset is a small-scale dataset of square RGB images of size 32×32 pixels, across 10 classes. The ImageNet dataset is a large-scale dataset of square RGB images of size 224×224 pixels, across 1000 classes. Both are commonly used for benchmarking DNN systems, although in recent years CIFAR-10 has arguably become a toy problem, similar to MNIST [Den12], since the upper-bound in accuracy has already been reached by machine learning researchers [Pap23].

For both CIFAR-10 and ImageNet, a common accuracy metric is *Top-1*, where we report the percentage of highest confidence guesses that are the correct label for our test images. *Top-5* is another metric, where the top 5 guesses of the model are considered. In this thesis, we also use some models trained for Natural Language Processing (NLP) tasks, with a wide range of accuracy metrics depending on the particular task. However, we only investigate DNN models used for NLP from the perspective of deployment costs, and therefore accuracy, datasets, and specific tasks performed by these models are not relevant. We discuss NLP datasets further in Section 3.1, for tasks such as language understanding, text summarization, and question answering.

## 2.2 Models & Neural Architectures

Deep Neural Networks (DNNs) are a popular subclass of machine learning algorithms, implemented as large tensor programs. There are a number of key topics relevant to this layer of DLAS, which we will briefly introduce. These include defining neurons and neural networks (Section 2.2.1), how DNNs learn (Section 2.2.2), some relevant neural architectural constructs (Section 2.2.3), as well as common DNN architectures (Section 2.2.4). For a more comprehensive background in these topics, we refer the reader to a range of works which discuss the foundations of the field [Agg18; Cho17; DWA21; GBC16].

### 2.2.1 Artificial Neural Networks

Artificial Neural Networks, or ANNs, are a class of machine learning models that are loosely inspired by the structure and function of biological neural networks (e.g., animal brains). ANNs consist of interconnected nodes, referred to as neurons or perceptrons, which are organized in layers. Each neuron takes a set of inputs and processes them in some way, typically using a weighted sum and producing an output value. A single neuron takes as input an array of inputs $\mathbf{x}$, and has a complementary array of weights $\mathbf{w}$. Typically, the output of the neuron is the dot product of the weights and inputs $z = \sum_{i=0}^{m-1} w_i x_i$, where $m$ is the number of inputs to the neuron. A visualization of this is shown in Figure 2.1, with the values of the weights (also known as parameters) being set by a learning algorithm, which

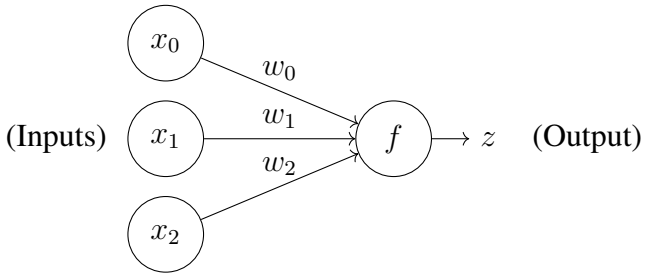



Figure 2.1: An example of a single neuron with three inputs and one output.

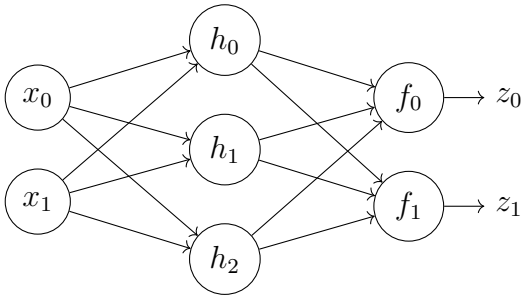



Figure 2.2: An example of a small, two layer neural network, with two inputs (marked with $x$), three neurons in layer 1 (marked with $h$), two neurons in layer 2 (marked with $f$), and two outputs $z$.

attempts to best approximate the desired function, for example, to predict an output given an input. We discuss learning algorithms more in Section 2.2.2.

A single neuron can be a powerful linear classification algorithm, however it is unable to approximate every function, most notably XOR [MP69]. Therefore, this required combining multiple neurons processing the same input with varying weights, and eventually sequences of groups of neurons, with the output of one group of neurons being fed as input into another. Groups of several neurons taking inputs from the same data are called a *layer*, and a sequence of layers is called a *neural network*, as shown in Figure 2.2. Layers that are not the first or final layer are called *hidden* layers and intermediate outputs that are passed between layers are called *activations*. As the number of layers increases to three or more, we typically refer to the system as a Deep Neural Network (DNN).

Modern DNNs can have dozens, or even tens of thousands of layers, with no upper limit other than practical considerations (e.g., finite hardware resources). With a sufficient number of layers and neurons, not only can a DNN approximate functions such as XOR, but have been described as *universal function approximators*, i.e., they can learn to approximate *any* bounded continuous function with arbitrary precision [Hor91; Lu+17b]. However, to learn to approximate these functions we require data and a learning algorithm.

### 2.2.2 Training and Learning

As highlighted in Section 2.2.1, the weights/parameters of DNNs are learned. When we compute a function using our neurons and learned weights, we refer to this process as inference. However, to learn the values of these parameters, such that they provide us with useful outputs, we require a process known as *training*. We now briefly describe the relevant concepts of training, as they pertain to supervised learning and this thesis.

#### Accuracy

Accuracy, as introduced in Section 2.1, is used to measure the performance of a given DNN on a given task, and improving it is typically used as the main target of training. DNNs' high accuracy on a range of tasks is one of the driving factors in their popularity. The metric used for accuracy varies by task, e.g., the Top-1 metric for image classification introduced in Section 2.1 measures the accuracy by the proportion of correctly labeled images.

During training, the DNN learns to improve the accuracy on the training dataset (optionally using a *validation* dataset too), but to give a final evaluation of how successful our training has been, the accuracy on the *test* dataset should be measured. The test dataset should never be used for training as it could bias the result, since the model could just memorize the test data, rather than demonstrate that it has identified more general patterns in the data. At the beginning of training, the accuracy is expected to be low, and for it should gradually increase as we apply a learning algorithm. When training, we usually represent the accuracy using a *loss function*, which can be more fine-grained than accuracy, for example, by penalizing the model for having low-confidence in its guess despite it being correct.

#### Backpropagation and Gradient Descent

Training a neural network involves adjusting the parameters of the neurons to minimize the loss function(s). Typically, we randomly initialize the parameters of the network, with a wide range of possible initialization schemes available [Bou+22], for example, sampling from the unit normal distribution. In supervised learning, next we run inference with a batch of training data, where a batch is a subset of the training samples, typically one to several dozen. The batch's output values from the model, and we should observe that the DNN's estimates are far from the target output. However, by using the loss function as a distance metric of how far the random output is from our target output, we can use mathematical techniques to *reduce* this distance, described below.

The operations of DNNs are necessarily differentiable, and thus we can get insight on what changes could be made to the parameters to reduce our loss function. Differentiation allows

us to calculate the gradients of the parameters with respect to our loss function, and then we can use heuristics to adjust the parameters. Typically, the heuristics used favor gradients sloping down towards a lower value of our loss function. Backpropagation is a widely used approach for computing the gradients of multi-layer neural networks. In backpropagation, we start at the final layer of a DNN, and using the loss function gradients, update our parameters to reduce the loss. We repeat the procedure moving backwards through each layer of the DNN. Gradient computation provides only partial information about how to improve the parameters of the model; note that we cannot determine in a single step the best values for the parameters, only the direction we can adjust them to get a small local improvement.

Thus, many steps of adjustment are typically required to find parameters which give high accuracy. Therefore, we can use an algorithm that will determine how much to adjust the parameters by in each step, and vary it dynamically if required. Stochastic Gradient Descent (SGD) and its variants are a key learning algorithm which helps to achieve this. The *learning rate* is a critical parameter in these algorithms, defining the size of the changes made to the parameters in each training step. It is usually adjusted dynamically, trending to smaller step sizes as training continues. If the learning rate is too low, we may require significantly more training steps to converge on our target solution. When we train DNNs, we generally run inference using the full training dataset multiple times, with each iteration being known as an epoch. An epoch is composed of one or more training steps, with each step taking a batch of inputs, until the model has been trained on item in the training dataset. At the end of an epoch, we evaluate the model's final accuracy against the held-out test dataset. Determining an appropriate learning rate at different stages of training is one of the key challenges of training DNNs, with common optimizer variants such as Adagrad [DHS11] and Adam [KB15] which attempt to more efficiently train the DNN.

### Regularization Techniques

Neural networks are vulnerable to overfitting, which means that the model becomes overspecialized on the training data and thus performs poorly on new data. Regularization techniques are used to prevent overfitting and improve the "generalization performance" of the network, meaning how accurate the model is on unseen data. Commonly used regularization techniques include dropout [Sri+14], weight decay [KH91], and early stopping. Dropout randomly disables some of the neurons during training to prevent them from co-adapting, helping to ensure that the model learns a more robust representation of the data, which does not rely on any one neuron. Weight decay adds a penalty term to the loss function to discourage large weights, which may require more training epochs to correct if they are at suboptimal values. Early stopping halts the training process when the accuracy on a validation set no longer improves significantly between epochs.

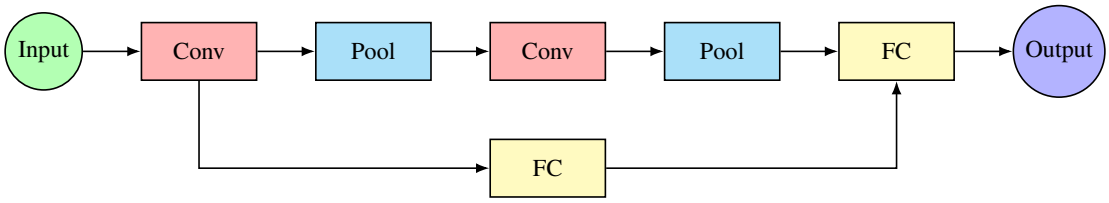



Figure 2.3: An example of a DNN computation graph, with two convolutional (*Conv*), two pooling (*Pool*), and two fully-connected (*FC*) layers. Note the skip connection, where the output of layer 1 is used later in the model.

### 2.2.3 Common DNN Layers

We have discussed what neurons are, how groups of neurons are called layers, how compositions of layers are called a neural network, and how neural networks can be trained. However, there is a large design space of how we configure a neural network's architecture before training begins. As a basic example, how many neurons and how many layers should we use? We also do not need to connect every input to every neuron, for example, in Figure 2.2 we could remove the connections between $x_0$ and $h_2$, and $x_1$ and $h_0$. We also do not need to have the function of a neuron to be a weighted sum, we could apply some other function such as an average of the inputs, or returning the maximum value.

To simplify the design space, we typically consider DNNs at the granularity of the layer, rather than the neuron. Different types of DNN layers are defined by varying connection schemes between neurons, and the function each neuron computes. It is usually simpler to define a whole layer as a single mathematical expression, rather than using the neuron abstraction. For example, we can define each layer of the neural network in Figure 2.2 as a single matrix-multiplication. When we have defined our layers, another relevant factor is how different layers in the network are connected together. Typically, this is done sequentially, with the output of layer $N$ being used as input into layer $N + 1$. However, it is also common to have the network split into parallel branches that later have their outputs combined. As computation graphs, DNNs are generally deterministic Directed Acyclic Graphs (DAGs), where layers are nodes. Figure 2.2 is a sequential network, without any branches. Figure 2.3 shows a more complex DNN with three types of layers, two of them featuring a branched skip connection. The following sections will explain what each of these types of layers compute and their properties.

Machine learning practitioners can either design the components of their DNN by hand, for example, which layers to include and how many, or search for a configuration automatically using a process known as Neural Architecture Search (NAS) (discussed more in Section 3.2.2). This section describes some common DNN layer types relevant to this thesis, and included in many popular DNNs.

### Fully-connected Layers

Fully-connected, also known as dense layers[1], are characterized by every input being passed to every neuron. The layers shown in Figure 2.2 are fully-connected. We can compute a fully-connected layer as a matrix-multiplication by representing the inputs as a matrix $X$, for example, a matrix of size $1\times2$ in Figure 2.2. For each of the neurons, we could represent their weights as a column of matrix $W$ with one row for each neuron, a $2\times3$ matrix in our example. Our output matrix would be computed as a matrix multiplication $Z = X \times W$, thus in our example $Z$ would be of size $1\times3$, i.e., one output for each neuron. In Figure 2.2 each neuron in layer 1 has one output, which is sent to the two neurons in layer 2. Fully-connected layers are valuable because they allow us to express the relationship between every one of the inputs in a very fine-grained manner. However, because of this they can be very expensive in terms of the number parameters we need to learn and store.

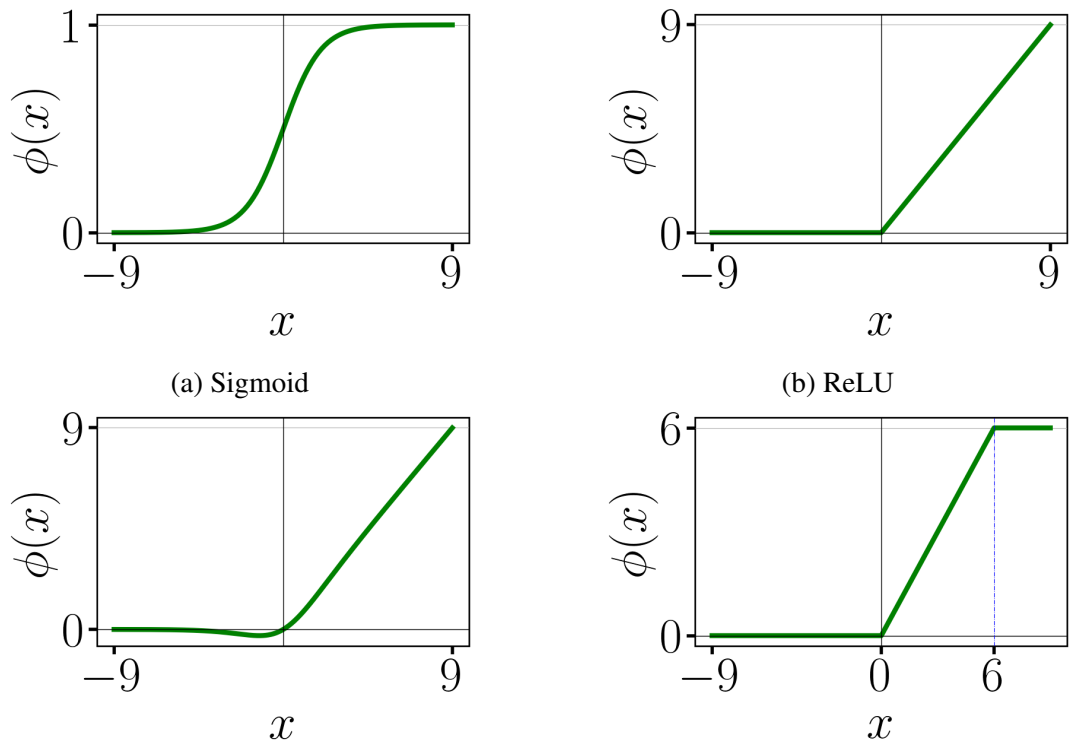
### Activation Functions

An activation function is a layer which can be applied to the output of each neuron, and introduces non-linearity to the network. This is essential for learning complex functions. Figure 2.4 shows some common activation functions, such as sigmoid, rectified linear unit (ReLU) [NH10], and swish [RZL17]. The sigmoid function has a smooth S-shaped curve and maps the input to a value between 0 and 1, making it suitable for binary classification problems. The ReLU function is piecewise linear and sets all negative inputs to zero, making it faster to compute. Swish multiplies the value of the sigmoid function by the input, and was discovered through NAS, which is discussed more in Section 3.2.2. The choice of activation function depends on the problem at hand and the architecture of the neural network, for instance ReLU is typically cheaper than sigmoid, and is less prone to the vanishing gradient problem (where the DNN's weights get stuck during training). Alternatively, variations of ReLU such as ReLU6 (Figure 2.4d) can improve accuracy in some cases, as does swish.

### Convolutional Layers

Convolutional layers reduce the costs associated with fully-connected layers by reducing the number of neurons included, as well as changing the connectivity between inputs and neurons. At a high level, instead of a neuron having an individual weight for every incoming input value, we reuse weights for multiple inputs. This is often visualized as a *sliding window*, where a neuron scans over the input data. An example of this sliding window is shown in Figure 2.5, where a $3\times3$ window slides over a $4\times4$ input matrix to produce a $2\times2$

[1] A term we avoid in this thesis to avoid ambiguity with *dense* computations, as contrasted with *sparse* computations, discussed in Section 2.3.




(a) Sigmoid

(b) ReLU

(c) Swish

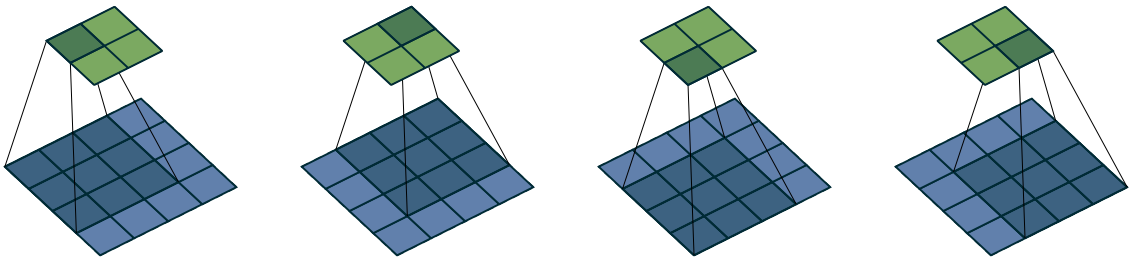
(d) ReLU6

Figure 2.4: Common non-linear functions applied as activation functions in DNNs, with the x-axis representing the input to the function ($x$), and the y-axis being the corresponding output ($\phi(x)$).

output matrix. Each application of the window produces a single output value. Therefore, convolutional layers reduce the number of parameters required by using the same weights for different input windows.

Convolutional layers are popular in image processing tasks, as they are effective at efficiently identifying patterns in the images such as curves, lines, textures, etc. We typically have multiple sliding windows, called *filters*, comprised of several neurons with their own weights, which together identify a given pattern. The more filters we have, the more weights we need to learn, and the more computations we need to perform in the layer. However, more filters increase the number of patterns we can identify, and may contribute to a higher accuracy. Algorithm 1 shows a simplified version of a convolution algorithm over 4D data.

Relevant configuration parameters for convolutional layers include the size of the filters, padding, and strides used. Increasing the filter size results in more parameters and computations for each output value. In Figure 2.5 the filter size is 3×3. Padding is when we include additional data, typically zeros, at the edge of the inputs. This can help ensure that our output shape meets a given size, and that inputs near the edge of the input volume are given an appropriate amount of processing. Strides is the step size that the filters make across the input



(a) (b) (c) (d)

Figure 2.5: A visualization of the sliding window of convolutional layer. The 2×2 output is shown above in green, and the 4×4 output is shown below in blue. In each step, we calculate a single output value by multiplying and accumulating a 3×3 slice of our inputs with a 3×3 weight matrix. The same weight matrix is reused for calculating the other three outputs. Figures adapted from Dumoulin and Visin [DV18] under an MIT license.

data, with higher strides typically meaning fewer computations are required. In Figure 2.5, we have no padding and a stride size of one. Note that padding, strides, and filter size are not necessarily symmetric, e.g., we could have 2×3 filters, padding on only the top of the input data, or a larger stride along the width dimension.

### Miscellaneous Layers

Other common and relevant layers include:

- *Pooling layers* such as max pooling and average pooling, which do not have learned parameters, and instead reduce the amount of input data we have by taking groups of input data and returning a single value, the maximum and average value for max and average pooling respectively.
- *Batch normalization layers* [IS15], which are used in training to ensure that our output data from other layers such as convolutions is normalized (i.e., zero mean and unit variance), which can aid in training.
- *Self-attention layers*, used in Transformer [Vas+17] architectures, which aim to capture the relationship between sequences of input tokens, regardless of distance. This allows the model to selectively focus on different parts of the input, and is especially useful in NLP tasks, where text can be represented as a sequence of tokens. For example, in NLP a self-attention layer can help identify how different words in a sentence relate to one another (i.e., grammar and semantics), and which words are most important for determining the overall meaning.

**Algorithm 1** Direct Convolution in `NCHW` Layout

**Require:** Input tensor $X$ of shape $N \times C \times H_{\text{in}} \times W_{\text{in}}$
**Require:** Filter tensor $W$ of shape $C \times M \times H_f \times W_f$
**Require:** Strides $S_h, S_w$
**Require:** Output height $H_{\text{out}} = \left\lfloor \frac{H_{\text{in}} - H_f}{S_h} + 1 \right\rfloor$
**Require:** Output width $W_{\text{out}} = \left\lfloor \frac{W_{\text{in}} - W_f}{S_w} + 1 \right\rfloor$

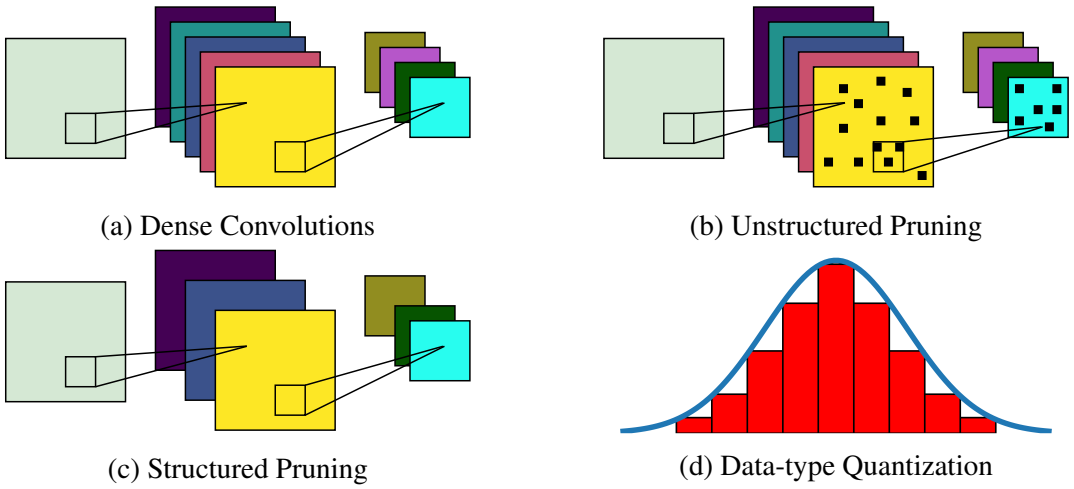
Initialize output tensor $Y$ of shape $N \times M \times H_{\text{out}} \times W_{\text{out}}$ with zeros
**for** $n = 0$ to $N - 1$ **do**
  **for** $m = 0$ to $M - 1$ **do**
    **for** $h_{\text{out}} = 0$ to $H_{\text{out}} - 1$ **do**
      **for** $w_{\text{out}} = 0$ to $W_{\text{out}} - 1$ **do**
        **for** $c = 0$ to $C - 1$ **do**
          **for** $h_f = 0$ to $H_f - 1$ **do**
            **for** $w_f = 0$ to $W_f - 1$ **do**
              $h_{\text{in}} = h_{\text{out}} \times S_h + h_f$
              $w_{\text{in}} = w_{\text{out}} \times S_w + w_f$
              $Y[n, m, h_{\text{out}}, w_{\text{out}}] \mathrel{+}= ($
                $X[n, c, h_{\text{in}}, w_{\text{in}}] \times W[c, m, h_f, w_f]$
              $)$
  **return** $Y$

### 2.2.4 Neural Architectures

Neural architectures are usually characterized by their inclusion of a given type of layer, or a particular quirk of their topology. There is a wide range of neural architectures [Vee16], however in this thesis the two most relevant ones are *Convolutional Neural Networks (CNNs)* and *Transformers*, as they are widely used in a range of applications, as well as the evaluations in this thesis. CNNs are characterized by their use of convolutional layers and are commonly used for computer vision tasks. Transformer architectures are characterized by their usage of self-attention [Vas+17], and are commonly used in NLP and computer vision tasks. Section 3.2.1 discusses some popular and influential examples of these architectures, and we highlight specific architectures evaluated in this thesis in the experimental setup of the relevant chapters.

## 2.3 Model Optimizations

A common observation is that neural networks are overparameterized, and similar accuracies can be achieved with smaller models [FC19]. As a result, a wide range of model optimization and compression techniques have been proposed in the machine learning community, which aim to reduce the size of a given DNN. Figure 2.6 shows some common



(a) Dense Convolutions (b) Unstructured Pruning

(c) Structured Pruning (d) Data-type Quantization

Figure 2.6: A visual representation of different Model Optimization techniques: (a) shows a slice of a typical DNN. Input on the left is computed with filters in the center, to produce output on the right; (b) shows the same network slice with weight pruning applied. A subset of parameters in the filters are forced to zero (visually represented with black holes) producing sparse matrices; (c) shows the network slice with channel pruning applied onto the filters, where there are fewer channels; (d) shows data-type quantization where the range of values that a given parameter of a DNN has been reduced.

compression techniques, with an uncompressed (also known as dense[2]) convolution shown in Figure 2.6a. Training is a computationally expensive process, potentially costing millions of dollars [Bro+20], and consuming significant amounts of energy [Pat+21]. Therefore, this makes it an attractive area for cost reduction. However, the model optimization techniques discussed in this thesis prioritize inference. This is because inference is more likely to be performed on more constrained hardware, where DNNs are deployed, as compared to the high-end hardware typically used for training. In addition, if we consider wide-scale long-term deployment, then inference will typically represent the majority of a DNN's costs over its lifetime, whereas training is an initial upfront cost.

For these reasons, this thesis focuses on techniques for inference acceleration. From a systems perspective, model optimization techniques do not necessarily provide speedups in-and-of themselves, unless lower levels of the stack adequately support it. For example, hardware supporting efficient computation using data-types with fewer bits, or algorithms and data formats that exploit pruning to skip operations and reduce memory usage.

### 2.3.1 Pruning

Pruning is the technique of setting parameters in a network to zero, which if exploited can reduce the computational or memory demands of a given DNN model, with potential accu-

[2]Not to be confused with fully-connected layers which are also referred to as 'dense' layers.

racy loss. Blalock et al. [Bla+20] characterize four key dimensions that distinguish pruning methods: *Structure*, *Scoring*, *Scheduling*, and *Fine-tuning*. We briefly discuss each of these features as they are relevant to this thesis, and discuss more examples in Section 3.3.3.

- *Structure*: the level of granularity of the pruning. We have two main categories of pruning: unstructured, where we prune individual parameters, as seen in Figure 2.6b; and structured, where we prune groups of parameters such as neurons, blocks, filters, or channels, as seen in Figure 2.6c.

- *Scoring*: how we decide which parameters to prune. We can prune parameters globally or layer-wise. With global pruning, given some pruning target (e.g., 50% of parameters should be pruned), the pruning algorithm will find the best parameters to prune across the *whole model*, meaning that some layers will be more or less pruned than others. For layer-wise pruning, we prune by a pre-defined amount per layer, e.g., every layer will be pruned by 50%. Given our choice of global or layer-wise, we need to rank the candidate parameters in some way to determine which are the least important, and therefore are likely to have the lowest impact on accuracy if removed. A popular approach is the *L1-norm*, where we prune parameters that have the lowest absolute value, i.e., closest to zero. Other ranking approaches include gradient-based methods and Taylor series expansion.

- *Scheduling*: how much we prune in each step. If we have a pruning target, for example, 70%, we might prune all the parameters in a single step, also known at *one-shot* pruning. Alternatively, we could prune a fraction of our target iteratively over several steps, e.g., first we prune 5%, then 10%, and so on until we reach our 70% target.

- *Fine-tuning*: how we recover lost accuracy due to pruning. Once we have pruned our model, we can retrain to recover lost accuracy using a method known as fine-tuning. Typically, we will use a lower learning rate than if we were training from scratch, ensure that our pruned parameters remain pruned, and tune for a small number of epochs. If we are pruning iteratively, we may want to fine-tune after each pruning step, however this can increase the time required significantly.

### 2.3.2 Quantization

Another compression technique is *data-type quantization*, which reduces the number of bits used to represent parameters or data. A visualization of this is shown in Figure 2.6d, where a continuous function is approximated with 9 distinct values. Typically, DNNs are trained using the `float32` data-type, however when they are deployed we can reduce the precision, often with minimal accuracy penalties. Common quantized data-types include `float16`

and `int8`, as well as emerging machine learning specific types such as `bfloat16`. In the extreme case, we can reduce our representation to using a single bit, also known as binarized neural networks, discussed more in Section 3.3.2.

For some types of data-type quantization it may be necessary to add additional operations to the DNN, as for example, in many `int8` DNNs we need to store the results of our computation as `int16` values, then rescale back into `int8`. This is because a sufficiently large `int8` value multiplied by another large `int8` value will require more than 8-bits to be represented. `float32` DNNs do not typically have this problem, since regularization means that our values are rarely large enough. However, this also highlights the problem: DNNs do not use the full range of values allowed by `float32`, meaning we are using more resources than we require, which is inefficient.

Similar to pruning, it may be necessary to use some form of fine-tuning, or calibration, to try and recover some of the lost accuracy. Calibration is the process of determining the values to use as our rescaling factors, and we may use calibration alone or in combination with fine-tuning of the model parameters.

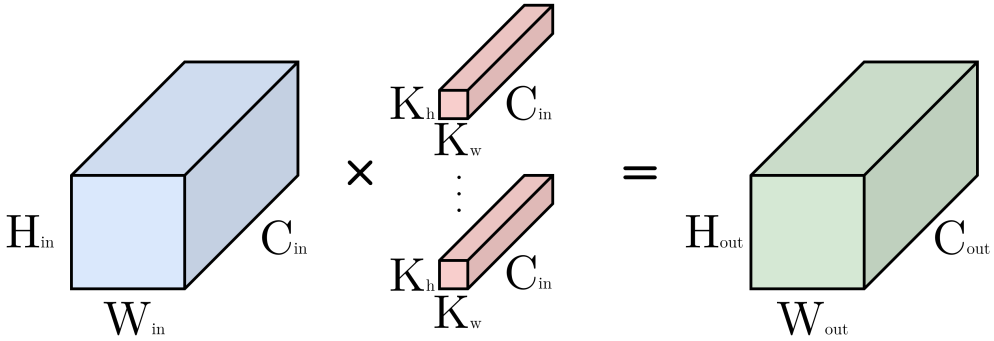
### 2.3.3 Cheaper Operations

The most expensive layers in a DNNs usually have a high number of parameters or operations. For CNNs, the convolutional layers are likely to represent the majority of the memory footprint and inference time, since it has a large number of Multiply-Accumulate (MAC) operations and weights. Therefore, it is desirable if we can find alternative versions of these layers which have similar representational capacity (and therefore accuracy), but with lower costs. This may allow us to deploy our DNN on more constrained hardware platforms.

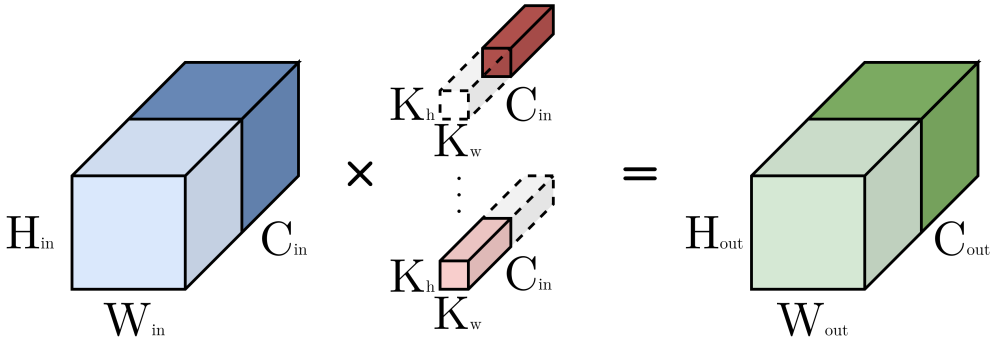
Grouped convolutions are a cheaper form of convolutional layers, where we divide our filters between subsets of the input data. This reduces the number of MACs we need to perform and the number of parameters we need to store. We denote these grouped convolutions as $G(g)$ where $g$ is the number of groups, and standard convolutions as $S$. Grouped convolutions allow for a trade-off between increased model compression against reduced accuracy as we increase $g$. The compression scales linearly with $g$.

Consider a standard convolution as depicted in Figure 2.7(a): its input consists of $C_{\text{in}}$ channels. Each of the $C_{\text{out}}$ filters is convolved with *all* of these input channels to produce a single $H_{\text{out}} \times W_{\text{out}}$ output. These outputs are concatenated to give the $C_{\text{out}}$ channel output of the convolution. Each filter uses $C_{\text{in}} \times K_h \times K_w$ parameters, where $K_h \times K_w$ is the filter size. As we have $C_{\text{out}}$ filters in total, then the overall parameter cost is $C_{\text{out}} \times C_{\text{in}} \times K_h \times K_w$.

Now, consider instead the case where *half* of our $C_{\text{out}}$ filters are convolved with the first half of our $C_{\text{in}}$ input channels, and the other half of our filters are convolved with the second




(a) Standard convolution.




(b) Grouped convolution (with 2 groups).

Figure 2.7: Standard vs. grouped convolutions: (a) In a standard convolution *S*, each filter is convolved with all of the input's channels; (b) In a grouped convolution with two groups $G(2)$, half of the filters are applied to each half of the input for a $2\times$ reduction in parameters used. More generally, a grouped convolution with $g$ groups uses $g\times$ fewer parameters than a standard convolution.

half of our $C_{\text{in}}$ input channels, as depicted in Figure 2.7(b). The filters now only use half as many parameters since each is now of size $C_{\text{in}}/2 \times K_h \times K_w$. This is a grouped convolution using two groups, and the total parameter cost and number of MACs is half of a standard convolution. For $g$ groups the parameter cost is reduced by a factor of $g$. The extreme case where $g = C_{\text{in}}$ is called *depthwise separable* convolutions [How+17; Sif14].

A disadvantage of grouped convolution is that it prevents information from different input channels from mixing across groups. This can negatively impact accuracy. To counter this, practitioners typically follow grouped convolutions with a pointwise convolution (i.e., a filter size of $1\times1$), which incurs an additional $C_{\text{out}} \times C_{\text{out}}$ parameter cost.

### 2.3.4 Knowledge Distillation

Knowledge distillation is a model optimization technique which can complement other approaches to compression. It allows us to use large pre-trained models to help train smaller models. Typically, we refer to the larger pre-trained model as the *teacher* and the smaller model as the *student*. Attention transfer [Tur+18b; ZK17] is a common knowledge distil-

lation technique, where the structure of the student and teacher networks are similar. In attention transfer, during training we pass the student and the teacher the same inputs, and store the intermediate activations of the teacher. Then, when we run backpropagation on the student network, as well as trying to minimize the loss with respect to accuracy, we also try and make the student's activations match the teacher's. The intuition for this technique is that the teacher network has already learned an internal representation of the data which is useful for high accuracy. Thus, if the student can mimic this representation, then it may learn more quickly than if it were trained from scratch.

## 2.4 Algorithms & Data Formats

The layers of a given DNNs can be implemented in a variety of ways, as long as we still get the same output. For example, for the fully-connected layer described in Section 2.2.3, we highlighted that it can be computed as a matrix multiplication. Note that there are many algorithms that we can use to implement a matrix multiplication, as well as many ways to arrange data in memory. However, each algorithm will have varying trade-offs and optimal cases. For example, some matrix multiplication algorithms are optimized for large matrices, others for small ones; some are optimized for square matrices, others for rectangular ones; we might order our data by columns, or by rows. The algorithms and data formats that we use to compute our operations are important to enabling accelerated deployment, and their behavior may be influenced by the size and shape of the data, the properties of the hardware we are running on, as well as the choices around sparsity if we want to exploit pruning.

### 2.4.1 Data Layouts

An important component of DNN computation is data layout, which is how we order our data in memory. The memory of the hardware device may have a different structure to our tensor (e.g., 1D memory, but with 2D data). For 2D arrays, common formats include *row-major* and *column-major*, with the former meaning that data in the same row is contiguous in memory, and the latter meaning that data in the same column is contiguous in memory. Similarly, for 4D data which is more relevant to CNNs, two common formats are `NCHW` and `NHWC`, with `N` representing the batch size (i.e., number of input samples); and `C`, `H`, and `W` representing the number of input channels, the input height, and input width respectively. We may also *tile* our data, where we group our data in memory by a more complex criterion, for example, by taking $2\times2$ squares of inputs from our larger tensor.

### 2.4.2 Convolutional Primitives

For the CNNs models we evaluate in this work, the most important algorithm to optimize is the one for convolutional layers, since they are generally the most compute and memory intensive. Algorithms used in this work implementing convolutional layers include *direct*, *GEMM*, and *spatial pack* convolution.

*Direct* convolution is one of the simplest algorithms, and is typically close to the textbook definition of convolution seen in Algorithm 1. It does not reshape input data and weights, which reduces memory overheads. The technique is called a *sliding window*, as shown in Figure 2.5, since the filters of size $H_f \times W_f$ 'slide' over the input data.

*GEMM* convolution reshapes 4D input data into a 2D array, potentially replicating elements using a reshaping algorithm known as *im2col*. This means that the convolution can be computed as a matrix multiplication (i.e., a GEMM function). Algorithm 2 gives a pseudocode representation of the im2col function when our data is in the `NCHW` layout. GEMM is a common algorithm used in many domains, not just machine learning, therefore there are many highly optimized GEMM algorithms and implementations which can be exploited, which can amortize the overheads of generating the im2col matrix. In the case of a $1\times1$ pointwise convolution, as described in Section 2.3.3, we can compute GEMM directly without the need for an im2col matrix, which saves on reshaping costs.

*Spatial pack* convolution reshapes both inputs and weights into tiles, packed such that each tile is ideally loaded once. The reshaping of weights can be performed offline, whereas input reshaping must be performed during inference. Unlike GEMM convolution, the size of the reshaped input data is the same size, and tiling the inputs and weights is intended to enable exploitation of data reuse and SIMD vectorization, as discussed in Section 2.6.3. The spatial pack convolution algorithm is elaborated on by Zheng and Chen [ZC18].

### 2.4.3 Sparsity

To achieve savings from the pruning introduced in Section 2.3.1, our algorithm must support *sparsity*, i.e., exploiting the zeros generated by pruning to skip computation. In addition, our data format should represent the sparse data in a more compressed form to reduce memory usage. The data format should also complement the algorithm such that non-zero values can be efficiently accessed. The computational savings are enabled by the fact that regardless of the input value, a multiplication by zero will always be zero, will have no impact on the final accumulated output, and thus can be skipped. We can achieve memory savings from encoding consecutive zeros in a more compressed format.

In Chapter 4, we use the popular compressed sparse row (CSR) format, which represents 2D

**Algorithm 2** im2col function in `CHW` format

**Require:** Input tensor $\boldsymbol{X} \in \mathbb{R}^{C \times H \times W}$
**Require:** Kernel size $(K_h, K_w)$
**Require:** Stride $(S_h, S_w)$
**Require:** Padding $(P_h, P_w)$
  Calculate output dimensions $H_{\text{out}} \leftarrow \frac{H+2P_h-K_h}{S_h} + 1$
  $W_{\text{out}} \leftarrow \frac{W+2P_w-K_w}{S_w} + 1$
  $K \leftarrow K_h \times K_w \times C$
  $M \leftarrow H_{\text{out}} \times W_{\text{out}}$
  Initialize output matrix $\boldsymbol{Y} \in \mathbb{R}^{M \times K}$
  **for** $c \leftarrow 0$ to $C - 1$ **do**
    **for** $h \leftarrow 0$ to $H_{\text{out}} - 1$ **do**
      **for** $w \leftarrow 0$ to $W_{\text{out}} - 1$ **do**
        **for** $k_h \leftarrow 0$ to $K_h - 1$ **do**
          **for** $k_w \leftarrow 0$ to $K_w - 1$ **do**
            $m \leftarrow (h \times W_{\text{out}}) + w$
            $j \leftarrow (c \times K_h \times K_w) + (h \times K_w) + w$
            $h_{\text{orig}} \leftarrow h \times S_h - P_h + k_h$
            $w_{\text{orig}} \leftarrow w \times S_w - P_w + k_w$
            **if** $h_{\text{orig}} \geq 0$ and $w_{\text{orig}} \geq 0$ and $h_{\text{orig}} < H$ and $w_{\text{orig}} < W$ **then**
              $Y_{m,j} \leftarrow X_{c,h_{\text{orig}},w_{\text{orig}}}$
            **else**
              $Y_{m,j} \leftarrow 0$
  **return** $\boldsymbol{Y}$

data using three arrays: 1. The non-zero elements of the parameters (*data*), 2. the original column index of the corresponding parameters (*indices*), and 3. the first non-zero elements in each row, as well as the final non-zero element (*indptr*). Using this format means that we require up to three values to represent a single non-zero element, and hence, the use of CSR only results in memory savings when at least two-thirds or more of the data is zeros. Equation 2.1 shows a sparse matrix $X$, and Equation 2.2 shows its CSR representation.

$$W = \begin{bmatrix} 0 & 9 & 0 & 0 \\ 8 & 0 & 0 & 7 \\ 0 & 6 & 5 & 4 \end{bmatrix} \tag{2.1}$$

$$\begin{aligned} \text{data} &= \begin{bmatrix} 9 & 8 & 7 & 6 & 5 & 4 \end{bmatrix} \\ \text{indices} &= \begin{bmatrix} 1 & 0 & 3 & 1 & 2 & 3 \end{bmatrix} \\ \text{indptr} &= \begin{bmatrix} 0 & 1 & 3 & 6 \end{bmatrix} \end{aligned} \tag{2.2}$$

Sparse algorithms may have different trade-offs when compared to their dense counterparts, as their data access patterns will be irregular. Therefore, the best algorithm in the dense case may not be the best in the sparse case. In addition, the sparse data format may also be better or worse in a given scenario, including how it interacts with a given sparse algorithm. Other sparse data formats include BSR (block sparse row) and COO (coordinate list). These issues are discussed more in Section 3.4.2.

## 2.5 Systems Software for DNNs

Software is required to implement, execute, and optimize all of the above layers of DLAS. Additionally, it acts as an interface with the underlying hardware. This software can be high-level, such as the frontend for developing DNN solutions, or lower-level, such as back-end software for code generation, which machine learning developers rarely interact with directly, but are critical for correct and efficient execution.

### 2.5.1 DNN frameworks

DNN frameworks are the front-end for machine learning practitioners to develop, train, or deploy their models. There are a range of popular DNN frameworks used, with a variety of interfaces, strengths, and weaknesses. Some popular examples include: PyTorch [Pas+19], TensorFlow [Aba+16], MXNet [Che+15], JAX [FJL18], and Keras [Cho15]. The interface for using most of these frameworks is Python, however optional bindings for other languages such as R, C++, and Julia exist for some frameworks. Listing 1 shows an example definition of a single CNN layer in PyTorch, with a ReLU activation function. A full range of common DNN layer definitions are expected to be available in DNN frameworks, so developers do not need to define them manually.

Typically, each DNN framework has its own file format for representing models, which are not necessarily interoperable. Motivated by this, ONNX has gained popularity as an interchange format [BLZ+19], where many frameworks can export to and import from ONNX files. This has increased the portability of DNN models between DNN frameworks, making it easier to deploy DNNs using frameworks other than the one they were trained. ONNX defines a set of common DNN operations, a format for storing parameters, and a format for structuring the data dependencies between operations (i.e., the computation graph). However, ONNX does not currently support every layer type from every DNN framework due to the rapid evolution of new methods. As a result, converting DNNs which implement exotic or novel operations can be challenging.

### 2.5.2 General Purpose Compiler Infrastructure

The hardware that we execute our software on is controlled using machine code and related hardware signals. Therefore, to run our higher level code (such as C++ or Rust) on real hardware we require a compiler. Compilers translate code from one language into another, which includes translating higher level languages into lower level languages (e.g., assembly) that are closer to machine code. Most commonly, we think of compilers as generating a binary

```
import torch

class WeeNet(torch.nn.Module):
    def __init__(
        self, in_c: int, out_c: int, kdim: int, stride: int, pad: int
    ):
        super(WeeNet, self).__init__()
        self.layer1 = torch.nn.Conv2d(
            in_c,
            out_c,
            kernel_size=(kdim, kdim),
            stride=(stride, stride),
            padding=(pad, pad),
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        out = torch.nn.functional.relu(self.layer1(x))
        return out

model = WeeNet(*args) # initialize model
y = model(x) # run inference using input data x
```

Listing 1: Single-layer CNN definition in PyTorch, with a ReLU activation function.

containing machine code, i.e., instructions which can be used to control a CPU. Compilers typically involve six stages:

1. **Lexical analysis (Lexing)**: This stage groups the input program into distinct tokens, such as variable names, operators, keywords, and literals.

2. **Syntax analysis (Parsing)**: This stage processes the tokens produced by the lexer and checks if they follow the basic rules defined by the language grammar, for example, every `if` token must have a corresponding `endif`.

3. **Semantic analysis**: This stage interprets the output of the parser, and checks for semantic correctness, such as ensuring that variables are declared before use, that function calls have the correct number and types of arguments, and that expressions have valid data types.

4. **Intermediate code generation**: This stage generates an Intermediate Representation (IR) of the program, which is a lower-level abstract representation that is more suitable for further processing, optimization, and code generation.

5. **Optimization**: This stage transforms the intermediate code to optimize its performance, either by making it consume fewer resources (such as memory) or by improving its execution speed. Optimizations can be local or global.

6. **Code generation**: This final stage translates the optimized intermediate code into low-level machine code or assembly that can be executed directly by the target machine or

platform. The code generator handles low-level details such as register allocation, instruction selection, and memory management.

Common general purpose compilers include Clang and GCC. Regarding the IR, a notable system is LLVM [Lat02], a collection of cross-platform compiler infrastructure and an industry standard, which is characterized by *LLVM IR*. As a program representation, LLVM IR is designed to be self-contained and platform independent, and preserves some program structure such as control-flow constructs and data-types, which can aid in optimization. Compilers such as Clang translate higher level code into LLVM IR, and LLVM defines a range of optimizations which can be applied to the IR. Part of LLVM's success has come from the reusability and relative ease-of-use for the development and application of optimization passes. This has resulted in a wide range of programming languages and tools using LLVM as a backend, so they can leverage its widely used and well-supported infrastructure. As previously highlighted, the IR preserves some program structure that is less evident in machine code, such as higher-level data-types; and also allows optimizations to be more platform and language independent, contributing to LLVM's widespread adoption.

### Compiler Optimizations

Compiler optimizations play a critical component of modern computing, by transforming the code being generated to improve some desirable attribute, for example, increased throughout or reduced memory footprint. Some common and relevant optimizations include loop unrolling, loop reordering, vectorization, dead code elimination (DCE), constant folding, and common sub-expression elimination (CSE).

**Loop unrolling** refers to replacing `for` loops with the body of the loop repeated once for each iteration. Listing 2 shows an example of a basic `for` loop that has three iterations. Listing 3 provides an example of the same loop unrolled. We could see execution time reductions due to the fact that we do not need to check our condition `i < 3`, and increment our counter `i` every iteration. However, we could also increase execution time if a loop has many iterations that could increase our binary size, and negatively impact our cache performance. We can also employ partial loop unrolling, where we keep the `for` loop, but reduce the number of iterations by a factor of $N$ repeating the body of the loop $N$ times. This can help us achieve speedups even if a given loop has many iterations.

**Loop reordering** refers to changing the order of nested `for` loops. This can provide performance improvements if the reordering better exploits the data layout and algorithm's data access patterns. For example, we could see performance improvements if the innermost loop iterates over data which is contiguous in memory.

```
for (size_t i = 0; i < 3; ++i)
    c[i] = a[i] + b[i];
```

Listing 2: Unoptimized `for` loop

```
c[0] = a[0] + b[0];
c[1] = a[1] + b[1];
c[2] = a[2] + b[2];
```

Listing 3: Unrolled `for` loop

**Vectorization** refers to replacing multiple sequential instructions involving an array of data with single instructions that manipulate multiple elements of the array at once. Section 2.6.3 discusses the hardware features required to enable this optimization.

**Dead code elimination (DCE)** is the removal of parts of the program that do not change the output, and may not even be executed. For example, we might be storing data to a variable that is never used later in the program, or we could have a branch of an `if` condition that will never trigger (e.g., `if False`). If we remove this 'dead' code, then we can reduce the number of operations we need to perform, as well as the binary size.

**Constant folding** involves propagating knowledge of any constant values in the program. For example, when calculating an index offset in a `for` loop, we may have an expression `(H × W)`, where `H` and `W` are constants. This optimization replaces the expression with a third constant variable which is calculated ahead-of-time (AOT) by the compiler, so that we do not need to reevaluate it at execution time.

**Common sub-expression elimination (CSE)** targets situations where there are expressions that compute the same result. For example, we could have two statements, $a_1 = b + c \times d$, and $a_2 = e + c \times d$. CSE could add a third statement $f = c \times d$, and update the two original statements to $a_1 = b + f$ and $a_2 = e + f$. This means that we only calculate $c \times d$ once, which could potentially bring us savings.

A wide range of compiler optimizations are available, and the above are only a fraction of them, chosen because they are common and more relevant to this thesis. A single compiler optimization may not always result in performance improvements on its own, as multiple passes may be required, with the order of passes potentially having an impact on performance. Section 3.5 discusses this problem more generally.

### AOT versus JIT

Some compiler optimizations exploit knowledge about the program's structure, for example, the number of iterations used in a loop, as leveraged by *loop unrolling*. If we have this knowledge at compilation time, then we can exploit it immediately using an approach known as ahead-of-time (AOT) compilation. This leverages optimizations such as those described above, exploiting the fact that we have some of our variables and control-flow as constants. This means that our code could be faster than if these components are left as variables.

However, we may not know all the features of our program at compile time, for example, our program could accept inputs of arbitrary size and shape. In this case, we may want to embed parts of the compiler in our program, so that we can leverage *Just-In-Time (JIT)* compilation. JIT is where we recompile parts of our program during execution time to optimize it based on features of the input data. This recompilation can come with non-negligible costs (e.g., time, memory), however these may be acceptable if the speedups achieved are sufficiently high. In addition, if we cache our optimized code, then we can reuse it with reduced overheads if we encounter data with similar features again.

### 2.5.3 Hardware APIs

Typically, CPUs are interfaced with using high-level programming languages such as C++, which are compiled or interpreted into machine code. The programming paradigms for hardware other than CPUs vary due to their distinctive architectures, as Section 2.6 will discuss, which may require interfacing with them in different ways. For hardware accelerators like Graphics Processing Units (GPUs), programming paradigms including CUDA [Nic+08] and OpenCL [SGS10] provide a programming interface using a C-like programming language. More specialized hardware accelerators such as Tensor Processing Units (TPUs) may define their own programming interfaces, which may be less flexible due to their specialized nature. Generally, when using hardware accelerators, we still require CPU-side host code to manage accelerator calls and data transfers between the CPU and the accelerator, with data transfers potentially being a significant bottleneck if not managed correctly.

### 2.5.4 Kernel Libraries

Kernel libraries are optimized subroutines for common or expensive operations (such as GEMM), and are often exploited by DNN frameworks. Within DNN frameworks, they may be responsible for executing expensive computations of the DNNs, such as convolutional layers. An example of a kernel library is cuDNN [Che+14b], which is a collection of optimized CUDA kernels and host code which executes common and compute-intensive DNN operations for Nvidia GPUs. However, a key drawback of kernel libraries is that they often fall behind the continuous innovation of new DNN architectures [BI19], which can lead to suboptimal performance, and potentially discourage experimentation from machine learning practitioners. They also may not be adequate in the face of increasing heterogeneity of hardware platforms, with variants of every critical kernel being required to suitably exploit each sufficiently unique platform.

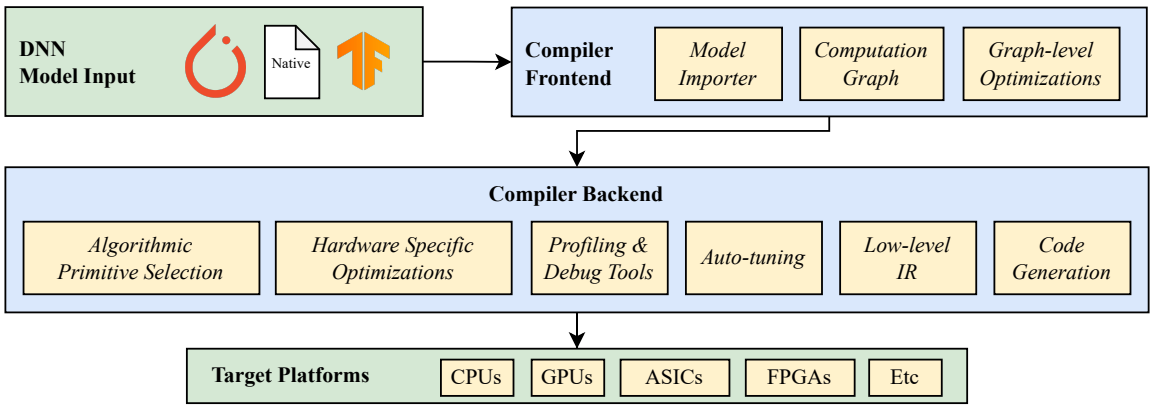



Figure 2.8: Overview of relevant components in modern tensor compilers

### 2.5.5 Tensor Compilers

An alternative to using optimized vendor libraries are *tensor compilers*, examples of which include Apache TVM [Che+18b] and IREE [The19]. Tensor compilers exist at a higher level of abstraction than other compiler infrastructure such as LLVM, with their purpose to represent and exploit domain-specific knowledge about the structure of potential optimizations available for DNNs. They generate code for a specific DNN on a specific hardware backend, and when leveraged correctly can outperform kernel libraries, especially for operations that may be less popular or optimized.

Apache TVM is the main tensor compiler leveraged in this thesis, due to its maturity, state-of-the-art performance, valuable features such as domain-specific languages to more easily develop new optimized algorithms (Section 2.5.6), as well as native support for auto-tuning the code (Section 2.5.7). Other relevant tensor compilers are discussed in Section 3.5.2, although they still broadly implement the same key components as TVM.

Figure 2.8 gives a high-level sketch of the key components that one would expect to find in a tensor compiler. They take a **DNN Model Input**, which is imported and optimized by the **Compiler Frontend**. Next, the **Compiler Backend** generates and optimizes the code that implements the particular layers and operations of the DNN model. Tensor compilers often leverage mature lower level infrastructure such as LLVM when generating their code, and do so after they have exploited and applied their domain-specific optimizations. Finally, the optimized code is lowered[3] into a representation that can be executed on the **Target Platforms**. Details of tensor compiler *frontends* and *backends* are given below.

[3] 'Lowered' meaning to compile from a higher level abstraction representation to a lower one.

```
for ni in range(N): # GEMM
  for mi in range(M):
    for ki in range(K):
      c[ni, mi] += (
        a[ni, ki] * b[ki, mi]
      )
for ni in range(N): # ReLU
  for mi in range(M):
    c[ni, mi] = relu(c[ni, mi])
```

Listing 4: Non-fused operators

```
for ni in range(N): # GEMM+ReLU
  for mi in range(M):
    for ki in range(K):
      c[ni, mi] += (
        a[ni, ki] * b[ki, mi]
      )
    c[ni, mi] = relu(c[ni, mi])
```

Listing 5: Fused operators

### Compiler Frontend

The first step is for the frontend to convert the **DNN Model Input** into a standard format. Often the **DNN Model Input** is pre-trained from another DNN framework such as PyTorch, or is in the ONNX format, however tensor compilers can also define models natively. The *Model Importer* reads the **Model Input**'s structure and parameters, and reproduces them using the tensor compiler's internal representation of a DNN.

This produces a *computation graph*, where the nodes are the atomic operations of the model, such as convolutional layers. This *computation graph* will usually be without lower level implementation details such as the algorithm used. An example of a DNN computation graph is shown in Figure 2.3, where we have convolutional, pooling, and fully-connected layers, as well as how they are connected together. Finally, *graph-level optimizations* will be applied, which can use techniques such as DCE and CSE to remove redundant nodes in the *computation graph*. The frontend may also leverage some domain-specific knowledge to apply techniques such as *operator fusion* to the graph.

*Operator Fusion* refers to combining nodes in the graph together into a single operation, which takes two main forms. The first combines the parameters of one operation with another, completely removing it from the graph. An example of this is with batch normalization and convolution: during inference we can simplify the operation sequence so that we are left with just a convolution with updated parameters. The second form of operator fusion moves the second operation into the loop nest of the first operation, so that the second operation is applied, but without having to perform a second full traversal of the data. An example of this is shown in Listings 4 and 5, where there is a GEMM followed by a ReLU. In the fused version, we apply ReLU immediately after the final MAC operation has been performed for a given `c[ni, mi]`. This means that it will still be in memory, which can bring significant savings compared to the non-fused version, where it will need to fetch data later, at which point it may have been evicted from the cache.

### Compiler Backend

After applying *graph-level optimizations*, the next stage is to use the compiler backend. The resulting code can be executed on a target hardware platform, which efficiently implements a given DNN. There are a variety of stages that can be exploited to realize and optimize this code while it is being generated. Note that not every tensor compiler will necessarily implement all of these stages, or may have additional ones. However, the stages presented here are the most relevant in the context of this thesis.

*Algorithmic primitive selection*: choice of the high-level algorithm that is used to implement each node in the computation graph. For example, for convolution we may choose one of the algorithms described in Section 2.4 such as direct or spatial pack. This stage may also select an algorithm implementation from a kernel library, rather than providing one from the tensor compiler itself. When using a kernel library, it will most likely be treated as a black-box by the tensor compiler, whereas in the latter case the tensor compiler backend will be able to apply further optimizations to the code of the algorithm.

*Hardware-specific optimizations*: manipulates the code of the algorithm to enable features like thread-level parallelism, SIMD-vectorization, and loop manipulations such as the ones discussed in Section 2.5.2. They should exploit the features available on the target hardware, for example, a multicore GPU would benefit more from higher degrees of parallelism. More discussion of this is given in Section 2.5.6.

*Lower-level IR*: the output of hardware-specific optimizations, closer to executable machine code, and may encode device specific features such as threads, blocks, or vectorization, which will be substituted with the relevant instructions and hardware API calls at the *Code Generation* stage.

*Auto-tuning*: automatically searches for additional optimizations for a given node and hardware platform, as discussed in more detail in Section 2.5.7.

*Profiling & Debug Tools*: inserts extra code into the program to collect information about the tensor program's execution, for example, storing intermediate outputs to debug incorrect output, or adding support for hardware counters that will report the resources such as the number of CPU clock cycles used by a given layer.

*Code Generation*: converts the lower-level IR to a representation that can be executed or optimized by the lower level compiler infrastructure. For example, we could convert from the tensor compiler's lower-level IR to LLVM IR, and then from LLVM IR to machine code. We may also generate code which targets other backends such as CUDA or OpenCL.

The previous sections cover all the main components in a typical tensor compiler. The following sections discuss some specific features seen in the TVM compiler in more detail, which make it a particularly effective tensor compiler.

```
A = te.placeholder((m, n), name="A")
B = te.placeholder((m, n), name="B")
C = te.compute(
  (m, n), # output shape
  lambda i, j: A[i, j] * B[i, j],
  name="Hadamard"
)
```

Listing 6: An example of an algorithm implementation in TVM.

```
sch = te.create_schedule([C.op])
# split the inner loop into two
no, ni = sch[C].split(
  C.op.axis[1], factor=32
)
# apply SIMD-vectorization
sch.vectorize(ni)
```

Listing 7: An example of a schedule implementation in TVM.

### 2.5.6 Compute Schedules

*Compute schedules* is a programming paradigm for tensor compilers which decouples the high-level description of an algorithm from the description of how it should be optimized for a given hardware platform. It has been seen in systems such as Halide [Rag+17], TVM [Che+18b], and RISE/ELEVATE [Hag+20]. Developers express their algorithms using a relatively simplified representation (e.g., NumPy-style syntax), which makes it easier to interpret and debug its behavior. The schedule is a separate function which defines the transformations to the algorithm's code, usually expressed in a domain-specific scheduling language. The schedule defines which optimization choices are applied, such as decisions about the intermediate storage and the order of computation, and to which parts of the algorithm they are applied. Often, there are several schedules defined for different platforms, for example, one for CPUs and one for GPUs, using the same algorithm definition.

The separation between high-level algorithm and platform specific transformations can allow clearer reasoning about the performance impact of optimizations, while ensuring that the semantics and readability of the algorithm is preserved. This is contrasted to defining the algorithm and its optimizations in the same code, as seen in system programming languages such as C, C++, and Rust. In these cases the original algorithm may be obfuscated by its optimizations. Additionally, as previously highlighted, a single algorithm can have multiple schedules for different cases, such as hardware architectures or kernel properties. This is seen in TVM, which defines different schedules for the same algorithm for CPUs, GPUs, and specialized hardware accelerators [Mor+19; SGC22]. Listing 6 shows an example of an implementation of matrix element-wise multiply in TVM. A complementary schedule for this kernel is shown in Listing 7, which optimizes the code by splitting the innermost loop and applying vectorization.

A well-designed schedule is key to achieving accelerated performance. Even small changes in the schedule can yield large changes in the speed of the generated code, sometimes by orders of magnitude. However, writing optimized schedules by hand requires domain expertise, including understanding of the algorithm's design [Gib+20] and the behavior of the

target hardware [ZC18]. In addition, an optimized schedule may not necessarily be efficient for all instantiations of the target algorithm. For example, if the schedule was optimized assuming that the size of the inner loop would always be small, it may experience slowdowns in cases where the inner loop is large. In this case, optimizations which were used to exploit this assumed kernel characteristic might not help, and may even hinder performance.

### 2.5.7 Auto-tuners and Auto-schedulers

Tensor compilers can be taken even further by automatically tuning the code for a given layer instance and hardware platform, also known as auto-tuning. This can improve the performance beyond what is achieved by generic optimizations, which may only optimize for common cases, leaving the majority of cases unoptimized. As a basic example of auto-tuning, consider the loop unrolling example in Listing 3. If the number of loop iterations is low, then loop unrolling may give performance improvements. However, if the number is high, then we may see a performance degradation, due to increased code size. Determining this threshold can vary based on the cost of the loop body and the hardware being used. An auto-tuner can take the code for a given kernel of a given size, and test its performance using different optimization configurations, e.g., if loop unrolling increases or decreases performance. We can imagine this approach being applied to many other optimizations, such as loop reordering. Many works have demonstrated that auto-tuning techniques can outperform human experts [Ash+18; LC20].

Auto-schedulers are a special case of auto-tuners for systems using the computer schedule paradigm. This allows us to automatically explore the full space of transformations exposed by a scheduling language, and potentially achieve higher speedups. However, this can also increase the size of the search space significantly, when compared to an auto-tuner which explores a pre-defined set of optimization parameters.

An example of an auto-tuner used in this thesis is AutoTVM [Che+18a], and an example of an auto-scheduler is Ansor [Zhe+20a], both of which are defined for TVM. AutoTVM explores predefined 'tuning knobs' defined by the creator of a given schedule, with possible configuration choices including checking if we unroll a loop, or the tile size to use. Ansor does not require a handwritten schedule, and only requires the instantiation of a given algorithm (which we refer to as a kernel, especially in Chapter 6) to begin optimizing it.

Much like NAS (introduced in Section 2.2.4, and discussed more in Section 3.2.2), the two key factors for auto-tuners and auto-schedulers are: 1. The search space described, since a larger space will be more likely to include faster tunings, but will be more expensive to search; and 2. the search strategy, which will determine how expensive the search process is, and now likely we are to find a good tuning in our space. The size of the search space

for AutoTVM is determined by the number of tuning knobs defined for a given handwritten schedule. The search space for Ansor can be orders of magnitude larger, since it can compose together an arbitrary number of schedule primitives.

Both AutoTVM and Ansor use cost models to estimate how fast a proposed configuration will be on the target hardware. This can help them exclude variants which are likely to perform poorly, reducing the search time. Also, for their search strategies, they can leverage evolutionary algorithms, which make small changes to their best performing programs rather than randomly jumping around their search space. However, Ansor can find more optimized schedules much faster than AutoTVM, since it has a more efficient search strategy coupled with a search space which is less constrained. The net result of this is that Ansor finds a *better* schedule in *less* time than AutoTVM.

## 2.6 Hardware for DNNs

Ultimately, DNNs must be executed on physical hardware, which will dictate the upper limit of their throughput (see the roofline model shown in Figure 1.2). For a given deployment, we may be limited to a given device, and thus must operate within the constraints available. Alternatively, we may have several devices available, and thus will want to choose the best device in terms of some metric of success, such as throughput, maximum supported model size, cost, or energy consumption. This section introduces the classes of devices relevant to this thesis, as well as some of their hardware features which can be exploited by other layers of DLAS. For a more comprehensive background on hardware as it pertains to deep learning, we refer the reader to the following overviews [CPJ21; Dhi+22; Hoo21].

### 2.6.1 Devices

Hardware that DNNs commonly execute on includes general purpose computing devices such as CPUs and GPUs, as well as more specialized accelerators such as TPUs and other ASICs such as MAERI [KSK18] and Eyeriss [Che+17; Che+19]. CPUs are generally complex independent processing cores, typically with one to several dozen cores on a single chip. GPUs are generally simpler interdependent processing cores, typically with dozens to several thousand cores on a single chip. Both CPUs and GPUs define Instruction Set Architectures (ISAs) which are an abstract software interface to the underlying hardware, defining the data manipulations that the hardware can perform. Examples of instructions include data handling and memory operations (such as loading and storing), arithmetic operations (such as addition and multiplication), and control flow operations (such as conditions and branching). The specific instructions available vary by device, with CPU ISAs relevant to

this thesis being x86-64 and Arm-V8, and a range of GPU ISAs which are more diverse and less documented. These instructions are usually generated by a compiler, however they can be handwritten by the intrepid.

More specialized accelerators are designed specifically with DNNs in mind, such as Tensor Processing Units (TPUs) and Neural Processing Units (NPUs). These accelerators may or may not have their own ISAs, but given their specialized nature they will usually define significantly fewer instructions, with a single instruction potentially representing the equivalent of thousands of CPU instructions, since it may be executing a more coarse-grained operation such as a GEMM. More details of hardware accelerators are given in Section 3.6, as the core contributions of this thesis are using CPUs and GPUs.

Key metrics that should be considered regarding DNN-processing hardware include throughput and energy consumption. If we are designing the hardware, then we must also consider the area of the chip, which strongly correlates with cost.

### 2.6.2 Edge versus Cloud

The range of applications and modes of DNN usage have varying computational requirements. The *edge/cloud dichotomy* has emerged as a way of delineating between two classes of hardware platforms, with *edge* referring to devices at the edge (or near it) of computer networks, such as smartphones and Internet of Things (IoT) devices, typically with fewer computational and networking resources; and *cloud* referring to devices in data-centers such as rack-mount servers with high-end GPUs, typically with more computational and networking resources. Typically, training is performed in the cloud, as we require a large amount of processing power, which powerful server-class GPUs or TPUs are well suited for.

The *cloud* is also ostensibly a more desirable inference deployment platform for similar reasons. However, the data which we want to process in AI applications is typically at the edge. If network connectivity or latency is limited, then sending data from the edge to be processed in the cloud may not be practical, with network scaling issues as the number of connected edge devices increases [CR19], and there may also be data privacy and security concerns sending data to the cloud [KTZ19]. Therefore, we may be required to run at the edge, which further motivates efficient acceleration using the techniques within DLAS, given the more constrained nature of the hardware.

### 2.6.3 Hardware Features

General purpose hardware such as CPUs and GPUs are Turing-complete, meaning they can theoretically execute any computational function. However, they may be more or less efficient for some computations, with varying hardware specialisms that accelerate different

classes of computations. At a high-level, key concepts around the speed of a computation include: how fast a single instruction can be executed, how fast data can be read from or written to memory, and how much parallelism can be and is exploited.

### Multicore Processing

If a single chip has more than one processing unit that can work in parallel, this can reduce the time needed for programs which can be split into parallel threads of computation. Many DNN computations can be parallelized, however often still require communication between cores, for example, we may need to combine the outputs calculated by each core for our final result. As discussed in Section 2.6.1, CPUs typically have fewer cores (one to several dozens) which are more complex, and GPUs typically have more cores (four to several thousand) which are less complex. However, this comparison mostly illustrative, to indicate that GPUs are typically designed around higher levels of parallelism. In practice the design CPUs and GPUs is different enough such that a direct comparison in number of cores is not fair or sufficiently informative. Computer scientists typically use Amdahl's law [Amd67] to calculate the theoretical speedups from leveraging parallelism, which compares parts of the program which can be parallelized against parts which cannot. Programs which are highly parallel, with regularity between computation units, may be better suited for GPUs, whereas less parallel programs may be better suited for CPUs, since individual CPU cores are typically faster than GPU cores.

Some CPU architectures are heterogeneous, for example, Arm big.LITTLE, where relatively lower power slower processor cores (LITTLE) are coupled with relatively more powerful and power-hungry ones (big). Applications can choose which cores to use for different tasks, which can help reduce overall all power consumption, while still allowing higher throughput computation when required.

### Memory System

Another important aspect of hardware is the memory system. Computation requires fast access to the data being computed upon, however there are practical limitations regarding how large and how fast our memory systems can be. Therefore, conventional computation devices have a memory hierarchy: with fast, small data storage (caches) being close to the processor cores, and larger, slower memories being higher up in the hierarchy. Data inside processor registers (L0) is the fastest to access, and we will typically have multiple levels of caches above that, with L1 being the fastest and smallest, usually with one for each core, with larger, slower levels of cache such as L2 and L3 being shared between cores. Nearer the top of the memory hierarchy we have large main memory, which has comparatively high

access times. It is critical for performance optimization to have a program use data as much as possible once it has been loaded into cache, since if it is evicted, and then is required again, we can incur high latency costs loading it from a higher level of the memory hierarchy.

As well as direct programmer/compiler control of the order data is loaded and used in a program, cache systems also have their own internal policies to determine which data should be retained, and which should be flushed. Often these policies are implemented in hardware, with examples including least recently used (LRU) and dynamic re-reference interval prediction (DRRIP) [Jal+10]. Cache policies can vary significantly between devices, and may be opaque to the user. Therefore, efficient algorithms and systems software need to account for this behavior, which may require trial-and-error to model accurately.

A related issue to the memory system is data-transfer between a host CPU and a hardware accelerator, which can be a critical bottleneck to efficient processing. In this case, we can think of an accelerator as our processor core, and the CPU as part of its memory hierarchy, with a similar optimization target of ensuring that we keep data transferred over the slow host-device data buses in the accelerator for as long as possible.

### SIMD processing

In DNNs, we are regularly working with linear algebra operations such as matrix multiplication. Conventional hardware instructions compute on single items of data at a time, for example, adding two numbers together and storing the result. Single instruction, multiple data (SIMD), or vector, instructions allow more than one element of data to be loaded or computed upon using a single instruction. For example, we might have a vectorized addition instruction which loads multiple pair of numbers at once, and computes the sum of each pair, returning a set of multiple outputs. As a more concrete example, the Intel SIMD instruction `MOVAPS` (`_mm_load_ps` as a vector intrinsic for C/C++ programmers) loads four `float32` values in a single instruction, which in theory represents a $4\times$ speedup compared to loading them one-by-one. Practically, micro-architectural considerations means that this speedup may vary. However, for some classes of programs, SIMD instructions can enable an additional level of parallelism to complement multicore processing, and may bring significant speedups. Different architectures may support varying maximum SIMD lengths, e.g., Intel's AVX instructions support up to 256 bits, with AVX-512 supporting up to 512 bits; whereas Arm Neon supports up to 128 bits.

### Data Types

As discussed in Section 2.3.2, DNNs are typically trained using `float32` data types, which generally have well optimized data paths for CPUs and GPUs. Different hardware may have

varying support and levels of optimization for different data-types, such that data types with fewer bits may not necessarily be faster to compute. Some devices may not have support for a given data-type at all, and it can only be computed upon using some emulation using a supported type. For example, some CPUs compute on `float16` data using `float32` instructions, potentially incurring additional overheads compared to using `float32` data. In comparison, a GPU may have explicit `float16` instructions, meaning that using `float16` data would be faster than the equivalent computation using `float32`. Also note that since many of the operations of DNN tend to be dot-products (see Section 2.2.1), this can mean that data-type of the accumulated value may need to be larger than the data-type of the product operands. For example, in the case of DNNs using the `int8` data-type, often we require at least `int16` values to accurately store our accumulated value.

## 2.7 Summary

This chapter provides background on the DLAS techniques relevant to this thesis, spanning across machine learning and systems. These topics are evolving rapidly, due to the high interest from both research and industry communities. For instance, many new and significant DNN architectures appear every year, and novel computing hardware targeted to DNNs also emerge with similar frequency. The following chapter surveys the wider research literature relevant to this thesis, further contextualizing DLAS as well as highlighting publications which have similar approaches to the techniques developed in this thesis.

# 3 Related Work

This chapter provides a survey of the literature in areas relevant to this thesis. Similarly to Chapter 2, it is structured following the layers of DLAS. In addition, Section 3.7 discusses the complementary publications highlighted in Section 1.5.

## 3.1 Datasets & Problem Spaces

The datasets and problem spaces targeted by deep learning based solutions are generally assumed to be static when designing an application. In research there is a wide range of common datasets which are used to evaluate performance on a variety of tasks, and are often used as standard benchmarks to compare solutions against each other. The website `paperswithcode.com` ranks state-of-the-art solutions to these learning tasks, and can be a valuable reference for this rapidly evolving and highly competitive domain. Here are some examples of popular public datasets and benchmarks where DNN-based solutions can be effective and appear often in the literature. This is not an exhaustive list, but it provides an idea of the breadth and variety of datasets for which DNNs can be an effective solution, and how state-of-the-art systems can be more fairly compared against each other.

- **Image Classification**: ImageNet [Den+09], CIFAR-10 [Kri09], MNIST [Den12], Fashion-MNIST [XRV17], SVHN (Street View House Numbers) [Net+11].
- **Natural Language Processing (NLP)**: GLUE [Wan+19a], SQuAD [Raj+16; RJL18], IMDb Reviews [Maa+11], BookCorpus [Zhu+15].
- **Object Detection**: COCO [Lin+14], PASCAL VOC [Don+21c], DOTA [Xia+18], Open Images [Kra+17].
- **Image Generation**: Flickr-Faces-HQ [KLA19], CelebA [Liu+15], Animal FacesHQ (AFHQ) [Cho+20a], MetFaces [Kar+20b].
- **Autonomous Driving**: CARLA (Car Learning to Act) [Dos+17], nuScenes [Cae+20], Waymo Open Dataset [Ett+21; Sun+20a].

- **Healthcare**: AlphaFold Protein Structure Database [Var+22], ChestX-ray8 [Wan+17], IGSR [Fai+20].
- **Games**: Gym [Bro+16], Minecraft, Starcraft, Atari, Chess, Go, DOTA 2.
- **Speech & Audio**: LibreSpeech [Pan+15], TIMIT [Gar+93], Google Speech Commands [War18], VoxCeleb [NCZ17].

Commercial products often use private datasets, which makes it difficult to evaluate them fairly. Furthermore, contemporary systems such as GPT-4 [Ope23] have the issue of using essentially all data available to them, via extensive automated web crawling. Thus, ensuring that the test dataset is independent and has not leaked into the training data poses a challenge. In the case of GPT-4, they validate using the company's internal codebases, as this is something they feel more confident would not have been included in the training dataset.

We can modify our dataset to make it better suited for a given deployment or to help the training process in some way. For example, techniques such as data augmentation, which has the effect of simulating having a larger dataset, and can help the model learn a more robust representation of our distribution. For image data, an example of this is applying transformations such as random rotations, crops, and flips. Alternatively, we may want to change our data in some way so that it can reduce the costs of our DNN solution. For example, we could make our images smaller, or switch from detecting full-color images to grayscale [XR19]. However, these techniques may have a negative impact on accuracy, but if we adjust the neural architectures accordingly we could potentially reduce the inference costs linearly as the images are scaled down, and by up to $3\times$ in the case of RGB-to-grayscale transformations [Pm+19].

Moreover, the ways in which we generate and process the data can be a significant contributor to the efficiency or success of the solution. Semi-supervised learning addresses the issue of limited labeled data, with a vast range of methods being employed [Yan+22]. For example, GPT-4 is trained on its own output [Ope23], generating critique of its generated text that is then used to improve the model further. A related problem for data generation is found in reinforcement learning, a problem domain where the model (or agent) must perform actions in an environment that maximize some goal. In this case, the agent must learn the dynamics of the environment given its actions. However, the cost of exploration can be very high. Thus, semi-supervised learning methods can be an approach to help learn the dynamics of the system, while reducing the cost of data that needs to be collected or labelled. For example, VPT [Bak+22] significantly improves the state-of-the-art for Minecraft agents by leveraging a dataset of unlabeled videos of gameplay, which is used to pre-train the model. This addresses two core problems: 1. The amount of quality labelled data is relatively small for the complexity of the task, and 2. allowing the system to learn all the dynamics from

active gameplay can be prohibitively expensive. Therefore, using this data the model requires less training in a real reinforcement learning environment.

Overall, there has been relatively limited exploration of this layer of DLAS compared to other layers, with most solutions taking the dataset as an immutable given. Moreover, in the research community, to fairly compare DNN solutions it is necessary to not change the learning task too much. However, given the wide range of datasets available, the possibility of training models using a mix of datasets (e.g., using transfer learning), and additional techniques listed above, this layer of DLAS is far from static.

## 3.2 Models & Neural Architectures

As mentioned in Section 2.2, there is a huge range of DNN layer types and architectures which we can leverage to solve a particular learning task [Vee16], with only a subset of them explored in this thesis. Section 3.2.1 gives a list of some significant DNN model architectures which have emerged over recent years. One notable aspect of DNN-based solutions to problems is that we need not consider the neural architecture as fixed. We can co-design our neural architecture using some other aspect of DLAS, for example, to fit the constraints of a given hardware platform. Architectures can designed by hand with this co-design perspective, however another approach involves Neural Architecture Search (NAS), a field of AutoML [HKV19] where we *automatically* search for a DNN design that meets certain criteria, such as accuracy or throughput. Although not directly explored in this thesis, we discuss some relevant NAS works in Section 3.2.2, since it fits into the broader themes of this work.

### 3.2.1 Significant Model Architectures

Section 2.2.4 highlighted two common classes of DNN architecture evaluated in this thesis: CNNs and Transformers. The relevant details of specific models will be discussed in the experimental setup of each chapter. This section highlights some popular DNN models that have been influential or have seen wide usage in research and industrial deployment. These include CNNs, Transformers, and more. We note relevant across-stack optimizations which the developers of a given model may have leveraged. Other popular architecture classes include graph neural networks (GNNs), generative adversarial networks (GANs) [Goo+20; Kar+20a], recurrent neural networks (RNNs), and diffusion models [Rom+22; Soh+15]. The distinction between different classes of architectures can be ill-defined, given that some architectures may share some attributes with other architecture types. For example, both GANs and diffusion models may still include convolutional layers.

The following lists some significant DNN architectures:

- LeNet-5 [Lec+98]: One of the earliest successful CNN architectures designed for handwritten digit recognition (MNIST [Den12]). It features three convolutional layers and two fully-connected layers.
- AlexNet [KSH12]: A breakthrough CNN that significantly outperformed previous state-of-the-art models for the ImageNet dataset [Den+09].
- VGG (Visual Geometry Group) Networks [SZ14]: A family of CNNs which increased their depth through small ($3\times3$) convolutional filters.
- Inception (GoogLeNet) [Sze+15]: A CNN which introduced the Inception module, which splits the network into parallel branches of layers (as introduced in Section 2.2.3). This allows the network to learn multiscale features by having different branches use varying filter sizes.
- ResNets [He+16]: A family of CNNs which introduced residual connections (skip connections), which reuses outputs from earlier layers later in the network. This process allows deeper networks to be trained.
- Capsule Networks [SFH17]: A DNN that introduces a novel structure called *capsules* to model hierarchical relationships between features, with the goal to improve robustness against viewpoint changes for computer vision tasks, for example, if an object is rotated. The novel structure was initially poorly supported by kernel libraries [BI19].
- MobileNets [How+17; San+18]: CNNs targeting edge-class hardware such as mobile phones. They leverage the depthwise separable convolution model optimization technique discussed in Section 2.3.3.
- DenseNets [Hua+17]: A family of CNNs that concatenates the outputs of multiple layers together via skip-connections, which increases feature reuse and thus accuracy.
- Transformer [Vas+17]: A novel architecture for NLP bringing the 'self-attention' mechanism, becoming the basis for a new family of Transformer architectures (e.g., BERT, GPT). See Section 2.2.3 for a brief explanation of self-attention.
- EfficientNets [TL19]: uses blocks discovered via NAS (see Section 3.2.2) with a novel scaling factor, enabling multiple sizes of model with increasing accuracy as the model size increases. Also leverages grouped convolutions as discussed in Section 2.3.3.
- BERT [Dev+19]: A Transformer-based architecture for NLP which uses context from both before and after a given token to better understand its meaning.
- MobileBERT [Sun+20b]: a compressed version of BERT using bottlenecking and knowledge distillation similar to Moonshine [CGS18] (see Sections 2.3.3 and 2.3.4).

- GPT (Generative Pre-trained Transformers) [Bro+20; Ope23; Rad+18; Rad+19]: a family of transformer models popular in a range of language tasks. The models are increasingly large scale, meaning that for the latest and largest version (GPT-4), model-specific tuning was prohibitively expensive. In this case, authors trained smaller models to predict the behavior of the full-scale model.
- YOLO (You Only Look Once) [Red+16; RF17; RF18]: a family of object detection models. The original YOLO unified the two tasks of classification and localization into a single architecture, targeting realtime inference, with newer versions further improving detection accuracy.
- BigGAN [BDS19]: A generative model based using a GAN architecture capable of producing high-quality, high-resolution images.

### 3.2.2 Neural Architecture Search

As introduced in Section 2.2.3, NAS automatically generates a neural architecture which aims to be accurate, as well as potentially targeting other metrics such as a low parameter count, or low inference times on a given hardware platform. The search space explored can vary, for example, NAS could vary the number or size of layers, which layer types to use, how layers are connected, or explore new types of layers. An appropriately designed search space is essential for NAS to successfully find a performant solution, and the cost of navigating this search space can be a critical bottleneck. For example, NASNet [Zop+18] generates blocks of layers, filters them using reinforcement learning, and stacks them together to form DNN models. However, they report that this process required over 3 days over 450 GPUs.

There have been some recent compiler-NAS works, which leverage techniques from the compiler community to reduce the size of the search space, or explore it more efficiently. This can reduce NAS search time, and produce more accurate and compact DNN architectures. For example, we can represent changes in the configuration of network layers as program transformations [TCO21], or in the case of $\alpha$NAS [JPR22] represent the search space as an abstract space of program properties.

Hardware-aware NAS looks to improve the NAS feedback loop by incorporating information about the target hardware platform into the search process. MnasNet [Tan+19] generates models for mobile devices, running inference of candidate models on the target hardware to evaluate their latency. FBNet [Wu+19a] outperforms models generated using MnasNet while reducing search costs by around 420$\times$ by using gradient-based methods to avoid evaluating every model architecture on real hardware. TinyNAS [Lin+20] reduces the size of the search space by representing the resource constraints of the target device, so models that cannot

fit are not evaluated. EdgeNAS [Luo+20] uses an accurate learning-based hardware latency estimator to reduce the costs of candidate architecture evaluation.

Weight-sharing based NAS methods, such as ENAS [Pha+18], Once-for-All [Cai+20], Convolutional Neural Fabrics [SV16], and SMASH [Bro+17], are particularly attractive for deployments across multiple heterogeneous hardware environments, as opposed to a single platform. The principle is that rather than searching for a specialized architecture and training from scratch for every deployment, they instead train a single large network with a structure from which smaller subnetworks can be generated. These subnetworks should ideally perform well on the target problem without training from scratch, or at most require a modest number of additional training epochs.

## 3.3 Model Optimizations

There is a wide range of model optimization techniques in the machine learning community, which can reduce the size and costs of increasingly large DNN architectures. This section provides an overview of some related works in the DNN optimization literature, in the areas of cheaper operations (Section 3.3.1), quantization (Section 3.3.2), and pruning (Section 3.3.3). For an overview of the wider context and field of model optimization techniques, beyond the topics directly touched upon by this thesis, we refer the reader to a variety of survey works on this rapidly developing area [Che+18c; Che+20; Cho+20b; Den+20; Sze+20]. Similar to the broader field of data compression, reducing the size of our DNN models reduces the representational capacity, and in-principle the maximum accuracy that it can achieve. However, as the Lottery Ticket Hypothesis identifies [FC19], most DNNs have a large amount of redundancy in their parameters, and thus an effective model optimization technique can achieve significant compression with very little accuracy penalty.

### 3.3.1 Cheaper Operations

It is widely recognized that many modern DNNs are over-parameterized [Han+15], which means that models are larger than they need to be. A focus on achieving state-of-the-art results has led to bloated network architectures with diminishing returns when new parameters are added [Hua+19]. For example, given that most of the energy consumption and execution time in CNNs is dedicated to convolution [LSC18], it is desirable to reduce convolutional over-parameterization for use in resource-constrained settings. As highlighted in Section 2.3.3, one popular method to exploit parameter redundancy is to split standard convolutions into *groups* along the channel dimension, and the approach has featured significantly in the network compression literature [CGS18; How+17; Hua+18; Ioa+17; San+18;

Tur+20]. As the number of groups is increased, the parameter cost of a grouped convolution decreases at the expense of representational capacity. In the extreme case where there are as many groups as convolutional channels we obtain *depthwise convolutions*. In MobileNets [How+17; San+18] for example, the authors replace standard convolutions with pairs of depthwise convolutions and pointwise ($1\times1$) convolutions, the latter of which allows for channel mixing to restore capacity. The technique is known as depthwise separable convolutions [Sif14]. Similarly, Moonshine [CGS18] takes the standard block used in ResNets [He+16] consisting of two standard convolutions, and replaces each convolution with a grouped and pointwise pair. The number of groups can be varied to trade off accuracy against the number of parameters.

For Transformers, much like with CNNs, there have been a number of similar compression techniques applied. Some Transformers include convolutional layers, and have used depthwise separable convolutions as a cheaper alternative [So+21]. Additionally, the concept behind depthwise separable convolutions has been adapted for the self-attention operation [Li+22c], along with other techniques, to reduce the cost of the attention mechanism by exploring alternative cheaper versions [Cho+23; Kat+20].

However, translating parameter and MAC reductions into better hardware performance remains difficult. Many frameworks transform convolution into GEMM in order to exploit pre-existing, highly optimized subroutines [Che+14b], which may not be composable with any dimensionality perturbations caused by the cheapening process. Although tensor compilers promise performance portability for custom convolutions [Bag+19; Che+18b; Vas+18; Ven+19], they have been shown to lack the generality required to adopt such radical neural architecture changes to a vast hardware landscape [BI19]. However, with the recent development of auto-schedulers, tensor compilers have been shown to provide optimized code generation for more novel operations [Zhe+20a].

Another approach for making operations cheaper is *temporal reuse*, where the DNN is passed multiple inputs in sequence, for example, frames of a video. Temporal reuse exploits the fact that in some cases there will be similarity between subsequent inputs, for example, in video there may only be small changes between frames. Therefore, we can cache and reuse computations from parts of previous inputs that are similar. Approaches to exploit temporal reuse may require additional specialized hardware, but with the potential for significant energy or inference time reductions [CSO22; Li+22b; RAG18].

### 3.3.2 Quantization

As discussed in Section 2.3.2, we can reduce the range of values which parameters or activations can take, for example, by using a data-type that has fewer bits than the typical

`float32`. Quantizing a network can result in a reduction in accuracy, however there is a range of approaches to mitigate this penalty, such as with post-training calibration discussed in Section 2.3.2; or quantization-aware training, where we emulate the impact of quantization during training to make the network more resistant to accuracy loss [Jac+18]. Quantization can offer advantages such as reduced memory footprint, reduced memory bandwidth requirements, and better energy efficiency.

Quantizing neural networks using lower-precision data-types was seen as far back as the 1990s [Bal+91; FCC90], with 8-bit weights being a popular level of compression [She+18; VSM11], and some works compressing further [HMD16]. The extreme is 1-bit quantized networks, often refereed to as *binary nets* or BNNs [CBD15; Hub+16]. Examples include XNOR-Net [Ras+16], with hardware implementations which exploit it including YodaNN [And+16] and BRein [And+17b]. Ternary quantization [Li+22a] can potentially improve the quality of BNNs, by adding an extra bit so we can represent $\pm 1$ and 0. Leveraging sparsity enables us to remove the overhead of this additional bit.

Another approach to quantization is `power2` quantization for `int`-like data types, which ensures that weights are limited to powers of two. Specifically, positive and negative versions of non-fractional powers, and zero, i.e., $\{\ldots, -2^2, -2^1, -2^0, 0, 2^0, 2^1, 2^2, \ldots\}$. `power2` quantization is notable, since it means that we can replace our multiplication operations with bit-shifts, since they are computationally equivalent in this case. If this is properly exploited by the hardware, bit-shifts can be significantly cheaper in terms of cycles or energy, thus making `power2` quantization an attractive technique for DNNs [GMG16; Lee+17; MLM16; Yao+22]. A bit-shift is when we move the bits representing a given datum to the left or the right by a given number of places, padding with 0s. For example, if we left shifted the nibble `0101` by one position, it would become `1010`, with the original bits underlined. This bit-shift is equivalent to multiplying the nibble by 2.

To fully leverage quantization the data-types should be supported natively by the underlying ISA, so that we can actually run our computations more efficiently, rather than converting our data to a supported type at execution time. Motivated by this, there are some emerging data-types specifically targeting DNNs which have been implemented in recent generations of hardware. For example, the `bfloat16` format (Brain Floating Point), found in some modern Intel processors, Google TPUs, and the ARMv8.6-A instruction set. The format uses 8 exponent bits, 7 mantissa bits, and 1 sign bit, and has the same dynamic range as `float32`. However, for MACs involving `bfloat16`, `float32` values are typically used for more accurate accumulation, similar to `int8` computations discussed in Section 2.6.3.

For training, higher precision may be needed, and `float32` remains the standard. Nvidia have adopted the `TensorFloat-32` type in some of their GPUs, which is claimed to accelerate training by up to $10\times$ compared to `float32`, while still achieving comparable

accuracy. Similar to `bfloat16`, 8 bits are used for the exponent, 1 for the sign, however the mantissa is expanded to 10 bits. In contrast to `bfloat16`, `TensorFloat-32` is used as the *accumulation type*, rather than the storage type. Both techniques take advantage of the fact that many neural network weights are small (predominantly in the range [-1, 1]), and thus we can dedicate more of the data-type's representational capacity to values closer to zero, and reducing precision for larger values.

A complementary approach to quantization is mixed precision, where we have different levels of quantization throughout the DNN. This can help us achieve a more optimized trade-off between accuracy and inference time. As a rule-of-thumb, earlier layers in the DNN require more bits to maintain accuracy on the target dataset than later layers, however developing more robust approaches to determining the ideal trade-offs between compression and accuracy using mixed-precision is an active area of research [CWC21; Lou+20; Uhl+20; Wan+19b]. In addition, there is a number of challenges for mixed precision quantization such as generating efficient code for each of the data-types [Zha+20]. As tensor compilers continue to mature, we may see reduced friction in adopting mixed precision techniques.

### 3.3.3 Pruning

As highlighted in Section 2.3.1, pruning is where we set some our DNN parameters to zero, with four key dimensions that distinguish pruning methods: *Structure*, *Scoring*, *Scheduling*, and *Fine-tuning*. For *Scoring*, we discussed the L1-norm method, which is commonly used and prunes parameters based on their absolute values. However, other methods exist such as adding 'importance coefficients' which are learned during training [Mol+19], or measuring a given parameter's contribution to the model's gradients or activations [LAT19]. *Scheduling* refers to how much we prune in each step, with iterative scheduling requiring fine-tuning after each step to maintain accuracy [Han+15; Tur+18a; ZG17]. However, fine-tuning after every iterative pruning step can be expensive. Therefore, *one-shot* methods look to prune and fine-tune in a single step, which requires more careful selection of the parameters to prune [Liu+19; SA20]. Some works have explored pruning during the first training run of a DNN, rather than pruning a pre-trained dense model [Bel+18; Evc+20].

A related issue to parameter pruning is ephemeral sparsity, such as activation sparsity. If we use activation functions such as ReLU, a non-trivial number of activations will be zero or close to it. However, this sparsity pattern will vary *per-input*, meaning that it will change dynamically depending on our input data. Thus, if we want to exploit activation sparsity, we will need to carefully co-design our algorithms, systems software, and potentially hardware to achieve any speedups. Dynamic sparsity also excludes many AOT compilation speedups, since the sparsity pattern changes for every input. We discuss some works which exploit activation sparsity in Section 3.6.2.

Finally, note that the field of pruning is large, and this section has only provided a brief overview of the relevant topics in the area. We refer to reader to several surveys on the topic for a broader overview of pruning [Bla+20; Hoe+21; Lia+21; Xu+20].

## 3.4 Algorithms & Data Formats

In this section, we discuss some popular or promising works in the literature regarding algorithmic primitives for DNNs, both dense (Section 3.4.1) and sparse (Section 3.4.2). We also discuss the inter-linked topic of data formats in Section 3.4.3.

### 3.4.1 Dense Algorithms

For convolutional algorithms, which are usually the most expensive in CNNs, a wide range of primitives are available to implement a given layer. Anderson and Gregg [AG18] characterize five main families of convolution algorithms.

- The *direct* family is characterized by a simple six-deep loop nest, as described in Section 2.4.2 and Algorithm 1. Direct convolution has been revisited in recent studies, including JIT-based recompilation [Geo+18] and alternative data layouts [ZFL18].
- The *im2* family is characterized by using the *im2col* transformation described in Algorithm 2, followed by a GEMM. This means that we can leverage the plethora of accelerated GEMM algorithms such as Strassen's algorithm [Str69], which has been explored in the context of DNNs [CX14]. However, there is a variety of accelerated GEMM algorithms for different scenarios.
- The *kn2* family is similar to the *im2* family since it uses GEMMs, however it does not construct the *im2col* matrix, which can reduces the memory overheads. However, an additional processing stage is required to combine the partial outputs [VAG17].
- The *Winograd convolution* family maps the data into the frequency domain [LG16]. This means that multiplications can be computed as additions. However, there are overheads in mapping and unmapping data into the required space. In addition, the algorithm has varying definitions depending on parameters such as filter size, and there may not be an implementation available for every configuration.
- The *fft* family use Fourier transforms to apply convolution, transforming the inputs and filters, and applying a pointwise multiplication [MHL14; Vas+14].

GEMM convolution, i.e., the *im2* family, is especially popular on GPUs since its is highly parallelizable, and there is a wide range of optimized kernel libraries which implement it, such as OpenBLAS [XQY12] and ATLAS [CPD01]. TVM's default approach to standard convolution on the CPU is an algorithm known as spatial packed convolution (SPC) [ZC18].

Note that in this thesis we have used the same algorithm for all layers of a given model, in a given evaluation. However, we could also vary the algorithm used per-layer, which could bring significant performance speedups [AG18; PPB19; Wen+19]. We might also expect some layer shapes and properties to be better suited to one algorithm or another on a given target platform, such that different algorithms are used for each layer in a given DNN. For example, for pointwise convolution, i.e., when the filter size is $1\times1$, *im2col* is the identity. Therefore, GEMM convolution is much cheaper to compute. However, for another layer, memory could be more constrained on the target platform, and an *im2* algorithm may be suboptimal, since the transformed inputs could use up too much memory. As we will discuss in the Section 3.4.3, we must also account for the fact that there may be transformation costs for passing data *between* algorithms which have their own data formats. Thus, we need to consider these interactions when choosing algorithms for adjacent DNN layers.

### 3.4.2 Sparse Algorithms

Sparse versions of many of the algorithms discussed in Section 3.4.1 exist or can be developed. For example, some works have found sparse direct convolution to be a fast algorithm on CPUs [Par+17b]; and sparse versions of Winograd convolution have been explored [LPT17]. However, the latter case requires additional co-design with techniques from the *Model Optimizations* layer of DLAS, since we are not directly pruning parameters, instead we are pruning the given parameter's representation in Winograd space.

Other works have focused on sparsity on GPUs, generally opting for sparse GEMM convolution [Gal+20]. Given GEMM is used extensively across scientific computing workloads, there is a wide range of algorithms supporting various cases where either one of the two matrices are sparse [ASA16; Koa+16], or both are [Gao+23; Gus78].

### 3.4.3 Data Formats

Although `NCHW` is the main dense data format explored in this thesis (for 4D image data), primarily in the interests of tractability in the face of a huge number of variables across DLAS, there is a range of other data formats for both 4D image data, and other classes of inputs. For example, there is `NHWC` and `CHWN` for 4D image data, or tile based approaches which split one or more of the 4 dimensions in some way; or for 5D video data we might add a *depth* dimension `D` to represent time. For tasks which are not computer vision, we may

not necessarily have names for the data format. Regardless, the format will be defined by how data is ordered in memory, and will still be an important consideration for accelerated deployment, potentially being better or worse suited for a given hardware platform.

The data layout can be tightly coupled with the algorithmic primitive used for a given layer, and like the algorithm, we do not need to keep the same data layout between layers. However, this may incur layout transformation costs, which should be considered when choosing a solution [AG18; PPB19; Wen+19]. Inference speed is an important factor to consider, however some data formats or conversion between them can incur memory overheads. Therefore, some approaches such as TASO [Wen+20][1] look at optimizing memory usage, with inference speed as a secondary objective.

There is a wide range of formats for representing sparse tensors, summarized by the TACO project [CKA18]. Much like dense data formats, there may be overheads in converting between sparse data formats at runtime [CKA20], and a number of tensor compiler tools are emerging [Bik+22; CKA20; Kjo+17; Ye+23] to help manage the complexity, heterogeneity, and range of sparse representations, algorithms, and data sizes.

## 3.5 Systems Software for DNNs

There is a wide-range of software which can be relevant to the performance of DNNs, such as kernel libraries, tensor compilers, and auto-tuning and auto-scheduling systems. This section discusses relevant works in this area, however does not cover all areas of software which may be used in DNN deployment. This includes software which may still influence performance, but is not yet a primary bottleneck or optimization front for DNNs such as operating systems, firmware, or networking protocols in the case of multi-device collaboration.

### 3.5.1 Kernel Libraries

There is a huge range of kernel libraries implementing optimized routines for DNNs targeting varying hardware platforms. Nvidia's cuDNN [Che+14b] is a popular library for Nvidia GPUs, while miOpen [Kha+19] and the Arm Compute Library [Arm17] are for Arm GPUs and CPUs respectively. oneDNN [Int20] provides cross-platform optimized kernels for CPUs and GPUs, as does the triNNity library, which implements over 70 different algorithmic variants for CNNs [And+17a]. Other kernel libraries that are worth highlighting include CLBlast [Nug18], OpenBLAS [XQY12], and SYCL-DNN [Tan+22].

When targeting common operations and data-sizes, kernel libraries can provide state-of-the-art performance on platforms that they support. However, as discussed in Section 2.5.4,

[1] Not to be confused with TASO (Tensor Algebra SuperOptimizer) [Jia+19].

kernel libraries' focus on common cases that can lead to significant performance degradation with workloads they have not been optimized for [BI19]. This can be a severe limitation in the quickly developing world of DNN research, as it increases the costs of exploring new operations. Recent kernel libraries have been looking more at performance portability and cross-platform support [Int20; Tan+22], and it is likely that the line between tensor compilers and kernel libraries will blur in the future. For example, Collage [Jeo+23] leverage's TVM's integration of kernel libraries to select the best backend for each layer. Often this is TVM's native code generator, however in some cases a kernel library implementation is preferable. This interoperability is partially enabled by the DLPack format [Chi+17], which is a standardized cross-platform data structure for N-dimensional arrays leveraged by several projects including TVM, PyTorch, TensorFlow, and NumPy. This reduces the overheads of switching between backends.

### 3.5.2 Tensor Compilers

As discussed in Section 2.5.5, tensor compilers can be a powerful tool for accelerating DNNs, and the position of this thesis is that they are playing an increasingly important role in managing the range of optimization choices available across DLAS. They exist at the intersection of many topics, since they can be extended to be responsible for a myriad of techniques across DLAS, such as managing sparsity, quantization the algorithms and data formats, generating and tuning efficient code, and managing the interfaces with hardware. TVM is the main tensor compiler used in this thesis, since it is relatively mature and has a variety of features which make it amenable to the studies undertaken, such as the ease of defining new algorithms, and state-of-the-art auto-tuning [Che+18a] and auto-scheduling infrastructure [Sha+22; Zhe+20a]. However, there is a range of other tensor compilers available, some of which pre-dated or were contemporary with TVM, and others which have emerged during the development of this thesis.

- Halide [Rag+17]: popularized the compute schedule programming paradigm (as discussed in Section 2.5.6), and served as one of the initial building-blocks of TVM. Its focus is on tensor programs for image processing (e.g., applying blurs and other filters), and thus does not provide all the domain-specific acceleration for DNNs available in TVM [Che+18b], such as support for models from multiple DNN frameworks, optimized DNN operator schedules, etc.

- TACO [Kjo+17]: an early tensor compiler which differentiated itself from contemporary kernel libraries by supporting a wider range of tensor programs while achieving competitive performance. It has support for dense and sparse computations [CKA18],

with a range of sparse data formats supported. It has recently added support for *auto-scheduling* sparse computations [AKA22].

- Tensor Comprehensions (TC) [Vas+18]: includes a domain-specific language for representing tensor program kernels, as well as a JIT-runtime. They use a polyhedral IR (see Section 3.5.5), as well as an auto-tuner. The project was formally archived in 2021, and had its last commit in 2019 (git hash `fd01443`).

- RISE/ELEVATE [Hag+20]: two well-defined functional languages for compute declaration and scheduling respectively. ELEVATE's schedule primitives are more composable than TVM's imperative programming style, and on some benchmarks outperforms TVM's handwritten schedules. However, the system is still a research compiler and is not currently production ready.

- Tiramisu [Bag+19]: a tensor compiler designed around the polyhedral model (see Section 3.5.5). It has four levels of IR and defines its own scheduling language to write optimizations. It also has its own auto-scheduling system [Bag+21], and supports dense and sparse computations. However, its auto-scheduling system has not been formally compared against Ansor, and only supports a limited number of backends (x86 CPUs, Nvidia GPUs, and Xilinx FPGAs).

- TensorRT [Nvi16]: designed by Nvidia specifically for generating optimized code for their GPUs. Its main focus is on graph-level optimizations, such as operator fusion, node scheduling, and efficient memory footprint management. It has native quantization support, to make it easier for developers to quantize their models.

- MLIR (Multi-Level IR) [Lat+21]: a compiler framework building upon the ideas of LLVM. It provides tools for defining domain-specific dialects, and translations (or *lowerings*) between dialects. Dialects represent program structure of varying levels of granularity and specificity, for example, the `linalg` dialect represents linear algebra operations such as matrix multiplications, whereas the `affine` dialect represents lower-level structures such as nested `for` loops. Therefore, MLIR is a compiler framework designed to enable the *development* of domain-specific compilers, providing a range of dialects and translations, with the goal of making it easier to define custom dialects that are relatively interoperable. The motivation for this is well-founded, as although there is a wide range of domain-specific compilers available (including tensor compilers), most are defined using their own bespoke IRs and toolchains, leading to significant fragmentation and repeated implementation of equivalent features. MLIR hopes to make interoperability and reuse between domain-specific compilers such as tensor compilers easier, and there are ongoing discussions in the TVM community to explore a transition to using MLIR as part of TVM's backend infrastructure.

- XLA [Tea17]: originally a compiler for TensorFlow, it has been adapted into an MLIR dialect featuring the HLO (high level optimizer) IR and a range of hardware-specific optimizations. Integrated as backend for the popular TensorFlow [Aba+16] DNN framework, as well as JAX [FJL18], it targets CPUs, GPUs, and TPUs. It includes auto-tuning support, and its design prioritizes large-scale inference, for example, large batch sizes of high-end hardware, and therefore has relatively poor performance at smaller scales [Li+21].

- IREE [The19]: built using MLIR, it is still in the early stages of development, with some initial results showing that it can generate very efficient runtimes for embedded CPUs [Liu+22]. It does not currently have the wealth of hand-optimized schedules that TVM has, nor any auto-tuning or auto-scheduling support. However, it has the advantage of existing in the MLIR ecosystem, and thus may benefit in the long term from the network effects of MLIR.

- PlaidML [ZB19]: a polyhedral-based tensor compiler from Intel, featuring an IR called Stripe. PlaidML represents programs using a nested polyhedral model, which means that it can hierarchically model different levels of abstraction for applying for optimizations, for example, device-level, DRAM-level, SRAM-level, or SIMD-level. The authors argue that this representation makes supporting niche hardware accelerators easier. The compiler has been transitioning to using MLIR, and has a embedded domain-specific language similar to TVM. By integrating with Keras, it can accelerate DNN training, a feature which is still under development in TVM.

- nGraph [Cyp+18]: focused on graph-level optimizations, such as how to partition graphs, nGraph uses kernel libraries for all layer implementations. It supports CPUs, Intel's Nervana hardware accelerators, and GPUs via integration with PlaidML.

- Glow [Rot+19]: uses instruction-based expressions to specify constant memory regions (e.g., inputs, weights, outputs), and locally allocated regions such as intermediate outputs and transformed inputs. This differentiates it from other tensor compilers in this list, which tend to use polyhedral or compute schedule programming models. The two classes of memory regions allows Glow to determine which transformations it can make. However, this programming model has been observed to make it harder to add new operators, and Glow is missing components which other tensor compilers include, such as thread parallelism support on CPUs and support for external kernel libraries [Li+21]. As a result, Glow's performance is not currently competitive with state-of-the-art tensor compilers [Li+21].

### Graph-level Optimizations

As discussed in Section 2.5.5, graph-level optimizations such as operator fusion can be an important optimization which tensor compilers can apply. Most systems, such as TVM and TensorRT use predefined rules to combine layers, for example, fusing batch normalization layers or element-wise operations such as activation functions. DLGR [Ma22] specifies a larger set of graph-level optimization rules than TVM, which developers can use to generate more complex graph-substitutions and potentially achieve higher speedups. However, by defining rules manually, these techniques may still lose out on potential speedups from graph-level optimizations not generated by the rules.

It has been shown that the task of optimizing DNN computation graphs is NP-hard [Fan+20], however this has not stopped more flexible graph-level optimization systems being developed which improve performance compared to using rigid transformation rules. Approaches such as FusionStitching [Lon+18; LYL19] construct a search space of possible fusion patterns, since we may have a wide range of choices to make; and TASO (Tensor Algebra SuperOptimizer) [Jia+19][2] automatically generates graph transformations, and uses a cost-based backtracking search with high correctness guarantees.

## 3.5.3 Auto-tuning & Auto-scheduling

Auto-tuning frameworks, especially for compute-schedule based systems like TVM, are a popular area of research. AutoTVM [Che+18a] takes hand-engineered schedules for operations and explores parameter tunings across a space defined by the schedule author, such as tiling sizes, unrolling factors, and others. However, it should be noted that AutoTVM and other parameter-based auto-tuners likely have lower maximum speedups than auto-schedulers such as Ansor, since pre-defined 'tuning-knobs' inherently constrain the search space, potentially excluding many performant schedules.

Methods to efficiently explore tuning search space vary, including approaches such as gradient boosting [CG16], genetic algorithms, and Chameleon [Ahn+20] which improves the search strategies of AutoTVM by leveraging reinforcement learning. To improve auto-tuning parameter selection for systems like AutoTVM, One-Shot Tuner [RPS22] leverages a pre-trained cost-model to reduce search costs. In ATF [Ras+21], the authors look at more efficient auto-tuning techniques with a focus on modeling interdependencies between tuning parameters. WBTuner (white-box tuner) [Lee+19] exploits the internal program states to achieve better tuning.

Regarding auto-scheduling, FlexTensor [Zhe+20b] is an auto-scheduling system similar to Ansor, although it relies on more handwritten templates, thus seeing worse performance on

[2] Not to be confused with TASO [Wen+20], an approach to select convolutional primitives.

some benchmarks. LIFT [SRD17] explores the use of rewrite rules on high level representations of programs to generate OpenCL code, although it does not explore its large search space as efficiently as Ansor. Tiramisu's auto-scheduler approaches auto-scheduling uses a DNN-based cost model [Bag+21] rather than Ansor's tree-boosting based model. In terms of performance, it has not yet been evaluated against Ansor.

The emerging *MetaSchedule* system [Sha+22] allows auto-scheduler search spaces to be modularly expanded to include niche hardware features such as tensor cores. As mentioned in Section 3.5.1, Collage [Jeo+23] tunes by choosing which backend to use for each DNN layer, for example, using TVM for some layers and cuDNN for others. A more extreme case of layer-partitioning is seen with Neurosurgeon, which allows collaborative processing between server and edge devices [Kan+17]. With Neurosurgeon, some layers of the DNN are executed on an edge device and others on the cloud. This approach can bring significant energy and inference time reductions compared to an all-cloud or all-edge approach.

From the wider compiler literature, beyond tensor compilers, MiCOMP [Ash+17] clusters LLVM optimizations into sub-sequences, and finds orderings for them, since as highlighted in Section 2.5.2 the ordering of optimization passes can have a significant performance impact. Like many tuners, MiCOMP uses latency information of candidate programs to inform optimization decisions. However, other helpful metrics can include hardware counters [Cav+07], instruction counts [Coo+05], and the polyhedral model which attempts to minimize the dependence distance of statements [Ben+10].

### 3.5.4 Re-use of Optimized Code

The main contribution of Chapter 6 is a technique called *transfer-tuning*, which reuses optimized schedules on new tensor programs. However, the reuse of bundles of optimizations has been explored in other works beyond tensor programs. For example, Martins et al. [Mar+16] looks at similarities between C functions to cluster them into groups and applies compiler passes based on group membership, representing programs using the DNA symbolic representation [SC10]. In transfer-tuning there is more domain-specific knowledge that we can leverage, since we are in the space of tensor programs with well-defined operations. CompilerGym [Cum+22] exposes LLVM compiler optimizations to reinforcement learning agents via the OpenAI Gym [Bro+16], a popular toolkit for reinforcement learning. Like Martins et al. [Mar+16], its focus is on more general purpose program optimization and does not exploit domain-specific knowledge of tensor programs.

### 3.5.5 Polyhedral Compilation

A significant part of the compiler community is focused in exploring compiler design from the perspective of the polyhedral model [Ben+10; VBC06]. This model represents programs as a series of constraints and attempts to optimize them for these constraints. Tensor compilers such as PlaidML [ZB19], TC [Vas+18], and Tiramisu [Bag+19] are based on the polyhedral model. Works such as PolyDL [Tav+21] use ideas from polyhedral compilation to generate efficient DNN kernels. Even systems such as Ansor take cues from polyhedral techniques, for example, by having common metrics such as reuse distance in its cost model; or recent techniques which use polyhedron scanning algorithms to significantly reduce the search space size for auto-scheduling convolutions, while still achieving high speedups [Tol+23].

An issue with many polyhedral-based systems comes from the complexity of modern hardware, where it can be difficult to represent an accurate model of the hardware being deployed to. It is for this reason that approaches which use real hardware trials to dynamically build a model have been so successful, for instance Ansor. However, it is clear that techniques from polyhedral compilation continue to be relevant, and further synthesis of polyhedral techniques into the tensor compiler space may unlock further performance gains.

### 3.5.6 Sparse Computation

Many tensor compilers support sparsity, with several emerging works. TVM has some support for sparsity, extended in Chapter 4, and has a new emerging sparsity abstraction called SparseTIR [Ye+23]. MLIR has a sparsity focused dialect [Bik+22], which should make supporting sparse computations easier in MLIR-based compilers such as IREE. The TACO compiler has support for a range of sparse data formats [CKA18], as does Tiramisu [Bag+19].

## 3.6 Hardware for DNNs

As DNNs increasingly become a key component of a range of applications, the motivation to develop hardware which better supports it grows, especially as Moore's law and Dennard scaling [Den+74] slows, meaning that improvements to hardware must become more targeted [HP18]. Approaches to improve hardware for DNNs include adding DNN-targeted extensions to existing general purpose hardware; creating DNN specific accelerator devices; or to better support heterogeneity in DNN architectures, developing *reconfigurable* DNN accelerators. The following sections discuss some key works across these topics.

### 3.6.1 General Purpose Hardware

A range of GPU and CPU hardware extensions have emerged in recent years to make DNN processing more efficient on them. For example, on Nvidia GPUs *tensor cores* were first introduced in the Volta architecture [Mar+18] and operate on the unit of sub-tensors (e.g., 4×4 GEMMs), rather than computing over individual values. Tensor core extensions are applicable to workloads beyond DNNs, since GEMMs are used in a wide-range of applications, however their main focus is on DNNs, with Nvidia even leveraging them in consumer products, running *Deep Learning Super Sampling (DLSS)* in tensor cores to increase the quality of video game graphics workloads. Future AMD GPUs will include a Wave Matrix Multiply-Accumulate (WMMA) instruction, which is documented to support 16×16×16 tensors using `float16` and `bfloat16` instructions. `bfloat16` has been highlighted in Section 3.3.2, and is a DNN-specific datatype. As well as being added to GPUs, `bfloat16` is also being added to CPUs, such as newer Intel processors. On Intel processors, they are specifically targeted by large SIMD instructions, with the `AVX-512_BF16` extension which allows up to 32 `bfloat16` to be used in a single operand. In addition, certain CPUs are being developed with support for even smaller bitwidths [Ask+23; Ott+20].

To better understand the behavior and interactions of other layers of DLAS with the hardware, it may be valuable to explore hardware counters which give statistics such as cache misses, core utilization, and more. Apache TVM has support for the PAPI library [Ter+10], which provides an extensible interface to a variety of performance counters across device classes. This can help developers understand the bottlenecks in their designs, and so that they can better target their optimizations.

Like many other layers of DLAS, we can split the components hardware layer into multiple sub-layers, which may be helpful for better understanding or improving our workload's performance. For example, the system's memory hierarchy can be a topic of study on its own, and more intelligently exploiting it can lead to significant performance improvements. If we have a heterogeneous non-hierarchical memory system, a critical question for systems designers is which memory type different data should be placed. The fastest memory is typically limited in size, and some emerging memory technologies have varying specialisms, such as reads being much faster than writes. There are some works exploring this question for dynamic scientific workloads [PB14], although for conventional DNN processing much of the memory usage can be known and planned ahead-of-time. However, some workloads may require runtime object placement heuristics, such as DNNs with more dynamic behavior, e.g., neural networks that exploit activation sparsity or graph neural networks. Processing in memory (PIM) [Asi+23] devices allow us to run some computations without moving data to a central processing unit, which exposes a large set of performance trade-offs to explore.

### 3.6.2 DNN Accelerators

A wide-range of fixed function accelerators for DNNs have seen commercial success, such as the TPU [Jou+17], Apple Neural Engine, and Qualcomm Cloud AI 100; as well as a wide range of research architectures, such as DaDianNao [Che+14a], ShiDianNao [Du+15], Origami [CB17], and more [Har+21; Har+23; Mor+19; Xi+20]. These accelerators may target a single type of DNN layer such as convolutional or fully-connected, a mix of layers, or run the whole DNN.

However, it is important to consider that there are limits in the possible improvements that any given style of accelerator can achieve [FW19]. These limits and how to avoid hitting them too early should be acknowledged by chip designers. Some constraints include lower limits in CMOS size, and upper limits of chip area for different technologies and budgets. However, with the optimization fronts of deep learning workloads being across-stack, design stagnation due to these limits is unlikely to occur for several generations of devices.

#### Reconfigurable Hardware

An emerging topic is reconfigurable hardware, which includes fully reconfigurable fabrics such as FPGAs (which is often used as a development platform for accelerators), as well as ASICs for DNNs which may have some level of reconfigurability, for example, via programmable network-on-chips (NoCs) which allow programmatically controlled alterations to the dataflow within the accelerator. This reconfiguration can be performed to better suit a particular DNN or DNN layer, achieving better inference time or energy consumption.

There is a variety of reconfigurable FPGA designs which target specific DNN architectures, such as FINN [Umu+17], VTA [Mor+19], DNNBuilder [Zha+18], FP-DNN [Gua+17], and others [Wei+17]. Some approaches co-design the FPGA accelerator design and DNN architecture, using a form of hardware-aware NAS [Don+21b; Hao+19; Jia+20; Li+20b].

For reconfigurable ASIC designs, examples include Eyeriss V1 [Che+17] and V2 [Che+19], FlexFlow [Lu+17a], and MAERI [KSK18]. These differ from a full FPGA fabric by limiting what components of the accelerator can be reconfigured, for example, we might only allow the data-flow to be changed keeping the processing elements unchanged. This can allow much higher clock speeds and energy efficiency when compared to FPGA designs. STONNE [Muñ+21] is a cycle-accurate simulator for reconfigurable DNN accelerators, with Bifrost [SGC22] as an interface to STONNE via TVM (see Section 3.7.4). Since reconfigurable DNN accelerators are still maturing as a technology, this can be valuable for researchers who want to more efficiently explore the design space.

### Sparsity-aware Hardware

There are a number of hardware accelerator designs which exploit sparsity. From the perspective of sparse weights, accelerators include Eyeriss V1 [Che+17] and V2 [Che+19] (which also support dense computations), Cambricon-X [Zha+16], and SIGMA [Qin+20] (which can also be used for training). EyerissV2 also supports sparse activations in addition to sparse weights, as do SparTen [Gon+19] and SCNN [Par+17a]. Accelerator designs which focus on sparse activations exclusively include Cnvlutin [Alb+16], cDMA [Rhu+18], and Spartan [Don+21a] (specifically for training).

### Simulators

Software simulation of hardware is a common practice in hardware design, as it allows cheaper prototyping and can provide insights into the internal performance of a design that is not available on real hardware, such as the internal state of otherwise opaque subcomponents. SCALE-Sim [Sam+19; Sam+20] is a cycle accurate simulator for systolic arrays. As highlighted earlier, STONNE [Muñ+21] is a cycle-accurate simulator for reconfigurable DNN accelerators. SECDA [Har+21; Har+23] is an FPGA design methodology that uses SystemC simulation, described more in Section 3.7.3. SMAUG is a DNN accelerator simulator [Xi+20] which uses gem5-Aladdin [Sha+16] to simulate the full system, including the accelerator and the host CPU and its memory system. This gives high fidelity information about the performance, however comes at high evaluation costs. More specifically, it can take several hours to evaluate a single image with ResNet50 [He+16]. MGPUSim [Sun+19] is a high-performance simulator for multiple GPUs, and can be used to model the performance of DNNs and other workloads, especially when exploring the impact of micro-architectural changes to the GPU. Written in Golang, it is a relatively efficient simulator running most simulated processes asynchronously across many CPU threads. This reduces the impact of one of the main disadvantages of DSE using simulators, namely the high evaluation time.

## 3.7 Complementary Publications

The following chapters discuss core contributions of this thesis, including a characterization of DLAS (Chapter 4), an algorithmic and compiler based approach to accelerate an underserved DNN compression technique (Chapter 5), and a novel approach to reduce the search costs associated with tensor compiler auto-scheduling (Chapter 6). However, other complementary works have been produced during the development of these core contributions, which exist in the wider context of DLAS, but do not constitute contributions of this thesis.

### 3.7.1 Orpheus DNN Inference Framework

Orpheus was a DNN inference research framework [GC20] created to make it easier to explore techniques across DLAS, and to tackle the challenges described in Section 1.2. Its goal was to reduce the codebase complexity relative to production quality DNN frameworks such as TensorFlow and PyTorch, which also had to support training. By reducing the complexity, the aim was to make it easier to prototype novel DNN acceleration techniques. Written in C++, Orpheus takes ONNX models as input and only implements inference. Orpheus was able to achieve better inference times for some DNN models compared to other frameworks, such as ResNet18, ResNet50 [He+16], and InceptionV3 [Sze+16]. It implements a range of algorithms for convolutional and fully-connected layers, along with optimized AVX and Neon SIMD-intrinsic implementations of these kernels.

As the upper limits of easily achievable performance were reached, Orpheus added a JIT compilation system to unlock further acceleration. However, this made it clear that the most promising optimization fronts in DNN acceleration were from a compiler perspective, something which the emerging TVM tensor compiler excelled at. Thus, development on Orpheus was put on-hold indefinitely due to this conclusion, and the challenges and significant research contributions of this thesis were approached using Apache TVM.

### 3.7.2 Productive Reproducible Workflows for DNNs

The position of this thesis is that tensor compilers can improve productivity and the ease of design space exploration for developing and accelerating DNN-based applications. However, there is other supporting infrastructure required for developers to effectively develop their solutions. While scaleable production quality tools of this type are readily available, their use within the research community is not widespread, suggesting researchers may be missing out on potential productivity gains.

Motivated by this, our work ([GC22]) presents a case study producing an end-to-end AI application for industrial defect detection. The paper describes the high level deep learning libraries, containerized workflows, continuous integration/deployment pipelines, and open source code templates leveraged during the case study's development. It also discusses the value which each of these supporting tool brought to the development process.

The final output of the case study was 62 DNN models for defect detection, the best of which achieved competitive results, matching the performance of other ranked solutions on the three target datasets (DAGM2007 [WSS16], KolektorSDD [Tab+19], and KolektorSDD2 [BTS21]). One of the core enabling tools was the Bonseyes AI Asset template system [Lle+17], shown in Figure 3.1. AI Assets encapsulate all the code and dependencies

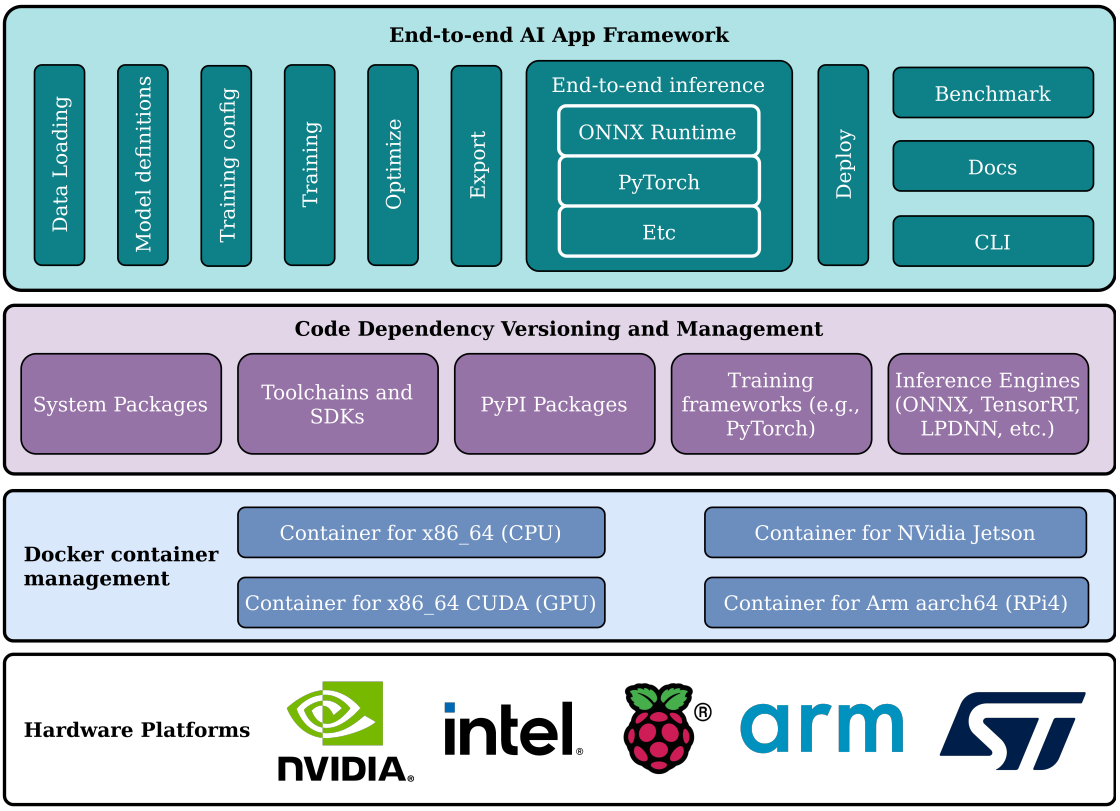



Figure 3.1: Simplified representation of the features provided by the Bonseyes AI Asset template system, adapted (with permission) from the AI Asset Generator documentation [Ass22].

required to develop an AI solution, providing pre-defined code for many common activities, such as benchmarking, report generation, model conversion, and inference.

The work highlights the value that exploiting such tools and systems can bring, even for research. It details how the solutions were developed, and presents the best results in terms of accuracy and inference time on a server class GPU, as well as inference times on a server class CPU, and a Raspberry Pi 4 CPU.

The work does not directly leverage tensor compilers, however the infrastructure and workflows it describes were used to help develop some of the solutions in the thesis. By reducing the overheads in setting up and developing the AI application, the workflows described in the paper can free up developer time for exploring more DNN acceleration options, and thus contributes to tackling the efficient DSE challenge described in Section 1.2.3.

### 3.7.3 SECDA: Efficient Hardware/Software Co-Design of FPGA-based DNN Accelerators

SystemC Enabled Co-design of DNN Accelerators (SECDA) [Har+21], and its follow-up SECDA-TFLite [Har+23], introduced a hardware-software co-design methodology and soft-

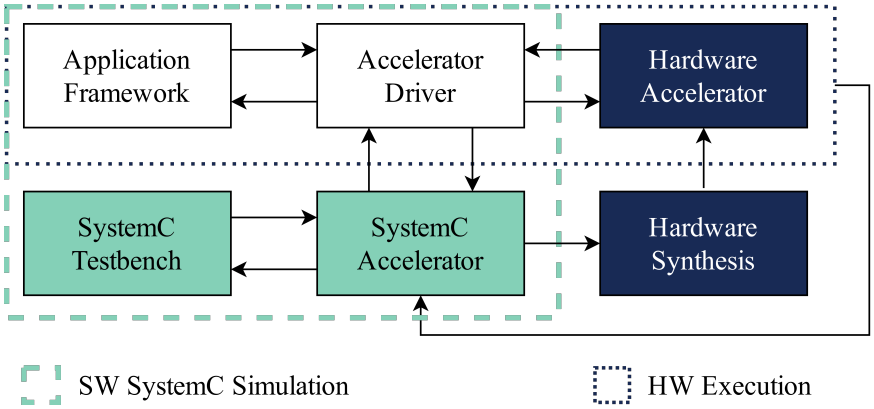



Figure 3.2: The SECDA methodology, figure obtained from the original publications [Har+21; Har+23]. Components in the dashed lines correspond to the design in simulation and components in the dotted lines correspond to the design running on real hardware. The *Application Framework* and *Accelerator Driver* software are common to both.

ware toolkit respectively for creating DNN accelerators using FPGAs. The methodology is shown in Figure 3.2, with the core idea being to reduce the costs associated with iterative design for accelerators. By leveraging SystemC [IEE12], a C++ library for system and hardware design, SECDA helps developers design their hardware using a low-cost SystemC-enabled simulation. When they have sufficiently optimized their hardware design, they can synthesize the design using an FPGA using the same code.

This 'single source' paradigm is enabled by SystemC, which can be compiled to both a simulation, or into a real hardware implementation on an FPGA, depending on the compilation configuration. Although hardware synthesis has a relatively higher cost (many minutes, compared to seconds for simulation), this allows designers to identify data-transfer overheads and other bottlenecks not evident in simulation. Specifically, these overheads can come from the Accelerator Driver code, which is responsible from transferring data from the host CPU to the accelerator. As Figure 3.2 shows, the Application Framework (e.g., a DNN library) and the Accelerator Driver are common between simulation and hardware execution, which also reduces the development costs of comparing simulated and real hardware execution.

SECDA is effective at tackling the DSE challenge described in Section 1.2.3 from the perspective of hardware-software co-design. SECDA-TFLite applies the SECDA methodology to develop an open source software toolkit within TensorFlow Lite, providing accelerator designs for CNNs and Transformers. By reducing the cost of the design loop for hardware design, SECDA tackles the efficient design space exploration challenge in Section 1.2.3.

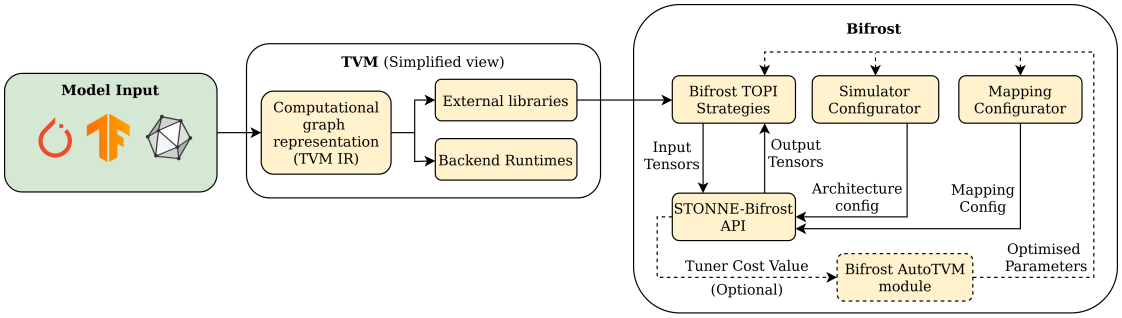



Figure 3.3: High-level overview of Bifrost's design, adapted from the original publication [SGC22]. Simulation and mapping configuration can be performed manually, or optionally auto-tuned using information from simulated execution to optimize the parameters.

### 3.7.4 Bifrost: Tensor Compiler Integration with Reconfigurable DNN Accelerators

Bifrost [SGC22] is a software tool which integrates the STONNE simulator [Muñ+21] with TVM, which allows DNNs to be more easily evaluated in simulated reconfigurable DNN accelerators. Previously, the process of evaluating a new model in STONNE was a manual and error-prone process, which could only import the model from PyTorch. In addition, the configuration of the accelerator had to be written by hand, which increased the costs of evaluating a range of configurations.

As shown in Figure 3.3, to help tackle these weaknesses, Bifrost leverages TVM's model loading infrastructure to make it easier to import DNNs from other DNN frameworks. This approach also allows more models to be supported, because DNN layers which are not supported by the chosen STONNE accelerator can be executed in TVM using other backends such as the CPU. Bifrost also defines utilities for configuring STONNE accelerators, including validation of parameters. It also integrates AutoTVM to find the mapping for some reconfigurable accelerators automatically. By combining the insights available from using a cycle-accurate hardware simulator, with the flexibility and tuning capabilities of a tensor compiler like TVM, Bifrost helps exploit across-stack interactions, contributing to further solutions to the challenge in Section 1.2.2.

### 3.7.5 Assessing Robustness of DNN Models

These two works [Lou+22; Lou+23] explore DNN deployment from a software testing perspective, which is increasingly necessary as DNNs are seeing increased usage in safety-critical systems, such as autonomous vehicles. Understanding the types of errors that can occur when deploying DNNs can help developers anticipate them, and tool developers mitigate them. Figure 3.4 gives an overview of possible sources of DNNs errors, with much

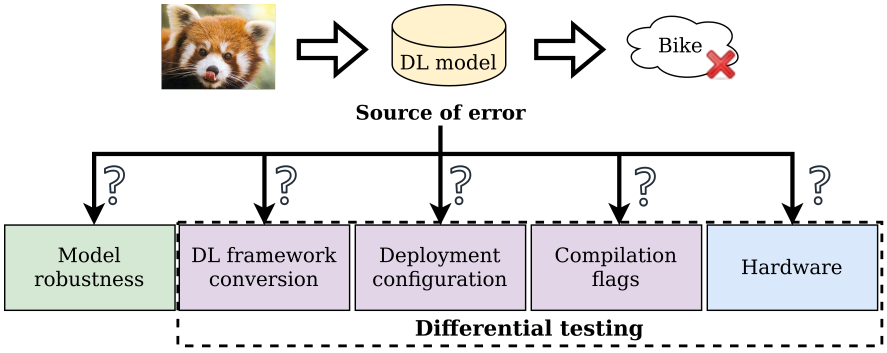



Figure 3.4: Motivating figure showing the potential sources of error in DNN deployment scenarios. Note that many works look at the problem for only a machine learning perspective, i.e., is the model robust to noise. However, there are many other factors that could introduce errors, especially from a systems perspective.

of the existing research exploring model robustness, for example, against adversarial inputs [Cha+18]. These works take a more systems-oriented approach, exploring the impact of changing the software and hardware environment of the DNNs. They use TVM as an evaluation framework, since it can import DNNs from most popular DNN frameworks, allowing a more easily controlled environment. TVM also allows the generation of code for a range of hardware platforms, exposes a range of debugging tools, and gives the user fine-grained control of code generation configuration, such as the code optimizations applied.

The first part of the work [Lou+22] found that definitions of the *same DNN architecture* from different DNN frameworks could have significant differences both in terms of output, and inference time. A followup work [Lou+23] found that the conversion process between DNN frameworks could introduce errors which could change the output labels in some cases. By developing a rigorous testing framework, the works helped identify robustness issues, either in terms of inference time or accuracy. Many of these problems were due to unexpected software errors, the mitigation of which may also help with the challenge described in Section 1.2.1, since when they were corrected, performance can improve.

### 3.7.6 ICE-Pick: Iterative Cost-Efficient Pruning for DNNs

As discussed in Sections 2.3.1 and 3.3.3, pruning can significantly reduce the number of parameters used in a DNN, and if sparsity is exploited, this can also reduce the inference time significantly. However, pruning can suffer from the challenge in Section 1.2.3, i.e., DSE can be prohibitively high. One of the largest contributors to these search costs is the fine-tuning time required to recover lost accuracy. Therefore, ICE-Pick [HGC23] proposes a technique to significantly reduce the amount of fine-tuning required when pruning, while still ensuring that accuracy is maintained.

The core idea of ICE-Pick borrows some concepts from transfer-learning, where we ‘freeze’

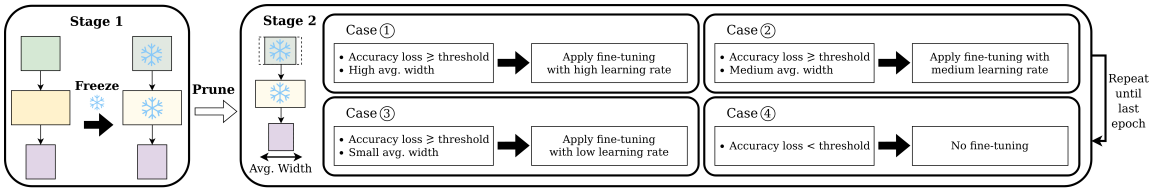



Figure 3.5: Overview of ICE-Pick, figure taken from original paper. First, less sensitive layers are frozen (Stage 1), then for each layer we prune, then fine-tune the model (Stage 2). The learning rate is adjusted dynamically, and fine-tuning is halted if our accuracy loss is lower than our threshold. This early halting, coupled with layer freezing, can save significant tuning time.

layers of the DNN during fine-tuning if they are unlikely to change much. This can reduce training costs, since we do not need to calculate all the gradients. In addition, ICE-Pick uses an 'accuracy loss threshold' during fine-tuning, where for a given fine-tuning step, if the accuracy loss is below a certain value, then fine-tuning stops early. These two approaches, when combined, can save up to 87.6% of the overall pruning time while maintaining accuracy. Figure 3.5 gives an overview of the technique, which shows how a single pruning step is performed using the ICE-Pick technique. In Stage 1, we identify and freeze less sensitive layers in the DNN, then in Stage 2 we iteratively fine-tune the model dynamically adjusting the learning rate as required. There are four cases where the learning rate is adjusted, with varying adjustments being made according to the level of pruning and accuracy loss. In the fourth case, when the accuracy loss is sufficiently low fine-tuning stops early, which can further reduce the overall time required.

## 3.8 Summary

In this chapter we gave an overview of the relevant techniques of DNN acceleration across DLAS. The pace of innovation is significant, as deep learning continues to garner interest as a solution to a wide range of problems. As discussed, there is already a range of techniques which look at tackling the key challenges identified by this thesis. However, given the size of DLAS, there is still significant scope for further research. In addition, it is still intractable to tackle the full DLAS as a holistic optimization space. The next chapter presents a case study choosing a variety of techniques from DLAS to demonstrate the importance of across-stack interactions. Further chapters tackle more targeted problems in DLAS.

# 4 Exploring the Deep Learning Acceleration Stack

In Section 1.1, we introduced DLAS, as a conceptual view of the relevant domains which are critical for accelerated DNN deployment. In this chapter, we further motivate the need for DLAS with a study into the impact of varying a small number of parameters at each layer. These parameters include two datasets, four models, three optimization techniques, three algorithms, two compilation techniques, and four hardware devices. To the best of our knowledge, this is the first study which examines a full vertical slice of DLAS. Other studies have focused on fewer layers of DLAS, with a wider evaluation of each layer. For example: models, DNN frameworks, and hardware platforms [Had+19]; datasets, models, and compression techniques [Bla+20]; and models, tensor compilers, and hardware platforms [Li+21]. Due to the limited range of parameters evaluated at each layer of DLAS, the study itself does not uncover significant new observations about the behavior of particular DLAS techniques. Therefore, the details of this study are given in Appendix A, with this chapter discussing how it highlights and motivates the challenges identified in Section 1.2.

## 4.1 Discussion

To realize an across-stack evaluation of DLAS requires significant engineering effort to allow a unified and consistent exploration framework, that fairly explores varying combinations of DLAS parameters. The particular parameters evaluated are listed below, and how they were implemented are given in more detail in Appendix A.1, with some of the most significant challenges discussed in Section 4.1.1.

- *Datasets & Problem Spaces*: CIFAR-10 [Kri09] and ImageNet [Den+09].
- *Models & Neural Architectures*: MobileNet (V1 [How+17] and V2 [How+17]), VGG-16 [SZ14], ResNet18 and ResNet50 [He+16], DenseNet161 [Hua+17], and EfficientNetB0 [TL19].

- *Model Optimizations*: Two methods of quantization (`int8` and `float16`), two methods of pruning - channel-wise structured pruning, and unstructured pruning (which we call 'weight pruning').
- *Algorithms & Data Formats*: Direct, general matrix multiplication (GEMM), and spatial pack convolution algorithms, both sparse and dense implementations.
- *Systems Software*: Apache TVM as the main evaluation framework, with both untuned and tuned (using Ansor [Zhe+20a]) variants of each model configuration.
- *Hardware*: CPUs: Intel i7, Arm Cortex-A73; GPUs: Nvidia AGX Xavier, Arm Mali-G72 (more details given in Table A.1).

Overall, the study required extending the Apache TVM compiler with new algorithmic primitives, adding new functionality to its limited sparse computation support, and fixing various bugs. In addition, pipelines for training, pruning and fine-tuning, deploying and evaluating, as well as auto-tuning the models on a range of hardware platforms had to be developed. This highlights a sub-problem of the efficient DSE challenge (Section 1.2.3), namely that software tooling is still maturing for DNN deployment, such that exploring varying combinations of DLAS parameters may require additional implementation by the user before the evaluation can take place. Naturally, this increases the cost of DSE. Similarly, despite the collection of inference time results for nearly 800 unique DNN deployment configurations, making definitive and generalized claims about the behavior of different layers of DLAS is difficult. This is because only a small number of parameters at each layer of DLAS were chosen. Section 4.1.2 discusses some of these issues, and what we can infer from the results.

### 4.1.1 Barriers to Evaluation

There were a number of technical barriers which made providing a full evaluation of DLAS prohibitively expensive in terms of engineering effort. The study endeavored to choose common acceleration techniques from across DLAS, and despite this many of these techniques were not available in the core TVM software stack. For example, for implementing algorithmic primitives for the convolutional layers, TVM only provided implementations for direct and spatial pack convolution, meaning that to realize GEMM convolution, a new implementation of im2col had to be created within TVM's embedded domain specific language. Note that TVM was chosen specifically because compared to other frameworks, it had the widest support for the DLAS parameters of interest, and relatively lower costs for adding parameters which were not supported.

Similarly, for sparse versions of these algorithms, TVM initially only supported $3\times3$ convolutions with a stride of one. Therefore, custom sparse implementations of each of the

three algorithms had to be written, and TVM's code generator was adapted to support sparse convolutions of any size and shape. It should be noted that this support only extends to a single sparse data format, namely CSR, described in Section 2.4.3. If we wanted to support additional sparse data formats, for example, coordinate list, block sparse row, or compressed sparse column, then we would need further extend TVM, as well as re-implement each of our sparse algorithms. To tackle the challenge of code generation for varying sparse data formats, the MLIR sparse dialect [Bik+22] was developed, which reduces the costs of exploration. However, this is only available in the context of MLIR, which is not interoperable with TVM, and currently has no DNN evaluation framework which is sufficiently mature enough allow an across-stack DSE such as the study undertaken in this chapter, at least not as easily as with TVM.

For quantization, not all DNN models using `int8` worked correctly. For example, for EfficientNetB0 TVM assumed that for a quantized multiplication operation only the left-hand operand would be pre-quantized. However, the structure of EfficientNet violated this assumption, which necessitated a bug-fix which we pushed upstream[1]. For DenseNet161, another bug stopped `int8` inference from being achieved, due to there being a mix of datatypes for a concatenation operation[2], but we were unable to fix this issue. From an accuracy perspective, EfficientNetB0 had the highest drops for `int8` quantization out of any model. We understood this to be due to EfficientNetB0's architecture not being amenable to post-training quantization. These issues were highlighted and corrected by EfficientNet-Lite [Liu20], which removes the squeeze-and-excitation networks, and replaces the swish activation functions with ReLU6 activations [How+17] (Figure 2.4d).

For evaluating auto-scheduling, there were several issues that increased the difficulty of evaluating all of the parameter configurations, and in some cases these issues meant that some results could not be collected. For instance, the cost of tuning is very high, and for the largest models on the most constrained hardware platforms, this was prohibitively expensive. On the HiKey platform, to tune each configuration of the large models required over 140 hours. With fifteen configurations for each model, this made collecting auto-scheduled results for these models on the HiKey platform impractical.

Another auto-scheduling issue was for sparse computations, which was not fully supported by TVM. Normally, TVM can generate the auto-schedule for a given computation automatically, with no addition setup steps required by the user. However, for auto-scheduled computations, TVM could not do this, and development of so called 'sparse sketch rules' was required for each of our sparse algorithms to enable auto-scheduling. Sparse sketch rules manually define the starting point for the auto-scheduler, rather than this starting point

[1] `https://github.com/apache/tvm/pull/14286`

[2] `https://discuss.tvm.apache.org/t/bug-qnn-type-mismatch-in-broadcastrel-for-8-bit-quantized-model/14518`

being inferred automatically. The sparse sketch rule solution worked for the CPU, however on GPUs TVM was unable to support auto-scheduling of sparse computations in the versions of TVM we evaluated. This is because the auto-scheduler has two conflicting requirements:

1. cross-thread reduction, requiring the partial sums (such as those used in convolutions) to be computed across multiple GPU threads simultaneously.

2. loops parallelized over threads which request a static number of threads.

Both of these conditions cannot be satisfied, since the size of our reduction loop for our algorithms varies depending on how many non-sparse elements there are in a given portion of the computation. Thus, we could not tune pruned models on the GPU in our evaluation, since it cannot be easily supported by TVM without significant changes to its core design.

A final issue we experienced with the auto-scheduler was on the AGX Xavier GPU platform, where the auto-scheduler did not produce any speedups. This problem was investigated extensively, including debug tracing, and remote auto-scheduling from other platforms. We hypothesize that the version of the CUDA libraries provided on the AGX Xavier GPU had a silent error which neither TVM or our own investigation were able to identify.

### 4.1.2 Observations and Caveats

In our evaluation (Appendix A.2) we observed many variations across the the experimental configurations. For example, the best algorithmic primitive in terms of inference time varied not only by hardware platform, but also by DNN model, and compression technique. This can be seen in Table A.3, where for instance, on the HiKey CPU, GEMM convolution is the fastest algorithm for VGG-16, however when it is quantized to `int8`, direct convolution is fastest, and the fastest configuration for this model and hardware platform overall. This can be compared to MobileNetV2, where GEMM convolution is the fastest algorithm for `int8`, however GEMM convolution using the weight pruning compression technique was the fastest overall. Since the evaluation is deep, but not wide[3], we cannot make any generalized observations about the behavior of a given DLAS technique, for example, which algorithmic primitive gives the lowest inference time on a given platform in the majority of cases. In fact, the lack of the ability to make generalized claims highlights the core thesis of this chapter: across-stack interactions of machine learning and systems optimizations can be non-trivial, hard to understand without significant data gathering (which can be prohibitively expensive, see the DSE challenge in Section 1.2.3), and bringing in additional features may significantly accelerate or impede a given technique in unexpected ways. This

[3]Meaning that we evaluate of the layers of DLAS (depth), but only a limited number of parameters at each layer (width).

is evidenced by the size of the study in Appendix A, with over 800 unique deployment configurations, with some configurations requiring dozens of hours to evaluate (as discussed in Section 4.1.1). Despite the ambitious size of the study, we do not include every popular technique such as Winograd convolution, and Transformers, since the combinatorial increase would make the study impractical to undertake.

In addition, even within our small set of DLAS parameters, there were simplifying assumptions which we made which arguable reduced the utilization of the chosen hardware platforms. The details of these hardware platforms are described in Appendix A.1.5, but include a Arm CPU, an Intel CPU, an Nvidia edge GPU, and an Arm edge GPU. Aspects of the hardware that could be better utilized include the big.LITTLE architecture (introduced in Section 2.6.3) of the Arm CPU, since we only leveraged the big cores, hyper-threading on Intel CPU, since we ran one thread per core, or leveraging both the CPU and GPU in parallel for the Nvidia and Arm platforms, since we only used one device for the computation in each experiment. However, across-stack optimizations would be required to exploit these features properly, otherwise it is likely that they would introduce a performance penalty [LCO18; Wan+20]. We also did not leverage the Nvidia GPU's 64 tensor cores in addition to its 512 general purpose CUDA cores. Though TVM supports tensor cores, it requires manual schedule re-design for each algorithm. MetaSchedule [Sha+22] can expand auto-scheduler search spaces to include hardware features such as tensor cores, which could allow us to more easily investigate this dimension, but this would have required additional engineering integration effort in an already ambitious evaluation.

Our experimental methodology makes assumptions about the metrics of interest, and if we varied the way data is collected, it could change the inference times significantly. For the experiments, we used a batch size of 1, took the median of 150 runs, and disregarded the first warm-up run. Although this is a common deployment and evaluation scenario, it is important to be aware that this is not the only one, and experimental design should reflect which deployment case is being considered when evaluating models [Tan+13; Wu+19b]. For instance, for edge deployment we may expect the batch size to be small, whereas on the cloud it may be large. Increased batch sizes mean increased memory requirements and inference latency, but also potentially higher throughput. For the use of 150 runs, disregarding the first run, there could be deployment scenarios where we are more interested in the performance of these initial warm-up runs, before the cache behavior becomes more regular.

Overall, even for the small number of DLAS parameters explored across the layers of the stack, the combinatorial explosion means that there is a significant amount of data to collect, and conversely the exclusion of a wide range of DLAS parameters means that this data does not give us conclusive information on the impact of different acceleration techniques. At most, we can make highly conditional statements, such as "for this given model, evaluated on this hardware platform, using this implementation of this algorithmic primitive, and this

compression technique, we observe that it achieved a lower inference time than the same configuration using a different compression technique." A deeper evaluation varying only one or two parameters of DLAS is more tractable, and is the approach which we take in Chapters 5 and 6. However, as this study reveals, exploring all dimensions of DLAS is prohibitively expensive (see the DSE challenge in Section 1.2.3), and effectively exploiting across-stack interactions can be a complex undertaking (see the challenge in Section 1.2.2).

## 4.2 Summary

This chapter presented a retrospective of a study which explored the impact of varying a small number of parameters at each layer of DLAS. In our study, we find a variety of across-stack interactions and a variety of scenarios where ideal performance improvements were not achieved due to lack of full exploitation across the stack. Our work does not purport to propose solutions to all of these limitations, rather highlights some of the complexities which emerge in deep learning acceleration and presents a conceptual framework for practitioners to approach their studies in the future. We believe this can be achieved through closer collaboration across the layers of the stack to enable more holistic co-design and co-optimization. For example, we may be able to sample from weight-sharing NAS systems (discussed in Section 3.2.2), and for a given hardware platform, generate a DNN which has very high performance in terms of some metrics of interest such as accuracy, inference time, or power consumption. This sampling could consider the impact of varying multiple DLAS parameters such as algorithmic primitives, systems software, and compression techniques simultaneously, and generate a more performant solution than varying just one technique alone. Such broad co-design and co-optimization is still impractical, however the remainder of this thesis makes contributions which bring this goal closer.

# 5 | Accelerating Grouped Convolutions

Many model optimization techniques revolve around replacing the standard convolutions in a neural network with grouped convolutions [CGS18; How+17; Hua+18; San+18], as discussed in Section 2.3.3. These allow for substantial savings in memory and compute with minimal loss of accuracy, and are becoming increasingly prevalent. However, when evaluating the implementation of grouped convolutions present in current state-of-the-art deep learning frameworks such as PyTorch [Pas+19], TensorFlow Lite [Goo19], and TVM [Che+18b], we observe that the measured inference times are significantly higher than the expected ones, given the rate of compression provided by grouped convolutions.

In this chapter, we identify that this gap is due to a suboptimal algorithm design, and propose, implement, and evaluate a new solution using the TVM tensor compiler: Grouped Spatial Pack Convolution (GSPC). The primary contributions of this chapter can be summarized as:

- We propose a new algorithm for grouped convolutions, GSPC, and implement it in TVM with a hand-optimized schedule, exposing auto-tuning parameters of the schedule (such as tile size) using AutoTVM [Che+18a].
- We evaluate the performance of GSPC using three DNN models on the CPU of three edge devices, using our handwritten schedule and auto-tuning.
- We compare GSPC against implementations of grouped convolutions present in widely used deep learning frameworks, and we show that our solution outperforms them.
- We quantify the performance gap between the theoretically expected inference times and the measured ones.

The chapter is organized as follows: the first Section 5.1 gives some motivation for our solution based on experimental observations. Then in Section 5.2 we discuss our GSPC solution and the details of our implementation in TVM. Section 5.4 shows our experimental setup and a performance evaluation of several networks with grouped convolutions on three

edge devices, discussing the time/accuracy trade off, analyzing different implementations on TVM and comparing GSPC with other existing implementations of grouped convolutions. Finally, Section 5.5 concludes the chapter.

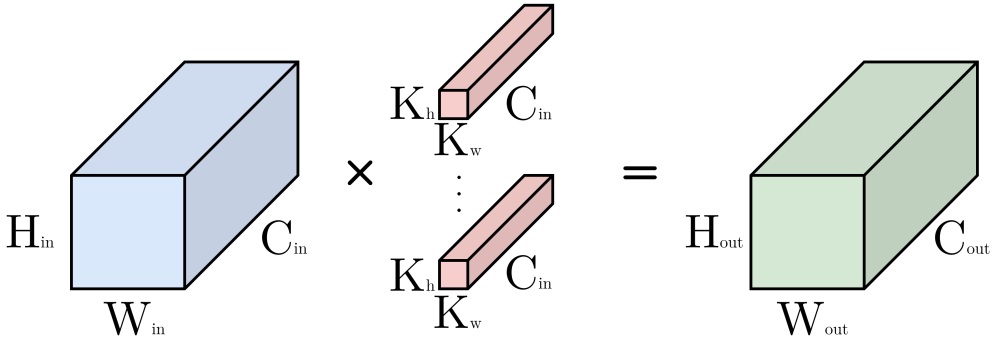
## 5.1 Motivation

As discussed in Section 2.3.3, leveraging grouped convolutions can allow us to reduce the number of parameters and MACs used in CNNs. Figure 5.1 shows the difference between group convolution and standard convolution. We would ideally expect that using $g$ groups would correspond to a $g\times$ speedup when compared to an equivalent standard convolution. However, this is a *theoretical* saving, and thus in this work we explored if this translates to real reductions in inference time. Figure 5.2 shows an initial experiment where we run WideResNet models (trained on CIFAR-10 [Kri09]) using standard (*S*) and grouped convolutions (*G*) on the CPU of a HiKey 970 board for three DNN deployment frameworks. Note that we use $G(g)$ to denote a grouped convolution using $g$ groups. As we can see, none of the frameworks provides the theoretical expected behavior, that is: 1. Models using grouped convolutions should execute in less time than the model implementing standard convolutions (*S*), since the overall number of Multiply-Accumulate (MAC) operations decreases when using groups (see Section 2.3.3); 2. As the number of groups increases (i.e., 2, 4, 8, etc.), the number of MACs decreases and thus the execution time should also decrease, with an expected linear decrease as $g$ grows.

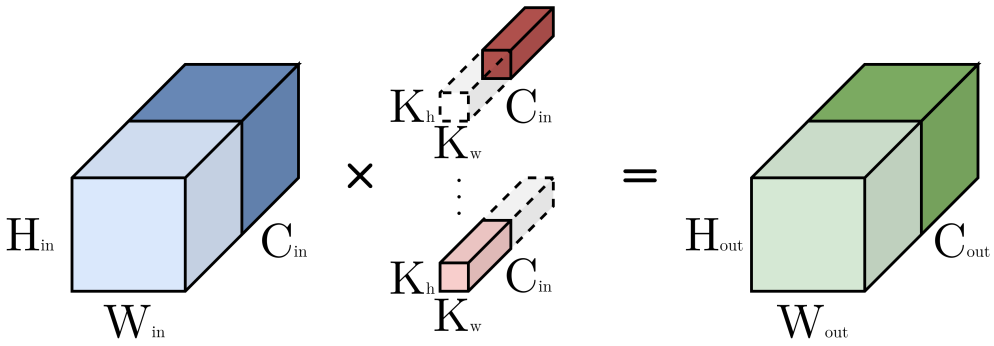
As highlighted in Sections 2.4.2 and 3.4.1, there are a number of popular algorithms that implement standard convolutional layers in DNNs, with a myriad of trade-offs. For example, direct convolution is the simplest conceptually requiring no data reshaping, and processes data using the *sliding window* that the convolution operation is often described as. Other approaches like GEMM convolution reshape and sometimes expand inputs or weights, which can reduce inference time by improving the locality of data. However, the increase in memory footprint and the computational cost of the reshape stages can reduce the improvement in inference time provided by the improved locality. Compared to standard convolutions, for grouped convolutions, the algorithms available and their potential trade-offs have been explored less in the literature.

In TVM, the algorithm used for grouped convolutions on CPUs is grouped direct convolution[1]. This is a variant of the standard direct convolution algorithm, and similarly has the advantage of requiring no additional memory footprint. However, without any reshaping of data or weights, it can suffer from poor data locality and thus performance can be suboptimal, especially for large layers since parameters may need to be reloaded from higher

[1] As of 2021, our GSPC was accepted as the default CPU algorithm in upstream Apache TVM.




(a) Standard convolution.




(b) Grouped convolution (with 2 groups).

Figure 5.1: Standard vs. grouped convolutions: (a) In a standard convolution *S*, each filter is convolved with all of the input's channels; (b) In a grouped convolution with two groups $G(2)$, half of the filters are applied to each half of the input for a $2\times$ reduction in parameters used. More generally, a grouped convolution with $g$ groups uses $g\times$ fewer parameters than a standard convolution.

levels of cache. For this reason, direct convolution is generally only used as the algorithmic primitive for standard convolutional layers in certain contexts, for example, when memory is particularly constrained. However, the effectiveness of this algorithm when adapted for grouped convolutions may have different considerations, especially for varying values of $g$.

We observe in Figure 5.2 that TVM's grouped direct convolution scales well as the number of groups increases, and outperforms both PyTorch and TensorFlow Lite for large values of $g$. This scaling makes sense since the increased $g$ reduces the number of MACs for grouped convolutional layers in the model. However, we observe that relative to the $S$ model in TVM, the time for $G(2)$, which reduces the number of MACs by 60%, is $4\times$ slower. Given the poor performance of $G(2)$, it is clear that an alternative approach to grouped convolutions is required to realize the potential performance improvements derived from the reduction in the number of MACs. Motivated by the poor performance of grouped convolutions shown in Figure 5.2, we propose a new solution called *Grouped Spatial Pack Convolution (GSPC)*, which has the potential to outperform existing implementations and approach the theoretically optimal performance level of grouped convolutions.

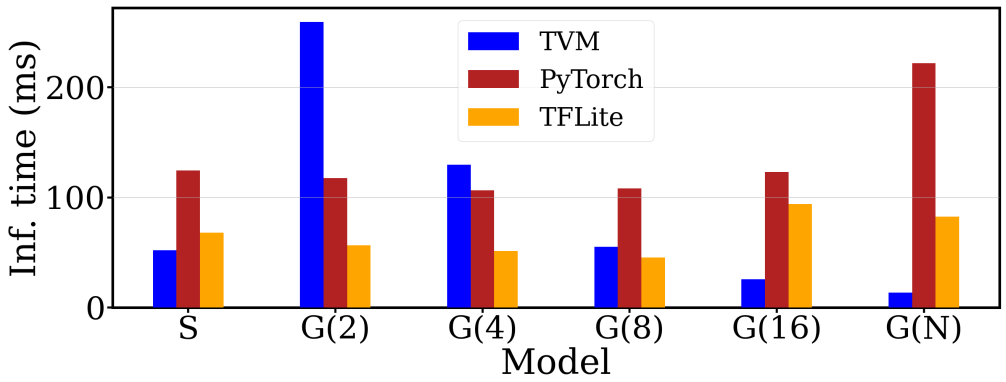



Figure 5.2: Inference time for WideResNet models using standard (S) and grouped convolutions ($G$) on the CPU of the HiKey 970 board for three common deep learning frameworks. No framework shows the expected behavior, defined as: i) faster execution than standard when using grouped convolutions, $G(g)$ where $g$ is the number of groups; ii) the time decreases as the number of groups increases.

## 5.2 Grouped Spatial Pack Convolutions

The GSPC algorithm is defined for the `NCHW` data layout[2]. We modify and extend the spatial pack convolutions (SPC) algorithm [ZC18], which does not cover grouped convolutions. Like SPC, GSPC reshapes data, kernels, and outputs to exploit data locality for the computation. Our extension splits and computes data along an additional outer dimension for groups. Since there is no data dependency between groups, this outer dimension can be leveraged to efficiently divide data between processing cores. We will describe the algorithm in more detail in Section 5.2.1. As we will discuss in Section 5.2.2, we implement the algorithm using TVM's compute schedule language, as it is portable across a wide variety of platforms, and can generate code which achieves state-of-the-art performance on many common deep learning benchmarks. We favored implementing the GSPC algorithm in TVM, since the predictable scaling of its default grouped convolution suggests that TVM's code generator would be more likely to give us reasonable scaling, as well as providing tensor compiler features such as auto-tuning.

### 5.2.1 General Description

At a high level, GSPC consists of four stages, takes two input volumes (input data and the weights/kernels), and produces one output volume. We expect a 4D input data volume of size $NC_{\text{in}}H_{\text{in}}W_{\text{in}}$, 4D kernels (weights) of size $C_{\text{out}}C_{\text{in}/g}K_hK_w$, and a 4D output volume of size $NC_{\text{out}}H_{\text{out}}W_{\text{out}}$.

[2]Input and output data are 4D arrays in row-major order, where $N$ is the batch size, $C$ is the number of channels, and $H$ and $W$ are height and width.

GSPC reshapes the input, weight, and output data to improve locality. The reshape has two values which represent tile size: $T_O$ and $T_I$, the former for tiling across output channels and the latter for tiling across input channels. Note that data in the same tile is related, and thus can enable further optimizations such as vectorization. We define $K_{PG}$ to be the number of kernels per group (i.e., $C_{\text{out}}/g$), and $C_{PG}$ as the number of input channels per group (i.e., $C_{\text{in}}/g$). The four stages of the GSPC algorithm are:

- Reshape 4D kernels into a new 7D volume:

$$C_{\text{out}}C_{\text{in}/g}K_hK_w \rightarrow g\lfloor\frac{K_{PG}}{T_O}\rfloor\lfloor\frac{C_{PG}}{T_I}\rfloor K_hK_wT_IT_O$$

- Reshape 4D padded input data into a new 6D volume:

$$NC_{\text{in}}H_{\text{in}}W_{\text{in}} \rightarrow gN\lfloor\frac{C_{PG}}{T_I}\rfloor H_{\text{in}}T_IW_{\text{in}}$$

- Perform the grouped convolution using the 7D weights and 6D inputs, storing the output in a temporary 6D volume. The computed 6D volume is:

$$gN\lfloor\frac{K_{PG}}{T_O}\rfloor H_{\text{out}}W_{\text{out}}T_O$$

- Reshape the 6D output volume to the desired 4D shape:

$$gN\lfloor\frac{K_{PG}}{T_O}\rfloor H_{\text{out}}W_{\text{out}}T_O \rightarrow NC_{\text{out}}H_{\text{out}}W_{\text{out}}$$

The kernel reshaping stage can be computed ahead-of-time and stored on disk in lieu of the default layout, since it does not depend on the input data. By reordering our weights and inputs, we can improve the memory locality of our computation, which can reduce the cost of loads for elements being computed on. Similarly, accumulating the convolution on a 6D intermediate array and reshaping to 4D output is preferable to accumulating directly onto 4D as improved locality can improve cache behavior. The tile sizes are constrained by $C_{PG}$ and $K_{PG}$ dimensions, since these are the dimensions over which we tile:

$$\begin{aligned} 0 < T_O \leq K_{PG} \\ 0 < T_I \leq C_{PG} \end{aligned} \tag{5.1}$$

However, the ideal values for these tiles which minimize inference time can vary. In the description of the stages of GSPC, note that the inner dimension of the reshaped kernels and

the intermediate outputs is $T_O$. The SIMD lane size for the target CPU can be a reasonable default for $T_O$, since data in the inner dimension is adjacent in memory and can thus be easily vectorized, however auto-tuning for each layer could discover if there is a better configuration of these schedule parameters.

Figure 5.3 illustrates the GSPC algorithm with a basic example. We use tile sizes $T_O = T_I = 2$, as these are the maximum values allowed by the constraints in Equation 5.1 for this example. The initial data layout is shown on the left, with the channels split by group for clarity. The 6D and 7D volumes are shown flattened. We observe how the input data and kernels are reshaped to improve data locality. Even in the original data layout, data is divided between groups and the GSPC reshape stages maintain this division. The MAC operations can be ordered to reduce the number of loads for each tile. In this example, each input value is used twice, thus computing these MACs in sequence could be a load-efficient approach. The outputs reshaping stage is trivial in this case due to the small output size, and thus from a 1D memory perspective the reshape is the identity. In the case of depthwise convolution, output reshaping is also the identity, which saves $N \times C_{\text{out}} \times H_{\text{out}} \times W_{\text{out}}$ copy operations.

Algorithm 3 describes GSPC for the `NCHW` layout. The strides hyperparameter is defined with $S_h$, $S_w$. Note that the number of input and output channels should be divisible by the number of groups, so we can evenly split data between the groups. The costs of GSPC's relatively complex index arithmetic are reduced by TVM's ahead-of-time compilation for each layer size. This means that expressions involving constants are simplified using constant folding and CSE (see Section 2.5.2), which greatly reduce the number of computations required during inference.

The main loop of kernel reshaping (lines 2-4) is nested with a depth of seven, with each loop representing a dimension of the reshaped kernel volume. The same is true for inputs reshaping (lines 6-8), with a nested loop depth of six. The main convolution loop (lines 10-18) is over the six dimensions of the temporary output volume, with an additional three loops over $C_{PG}$ and the kernel dimensions. These loops can be reordered to improve performance, but again the best ordering may need to be determined via auto-tuning.

### 5.2.2 Implementation

We implement GSPC in TVM, as it can generate efficient code for tensor programs, provides the best time for the $S$ model, and scales well as $g$ increases, despite the poor performance of $G(2)$. We leverage TVM's compute schedule programming paradigm, described in Section 2.5.6. Our high level algorithm is described following to Algorithm 3, using TVM's embedded domain-specific language. Our hand-tuned schedule applies a number of optimizations such as loop reordering, thread parallelism, loop unrolling, and SIMD-vectorization. These are intended to maximize data-reuse and parallelism.

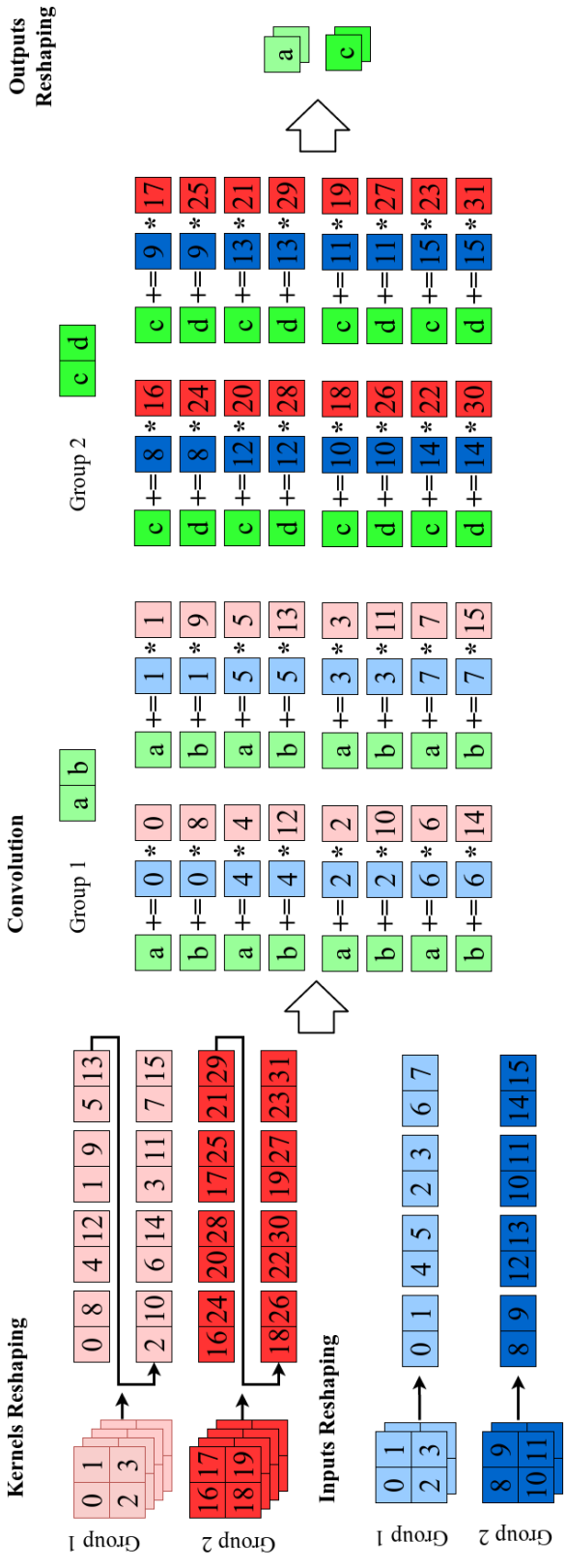



Figure 5.3: Overview of GSPC using a concrete example: two groups, four filters, four input channels, strides of one, and no input padding. Note that the input size is $1\times4\times2\times2$, the kernels size is of dimension $4\times2\times2\times2$ and numbers represent indices in the `NCHW` format.

**Algorithm 3** Grouped Spatial Pack Convolution (GSPC)

$X$ : inputs of shape $NC_{\text{in}}H_{\text{in}}W_{\text{in}}$
$W$ : kernels of shape $C_{\text{out}}C_{\text{in}/g}K_hK_w$
$T_I, T_O$: Tile sizes (constrained by equation 5.1)
$K_{PG} \leftarrow \frac{C_{\text{out}}}{g}, \quad C_{PG} \leftarrow \frac{C_{\text{in}}}{g}$

**Kernels Reshaping**

Allocate $W'$ of dimension $g\lfloor\frac{K_{PG}}{T_O}\rfloor\lfloor\frac{C_{PG}}{T_I}\rfloor K_hK_wT_IT_O$
**for** Dimensions of $W' : j, k, c, h, w, ci, co$ **do**
    $w \leftarrow W[c \times T_O + co + j \times K_{PG}][c \times T_I + ci][h][w]$
    $W'[j][k][c][h][w][ci][co] \leftarrow w$

**Inputs Reshaping**

Allocate $X'$ of dimension $gN\lfloor\frac{C_{PG}}{T_I}\rfloor H_iT_IW_i$
**for** Dimensions of $X' : j, n, C, h, c, w$ **do**
    $x \leftarrow X[n][C \times T_I + c + C_{PG} \times j][h][w]$
    $X'[j][n][C][h][c][w] \leftarrow x$

**Perform Convolution**

Allocate $Y'$ of dimension $gN\lfloor\frac{K_{PG}}{T_O}\rfloor H_oW_oT_O$
**for** Dimensions of $Y'$: $j, n, occ, oh, ow, ocv$ **do**
    $y \leftarrow 0$
    **for** $c = 0$ to $C_{PG}$ **do**
        **for** $kh = 0$ to $K_h$ **do**
            **for** $kw = 0$ to $K_w$ **do**
                $x \leftarrow X'[j][n][\lfloor\frac{ic}{T_I}\rfloor][oh \times S_h + kh][c \bmod T_I][ow \times S_w + k_w]$
                $w \leftarrow W'[j][occ][\lfloor\frac{ic}{T_I}\rfloor][kh][kw][c \bmod T_I][ocb]$
                $y \mathrel{+}= x \times w$
    $Y'[j][n][occ][oh][ow][ocb] \leftarrow y$

**Outputs Reshaping**

Allocate $Y$ of dimension $NC_{\text{out}}H_{\text{out}}W_{\text{out}}$
**for** Dimensions of $Y$: $n, c, h, w$ **do**
    $y \leftarrow Y'[\lfloor\frac{c}{K_{PG}}\rfloor][n][\lfloor\frac{c}{T_O}\rfloor \bmod \lfloor\frac{K_{PG}}{T_O}\rfloor][h][w][(c \bmod T_O) \bmod K_{PG}]$
    $Y[n][c][h][w] \leftarrow y$

Tuning the parameters exposed by our GSPC schedule includes varying the tile sizes, and optionally unrolling the $K_w$ loop of the convolution stage. There may be scope for additional improvements to the GSPC schedule, which could further reduce inference time. For example, a potential optimization could investigate the impact of interleaving portions of the reshaping and computation stages to reduce the footprint of the intermediate arrays by reusing a subset of their memory. In early 2020, when GSPC was developed, auto-scheduling systems like Ansor [Zhe+20a] were not available, therefore they are not included in the evaluation. This extension will be considered in future work.

## 5.3 Experimental setup

In this section, we describe our experimental setup for evaluating the effectiveness of GSPC. Section 5.3.1 describes the DNNs and datasets used, and Section 5.3.2 describes the hardware platforms evaluated.

### 5.3.1 Datasets and Networks

We consider two datasets widely adopted for image classification tasks, CIFAR-10 [Kri09] and ImageNet [Den+09], and we use the `float32` type to represent data values. We evaluate three DNN models, WideResNet-40-2 and ResNet34 which are good representatives of residual network types, and MobileNetV2 which is a widely used model for edge devices. Some relevant details of these models include:

- WRN-40-2: a Wide Residual Network (WRN) [ZK16] with 40 layers and a width-multiplier of 2 that requires 2.2 million parameters. We use a CIFAR-10 classification definition of the network.
- ResNet34: a Residual Network [He+16] with 34 layers that requires 21.8 million parameters. We use an ImageNet classification definition of the network.
- MobileNetV2 [San+18]: a DNN with 53 layers optimized for edge platforms that requires 3.5 million parameters. By default, the architecture uses depthwise separable convolutions (i.e., the maximum number of groups). We use an ImageNet classification definition of the network.

For WRN-40-2 and ResNet34, we take pretrained versions of the DNNs and define versions of models where the standard convolutions are replaced by a grouped convolution followed by a pointwise standard convolution, as discussed in Section 2.3.3. For MobileNetV2, we define versions of the model where the number of groups decreases, and an $S$ model where the grouped and pointwise convolution are replaced by a single convolution. We consider the following grouped convolutions: $G(g) \forall\, g \in \{2, 4, 8, 16, N\}$, where $N$ is the number of input channels to each convolution. Note that although pointwise convolutions incur a parameter cost, their inference time is negligible relative to grouped convolutions. This is because the operation is equivalent to a matrix multiplication over inputs and parameters, where we can perform GEMM convolution without reshaping the inputs, and given the lower number of parameters, we also have relatively fewer MACs.

For WRN-40-2 and ResNet34, for each value of $g$, models were trained with attention transfer [ZK17] (see Section 2.3.4) on 1 and 4 Nvidia TITAN X GPUs for 200 and 100 epochs

respectively. For training, we use Stochastic Gradient Descent (SGD) with momentum 0.9 to minimize cross-entropy loss, learning rate of 0.1, and a weight decay of $5\times10^{-4}$ and $1\times10^{-4}$ for WRN-40-2 and ResNet34 respectively. For MobileNetV2, since there was not a pretrained $S$ model to use for attention transfer, we trained each value of $g$ from scratch on 1 Nvidia TITAN RTX GPU for 150 epochs using SGD with momentum 0.9, a learning rate of $5\times10^{-2}$, and a weight decay of $4\times10^{-4}$.

### 5.3.2 Hardware Platforms

Table 5.1 lists the platforms used in this chapter. There are two edge boards (HiKey 970, Raspberry Pi3B) that include both a CPU and GPU, however in this work we focus on CPU evaluation, and we leave GPU investigations for future work. We also analyze a standard desktop Intel i7 CPU. Therefore, we evaluate Arm and Intel processors that implement two different ISAs with frequencies ranging from 1.2GHz to 3.2GHz. Note that the CPU of the HiKey board implements the *big.LITTLE* architecture (4 *big* cores + 4 *LITTLE* cores), but in this work we only use the *big* cores. Finally, the memory hierarchy varies significantly across platforms, for example, the Intel i7 features an L3 cache, whereas the other CPUs do not. All these features give us a diverse set of configurations for evaluation.

## 5.4 Evaluation

In this section, we evaluate GSPC using the three DNNs, and three CPU devices as described in Section 5.3. Section 5.4.1 shows how the accuracy, number of parameters, number of MACs, and inference times using GSPC. Next, Section 5.4.2 performs a deeper analysis of GSPC against baseline TVM, investigating ideal versus observed performance and exploring the impact of auto-tuning the schedule. Finally, Section 5.4.3 compares the performance of GSPC against other DNN frameworks.

### 5.4.1 Speed versus Accuracy Analysis

Tables 5.2, 5.3, and 5.4 show the inference time in milliseconds for all the DNN models considered using standard ($S$) and grouped ($G$) convolutions for WRN-40-2, ResNet34, and MobileNetV2 respectively when running on the three platforms under study[3] using our GSPC implementation in TVM. The tables also show the total parameter cost, the number of MACs,

[3]Note that all times are for single thread execution, since we verified that threads affect quite differently the performance of each platform, thus not providing a completely fair comparison. We leave the threads analysis for future work.

Table 5.1: Hardware features of the devices used in the grouped convolutions experiments.

| Device | CPU | L1 Cache (I+D) | L2 (+L3) Cache | RAM | Instruction Set |
|---|---|---|---|---|---|
| Desktop | Intel i7-8700 (6 cores) @ 3.2 GHz | 192K + 192K | 1.5M (+12M) | 16GB DDR3 | x86 64-bit |
| HiKey 970 | Arm Cortex-A73 (4 cores) @ 2.4 GHz<br>Arm Cortex-A53 (4 cores) @ 1.8 GHz | 256K + 256K<br>128K + 128K | 2M shared<br>1M shared | 6GB LPDDR4 | ARMv8-A 64-bit |
| Raspberry Pi3B | Arm Cortex-A53 (4 cores) @ 1.2 GHz | 64K + 64K | 512K shared | 1GB LPDDR2 | ARMv8-A 64-bit |

Table 5.2: Inference time in *ms* for WRN-40-2 models with standard ($S$) and grouped ($G$) convolutions when running on the platforms in Table 5.1.

| Model | Model Info | | | Inference Time | | |
|---|---|---|---|---|---|---|
| | Params | MACs | Top-1 | Desktop | HiKey | RPi3 |
| S | 2242.26K | 328.30M | 4.79 | 8.23 | 65 | 811 |
| $G(2)$ | 1357.68K | 198.15M | 4.87 | 9.20 | 51 | 530 |
| $G(4)$ | 813.36K | 118.52M | 5.00 | 5.84 | 34 | 307 |
| $G(8)$ | 541.20K | 78.71M | 5.05 | 4.65 | 24 | 199 |
| $G(16)$ | 405.12K | 58.80M | 5.13 | 4.51 | 20 | 158 |
| $G(N)$ | 292.22K | 44.83M | 6.57 | 2.14 | 16 | 122 |

Table 5.3: Inference time in *ms* for ResNet34 models with standard ($S$) and grouped ($G$) convolutions when running on the platforms in Table 5.1.

| Model | Model Info | | | Inference Time | | |
|---|---|---|---|---|---|---|
| | Params | MACs | Top-1 | Desktop | HiKey | RPi3 |
| S | 21.79M | 3.67G | 26.73 | 107 | 1096 | 7466 |
| $G(2)$ | 13.22M | 2.25G | 26.13 | 99 | 636 | 5700 |
| $G(4)$ | 8.14M | 1.39G | 26.58 | 62 | 426 | 3334 |
| $G(8)$ | 5.60M | 0.97G | 27.24 | 41 | 304 | 2344 |
| $G(16)$ | 4.34M | 0.75G | 27.99 | 34 | 259 | 1749 |
| $G(N)$ | 3.13M | 0.56G | 30.16 | 23 | 204 | 1285 |

Table 5.4: Inference time in *ms* for MobileNetV2 models with standard ($S$) and grouped ($G$) convolutions when running on the platforms in Table 5.1.

| Model | Model Info | | | Inference Time | | |
|---|---|---|---|---|---|---|
| | Params | MACs | Top-1 | Desktop | HiKey | RPi3 |
| S | 44.05M | 5.56G | 26.03 | 166 | 1207 | 13770 |
| $G(2)$ | 23.75M | 2.92G | 25.90 | 135 | 776 | 7603 |
| $G(4)$ | 13.59M | 1.60G | 26.34 | 75 | 733 | 4608 |
| $G(8)$ | 8.52M | 0.95G | 26.84 | 47 | 495 | 2625 |
| $G(16)$ | 5.98M | 0.62G | 27.06 | 37 | 429 | 1808 |
| $G(N)$ | 3.50M | 0.31G | 28.20 | 15 | 134 | 812 |

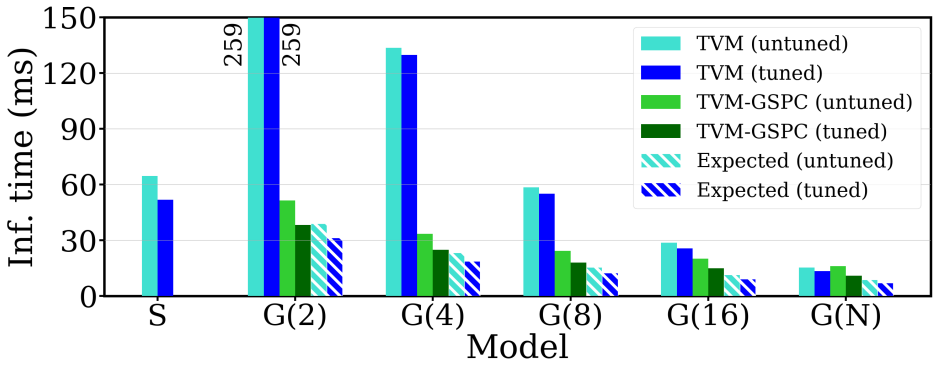
and the Top-1 error for each DNN model. The number of MACs for each convolutional layer is obtained with the following formula:

$$\text{MACs} = \frac{N \times C_{\text{in}} \times C_{\text{out}} \times K_h \times K_w \times H_{\text{out}} \times W_{\text{out}}}{g} \tag{5.2}$$

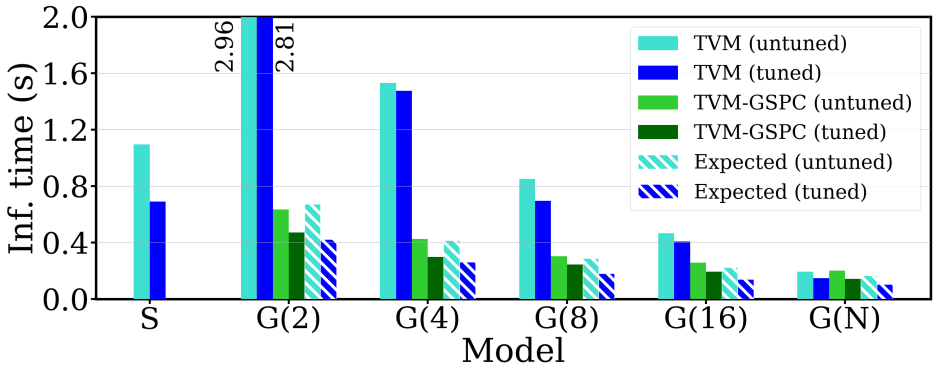
where $N$ is the batch size ($N = 1$ for all experiments), $C_{\text{in}}$ is the number of input channels, $C_{\text{out}}$ is the number of output channels, and $H_{\text{out}}$ and $W_{\text{out}}$ are the height and width of the layer's output respectively. $K_h \times K_w$ is the kernel size of each convolution, and $g$ is the number of groups.

As we can see in the tables, the reduction in the number of parameters, and thus the number of MACs, derived from using grouped convolutions provides between ~4-17$\times$ of speedup in the inference time across platforms and DNNs, the Raspberry Pi device and the MobileNetV2 network being the combination that provides the highest improvements. We also observe that on the desktop the inference time for $G(2)$ is not reduced with respect to the corresponding $S$ model as on the other two platforms, it even increases for WRN-40-2. However, the time decreases for every subsequent $G$ model. This observation suggest that the schedule is not less optimized for the Intel x86-64 architecture of the desktop. In TVM, the schedules can be optimized for a given hardware architecture and the default $S$ model is taking advantage of this, as we checked that it has schedules for both Intel and Arm architectures. However, we optimized the schedule of our GSPC code primarily for the HiKey platform, as we performed most of our experiments on it. Optimizing GSPC for the Intel architecture should provide better times for the $G$ models, but we leave this optimization for future work.

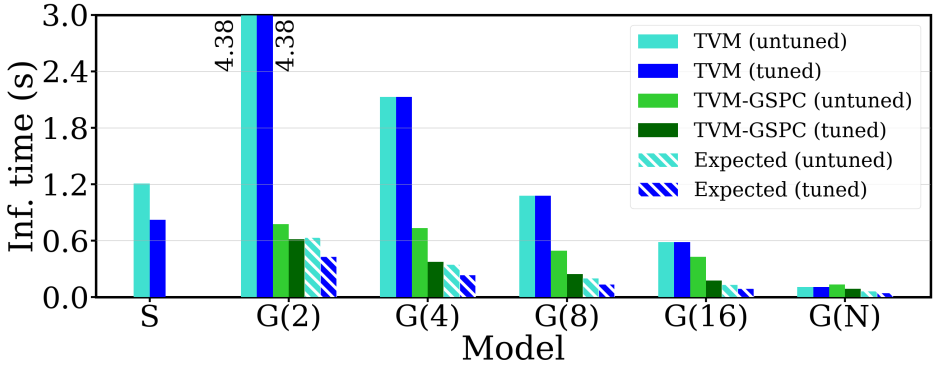
Related to the accuracy of the models, we see in Tables 5.2, 5.3, and 5.4 that the increase in Top-1 error can vary from almost 2% for WRN-40-2 to 3.5% for ResNet34 when we compare the $S$ and $G$ models. We also see that the overall error is much higher for the models using the ImageNet dataset (~30% vs ~7%), since a 1000-way classification is harder than a 10-way one. Therefore, these results provide different options to the user for selecting a model based on the time/accuracy trade off. The best solution for a given application will depend on its specific requirements and the hardware platforms available. For example, if the target platform is more constrained like the Raspberry Pi, it could be better to sacrifice some accuracy in favor of speeding up the inference time. However, for a more powerful platform like the desktop it can be better to maximize accuracy, as all times are below 166ms.




(a) WRN-40-2




(b) ResNet34




(c) MobileNetV2

Figure 5.4: Inference time in *ms* for DNN models with standard ($S$) and grouped ($G$) convolutions when running on the CPU of the HiKey 970 board. We compare the measured and expected times of our GSPC and the default TVM implementation for both tuned and untuned versions of the code.

### 5.4.2 TVM Analysis

Figure 5.4 shows the *Measured* versus the *Expected*[4] inference time for all the models considered for the three DNNs under study when running on the HiKey 970 platform. We compare GSPC with the default implementation of grouped convolutions in TVM, and we consider the tuned and untuned versions of the code in both cases. Note that the times in Tables 5.2, 5.3, and 5.4 correspond to GSPC untuned times. We only report the tuned inference time on the HiKey 970, since the auto-tuning process is very time-consuming, with an estimated search time of multiple weeks on the Raspberry Pi board. Our key observations in Figure 5.4 are as follows:

- GSPC improves the times of the default TVM implementation of the $G(2)$-$G(16)$ models for the three DNNs for both tuned and untuned versions of the code. However, for $G(N)$ the default TVM implementation is slightly better than GSPC (~5-22% across DNNs) for the untuned version. We believe that this is due to the overhead created by the reshaping stages of GSPC, which for $G(N)$ are maximized relative to the computation time. We leave for future work to investigate this problem further.

- When we consider the tuned versions of $G(N)$, GSPC provides better times than the default TVM implementation (~3-34% across DNNs). However, note that for MobileNetV2 the tuned times that we obtained for the default TVM implementation were worse than the untuned ones for all $G$ models. For this reason, in Figure 5.4c the tuned times for default TVM with $G(N)$ are the same as the untuned ones. We could not find an explanation for this strange result, but reproduced the behavior multiple times.

- There are differences between the expected and measured times for both tuned and untuned versions across all $G$ models. This performance gap is ~7-99% for untuned and ~27-72% for tuned versions respectively. Note that the expected times are theoretical estimations based on the structure of the code for the standard convolution, which should not be considered as true optimal values. In some cases, it can be possible to outperform the expected time (see $G(2)$ in Figure 5.4b), for reasons such as more data fitting in a lower level of cache. In this case, we incur reduced overheads for fetching data since it is closer to where it is required.

Overall, our GSPC implementation is on average 3.4× faster than the default TVM version for all the tuned/untuned $G$ models, when using the arithmetic mean.

### 5.4.3 Frameworks Comparison

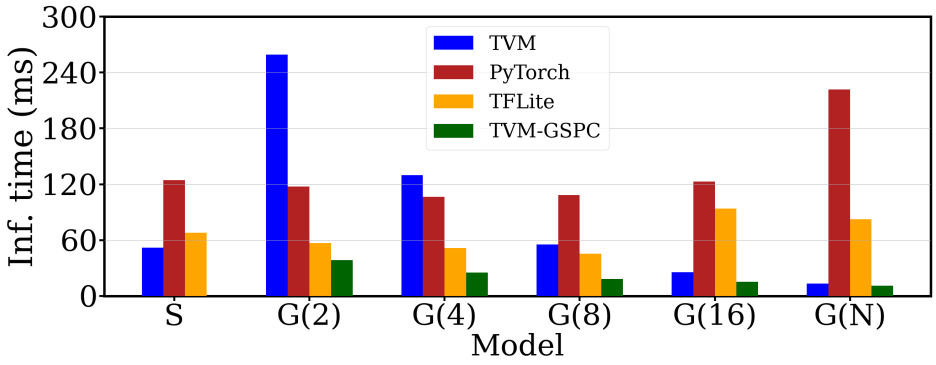
Figure 5.5 shows the inference time on the CPU of the HiKey board of GSPC and other implementations of grouped convolutions in current deep learning frameworks for a varying number of groups of our three DNNs, We report tuned versions of both GSPC and default TVM. The other frameworks analyzed are PyTorch [Pas+19] and TensorFlow Lite [Goo19].

As we can see, GSPC provides the best results for all the $G$ models of the three DNNs, clearly outperforming the default TVM and the other two frameworks, up to $8\times$ and $4\times$ better than PyTorch and TensorFlow Lite respectively. To the best of our knowledge, in terms of inference time GSPC is the most efficient implementation of grouped convolutions on CPUs available. We also observe that TensorFlow Lite performs much better than PyTorch for all the $G$ models of WRN-40-2 and for the $G(2)$-$G(16)$ models of ResNet34, whereas PyTorch is better for $G(N)$ of ResNet34 and all the $G$ models of MobileNetV2. However, none of these frameworks scale as expected for the $G$ models according to the number of MAC operations, where for example, we expect $G(2)$ to be around half of the inference time of $S$, $G(4)$ to be around half of $G(2)$, etc.

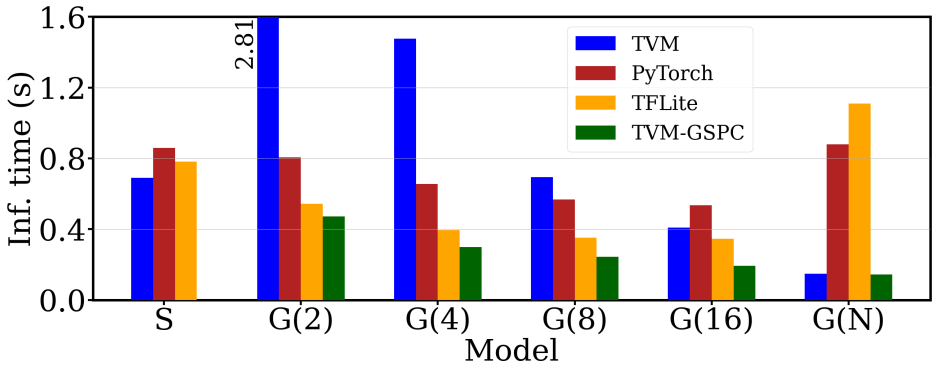
## 5.5 Summary

In this chapter we proposed Grouped Spatial Pack Convolution (GSPC) as a new and more efficient implementation of grouped convolutions. We have implemented GSPC in TVM, which provides state-of-the-art performance on CPUs, providing a hand-optimized schedule and auto-tuning parameters. We evaluated three DNNs implementing grouped convolutions for two datasets on three edge devices with varying hardware architectures. We also compared our implementation against existing solutions in current deep learning frameworks, outperforming them in all settings. Finally, we observed that even though DNNs using GSPC significantly improve their performance, there is still a gap between the expected inference time and the observed one.

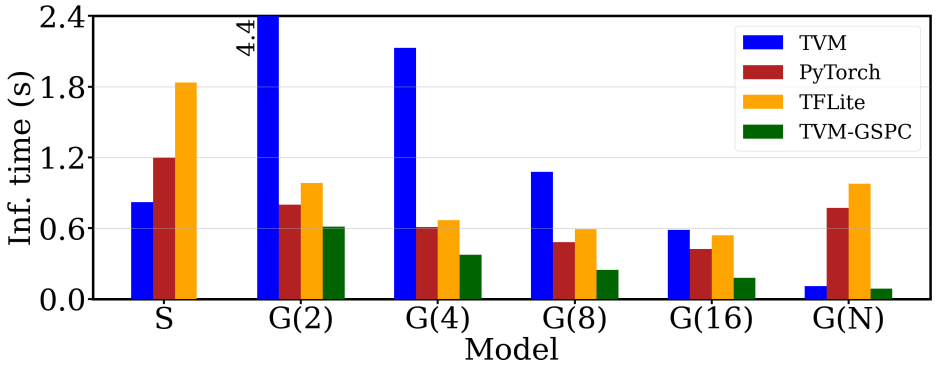
[4]Computed from the inference time of the $S$ model on a given platform by obtaining the time of a single MAC operation and then extrapolating.




(a) WRN-40-2




(b) ResNet34




(c) MobileNetV2

Figure 5.5: Inference time in *ms* for DNN models with standard ($S$) and grouped ($G$) convolutions when running on the CPU of the HiKey 970 board. We compare the tuned version of GSPC and default TVM against PyTorch and TensorFlow Lite.

# 6 Reusing Auto-Schedules for Efficient Tensor Program Code Generation

As we have highlighted throughout this thesis, tensor programs such as DNNs can be computationally expensive. As discussed in Section 2.5.7, an emerging approach to accelerate DNNs is using *auto-schedulers*, such as Ansor [Zhe+20a], which generate efficient code for a given DNN model and hardware platform via extensive automated program transformation exploration. This approach can produce state-of-the-art inference time performance on a range of platforms and programs, and in particular can show improvement compared to existing approaches on novel operations such as capsule 2D convolution [SFH17].

However, this tuning process can be very time-consuming and specific to a given kernel of a given size. To help tackle these two challenges, in this chapter we introduce *transfer-tuning*, a novel approach which can improve execution performance for a given tensor program with reduced tuning time. Transfer-tuning exploits the similarity between kernels containing the same operations with varying data sizes, such that we can reuse schedules from other tensor programs. Therefore, we can achieve performance improvements while reducing the search time costs associated with auto-scheduling. Transfer-tuning's main value comes in use-cases where tensor program deployment requires performance efficiency but has reduced resources to perform costly auto-scheduling.

The rest of this chapter is organized as follows: Section 6.1 gives some motivation regarding the overheads involved in auto-scheduling and introduces how transfer-tuning tackle this, Section 6.2 describes the key components of transfer-tuning and how it is enabled by the features of the compute schedule programming paradigm. Section 6.3 presents a range of experimental evaluations demonstrating and implementing the principles of transfer-tuning, and Section 6.4 concludes the chapter.

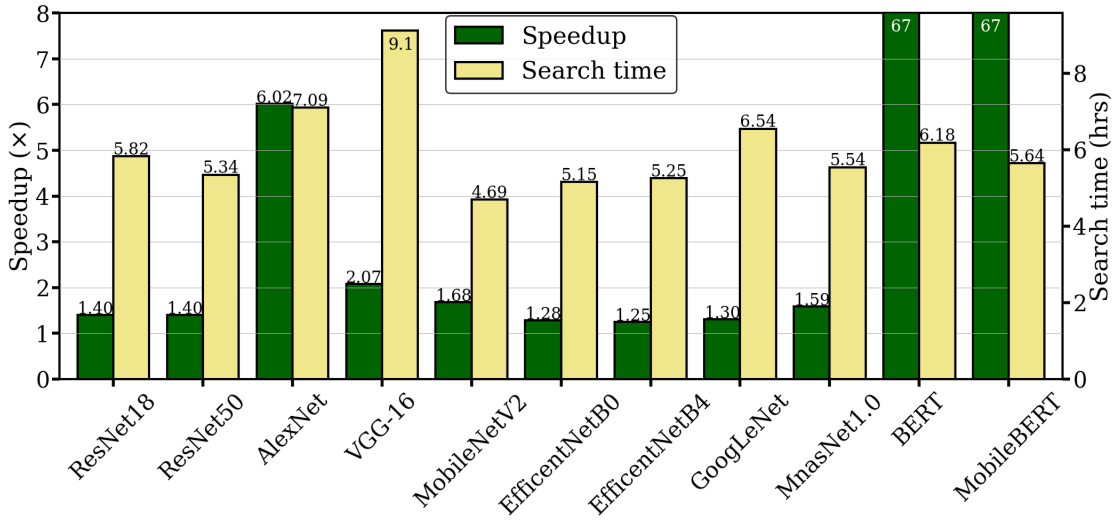



Figure 6.1: Inference time speedup and auto-scheduling search time when running Ansor on an Intel Xeon E5-2620.

## 6.1 Motivation

Figure 6.1 shows the tuning time required and speedups achieved by the Ansor auto-scheduler for a number of widely used DNN models on a common server-class CPU, an Intel Xeon E5-2620. We observe that the maximum speedup varies between models, and the search time can be several hours, which is a large upfront cost, especially as the number of models or platforms increases. If a range of applications are to be deployed on a given platform, this upfront cost may be further exacerbated, especially if the platform is more constrained, and thus make the potential performance improvements of auto-scheduling impractical to achieve, since tuning costs may be too high.

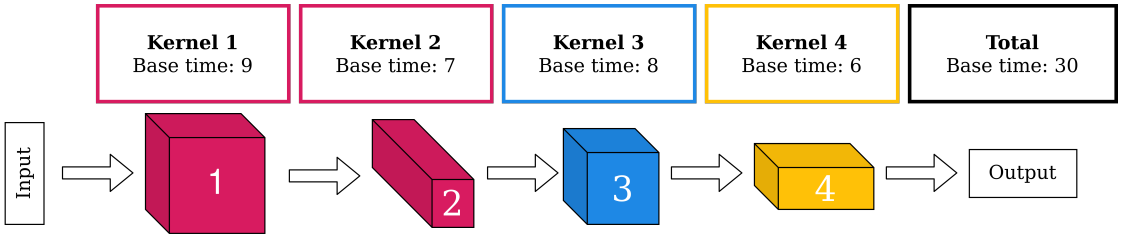
To reduce tuning time we could sacrifice potential performance improvements by stopping early or only tuning a subset of the kernels. However, Ansor gets its improvements by evaluating a vast array of possible schedules, and both early stopping and tuning a subset of kernels may miss performant schedules. Our main observation is that many applications, such as DNNs, feature high similarity between kernels in terms of the types of operations they compute and the sizes of their tensors. For example, most DNNs contain a limited set of operations such as convolutional, dense, and pooling layers. The DNN models in Figure 6.1 contain 22 unique kernel types (which we call *kernel classes*), with every model having at least one kernel class in common with every other model, and often many more. Thus, if we have already found performant schedules for some tensor program, perhaps we could reuse this information on other tensor programs which contain similar kernels. This observation motivates the contribution of this chapter, however let us further discuss the practical underpinnings of auto-schedulers to make this clearer.

Auto-schedules, such as those provided by Ansor [Zhe+20a], can yield state-of-the-art performance on a number of tensor programs (such as DNNs) and hardware devices by automatically generating optimized schedules. Figure 6.2a shows an example of a simple computation Directed Acyclic Graph (DAG) for a tensor program, such as a DNN, with default untuned kernels. Each node of the graph represents a unit of computation, also known as a *kernel*. The colors of the nodes represent the *kernel classes*, meaning the types of operations contained within (e.g., DNN layer types such as convolutional layers), with there being three classes of kernels in this example and four kernels in total. The shapes of the nodes represent the size of the data computed in the kernel, which in this example varies between kernels. This illustrative example shows how we can accelerate the inference time by running auto-scheduling individual kernels. Figure 6.2b shows that we can speed up the inference time by $2\times$ at the cost of a tuning search time of 36, where all values are for illustrative purposes.

However, it should be noted that auto-scheduling can be a very time-consuming process, as shown in Figure 6.1, where the tuning search time can be on the order of several hours for a whole DNN model. This depends on the complexity of the tensor program, the number of schedule variants chosen to evaluate, and the resources available of the target hardware device. The high cost of search can be a bottleneck to deployment, since if we want to get the best inference time for a given DNN model we must spend a long period of time exploring the schedule space. Approaches to reduce search time include: reducing the time we allow the tuner, as shown in Figure 6.2c; tuning a subset of the model's kernels, as shown in Figure 6.2d; or some combination of the two. These trade-offs allow users to sacrifice potential improvements in performance for reduced tuning search time.

Our main observation in Figure 6.2 is that kernels of the same class (where class is represented as color) may produce auto-schedules with similar properties. This is because they define the same high-level algorithm over varying data sizes. A question that emerges from this observation is *'Can an auto-schedule for one kernel be reused for another kernel of the same class?'* If so, a further question is *'How different will the optimizations found via auto-scheduling be between two different kernels of the same class?'* The answer to the second question will vary depending on the structure of the computations defining the class. Factors such as access patterns and costs of the loop body are likely to play important roles. In addition, the architecture of the target platform that the auto-schedule exploits is also likely to be important, since the organization of the memory hierarchy and features such as SIMD-instruction size may make some optimizations more or less relevant. Perhaps having some data size dimensions being similar (such as the extent of the innermost loop) could be more important than others (such as the extent of the outermost loop).

Therefore, answering the first question, we define the process of reusing an auto-schedule for a given kernel on a different kernel as *transfer-tuning*. In Figure 6.2e we show an illustrative example of transfer-tuning, where we reduce tuning search time by reusing auto-schedules.




(a) Kernels of a tensor program, 4 kernels of 3 classes, where classes define the same operations over varying input sizes (e.g., convolutions).

| **Schedule 1** | **Schedule 2** | **Schedule 3** | **Schedule 4** | **Total** |
|---|---|---|---|---|
| Base time: 9 | Base time: 7 | Base time: 8 | Base time: 6 | Base time: 30 |
| Tuned time: 4 | Tuned time: 3 | Tuned time: 4 | Tuned time: 4 | Tuned time: 15 |
| *Tuning time: 12* | *Tuning time: 8* | *Tuning time: 10* | *Tuning time: 6* | *Tuning time: 36* |

(b) Inference time and tuning costs when kernels are fully tuned.

| **Schedule 1** | **Schedule 2** | **Schedule 3** | **Schedule 4** | **Total** |
|---|---|---|---|---|
| Base time: 9 | Base time: 7 | Base time: 8 | Base time: 6 | Base time: 30 |
| Tuned time: 7 | Tuned time: 6 | Tuned time: 7 | Tuned time: 5 | Tuned time: 25 |
| *Tuning time: 6* | *Tuning time: 4* | *Tuning time: 5* | *Tuning time: 3* | *Tuning time: 18* |

(c) Inference time and tuning costs when tuning with reduced time.

| **Schedule 1** | | **Schedule 3** | | **Total** |
|---|---|---|---|---|
| Base time: 9 | Base time: 7 | Base time: 8 | Base time: 6 | Base time: 30 |
| Tuned time: 4 | | Tuned time: 4 | | Tuned time: 21 |
| *Tuning time: 12* | | *Tuning time: 10* | | *Tuning time: 22* |

(d) Inference time and tuning costs when tuning fewer kernels.

| **Schedule 1** | **Schedule 1** | **Schedule 3** | **Schedule 4** | **Total** |
|---|---|---|---|---|
| Base time: 9 | Base time: 7 | Base time: 8 | Base time: 6 | Base time: 30 |
| Tuned time: 4 | Tuned time: 5 | Tuned time: 4 | Tuned time: 4 | Tuned time: 17 |
| *Tuning time: 12* | *Tuning time: 0* | *Tuning time: 10* | *Tuning time: 6* | *Tuning time: 28* |

(e) Inference time and tuning costs when using transfer-tuning to reuse the auto-schedule of Kernel 1 with Kernel 2.

Figure 6.2: Illustrative example of the costs and benefits of different approaches of auto-scheduling for a tensor program.

We reuse the auto-schedule for Kernel 1 with Kernel 2 to reduce the inference time without requiring any additional tuning. We also use the auto-schedule of Kernel 1 with itself, which we refer to as the 'native schedule'. This makes the overall search time lower than our full tuning in Figure 6.2b. Note that we should expect some penalty when running Schedule 1 with Kernel 2 compared to running a native auto-schedule for Kernel 2, since a native schedule will exploit the specific data sizes of the computation to find optimizations for the target hardware and data size specific optimizations (see Schedule 2 in Figure 6.2b). This kernel specific information would not be exploited by using Schedule 1 for Kernel 2, as the schedule is tuned for Kernel 1. Therefore, in Figure 6.2e we observe that for Kernel 2, the transfer-tuned schedule achieves an accelerated execution time of 5, whereas when executed with Schedule 2 in Figure 6.2b, it is faster, with an execution time of 3.

The target of transfer-tuning is to improve the inference time of the overall tensor program, while being cheaper than running an auto-scheduler. The trade-off between search time and performance improvement is interesting to explore and exploit, as long search times may not always be acceptable. For example, a developer of AI applications for smartphones may not have the resources to provide auto-scheduling for their DNN model for the wide range of heterogeneous devices their app will be deployed on. Similarly, it will be unlikely that smartphone users be willing to wait several hours for the DNN model to auto-schedule on their device. Therefore, in this case transfer-tuning could provide some performance speedups in a shorter period of time. This reduced search time may also translate to reduced energy usage, as auto-scheduling is an energy intensive process that can saturate all CPU cores. However, in this work we focus purely on the inference time performance improvements of transfer-tuning, compared against the reduced search costs.

## 6.2 Transfer-Tuning

This section builds upon the initial observations and research questions of Section 6.1 to build a more concrete underpinning and conceptualization of transfer-tuning. First, in Section 6.2.1 we discuss some of the types of optimizations used by tensor program auto-schedulers, how transfer-tuning is possible, and its potential benefits. Then in Section 6.2.2, we further discuss the idea of kernel classes introduced in Section 6.1 and how they are a key part of transfer-tuning. In Section 6.2.3 we explore some of the behaviors of transfer-tuning on a full DNN model (ResNet18), and finally in Section 6.2.4 we discuss some other practical considerations for transfer-tuning.

### 6.2.1 Principles of Transfer-Tuning

Before delving into transfer-tuning, it is important to briefly review the concepts of schedules and auto-schedules, introduced in Section 2.5. Let us consider an operation such as a matrix-multiplication, which has a fixed loop structure but may have varying data sizes, as shown in lines 1-5 of Algorithm 4.

There are a variety of code transformations which can be applied to this operation, some of which are applicable to all instantiations of the operation and others which are specific to a particular input matrix size. For instance, transformations which are data-shape agnostic include unrolling a loop to its maximum depth, as introduced in Section 2.5.2. Data-shape-specific transformations such as loop-splitting transformations that may only apply to a specific loop size. In the case of unrolling, no matter how many iterations are in a loop, the transformation can be applied as long as we know the number of iterations ahead-of-time, it is a valid regardless of if there are 3 iterations or 300,000. However, the performance benefit of the transformation will vary depending on the number of iterations, with relevant factors including the architecture of the underlying hardware (e.g., the features of its cache) and properties of the computation such as the cost of the loop body.

In contrast, for data-shape-specific transformations, we may not have this flexibility. Taking loop-splitting as an example, if we have a loop over the range $(0, N)$ where $N = 32$, we could apply a loop splitting optimization defined as $\texttt{Split}(N, 4, 8)$ that breaks the loop into two loops in the ranges $(0, 4)$ and $(0, 8)$, which would allow us to traverse the full 32 elements. If we try to apply this optimization to a similar loop where $N = 128$, then splitting it into the two prior ranges would produce invalid code, since we will not be able to cover the full space of the loop. However, if we reformulate our transformation such that we apply it as $\texttt{Split}(N, (N/8), 8)$ our transformation becomes valid for all programs where $\{N \in \mathbb{N} : 8 \mid N\}$ (i.e., when $N$ is divisible by 8).

Therefore, some transformations can be applied to a kernel regardless of the data-shape, others can be reformulated to be valid for more than one data-shape, and some may not be valid for any data-shape other than the one they were originally defined for. The performance benefits of these transformations may be data-shape dependent, for example, a loop unrolling that brings benefit for a small loop range could bring a penalty for a larger loop range. However, we argue that even with large data-shape differences some of these transformations can potentially improve performance, as compared to a generic schedule.

As discussed in Section 2.5.7, the process of auto-scheduling takes a set of kernels representing operations in a tensor program (such as a DNN) and iteratively explores the space of transformations that we can apply to each of them. Transfer-tuning involves applying a schedule produced for a given kernel via auto-scheduling to a different kernel, with poten-

tially varying data shapes. The technique exploits the fact that many schedule transformations can be formulated to be data-shape agnostic, meaning that we can adapt schedules for kernels that they were not tuned for.

Observe two examples of auto-schedules for our row-major square matrix-multiply as defined in Algorithm 4, for two data sizes. Auto-scheduled kernels, such as kernels containing convolutional layers, can be verbose, difficult to interpret, and intuitions as to why they provide good performance may be unclear. This is because they are automatically generated to exploit hardware performance dynamics that may not be evident, such as cache behavior. Even for this relatively simple operation the schedules have many steps.

For our example, we have two instances of a schedule for varying sizes of our operation: $C_1 = A_1B_2$ which multiplies two 512×512 matrices, and $C_2 = A_2B_2$ which multiplies two 1024×1024 matrices. We use Ansor to produce auto-schedules for the two kernels, observing an improvement of 246× and 308× for $C_1 = A_1B_2$ and $C_2 = A_2B_2$ respectively compared to using an unoptimized schedule on the Intel Xeon E5-2620 CPU. The auto-scheduling of the two kernels produces different schedules since they have different sizes. Additionally, auto-scheduling in Ansor is non-deterministic due to the use of genetic algorithms to mutate schedules, and a stochastic learned cost model to reduce evaluation costs; thus differences in the auto-schedule may emerge even when re-running Ansor for the same kernel. Lines 6-17 and 18-35 of Algorithm 4 show a simplified representation of auto-schedules generated for $C_1 = A_1B_1$ and $C_2 = A_2B_2$ respectively. Next, we briefly explain the schedule primitives used in this example, which are a subset of all the primitives available to Ansor and by extension, transfer-tuning:

- `Split`([range], [factor]): split a loop range into inner and outer ranges.
- `Reorder`([set of ranges]): specify a reordering of a set of nested loops.
- `Fuse`([range], [range]): fuse two consecutive loop ranges into a single range.
- `Parallel`([range]): mark an axis to be used for multithreaded computation.
- `Unroll`([range], [max unroll factor]): unroll a loop range up to a maximum depth.
- `Vectorize`([range]): apply SIMD vectorization to a loop range.
- `ComputeAt`([output tensor], [axis]): move a loop body computation such that it is computed at a given axis.

Applying transfer-tuning to these GEMM computations, i.e., using the schedule generated for $C_1 = A_1B_1$ with $C_2 = A_2B_2$ and vice-versa, we observe that we still produce valid code, obtain inference time performance within 5% of the native tuning for both kernels, and a

**Algorithm 4** Auto-schedules for a GEMM operation

$\boldsymbol{A}$ : input matrix of size $N \times K$
$\boldsymbol{B}$ : input matrix of size $K \times M$
$\boldsymbol{C}$ : output matrix of size $N \times M$

**Unmodified row-major matrix-multiply computation**

**for** $n \leftarrow 0$ to $N$ **do**
  **for** $m \leftarrow 0$ to $M$ **do**
    $C[n][m] \leftarrow 0$ ▷ Initialize output value to zero
    **for** $k \leftarrow 0$ to $K$ **do**
      $\boldsymbol{C}[n][m]$ += $\boldsymbol{A}[n][k] \times \boldsymbol{B}[k][m]$

**Simplified auto-schedule where** $N = P = K = 512$

$N_o, N_i \leftarrow \texttt{Split}(N, 8)$
$N_{oo}, N_o \leftarrow \texttt{Split}(N_o, 1)$ ▷ note $N_o$ redefined
$N_{ooo}, N_{oo} \leftarrow \texttt{Split}(N_{oo}, 16)$
$M_o, M_i \leftarrow \texttt{Split}(M, 8)$
$M_{oo}, M_o \leftarrow \texttt{Split}(M_o, 1)$
$M_{ooo}, M_{oo} \leftarrow \texttt{Split}(M_{oo}, 16)$
$K_o, K_i \leftarrow \texttt{Split}(K, 1)$
$\texttt{Reorder}(N_{ooo}, M_{ooo}, N_{oo}, M_{oo}, K_o, N_o, M_o, K_i, N_i, M_i)$
$F_{NM} \leftarrow \texttt{Fuse}(N_{ooo}, M_{ooo})$
$\texttt{Parallel}(F_{NM})$
$\texttt{Unroll}(F_{NM}, 512)$
$\texttt{Vectorize}(M_i)$

**Simplified auto-schedule where** $N = P = K = 1024$

$N_o, N_i \leftarrow \texttt{Split}(N, 32)$
$M_o, M_i \leftarrow \texttt{Split}(M, 256)$
$\texttt{Reorder}(N_o, M_o, N_i, M_i)$
$\hat{N} \leftarrow N_i, \hat{M} \leftarrow M_i$
Create Local Cache Buffer $\boldsymbol{D}$ of size $\hat{N} \times \hat{M}$
$\hat{N}_o, \hat{N}_i \leftarrow \texttt{Split}(\hat{N}, 1)$
$\hat{N}_{oo}, \hat{N}_o \leftarrow \texttt{Split}(\hat{N}_o, 16)$
$\hat{N}_{ooo}, \hat{N}_{oo} \leftarrow \texttt{Split}(\hat{N}_{oo}, 2)$
$\hat{M}_o, \hat{M}_i \leftarrow \texttt{Split}(\hat{M}, 8)$
$\hat{M}_{oo}, \hat{M}_o \leftarrow \texttt{Split}(\hat{M}_o, 4)$
$\hat{M}_{ooo}, \hat{M}_{oo} \leftarrow \texttt{Split}(\hat{M}_{oo}, 8)$
$K_o, K_i = \texttt{Split}(K, 4)$
$\texttt{Reorder}(\hat{N}_{ooo}, \hat{M}_{ooo}, \hat{N}_{oo}, \hat{M}_{oo}, K_o, \hat{N}_o, \hat{M}_o, K_i, \hat{N}_i, \hat{M}_i)$
$\texttt{ComputeAt}(\boldsymbol{D}, M_o)$
$F_{NM} \leftarrow \texttt{Fuse}(N_o, M_o)$
$\texttt{Parallel}(F_{NM})$
$\texttt{Unroll}(F_{NM}, 64)$
$\texttt{Vectorize}(\hat{M}_i)$

speedup of nearly 270× when compared to the unmodified computation without a schedule. The core difference in the auto-schedules produced for $C_1 = A_1B_1$ and $C_2 = A_2B_2$ is that the latter uses a temporary cache buffer to store intermediate results, as seen on Line 22. Other differences are the unroll factors chosen by the auto-scheduler, 512 for $A_1B_1$ as seen in Line 16 and 64 for $A_2B_2$ as seen in Line 34. Note that in this case, when applying transfer-tuning all the transformations being applied are still valid, since both computations are defined with the same initial loop structure and no transformation is strongly dependent on a given data size.

### 6.2.2 Kernel Classes

We briefly introduced the idea of kernel classes in Figure 6.2, where we can reuse auto-schedules between kernels if they contain the same operations. We now discuss the concept in more detail. Kernels are the units of computation which we pass to the auto-scheduler, for example, in DNNs a kernel may be a layer, or a set of layers that can be composed together. Often kernels can contain several operations, especially when they can be fused to encompass the same loop structure, such as in the case of many activation functions like ReLU. Fusion of loop nests is discussed more in Section 2.5.5.

In this chapter, we implement transfer-tuning using TVM, and defer to the graph partitioning into kernels generated by TVM for a given DNN model. We use TVM's partitioning, since the choices it makes are reasonable, such as combining activation functions and bias additions with larger layers such convolutional layers, and leads to state-of-the-art performance in many benchmarks [Che+18b]. Other partitioning schemes, and their relative advantages and disadvantages, are discussed more in Section 3.5.2. The purpose of having distinct kernels, rather than treating the whole program as a single function to be optimized, is that it allows the kernels to be optimized independently and in parallel. We define a *kernel class* to be a set of kernels that share the same sequence of operations, regardless of their data sizes. For example, one kernel class could be characterized by containing only convolutional layers, another by containing a composition of fully-connected and ReLU layers, etc.

In Table 6.1 we observe the characteristics of the kernels in ResNet18 [He+16], a DNN defined on the ImageNet dataset [Den+09]. Most kernels include a 2D convolutional layer, some of which include an activation function, or a bias or a skip-connection addition. However, we also observe some pooling layers and a fully-connected layer. Some kernels are repeated more than once in the model[1], as represented by the 'Use Count' column. However, for the purposes of auto-scheduling, repeated kernels are only tuned once, although a given kernel may be given a higher proportion of the search time. Overall, in ResNet18 we identify

[1]Note that in ResNet18 the 18 refers to the total number of convolutional and fully connected layers.

Table 6.1: Features of kernels in ResNet18, where *class* is a label for the operations in the kernel (seen in *TVM Ops*).

| ID | Class | input_shape | kernel_shape | TVM Ops | Use Count |
|---|---|---|---|---|---|
| **1** | A | $[1, 256, 14, 14]$ | $[512, 256, 7, 7]$ | conv2d_add | 1 |
| **2** | A | $[1, 128, 28, 28]$ | $[256, 128, 14, 14]$ | conv2d_add | 1 |
| **3** | A | $[1, 64, 56, 56]$ | $[128, 64, 28, 28]$ | conv2d_add | 1 |
| **4** | E | $[1, 3, 224, 224]$ | $[64, 3, 112, 112]$ | conv2d_bias_relu | 1 |
| **6** | E | $[1, 64, 56, 56]$ | $[64, 64, 56, 56]$ | conv2d_bias_relu | 2 |
| **7** | F | $[1, 64, 56, 56]$ | $[64, 64, 56, 56]$ | conv2d_bias_add_relu | 2 |
| **8** | E | $[1, 64, 56, 56]$ | $[128, 64, 28, 28]$ | conv2d_bias_relu | 1 |
| **9** | E | $[1, 128, 28, 28]$ | $[128, 128, 28, 28]$ | conv2d_bias_relu | 1 |
| **10** | F | $[1, 128, 28, 28]$ | $[128, 128, 28, 28]$ | conv2d_bias_add_relu | 2 |
| **11** | E | $[1, 128, 28, 28]$ | $[256, 128, 14, 14]$ | conv2d_bias_relu | 1 |
| **12** | E | $[1, 256, 14, 14]$ | $[256, 256, 14, 14]$ | conv2d_bias_relu | 1 |
| **13** | F | $[1, 256, 14, 14]$ | $[256, 256, 14, 14]$ | conv2d_bias_add_relu | 2 |
| **14** | E | $[1, 256, 14, 14]$ | $[512, 256, 7, 7]$ | conv2d_bias_relu | 1 |
| **15** | E | $[1, 512, 7, 7]$ | $[512, 512, 7, 7]$ | conv2d_bias_relu | 1 |
| **16** | F | $[1, 512, 7, 7]$ | $[512, 512, 7, 7]$ | conv2d_bias_add_relu | 2 |
| **ID** | **Class** | **input_shape** | **pool_size** | **TVM Ops** | **Use Count** |
| **5** | B | $[1, 64, 112, 112]$ | $[2, 2]$ | max_pool2d | 1 |
| **17** | C | $[1, 512, 7, 7]$ | $[7, 7]$ | global_avg_pool2d | 1 |
| **ID** | **Class** | **input_shape** | **weights_shape** | **TVM Ops** | **Use Count** |
| **18** | D | $[1, 512]$ | $[1, 1000]$ | fully_connected_add | 1 |

6 kernel classes, labeled A-F: with classes A, E, and F representing kernels featuring convolutional layers; B and C being max-pooling and average-pooling layer kernels; and D being the final fully-connected layer. We highlight that there are a variety of kernel classes featuring convolutional layers (class labels A, E, and F), with kernels of class E also including a bias addition followed by a ReLU activation, an overview of which we describe in Algorithm 5. The operations of class E can be further decomposed into lower level loop structures such as those describing the convolutional algorithm. This could be a direct convolution as shown in Algorithm 1, or some other primitive. On the CPU, TVM uses the *spatial pack* algorithmic primitive by default.

Along with the characterizing operations of its class, a given kernel is also defined by the data size of its inputs and weights. A schedule for a kernel would apply transformations to the code in a manner similar to the one seen in Algorithm 4, albeit with the transformations being applied to a more complex initial loop structure. Much like the GEMM example in Section 6.2.1, we observe that schedules can be reused between kernels of the same class in ResNet18, even if they are defined with different sizes. Thus, we can run transfer-tuning using a schedule of class E on another kernel of class E. In some cases the generated code may be invalid, for example, if the schedule defines a loop splitting factor which is larger than the loop itself. Attempting to apply a schedule from class E to another class, such as one defined by a fully-connected (dense) layer of class D, would always be invalid as the schedule would try to apply transformations to computations and loops not present in the computation. In principle, kernel classes which have some operations in common (e.g., classes E and F) could have their schedules adapted to allow a form of transfer-tuning across kernel classes. The exploration of this idea, as well as its impact on performance, are outside the scope of this thesis.

### 6.2.3 Applying Transfer-Tuning

Now that we have discussed the principles of transfer-tuning, including how it works and why kernel classes are relevant, next we look at how it performs using a real DNN model. We take the ImageNet definition of ResNet18 and use the auto-schedules of ResNet50 generated by Ansor, a model chosen because it is likely to have potential for a successful transfer-tuning due to its similar structure.

First, we evaluate each of the 18 kernels of ResNet18 with all compatible schedules of ResNet50. Figure 6.3 shows the inference time of all of these kernel/schedule pairs, running as distinct programs. The purpose of this evaluation is to give us insights into which schedules provide good performance improvements for each kernel. We also compare against the performance of the kernel when it uses the default schedule provided by TVM, which we refer to as ‘untuned’. Negative values represent kernel/schedule pairs which produced in-

valid code, i.e., our schedule transformations for transfer-tuning did not make the schedule sufficiently data-shape agnostic.

We observe that there are six kernel classes in ResNet18, with no schedules for classes F found in ResNet50. For class F we use the default schedule provided by TVM, represented as a black bar. For kernels of class A we have 4 compatible schedules to try from ResNet50, kernels of class E can be compiled with 16 possible schedules, and for kernels of classes B, C, and D note that we only have one compatible schedule each. For class E we observe that some schedules on some kernels produce invalid code, which we represent with a value of $-1$. There are 16 schedules of class E from which 7 produce invalid code for the kernels of ResNet18, hence we do not represent them in the graph. We also observe significant differences in inference time between schedules for some kernels, for example, for kernel 2 schedule A3 has over double the inference time of A4.

Taking the best schedule found via transfer-tuning for each kernel of ResNet18 and using them when compiling the full model we can observe a speedup of $1.2\times$, as shown in the leftmost bar of Figure 6.4a. The bar next to it shows that given the same search time Ansor can only achieve a speedup of $1.01\times$. Speedup is relative to the default generic schedules of TVM, which give good inference performance but do not perform any specific per-layer tuning. Search time for transfer-tuning means the time for testing each kernel of the target model with each valid schedule of the model chosen for transfer-tuning, and choosing the best in terms of inference time. This search time (around 1.2 minutes for ResNet18) is shown in Figure 6.4b. In addition, we compare how long Ansor requires to match our speedup, which in the case of ResNet18 is $4.8\times$ longer, or 5.8 minutes. This validates that transfer-tuning can work in the context of a full DNN model. However, we chose the model to tune with (ResNet50) arbitrarily. Thus, in Section 6.2.4 we give an overview of how we might select a model in a more systematic manner.

### 6.2.4 Model Selection

In Section 6.2.3 we demonstrated the core concepts of transfer-tuning using the ResNet18 model tuned using schedules from ResNet50. This was a reasonable choice, since the model architectures are similar (they belong to the same family of models). However, we need a more robust approach to select the model we will use for transfer-tuning. In this section we explore this design question using ten other models.

#### Selection heuristic

Table 6.2 shows a set of DNN models, their kernel classes, the frequency of kernels of each class, and the proportion of the untuned inference time that kernels of a given class represent.

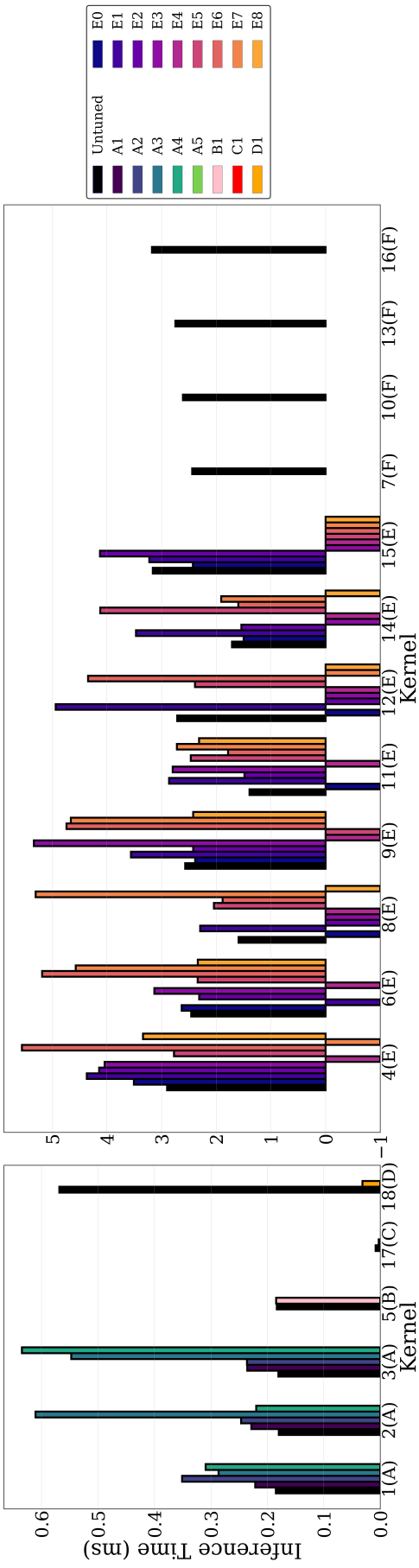



Figure 6.3: Inference time of ResNet18 kernels using ResNet50 schedules, with lower being better. Negative values denote schedules which produced invalid code.

For example, ResNet50 has 6 kernel classes representing 27 unique kernels (some kernels are repeated in the model), with kernels of class E representing the majority of the untuned inference time (67%), and kernels of classes B, C, and D representing a negligible proportion of the inference time. For brevity, we do not include details of each class, however expensive kernel classes tend to include convolutional layers or fully connected layers, while cheaper classes tend to contain operations such as pooling layers.

When choosing a model we want to maximize the likelihood that it will provide a good tuning for a target model. We hypothesized in Section 6.1 that perhaps similarities between the kernels (e.g., having the same convolutional kernel size, or similar memory footprint) could be used to predict how successful a given transfer-tuning for a kernel using a given schedule would be. However, in our initial study we did not find any feature which had strong predictive power. Thus, in this work we adopt a more coarse-grained approach which chooses a model to tune from based on the number of available schedules of a given class, and the proportional cost of that kernel class in untuned inference in the target model.

We define a selection heuristic which for a target model chooses a model to tune with which maximizes the number of available tuned schedules, giving preference for kernel classes that represent a higher proportion of the untuned inference time. To avoid models with very high numbers of schedules dominating the heuristic, we increase the influence of the untuned inference costs by squaring it and reduce the influence of the number of schedules in the tuning model by taking the square root. This scaling approach is arbitrary, and other scaling implementations are likely to exhibit similar behavior. Thus, we formulate our heuristic for a given target model $M$, which has a set of kernel classes $C$, as choosing a tuning model $T$ which maximizes the following:

$$\sum_{c \in C} P_c^2 \sqrt{|W_{Tc}|}, \tag{6.1}$$

where $P_c$ is the proportional cost of kernel class $c$ in $M$, and $W_{Tc}$ is the set of kernels of class $c$ in the candidate model $T$. Looking at Table 6.2, we can see for ResNet50 that the model which maximizes Equation 6.1 is GoogLeNet, and the two versions of EfficientNet maximize each other. For BERT and MobileBERT it is clear why they are chosen for each other, as both contain kernels of class Q (containing only a 'dense' fully-connected layer operation) representing 98% and 97% of the inference time respectively. These fully-connected layer operations are the subcomponents of the self-attention mechanism used in Transformer models, briefly discussed in Section 2.2.3.

However, this heuristic is not guaranteed to make optimal decisions. For example, the heuristic chose GoogLeNet for ResNet50 in part because it had a high number of schedules for class E, which represents 67% of ResNet50's untuned inference time. This means that for

each of the 16 kernels of class E in ResNet50 there are 49 schedules that may reduce the inference time. However, in theory, the 9 schedules of class E in VGG-16 may be better at reducing the overall inference time in ResNet50, even though there are fewer of them. This is not observed in our experiments, but should be noted as a possibility. The heuristic could be improved by accounting for this possibility if we had a better predictive model of which schedules may perform well for transfer-tuning, to reduce the likelihood of disregarding performant schedules. However, in this work we observe that the basic heuristic demonstrates sufficient improvements to validate the core ideas of transfer-tuning.

Table 6.3 shows transfer-tuning's maximum speedup by applying the top 3 models suggested by the heuristic. As we can see, the trend is that the best speedup is achieved by Choice 1, and the maximum speedup decreases with subsequent options. Note that for BERT and MobileBERT, every other model ties for second and third place and gives no speedup, hence we leave these entries blank (represented with '-'). This is because the only kernel class in BERT and MobileBERT shared by other models is class D, which represents less than 0.1% of their inference time. This is because both models are Transformers, whose most expensive operations come from self-attention layers, rather than convolutional layers.

### Alternative heuristics

In Figure 6.4 we provide an evaluation of the choices made by the heuristic described in Equation 6.1, demonstrating that it can outperform tuning the DNN models from scratch with Ansor. However, we could explore extensions to this heuristic which may allow greater exploitation of transfer-tuning, improving the speedup and/or reducing the search time. For example, the heuristic chooses a single model to transfer-tune from, however in principle we could use all the tuned schedules we have available in Table 6.2. We evaluate the impact of using all the available tuned schedules in Section 6.3.5. The caveat to consider is that this could translate into a very high number of schedules to evaluate, which would increase search time significantly. Thus, a more intelligent heuristic might discard schedules that are less likely to improve performance, and prioritize kernel classes by the potential improvement they could get, since we observe that the average speedups achievable by different kernel classes vary. We will explore this, and other potential extensions to transfer-tuning in future work, discussed more in Section 7.3.2.

## 6.3 Evaluation

In this section we evaluate the performance of 11 common DNN models, ResNet18 and the 10 more shown in Table 6.2, applying transfer-tuning using auto-schedules from the

Table 6.2: Kernel classes of DNN models, with the number of kernels of each class, and the proportion of the untuned inference time these kernels represent. Also shown is the model chosen for transfer-tuning.

| ID | Model | Kernel classes (number of kernels, percentage of inference time) | Tuning Model |
|---|---|---|---|
| **M1** | ResNet50 | **A** (4, 17%); **B** (1, 0%); **C** (1, 0%); **D** (1, 6%); **E** (16, 67%); **G** (4, 10%) | GoogLeNet |
| **M2** | AlexNet | **B** (3, 0%); **D** (1, 6%); **E** (5, 14%); **H** (2, 80%); **I** (1, 0%) | VGG-16 |
| **M3** | VGG-16 | **B** (5, 0%); **D** (1, 1%); **E** (9, 59%); **H** (2, 40%); **I** (1, 0%) | GoogLeNet |
| **M4** | MobileNetV2 | **A** (7, 15%); **C** (1, 0%); **D** (1, 24%); **J** (8, 32%); **K** (5, 15%); **L** (10, 14%) | EfficientNetB4 |
| **M5** | EfficientNetB0 | **A** (14, 9%); **C** (11, 4%); **D** (1, 12%); **K** (5, 9%); **M** (8, 39%); **N** (12, 27%); **O** (7, 0%) | EfficientNetB4 |
| **M6** | EfficientNetB4 | **A** (16, 11%); **C** (13, 3%); **D** (1, 10%); **K** (7, 14%); **M** (9, 39%); **N** (14, 23%); **O** (9, 0%) | EfficientNetB0 |
| **M7** | GoogLeNet | **B** (10, 1%); **C** (1, 0%); **D** (1, 4%); **E** (49, 95%) | ResNet50 |
| **M8** | MnasNet1.0 | **A** (7, 17%); **D** (1, 25%); **E** (9, 31%); **K** (5, 15%); **P** (12, 13%) | GoogLeNet |
| **M9** | BERT | **D** (1, 0%); **Q** (3, 98%); **R** (2, 2%); **S** (1, 0%); **T** (1, 0%); **U** (1, 0%); **V** (1, 0%) | MobileBERT |
| **M10** | MobileBERT | **D** (1, 0%); **Q** (4, 97%); **R** (2, 3%); **S** (1, 0%) | BERT |

**Algorithm 5** High level definition of a kernel class with a convolutional layer, bias addition, and ReLU activation.

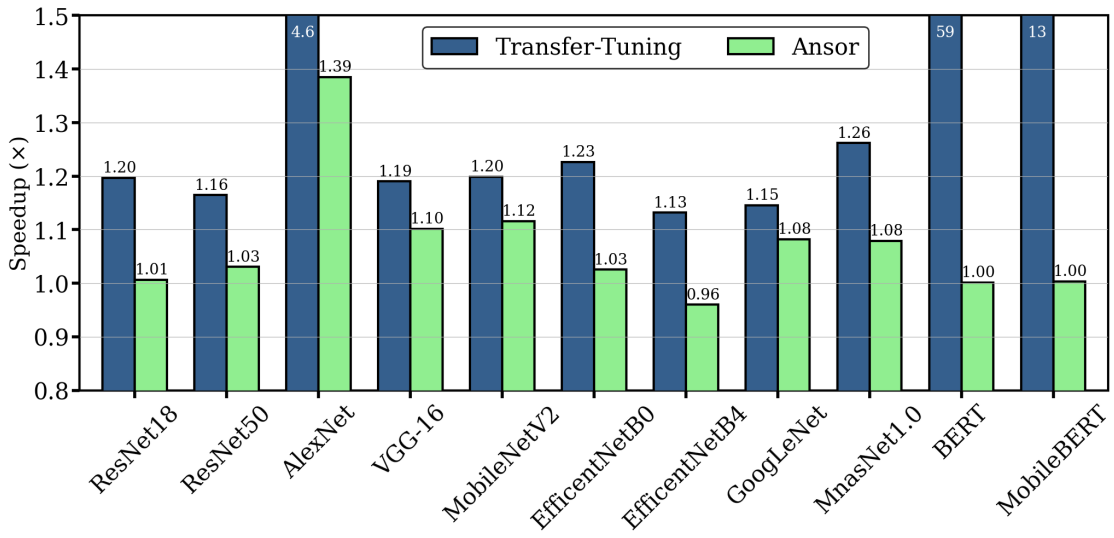
1: Define placeholders for inputs $X$, weights $W$, and bias $B$
2: Pad the input $X\prime \leftarrow \texttt{Pad}(X)$
3: $Y \leftarrow \texttt{Conv2d}(X\prime)$
4: $Y \leftarrow Y + B$
5: $Y \leftarrow \texttt{ReLU}(Y)$
6: Return $Y$

Table 6.3: Transfer-Tuning performance in terms of speedup using the top three choices from the heuristic.

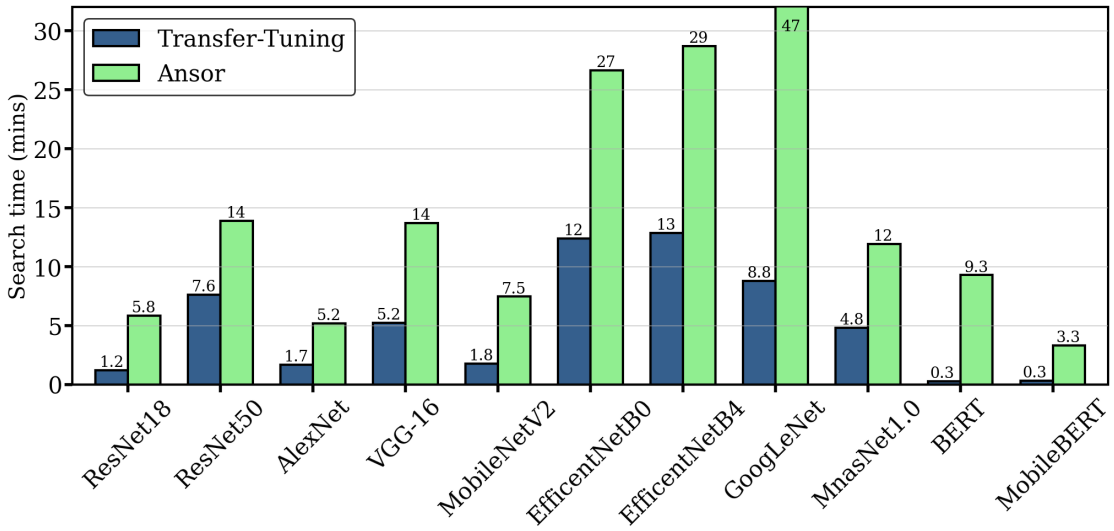
| Model | Choice 1 | Choice 2 | Choice 3 |
|---|---|---|---|
| ResNet50 | **M7** (*1.16*×) | **M8** (*1.0*×) | **M3** (*1.09*×) |
| AlexNet | **M3** (*4.6*×) | **M7** (*1.05*×) | **M1** (*1.03*×) |
| VGG-16 | **M7** (*1.19*×) | **M1** (*1.0*×) | **M8** (*1.0*×) |
| MobileNetV2 | **M6** (*1.20*×) | **M5** (*1.21*×) | **M8** (*1.19*×) |
| EfficientNetB0 | **M6** (*1.23*×) | **M4** (*1.08*×) | **M8** (*1.09*×) |
| EfficientNetB4 | **M5** (*1.13*×) | **M4** (*1.04*×) | **M8** (*1.03*×) |
| GoogLeNet | **M1** (*1.15*×) | **M3** (*1.04*×) | **M8** (*1.0*×) |
| MnasNet1.0 | **M7** (*1.26*×) | **M1** (*1.25*×) | **M3** (*1.18*×) |
| BERT | **M10** (*59*×) | -† | -† |
| MobileBERT | **M9** (*13*×) | -† | -† |

† Note that models M1-M8 tie for Choices 2 and 3 giving no speedup (i.e., 1.0×), hence we leave these entries blank (represented with "-").

model selected using the heuristic described in Section 6.2.4 on an edge class CPU. In Sections 6.3.3, 6.3.4, and 6.3.5 we explore transfer-tuning on an edge platform, varying the sequence length, and the impact of using a larger pool of schedules respectively.




(a) Speedup for transfer-tuning and Ansor given the same search time.




(b) Search time for transfer-tuning, and Ansor to match its speedup.

Figure 6.4: Transfer-Tuning results for several models on a server-class CPU (Intel Xeon E5-2620).

### 6.3.1 Experimental Setup

In addition to ResNet18 discussed in Section 6.2 we evaluate ten more DNN models. The first eight are CNNs defined on the ImageNet dataset [Den+09] for image classification, and the final two are Transformer [Vas+17] models for natural language sequence classification. The machine used to evaluate the models includes an eight core Intel Xeon E5-2620 CPU.

Auto-scheduling, compilation, and inference are executed on the CPU, using one thread per CPU core. For our baselines, we take the median inference time for each model over 10 runs, compiled using TVM's standard untuned schedules and the `-o3` flag.

**ResNet18** and **ResNet50** [He+16] have 18 and 50 layers respectively and consist of residual blocks. Each block contains two convolutional layers (that include between 64 and 2048 filters of size $3\times3$ and $1\times1$) and blocks are connected in a feed-forward manner.

**AlexNet** [KSH12] is a canonical CNN model and consists of 5 convolutional, 3 max-pooling, and 3 fully-connected layers. Note that newer DNNs often use fewer fully-connected layers to increase efficiency.

**VGG-16** [SZ14] is a CNN with 13 convolutional layers and 3 fully-connected layers. Some versions include batch normalization layers, but in TVM these are removed/fused for inference, see the discussion of *operator fusion* in Section 2.5.5.

**MobileNetV2** [San+18] is a lightweight model with 53 layers, many of which feature depth-wise convolutions which reduce the number of parameters and operations required. This makes it ideal for constrained edge devices.

**EfficientNet** [TL19] is a family of models with a focus on scalability. The architecture of the smallest model (EfficientNetB0) was found using neural architecture search (NAS) [ZL17], and the accuracy of the model is improved by applying a novel scaling method which changes the architecture to efficiently increase the number of parameters and operations. There are sizes ranging from B0-B7, and in this evaluation we use EfficientNetB0 and EfficientNetB4.

**GoogLeNet** [Sze+15] (or InceptionV1) is a 22 layer model containing 9 so-called 'inception' modules. This technique allows a deeper model to be trained more efficiently.

**MnasNet** [Tan+19] is a model architecture designed for edge devices such as mobile phones with an architecture generated using NAS. We evaluate the model using a depth multiplier of $1.0$, which contains 52 convolutional layers and a dense layer.

**BERT** [Dev+19] is a Transformer-based [Vas+17] model which excels in several NLP tasks. It contains 12 layers, where a layer is a so called 'transformer block'. We take a definition of BERT for sequence classification tasks.

**MobileBERT** [Sun+20b] is a compressed model inspired by the BERT architecture. It has 24 layers and around $4.4\times$ fewer parameters than BERT. For both BERT and MobileBERT which can take variable length input, we fix the sequence length at 256, with a discussion of the impact of varying the sequence length in Section 6.3.4.

### 6.3.2 Comparing Transfer-Tuning with Ansor

Figure 6.4 shows the results of running transfer-tuning across the 11 models, with each model being tuned using the model suggested by our heuristic described in Section 6.2.4. The only exception to this is ResNet18, which was used as in illustrative example in Section 6.2. As highlighted in Section 6.3.1, the results are collected on the Intel Xeon E5-2620 CPU. Figure 6.4a shows the speedup achieved by transfer-tuning and Ansor given the same search time. Figure 6.4b shows the search time required by transfer-tuning and how much time Ansor requires to match transfer-tuning's speedup.

For ResNet50 we observe a speedup of 1.16×, requiring 7.2 minutes to achieve. Given the same search time, Ansor gets a speedup of 1.03× and requires 1.8× as much search time to reach the same speedup. For AlexNet we observe a speedup of 4.6× which takes 1.7 minutes to achieve. The maximum search time for AlexNet is lower than ResNet50, as it has a smaller number of kernels. Ansor given the same search time gets a speedup of 1.39× and requires 3.1× more search time to reach the same speedup. For VGG-16, we observe a speedup of 1.19× which takes 5.2 minutes to achieve. To achieve the same speedup Ansor requires 2.6× as much time.

MobileNetV2 (tuned using schedules from EfficientNetB4) obtains a maximum speedup of 1.2×, which takes 1.8 minutes. Ansor given the same time gets a speedup of 1.12× and requires 4.2× more search time to reach the same speedup. We also observe that over half of its kernels (those of classes J and L), representing around 46% of the untuned inference time, are not transfer-tuned by EfficientNetB4 since it does not contain them. This suggests that there is further scope for improvement, for example, tuning using schedules from a model which included those kernel classes could increase the maximum speedup obtained.

EfficientNetB0 (also tuned with EfficientNetB4) gets a maximum speedup of 1.23×, which takes 12 minutes. The search time is much higher than most other models, since there are 58 kernels to evaluate with 764 unique kernel/schedule pairs. Ansor given the same time gets a speedup of 1.03× and requires 2.15× as much search time to reach the same speedup.

For EfficientNetB4 (tuned with EfficientNetB0) the situation is similar to EfficientNetB0 with a high search time of 13 minutes due to having 69 kernels, or 775 kernel/schedule pairs to evaluate. The speedup is 1.13× which is lower than EfficientNetB0, with Ansor requiring 2.23× more time to reach the same speedup. We observe that given the same time as transfer-tuning Ansor sees a slowdown by 0.96× compared to the baseline. This is not unexpected, as due to their stochasticity, auto-schedulers can sometimes hurt performance initially even if they eventually converge on an improved schedule.

For GoogLeNet, we observe a speedup of 1.15× which takes 8.8 minutes to achieve. Like the two EfficientNet models, a higher number of kernels (61) make the search time higher

than other models. Given the same time, Ansor achieves a speedup of 1.08× and requires 5.3× more time to reach the same speedup.

As the final ImageNet model, MnasNet1.0 (tuned with GoogLeNet) achieves a maximum speedup of 1.26×, taking 4.8 minutes. Given the same time, Ansor takes 1.08× and requires 2.5× as much time to achieve the same speedup.

Finally, BERT and MobileBERT see the most dramatic performance improvements of 59× and 13× respectively. In addition, they see the largest relative difference in search time required compared to Ansor, 33× and 10× respectively. These higher performance improvements are due to their inclusion of self-attention layers (see Section 2.2.3), which are computed as GEMMs. Compared to the convolution operation, the untuned version of GEMMs in TVM is less optimized, which means that it is easier to achieve higher performance improvements using tuning.

Overall, these results show that transfer-tuning can outperform the state-of-the-art Ansor auto-scheduler when given a limited amount of search time. Figure 6.1 shows that each model varies in the potential maximum speedup it can achieve, where we take the maximum speedup to be achieved using Ansor's recommended[2] 20,000 schedule variants (or iterations). Therefore, to compare the performance of our DNN models fairly we show the proportion of this maximum speedup transfer-tuning achieves in Table 6.4. On average using the arithmetic mean, transfer-tuning achieves 49.1% of Ansor's maximum speedup, with VGG-16 being the lowest with 17.7% and BERT being the highest with 88.4% of the maximum speedup. Compared to the search time required by Ansor to explore 20,000 schedule variants, transfer-tuning requires only 2.1% of this time on average. However, the values in Figure 6.4b give a more informative comparison showing that to achieve the same speedup as transfer-tuning, Ansor requires over 6.5× more time on average, with the lowest relative difference being for ResNet50 (1.8×), and the highest being for BERT (33×).

### 6.3.3 Exploring a Constrained Edge Platform

To further validate transfer-tuning we evaluate our models on a Raspberry Pi 4B, a common low-power edge device with an Arm Cortex-A72 CPU. Such devices represent another potential application of transfer-tuning, as they may not have the resources to undertake auto-scheduling themselves. Ansor allows edge devices to be connected to a more powerful server which runs auto-scheduling over RPC. However, this process can still be slow, may not always be available, and is not scalable to deployment across large heterogeneous fleets of devices, such as consumer smartphones.

[2] https://github.com/apache/tvm/blob/44549e623433dd10d9e97e442ef529fb44c46c14/gallery/how_to/tune_with_autoscheduler/tune_network_x86.py#L207

Table 6.4: Transfer-Tuning versus 20,000 Ansor iterations on the Intel Xeon E5-2620 CPU.

| Model | Speedup (%) | Search time (%) |
|---|---|---|
| ResNet18 | 49.20 | 0.48 |
| ResNet50 | 40.65 | 2.91 |
| AlexNet | 71.54 | 0.64 |
| VGG-16 | 17.69 | 1.41 |
| MobileNetV2 | 29.16 | 1.10 |
| EfficentNetB0 | 80.14 | 5.34 |
| EfficentNetB4 | 52.35 | 6.41 |
| MnasNet1.0 | 44.18 | 2.40 |
| GoogLeNet | 48.00 | 2.00 |
| BERT | 88.41 | 0.08 |
| MobileBERT | 18.96 | 0.10 |
| **Mean** | **49.12** | **2.08** |

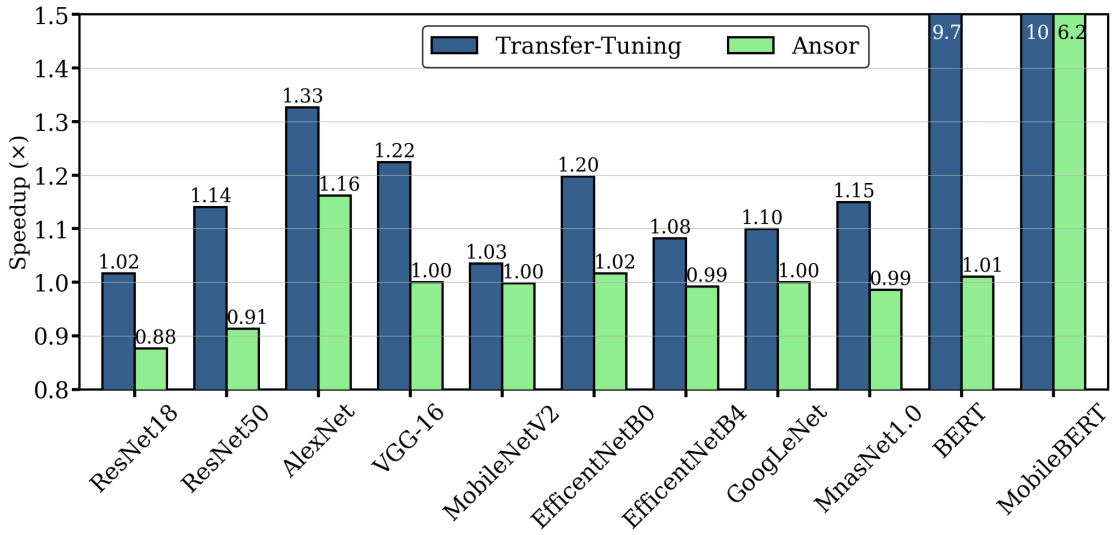
Figure 6.5a shows the speedups achieved on the Raspberry Pi 4, and Figure 6.5b shows the search time required. We observe that the relative differences between transfer-tuning and Ansor become exacerbated in terms of tuning time, with Ansor requiring over 10.8× as much time to reach the same speedup on average (using the arithmetic mean), which is significantly higher than the 6.5× difference observed on the x86 platform. In future work we will explore if transfer-tuning is viable between hardware platforms.

### 6.3.4 Varying Sequence Length

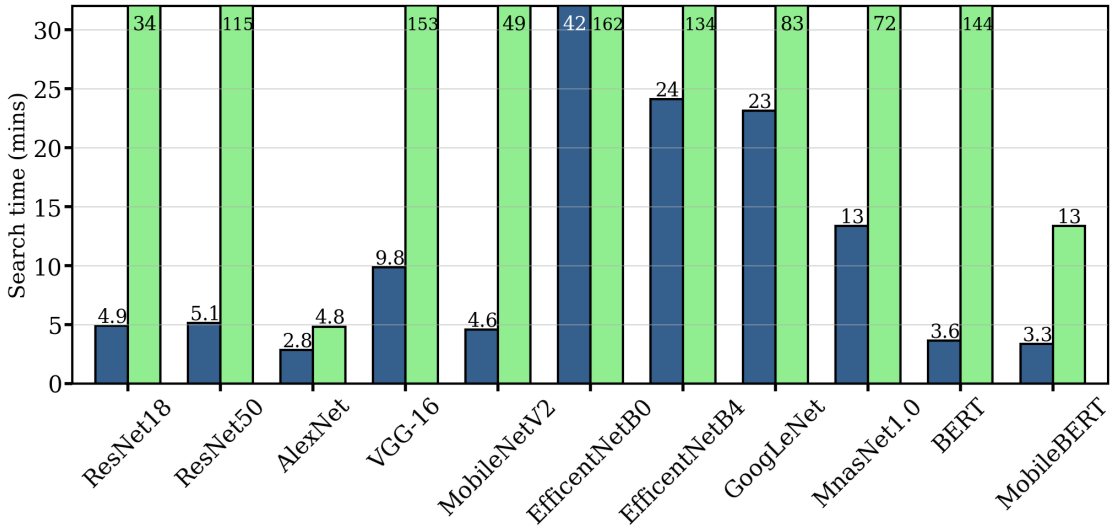
Unlike ImageNet models, which take input data of fixed sizes (224×224), sequence models such as BERT and MobileBERT can take variable input sizes, for example, a longer or shorter sentence. However, from the perspective of Ansor varying the input size means the whole model is different, since every single kernel has different data sizes to process. In principle, auto-scheduling could occur with a given dimension being specified as being dynamic. However, to date TVM has poor support for this and no support for this when tuning[3]. In addition, this could potentially lose out on some AOT optimizations by keeping the input size fixed.

Therefore, as an alternative view on transfer-tuning, we evaluate models of the same architecture but with different input sizes, namely BERT and MobileBERT for sequence lengths of 128 and 256. In Section 6.3.2 we evaluated these models for a sequence length of 256, therefore we must also tune versions of these models with sequence length 128. Figure 6.6 shows the results, with for example, 'BERT-128' representing the BERT model for sequence length 128 being tuned using schedules from 'BERT-256'.

[3]This changed with the introduction of DietCode [Zhe+22], which was not available during this study.




(a) Speedup for transfer-tuning and Ansor given the same search time.




(b) Search time for transfer-tuning, and Ansor to match its speedup.

Figure 6.5: Transfer-Tuning results for several models on an edge CPU (Arm Cortex-A72).

We observe that the improvement is greater applying tuning from a larger sequence length to a smaller sequence length, 3.3× as much improvement on average. We also note that compared to the results of Figure 6.4 (which shows BERT and MobileBERT with sequence length 256 being tuned with each other), BERT in Figure 6.6 gets less of a speedup (3.87× less) and MobileBERT gets approximately the same speedup (around 13×).

Varying input data sizes are common in sequence models such as BERT and MobileBERT. However, CNNs for computer vision often have publicly available models trained on a common dataset (e.g., ImageNet), which are then later fine-tuned using transfer-learning on a new dataset which may have a different input data size. Thus, this could represent another use-case for transfer-tuning, which we also leave for future work.

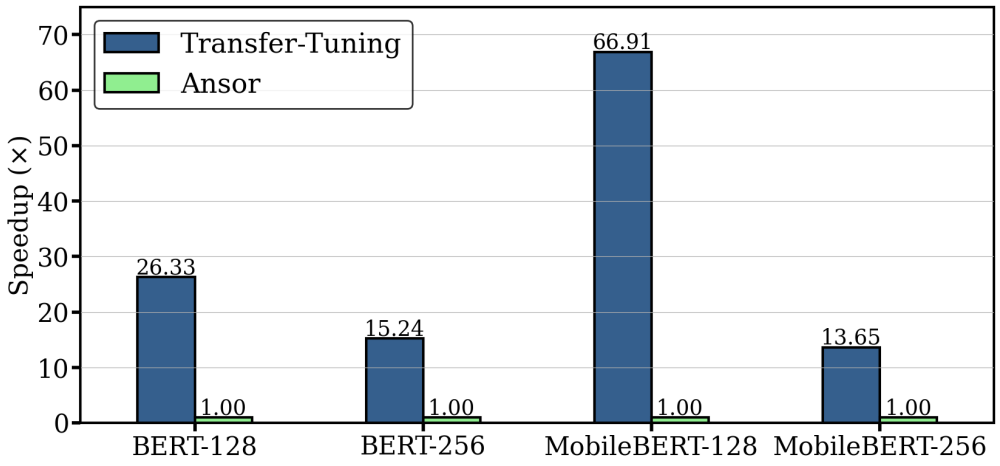



Figure 6.6: Transfer-Tuning varying the sequence length of BERT models (Intel Xeon E5-2620).

### 6.3.5 Alternative Heuristics

As discussed in Section 6.2.4, there is more than one way for transfer-tuning to select schedules, and this can be an implementation detail to suit the needs of a given use-case. Throughout the chapter we have demonstrated the core concepts and functionality of transfer-tuning by implementing 'one-to-one' model transfer-tuning, described by our heuristic in Section 6.2.4. This heuristic was devised from analytical observations about the features of models and their kernel classes, and has demonstrated speedups successfully, as shown in Table 6.3 and our wider results in Section 6.3.2. However, as discussed in Section 6.2.4, if we have tuned schedules available for a set of DNN models, as an alternative approach we could explore exploiting all of these schedules regardless of model.

Thus in this section we provide a brief evaluation of how transfer-tuning can be implemented to deal with this possibility. For each of our models described Section 6.3.1, we take the pool of schedules from Table 6.2 and make *all* of them available to the target model. Note that the

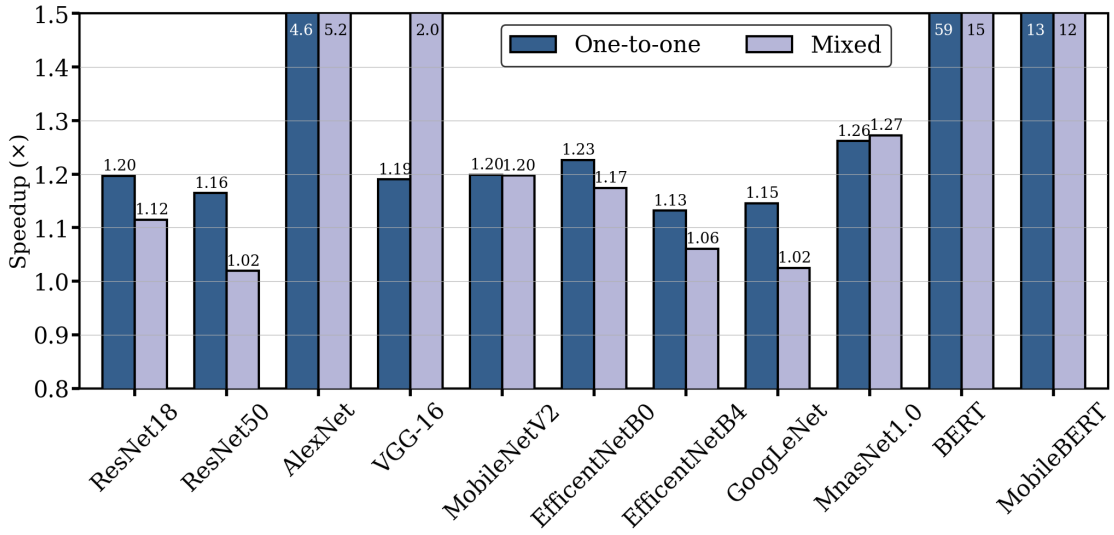
concept of 'models' is irrelevant to the pool, and for every kernel in the target model transfer-tuning picks the best schedule according to the standalone performance of its kernel.

We show the results of this evaluation in Figure 6.7. Our first observation in Figure 6.7b is that the search time increases by around 2× on average, with the highest being ResNet18 with a 5.34× increase. An increase in search time is expected, since we increased the number of kernel/schedule pairs we evaluate. However, because each model contains varying kernel classes, this increase varies between models. For instance, BERT and MobileBERT see a negligible increase in search time, as only kernels of class D are given new schedules to explore. In situations with many kernel/schedule pairs, we could reduce the search time by sampling a subset of schedules, either randomly or using some other selection heuristic.

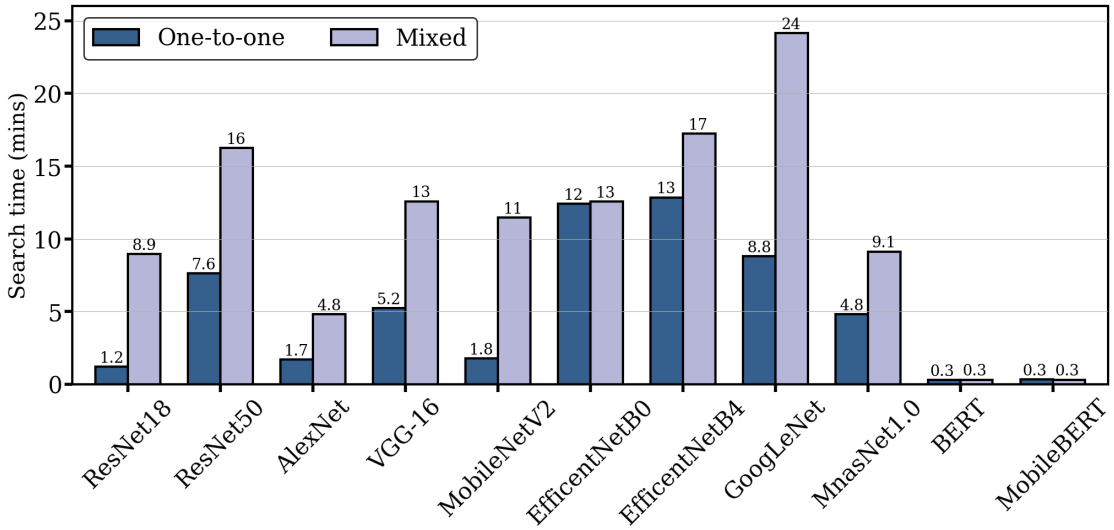
Regarding speedup, we can make several observations from Figure 6.7a. First we see that the maximum speedups achieved for AlexNet, VGG-16, and MnasNet1.0 increase when compared to the 'one-to-one' method. For MnasNet1.0 the improvement is modest (1.27× speedup compared to 1.26×), however for AlexNet and VGG-16 it is more significant: from 4.6× to 5.2× and from 1.19× to 2.0× respectively. This makes intuitive sense: either we use the best schedules found in the one-to-one approach, or we have new schedules in the pool that allow us to improve further. We observe that MobileNetV2 chooses the same schedules as before and provides the same speedup.

However, contrary to this intuition, we observe that seven models see a *reduced* speedup. Despite the fact we are selecting the kernel/schedule pairs with the lowest standalone inference time, our overall speedup when running the full model is lower than the initial one-to-one approach in Figure 6.4, even though the kernel/schedules used in the one-to-one case gave higher standalone inference times.

Our conclusion is that this is at least in part due the interaction between kernels when executing in a full model. Although the performance of kernels running in a standalone manner are a proxy to their performance in the context of a full tensor program, they do not capture all potentially relevant interactions. Our implementation of transfer-tuning assumes that the fastest kernel running as a standalone program will also be the fastest when running in the context of the full tensor program. This assumption of the independence of kernels is also used by Ansor, which tunes all kernels as standalone programs, and combines the best schedules together. It is clear that this assumption is sufficient for transfer-tuning to provide improvements over Ansor, however the results in Figure 6.7 demonstrate that there may be performance considerations of 'inter-kernel' relationships that are not captured by standalone kernel evaluation. For example, the output data of one kernel may be used as the input data for a subsequent kernel. The data access patterns of the first kernel will dictate the cache placement of the output data, which will impact the read times of the data when it is used in the second kernel. We can imagine an extreme case where the average reuse distance of the




(a) Speedup for transfer-tuning using schedules from a single model, and a mixed pool of models.




(b) Search time for transfer-tuning using schedules from a single model, and a mixed pool of models.

Figure 6.7: Transfer-Tuning using a schedule pool of several models on a server-class CPU (Intel Xeon E5-2620).

output/input data between kernels is at its maximum. This could significantly increase the inference time of the second kernel.

Therefore, awareness and exploitation of this dynamic may enable further optimizations for transfer-tuning and related methods. We leave a thorough exploration of inter-kernel relationships for future work, however approaches could include per-kernel profiling when running the full program, and evaluating kernels pairwise.

## 6.4 Summary

In this chapter we proposed *transfer-tuning* as a new approach to exploit similarities in tensor programs to reuse efficient schedules found via auto-scheduling. We have discussed how transfer-tuning is feasible in a compute/schedule programming paradigm, and explained the key components necessary to accelerate tuning of a full model. We defined an implementation of transfer-tuning in TVM and evaluated the performance on eleven models on a server-class x86 CPU, achieving 49.1% of the maximum speedup achieved by the Ansor auto-scheduler on average (using the arithmetic mean), and with Ansor requiring over 6.5× as much time to match our performance. We also evaluated transfer-tuning on a constrained edge device, the Raspberry Pi 4B, showing that the gap between Ansor and transfer-tuning is exacerbated, with Ansor requiring over 10.8× as much time to match our performance.

Transfer-tuning leverages the ideas of schedule based programming paradigms such as Halide and TVM, as well as auto-scheduling introduced by Ansor. Other works have exploited similarity between programs to make compilation optimization more efficient. However, transfer-tuning's novelty comes from leveraging this workload similarity in the domain of schedule-based tensor compilers to reduce search costs. In future work, we will explore the impact of across-kernel interactions, the viability of applying transfer-tuning across similar kernel classes, and transfer-tuning across hardware devices.

# 7 Conclusions

This thesis presents new techniques which exploit and develop the emerging tensor compiler paradigm as it applies to the Deep Learning Acceleration Stack (DLAS). These techniques help to address the challenges identified in Section 1.2. Chapter 4 gives a vertical slice of DLAS, demonstrating how a variety of techniques from across the stack can be combined, the engineering effort required, and the highly inter-dependent interactions between techniques that can occur. Chapter 5 addresses a specific issue observed from a Model Optimization technique (grouped convolutions) not achieving the expected inference time speedups given the reduction in MACs, and developing a novel algorithm and compilation pipeline to better realize this performance improvement. Chapter 6 looks more deeply at the tensor compiler itself, developing a new technique (transfer-tuning) to reduce the search time costs associated with auto-scheduling.

This chapter is structured as follows: Section 7.1 summarizes the main contributions of this thesis, Section 7.2 presents a critical analysis of the work, Section 7.3 describes future research directions, and finally Section 7.4 provides concluding remarks.

## 7.1 Contributions

The challenges identified in Section 1.2 highlight types of friction which researchers and developers may encounter when exploring across-stack acceleration of DNN workloads. This section summarizes the main contributions of this thesis and how they tackle these challenges, making a strong argument for the need for both across-stack optimization of, and a compiler-centric perspective for DNN acceleration.

### 7.1.1 Exploration of Varying DLAS Parameters

The Deep Learning Acceleration Stack as described in Section 1.1 is a key concept used throughout this thesis. There are a variety of parameters and techniques that can be applied at each layer of DLAS, with more emerging frequently. However, many of these parameters and techniques require support from other layers of DLAS to realize their potential, for

example, exploiting pruning (a Model Optimization technique) requires appropriate sparse algorithms and data formats which best utilize the target hardware. Chapter 4 explored some interactions across DLAS by describing the experience of running a perturbation study with a small number of parameters being varied at each layer. Exploring multiple DLAS parameters simultaneously exposes the lack of maturity of existing toolchains, and the need for significant engineering work to more easily allow more effective DSE. Additionally, given the combinatorial explosion from the wide range of DLAS parameters available, it is difficult to make definitive claims about the effectiveness of a given acceleration technique. The study itself is provided in Appendix A, and generated a large number of results and exposed several instances of suboptimal performance under some combinations of parameters, as well as unsupported configurations (such as auto-scheduled sparse computations on the GPU).

The results were collected using a modified version of TVM, further improving TVM's suitability as a way to perform design space exploration (DSE) for tensor programs, for example, by allowing the algorithmic primitive to be more easily changed. These modifications will be upstreamed to TVM upon publication, and go some way towards tackling the efficient DSE challenge described in Section 1.2.3, by exposing new optimization choices to the compiler. However, the cost of exploration is still high, both in terms of initial setup costs, and the collection of experimental results, as addressed by transfer-tuning (see Section 7.1.3), with the broader problem discussed more in Section 7.2.1.

### 7.1.2 Using a Tensor Compiler to Unlock the Potential of an Underserved Model Optimization Technique

Grouped convolutions have been adopted in a range of DNN architectures (such as EfficientNets [TL19] and Moonshine [CGS18]), which allows the number of MAC operations and parameters in a model to be linearly scaled down, at the cost of some accuracy loss (see Section 2.3.3). However, the performance improvement that grouped convolutions provides was mostly theoretical, with real software implementations of grouped convolution showing significant slowdowns compared to the expected performance improvements in terms of inference time, given the theoretical reduction in the number of MACs. In the worst case, in our initial evaluation in Section 5.1, we observed a model using grouped convolutions which was around 10× slower than the expected speedup, with none of the DNN frameworks we evaluated exhibiting behavior close to the expected performance scaling. This identification of an unrealized gain is an example of addressing the challenge described in Section 1.2.1, by comparing the expected speedup against actual performance. This of course requires having a realistic expectation of the impact of a given optimization, and being able to compare fairly against baselines and other solutions.

Motivated by this performance gap, Chapter 5 develops a novel algorithm to accelerate

grouped convolution, implemented using TVM, with optional compiler auto-tuning support. The inference time we achieved on the CPU outperformed PyTorch and TensorFlow Lite by 8× and 4× on average respectively (using the arithmetic mean), and improved against TVM's existing approach by 3.4× on average. The solution (GSPC) also leveraged auto-tuning to search for improved schedule parameters for each layer of a given DNN on a given hardware platform. Auto-tuning parameters included the tile sizes used for different stages of the algorithm and if loop unrolling was beneficial, and if so to what unrolling depth. The chapter tackles the challenge described in Section 1.2.2 by exploiting across-stack interactions to better realize the unrealized gains from grouped convolutions. The problem was tackled in an across-stack manner by developing a new algorithm to exploit this Model Optimization technique, as well as using Systems Software in the form of TVM to develop an optimized schedule, further accelerated by leveraging auto-tuning for specific hardware platforms. Due to its performance improvements, GSPC was accepted as the new default implementation of grouped convolution for CPUs in upstream Apache TVM in 2021.

### 7.1.3 Reuse of Auto-schedules to Accelerate Tensor Programs with Reduced Search Costs

The promise of auto-scheduling systems, as described in Section 2.5.7, is to tailor code for specific DNNs and hardware platforms. This can allow DNNs to have significantly improved inference times, or some other semantic-preserving target such as energy usage. Practically, these systems have shown impressive results, with Ansor outperforming state-of-the-art inference times on Intel CPUs, Arm CPUs, and Nvidia GPUs by up to 3.8×, 2.6×, and 1.7×, respectively [Zhe+20a]. However, these improvements do not come without a cost, namely the high search times required. With the number of program variants recommended by the Ansor developers, optimizing a typical DNN can take several hours when using a server-class CPU. These search times can increase significantly as the device becomes more constrained, for example, over 140 hours were required for some of our models in Chapter 4. These problems are typical of the efficient DSE challenge described in Section 1.2.3, where we have an acceleration technique for DNNs which comes with high search costs to achieve an adequately performant solution.

This could be considered a small price to pay for significant acceleration, especially when taken in the context of the high costs of training. However, there are two main issues with this conclusion. First, we may be designing our neural architecture with a perspective of optimized deployment costs, either by hand, or with NAS. We need an accurate estimate of what the deployment costs of our architecture will be, and high overheads in optimizing the candidate architecture using auto-scheduling makes this impractical. Secondly, we may be deploying our DNN to a wide range of hardware platforms, for example, as part of an

application run on end-user devices such as laptops and smartphones. It is unreasonable to expect users to dedicate hours of tuning time on their device when the application is installed, and application developers will not necessarily have access to every platform that users have.

Chapter 6 proposes a technique which can be used as a building block to tackle both issues. Instead of tuning from scratch for every new tensor program, we keep a cache of previously optimized schedules for other tensor programs, and adapt the schedules so that they are compatible with our new program. We then explore which pre-tuned schedules are the best for our new program. We describe the formulation of our technique 'transfer-tuning', with a heuristic that selects a single DNN model from a pool of pre-tuned models to use as a source of schedules. We show that compared to auto-scheduling from scratch, Ansor requires 6.5× more search time on average (using the arithmetic mean) to match transfer-tuning's speedups on a server-class Intel CPU. We also evaluate transfer-tuning on a constrained edge CPU and observe that the differences in search time are exacerbated, with Ansor requiring 10.8× more search time than transfer-tuning on average.

The maximum speedup achievable by transfer-tuning is capped by the quality of the pre-tuned schedules available. Therefore, it is unlikely that transfer-tuning can achieve the maximum possible speedup, without using other techniques. However, transfer-tuning still achieves speedups against TVM's untuned baseline (which itself is competitive against other state-of-the-art implementations [Che+18b]), with those speedups being achieved much faster than Ansor. If we let Ansor auto-schedule for a longer period of time, then we would expect that it will always outperform transfer-tuning. However, if we take the best speedups that Ansor can provide and the time required, overall transfer-tuning is able to achieve up-to 88.4% of Ansor's speedup, in at most 6.4% of the search time. On average, across our 11 models, transfer-tuning achieves 49.1% of the maximum speedup in 2.1% of the search time. By achieving high speedups at significantly reduced search costs, *transfer-tuning* tackles the efficient DSE challenge described in Section 1.2.3.

## 7.2 Critique

This section presents a critical analysis of the techniques introduced in this research. It will contextualize the strengths and weaknesses of the contributions, and evaluate its relevance in the context of contemporary and subsequent literature.

### 7.2.1 Limited Systematic Full-stack Optimization

Chapter 4 provides a study highlighting the difficulties of systematic exploration of parameters across DLAS. However, the work only discusses a characterization study which chooses

a small number of parameters at each layer of DLAS, and does not perform the next step, i.e., responding to and exploiting the observations to further accelerate the workloads, and making generalizable conclusions about the effectiveness of the techniques under study. This means that although the work highlights that across-stack interactions *can* occur, and that optimization techniques are not guaranteed to bring accelerated performance without wider support, it does not address the problem of how each technique can be better realized. Solving this for particular observations could be publications in and of themselves, since each may require a unique and novel solution.

In addition, there are several cases in the study where some combination of parameters did not work. For example, DenseNet161 did not work in TVM with 8-bit quantization and auto-scheduling sparse computations did not work on the GPUs. A bug fix was pushed to upstream TVM to support 8-bit quantized EfficientNet[1], and the sparse algorithms for 2D convolution were all written from scratch. However, this still gives an incomplete picture of the design space, since some of these problems appear to be tooling weakness and engineering problems, rather than fundamental challenges. It could be argued that as a characterization, missing results highlight wider issues in the existing infrastructure, however this still diminishes the contribution.

The issue of limited full-stack optimization is also seen in Chapters 5 and 6. Chapter 5 explores how a Model Optimization technique (grouped convolutions) can be accelerated by appropriate algorithm design, and leveraging an optimizing tensor compiler. However, the chapter leaves many unanswered questions regarding other aspects of DLAS, such as the impact of data formats, other devices such as GPUs, or what level of grouped convolution maximizes hardware utilization and accuracy while minimizing inference time. As a result, although Chapter 5 achieves the goal of tackling the unrealized gains of grouped convolutions, it would be stronger if it explored more dimensions of DLAS. Similarly, in Chapter 6, the focus is on improving the efficiency of DSE for generating efficient code for a given DNN and hardware platform. Questions about how this can be combined with other techniques such as NAS and algorithmic primitives are left unexplored.

However, it is clear that exploring all of these DSE questions is prohibitively expensive, as the challenge described in Section 1.2.3 highlights. Thus, for robust co-optimization search to be explored, the costs must be further reduced at individual layers and intersections of DLAS. Focusing on particular sub-problems can lay the foundation of more holistic acceleration. With the overarching goal of across-stack acceleration, this thesis makes contributions to make holistic acceleration more practical. However, it is still unfortunate that the studies did not combine more features of DLAS to address this goal.

[1] `https://github.com/apache/tvm/pull/14286`

### 7.2.2 Relevance of Grouped Convolutions on CPUs

Chapter 5 developed and evaluated a new algorithm and auto-tuning configuration called GSPC, on three CPUs. However, in the face of increasingly varied hardware for executing DNNs, as discussed in Section 3.6, is a CPU analysis relevant? Increasingly, we observe DNNs being deployed to GPUs and other hardware accelerators. Many devices, such as microcontrollers and other IoT platforms only contain CPUs. However, newer generations of devices are increasingly including hardware accelerators specifically for DNNs.

However, it can be argued that CPUs will continue to be a widely used compute platform, and grouped convolutions are especially relevant on more constrained devices that are less likely to have powerful accelerators. Therefore, an optimized CPU implementation of grouped convolution brings value, especially when the development cycle and lifetime of hardware is significantly longer than machine learning solutions. It should also be noted that by implementing the algorithm in TVM's compute schedule language, it can be evaluated on other platforms such as GPUs.

### 7.2.3 Upper Limits of Transfer-Tuning

Chapter 6 presents *transfer-tuning*, an approach for reusing pre-tuned auto-schedules on new DNNs on a given hardware platform. Given a set of pre-tuned auto-schedules, we can achieve speedups for new tensor programs with a reduced search time. However, the maximum speedup that transfer-tuning can achieve is limited by the quality of the pre-tuned schedules available. If we want to improve the performance, we must introduce more pre-tuned schedules. As Section 6.3.5 shows, evaluating additional schedules could increase the search time required, and thus reduce the relative efficacy compared to tuning from scratch. Alternatively, we could explore fine-tuning on our transfer-tuned schedules. Based on preliminary exploration, it is currently unclear if fine-tuning transfer-tuning is a viable research direction, and this thesis does not contribute any results demonstrating this.

A recent work that cites transfer-tuning [Tol+23] showed that they could significantly reduce the search space for auto-scheduling convolutional layers using techniques from polyhedral compilation. This reduced search costs by at least an order of magnitude without harming the final speedup that auto-scheduling achieves. Reduction of these search costs also reduces the motivation for approaches such as transfer-tuning. However, in the face of increasingly large and varied DNN architectures, DNN deployment to constrained hardware, and significant scope for reducing transfer-tuning's search time, transfer-tuning is likely still relevant. Future work on transfer-tuning should focus on reducing search times and increasing speedups to ensure that the technique continues to bring valuable benefits. Transfer-tuning could also

leverage similar techniques from polyhedral compilation to reduce its search times, and potentially improve the available schedules.

Another issue highlighted by transfer-tuning, and especially in Section 6.3.5, is that assuming that kernels are independent is too strong an assumption. Optimizing DNN operations as separate programs makes sense in principle, however it is clear that there is an interdependence between kernels in the form of their inputs and outputs. Some works in the algorithm selection literature (see Section 3.4) already account for this in the context of data format transformation costs. However, transfer-tuning does not leverage these techniques, and thus selects its schedules using the independence assumption, losing out on potential speedups.

## 7.3 Future Work

Throughout this thesis, we have highlighted how the layers of DLAS can expose significant scope for across-stack acceleration, or may impede acceleration techniques from other layers if an across-stack perspective is neglected. This work has made contributions in these topics, while centering the tensor compiler as a key component to manage this increasing complexity. This section outlines some promising directions for future research, in part enabled by contributions of this thesis.

### 7.3.1 Grouped Convolutions Exploration

We could investigate the performance of GSPC on the *big.LITTLE* architecture and embedded GPUs (e.g., Arm Mali) present in many edge devices. Additionally, we could explore translation of other algorithmic primitives for standard convolution into grouped variants, e.g., GEMM and Winograd convolution, and investigate their performance trade-offs across different benchmarks and devices. Auto-scheduling, which emerged after the initial development of GSPC, could push the performance even higher relative to the hand-optimized and auto-tuned schedule presented in Chapter 5. Finally, considering energy consumption is also an area for future research, since grouped convolutions are particularly relevant in scenarios where power usage is an important factor, for example, IoT deployment.

### 7.3.2 Improved and Expanded Transfer-Tuning

Transfer-tuning in its proposed form has clear applicability, however there is also significant scope for further improvement. The two main dimensions of what makes for a successful transfer-tuning are the search time required and the speedup achieved for a given model on a given hardware platform. From the search time perspective, the lower the search time, the

more effective transfer-tuning is. The implementation of transfer-tuning described in Chapter 6 compiles every available schedule with every candidate kernel, and evaluates it on real hardware. Although this search process is cheaper than auto-scheduling from scratch, we could further reduce the search time if we developed approaches to pre-screen schedules, discarding schedules which are unlikely to give good speedups for a given kernel. These approaches could look at features such as data shapes, and find a predictive mapping for speedup estimation. In addition, there may be scope to enable 'across-class' transfer-tuning, where we allow reuse of schedules on kernels which do not contain the exact same operations. For example, if our pre-tuned schedule was for a kernel containing a convolution followed by a ReLU, transfer-tuning may still be successful if a different activation function is used. By relaxing the 'same kernel class' constraint, we could increase the number of potential schedules available to transfer-tuning.

From the speedup perspective, a higher speedup is more desirable, and the maximum speedup that we can achieve is inherently limited by the pre-tuned schedules that we have available. The maximum speedup is also limited by our hardware, as illustrated by the roofline model shown in Figure 1.2. However, as discussed in Sections 6.3.5 and 7.2.3, we make the simplifying assumption that kernels are independent sub-programs, rather than interconnected parts of the same tensor program. Future work could explore this dimension, potentially comparing the performance of *pairs* of kernels, and then finding a set of schedules which finds a lower overall inference time. In addition, if we reduce the search time sufficiently, it could become more attractive to consider pre-tuned schedules from a larger pool, which would increase the probability of finding a good schedule for a given kernel.

### 7.3.3 Holistic Compiler-centric NAS

As discussed in Section 7.2.1, this thesis provides an across-stack characterization of some common techniques present in DLAS, exposing some of the challenges inherent with achieving accelerated performance. It also tackles some specific problems in subcomponents of the stack, such as slow grouped convolutions and expensive auto-scheduling.

In future work, accelerated deep learning solutions will be achieved by designing and efficiently exploring an increasingly large design space of techniques from across DLAS. NAS (as discussed in Section 3.2.2) is one of the highest levels of automated DSE in the contemporary DLAS literature, and demonstrates a relatively unique attribute of deep learning, namely that we can develop radically different program structures to solve similar problems. Coupled with the position of this thesis that tensor compilers are increasingly important for DNN acceleration, NAS techniques which combine compiler approaches along with other aspects of DLAS such as algorithm choices and hardware features are a rich vein for future work. As Section 3.2.2 highlights, some works have already begun to explore narrow slices

of this, however adding more parameters from across DLAS may enable even more efficient and effective solutions. However, the challenge of efficient DSE from Section 1.2.3 remains, and thus future work should also endeavor to manage the new bottlenecks which may emerge as the design space increases. Tensor compilers can help significantly with this, since if they are designed in a composable manner, they should be able to more gracefully manage the combination of multiple DLAS techniques. These combinations may be as yet unexplored in the literature, and poorly supported by more rigid infrastructure such as kernel libraries. Therefore, future tensor compilers could help the design of machine learning systems break out of their 'rut' as described by Barham and Isard [BI19], where highly optimized but inflexible kernels discourage exploration of more novel DNN architectures.

### 7.3.4 Improved Compiler-driven Mixed-precision

As highlighted in Section 3.3.2, mixed precision quantization is where we vary the data-type used at different points in a DNN. For example, we may use `float32` in earlier layers and `int8` in later layers. Mixed precision is an active area of research, since it can find a better balance between accuracy maintenance and inference acceleration. Some mixed precision works have already started to approach the problem from a systems perspective [Wan+19b]. However, the tensor compiler could play a significant role in future work. For instance, much like the data layout transformation costs associated with varying algorithmic primitives, there may be overheads going from one data-type to another. In addition, as shown in Chapter 4, the most performant algorithm can also vary depending on the data-type used, and a given data-type may not necessarily provide a speedup. However, unlike the exploration of different algorithms per-layer [AG18; PPB19; Wen+19], varying the data-type may also have consequences for model accuracy. A tensor compiler could help generate efficient code for these cases, as well as exposing information about the design space to the developer. However, a design challenge of this work is how to efficiently explore this space, especially regarding accuracy estimation.

### 7.3.5 Compiler-centric Exploitation of Heterogeneous Hardware

As highlighted in Section 3.6, the improvements in computing hardware due to Moore's law and Dennard scaling are slowing down, meaning that we cannot necessarily expect future iterations of general purpose hardware to improve as significantly as it did in the past. This also means that to accelerate workloads such as DNNs, increasingly specialized and co-designed hardware will be required. This requirement and trend has been described by some as a 'new golden age for computer architecture' [HP18], i.e., computer architects will have new challenges and opportunities for novel designs. Similarly, it has been said that we are

entering a 'new golden age of *compiler* design' [Lat21], one which will require increased exploitation of domain-specific knowledge and co-design. As both observations describe, there is significant scope for impactful work in domain-specific hardware and compilers over the coming years. At their intersection, as the variety of computing hardware available grows, compilers are well placed to manage this increasing heterogeneity. If a platform includes multiple hardware accelerators with varying properties and capabilities, how will the compiler decide what parts of the DNN to run on which accelerator?

Some works described in Section 3.5, such as Collage [Jeo+23] and Neurosurgeon [Kan+17], have begun to explore this question for CPUs and GPUs. However, future work may need to tackle the problem with an even wider range of devices, including reconfigurable hardware (see Sections 3.6.2 and 3.7.4) which comes with its own challenges around generating efficient mappings, as well as efficient management of data-transfers between devices. It has also been argued that the isolation of layers of the systems stack can lead to so-called 'hardware and software lotteries' [Hoo21], where certain workloads are favored in terms of practicality by the quirks of the most optimized hardware and software libraries available. A similar dynamic has been observed with DNNs [BI19]. This can have a significant influence in the direction technology develops, and may come with high opportunity costs if not adequately modeled. More flexible and extensible compiler infrastructure and reduced barriers for hardware design may help alleviate this problem.

## 7.4 Summary

This thesis explores and develops across-stack deep learning acceleration techniques, enabled by tensor compilers. The outcomes demonstrated in this thesis open new lines of research, in the already fertile landscape of DNN acceleration. The preliminary results are promising, with some of the contributions of this thesis (such as GSPC) being integrated into production ready open source systems. By further developing the idea of the Deep Learning Acceleration Stack (DLAS), the work underlines a conclusion which many practitioners in the machine learning and systems communities have also observed: to be fully effective, DNN acceleration techniques require careful co-design, combining approaches from multiple domains. The overall position of this thesis is that the tensor compiler will increasingly become the center of DLAS, and act as key tool in realizing this co-design. As innovations continue to emerge from both machine learning and systems communities, the increasingly large design space needs to be managed in a scalable and efficient manner. Tensor compilers are well suited for this task, since they can represent the logical requirements of different techniques, thus helping ensure that generated code is coherent. They also provide their own advantages such auto-scheduling code, and reducing the required binary size compared to a

general purpose library. However, tensor compilers are still in their infancy, and although they are increasing in popularity, it will require continued collaboration and innovation from the tensor compiler community and beyond to fully realize their potential.

# A DLAS Characterization Study

## A.1 Experimental Setup

Our experiments are intended as a vertical slice of DLAS, which demonstrates the design choices available and interactions that occur. We have sought to follow best practices in experimental design where possible. However, since this work is a characterization study, we do not optimize for every factor that may influence a given result. For example, our results comparing convolutional primitives are not necessarily a definitive claim on which algorithm is the best in a given circumstance. That would require focused optimization factoring in all layers of the stack, which would necessitate solving all three challenges in Section 1.2 simultaneously. Additional techniques from the literature that could be used to further optimize performance, as well as discussion of the trends and issues observed in the results, are given in Section 4.1.

### A.1.1 Models & Neural Architectures

We investigate two datasets for image classification: CIFAR-10 and ImageNet, both introduced in Section 2.1. For CIFAR-10, we use model definitions from an open-source PyTorch-based library [kG23], which we train from scratch. We consider four architectures based on ResNet18 [He+16], MobileNetV1 [How+17], MobileNetV2 [San+18], and VGG-16 [SZ14]. ResNet18 and VGG-16 are larger models, and MobileNets V1 and V2 are designed to be more resource efficient. To train the models we used SGD to minimize the cross-entropy loss (averaged across all data items), which penalizes the network for making incorrect classifications. We used a 1cycle learning rate scheduler [ST18], with momentum 0.9, weight decay $5\times10^{-4}$, and an initial learning rate of $5\times10^{-2}$, trained for 200 epochs.

For ImageNet, we use pre-trained models from the TorchVision repository [mc16]. We consider four architectures: DenseNet161 [Hua+17], EfficientNetB0 [TL19], ResNet50 [He+16], and MobileNetV2 [San+18]. ResNet50 and DenseNet161 are larger models, whereas MobileNetV2 and EfficientNetB0 are designed to be more resource efficient. The models are

pre-trained, with the training configurations described in the TorchVision documentation[1].

### A.1.2 Model Optimizations

We explore three approaches to compression: 1. global L1 unstructured pruning, which we call 'weight pruning' (Figure 2.6b); 2. global L1 structured pruning over convolutional channels, which we call 'channel pruning' (Figure 2.6c); and 3. data-type quantization exploring `float16` and `int8` types (Figure 2.6d).

Our pruning techniques explore the impact of increasingly higher compression ratios, thus for our evaluation of inference time and for each model and pruning technique, we select the pruning level which has the highest compression ratio before a significant accuracy drop. To implement our pruning, we leverage the PyTorch Lightning library [FT19], a wrapper of PyTorch which simplifies the pruning interface. We apply pruning iteratively, starting with the pre-trained unpruned dense models, and prune a fraction of the weights. To reduce accuracy loss, at each pruning step we apply fine-tuning, where we use training data to adjust the non-pruned weights to compensate for lost accuracy. For weight pruning, we start by pruning at 50%, then increase in step sizes of 10%, additionally pruning at 95% and 99%. For channel pruning, we start by pruning at 5%, then increase in step sizes of 5%, additionally pruning at 99%. In total, each pruning technique gets the same number of fine-tuning epochs, however we perform channel pruning in a more fine-grained way to compensate for its more coarse-grained approach to removing weights. For the CIFAR-10 models we use: 210 epochs of fine-tuning, shared evenly between each pruning step; an initial learning rate of $5\times10^{-2}$; and SGD with momentum 0.9, a weight decay $5\times10^{-4}$, and the 1cycle learning rate scheduler [ST18]. For the ImageNet models we use: 140 epochs of fine-tuning, shared evenly between each pruning step; an initial learning rate of $1\times10^{-3}$; and SGD with momentum 0.9, weight decay $5\times10^{-4}$, and the cosine annealing learning rate scheduler [LH17]. We store a copy of each model after every pruning step.

For data-type quantization we use TVM's native conversion tool. To recover the accuracy for `int8`, we use ONNXRuntime's [ONN18] post-training quantization tool. We perform calibration using the validation dataset to set rescaling constants. We use ONNXRuntime's default static quantization parameters using the `QInt8` weight type `QOperator` quantization format. For `float16`, as we show later in our results, there is no accuracy loss, thus we do not perform any additional steps to recover lost accuracy.

[1] `https://pytorch.org/vision/stable/index.html`

### A.1.3 Algorithms & Data Formats

We evaluate three algorithmic primitives for the convolutional layers: 1. *direct*, 2. *GEMM*, and 3. *spatial pack* convolution. We use both dense and sparse versions of these algorithms, which we implement or extend within TVM v0.8.0. The same high level algorithm implementation is used for both CPU and GPU, thanks to TVM's 'compute schedule' programming paradigm described in Section 2.5.6. All of our algorithms use the `NCHW` data layout, and for both weight and channel pruning we use the CSR sparse data format (described in Section 2.4.3).

### A.1.4 Systems Software

For our algorithms defined in TVM, we implement a minimal schedule such that the generated code exploits thread parallelism, and code can be generated correctly. However, since TVM's performance comes from having optimized schedules for each algorithm, unoptimized algorithms may give an unrealistic indication of the best algorithm in each setting. Thus, rather than hand optimize the schedule for every algorithmic variant that we explore, and risk introducing bias from inconsistent levels of optimization, we also leverage the Ansor auto-scheduler [Zhe+20a] to generate optimized schedules for each DNN layer.

For our CPU code generation, we use TVM's LLVM backend with AVX and Neon SIMD extensions for our Intel and Arm based CPUs respectively. Our untuned schedules do not define explicit vectorization and Ansor can automatically apply vectorization as one of its schedule primitives. For GPU code, we generate OpenCL and CUDA kernels for our Arm and Nvidia GPUs respectively, with CPU-side host code being similarly optimized.

For our auto-scheduling (or 'tuned') experiments, we allow Ansor to explore up-to 20,000 program variants, with early stopping permitted if no speedups have been observed after 1,000 variants. Auto-scheduling sparse computations is not fully supported by TVM. Thus, we employ an approach in TVM called 'sparse sketch rules', where we describe a starting point for the auto-scheduler to begin schedule generation. This works for the CPU, however TVM is unable to support auto-scheduled sparse computations on GPUs in the versions of TVM we have evaluated. This is because the auto-scheduler has two conflicting requirements: 1. cross-thread reduction, requiring partial sums to be computed across GPU threads simultaneously; and 2. loops parallelized over threads which request a static number of threads. Both of these conditions cannot be satisfied, since the size of our reduction loop for our algorithms varies depending on how many non-sparse elements there are in a given portion of the computation. Thus, we cannot tune pruned models on the GPU in our evaluation, since it cannot be easily supported by TVM.

### A.1.5 Hardware

Table A.1 shows the hardware platforms used in our experiments. For CPU experiments, we use a workstation machine featuring an Intel i7, and the HiKey 970 development board. The i7 has 6 cores, but due to hyper-threading 12 threads are exposed[2]. By default, TVM uses one thread per core, a default we follow in our experiments. The HiKey board has an Arm big.LITTLE architecture, meaning that it has 4 more powerful cores (A73@2.4GHz) and 4 less powerful cores (A53@1.8GHz). For our experiments, we use only the A73 (big) cores, which is the default for TVM. In principle, with appropriately configured load balancing between cores, using all cores could bring a performance improvement. However, this is outside the scope of this work, and as discussed in Section 4.1.2, exposes further across-stack considerations. For our GPU experiments, we leverage the GPUs of the HiKey 970 and an Nvidia AGX Xavier.

### A.1.6 Evaluation Methodology

When we have chosen all of our parameters for a given experiment, we then must evaluate the model and collect the inference time results. On both our CPU and GPU experiments, we ensure that devices are single-tenant and repeat experiments to mitigate potential interference from background processes. We run each experiment 150 times, using TVM's `time_evaluator` function with a single input image (i.e., batch size 1). For auto-scheduling, we run our search across 20,000 program variants once per experiment and evaluate the optimized binary 150 times. We report the median inference time, disregarding an initial warm-up run.

## A.2 Evaluation

For our evaluation, we split the results between CIFAR-10 (Section A.2.1) and ImageNet (Section A.2.2). We first analyze the accuracy impact of our optimization techniques, then choose maximally compressed models for each technique which maintain accuracy. We analyze the inference performance of these models using varying configurations on our CPUs and GPUs. Section 4.1 gives a high-level discussion of the results.

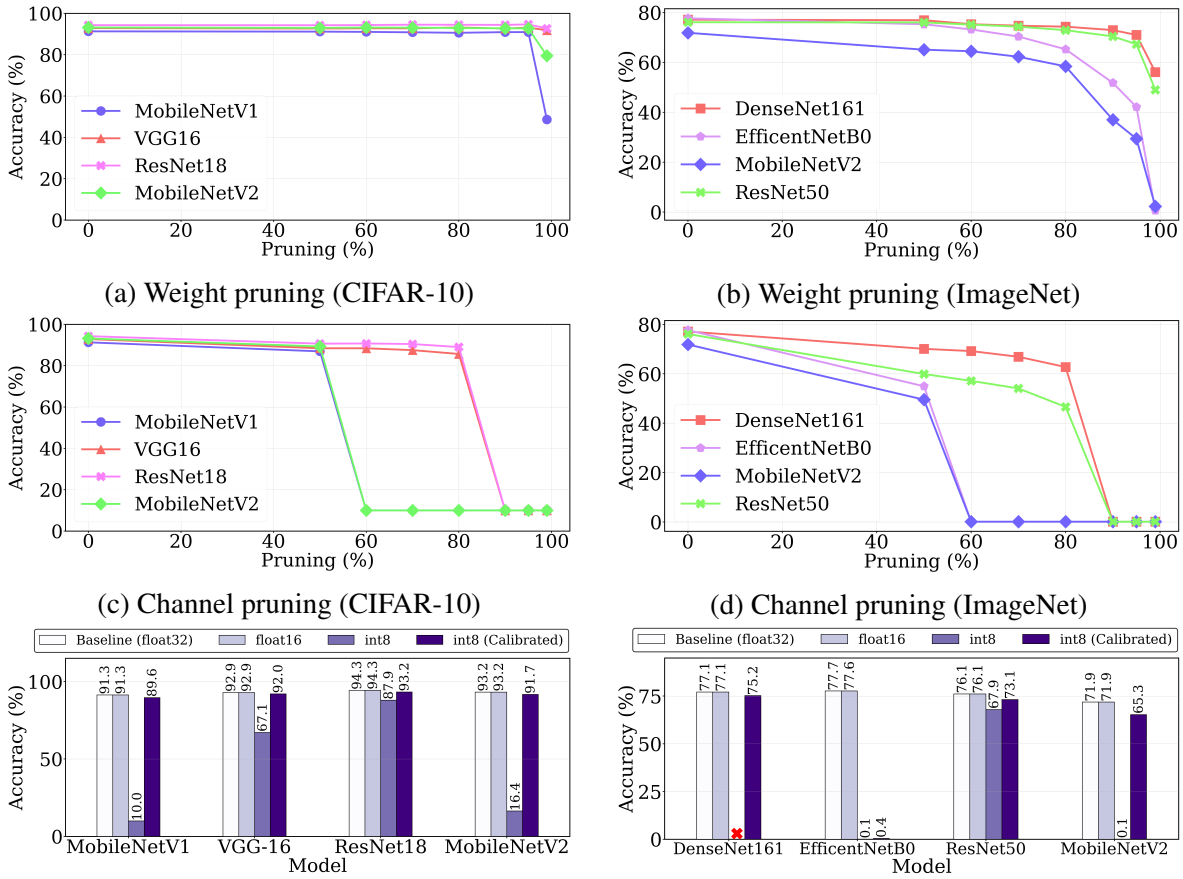
[2]Hyper-threading is Intel's approach to simultaneous multithreading).

Table A.1: Hardware features of the devices used in the experiments. For the HiKey 970, we only use the A73 CPU.

| **Device** | **CPU** | **L1 Cache (I+D)** | **L2 (+L3) Cache** | **RAM** | **GPU** | **GPU API** |
|---|---|---|---|---|---|---|
| Intel i7 | Intel i7-8700 (6 cores) @ 3.2 GHz | 192K + 192K | 1.5M (+12M) | 16GB DDR3 | - | - |
| HiKey 970 | *Arm Cortex-A73 (4 cores) @ 2.4 GHz*<br>Arm Cortex-A53 (4 cores) @ 1.8 GHz | *256K + 256K*<br>128K + 128K | *2M shared*<br>1M shared | 6GB LPDDR4 | Mali-G72<br>(12 cores) | OpenCL |
| AGX Xavier | Arm v8.2 Carmel (4 cores) @ 2.19 GHz | 64K + 128K | 2M (+ 4M) | 16GB LPDDR4x | Volta<br>(512 cores) | CUDA |

Table A.2: CIFAR-10 models, including baseline accuracy (Top1), and our chosen compression ratios and corresponding accuracies.

| Model | Params | MACs | Top1 | Model Optimization Accuracy (& Compression Ratio) | | | |
|---|---|---|---|---|---|---|---|
| | | | | Weight Pruning | Channel Pruning | `float16` | `int8` |
| MobileNetV2 | 2.3M | 98M | 93.2% | 92.7% (95%) | 89.3% (50%) | 93.2% (50%) | 91.7% (75%) |
| ResNet18 | 11.2M | 557M | 94.3% | 94.7% (95%) | 89.0% (80%) | 94.3% (50%) | 93.2% (75%) |
| VGG-16 | 14.7M | 314M | 92.9% | 93.1% (95%) | 85.7% (80%) | 92.9% (50%) | 92.0% (75%) |
| MobileNetV1 | 3.2M | 48M | 91.3% | 90.9% (95%) | 86.9% (50%) | 91.3% (50%) | 89.6% (75%) |




(a) Weight pruning (CIFAR-10)

(b) Weight pruning (ImageNet)

(c) Channel pruning (CIFAR-10)

(d) Channel pruning (ImageNet)

(e) Data-type quantization (CIFAR-10)

(f) Data-type quantization (ImageNet)

Figure A.1: Accuracy and compression trade-offs for our CIFAR-10 (a/c/e) and ImageNet (b/d/f) models: (a/b) shows the accuracy of each model after iterative weight pruning of the convolutional layers, and fine-tuning the model; (c/d) shows a similar setup for channel pruning; and (e/f) shows `float16` and `int8` quantization accuracy, with `int8` accuracy shown before and after calibration.

## A.2.1 CIFAR-10

### Accuracy

The accuracy of our models with varying levels of compression can be seen in the first column of Figure A.1. For our four models, the baseline (dense) top-1 accuracy on CIFAR-10 is shown in Table A.2. We observe that for unstructured pruning (Figure A.1a), the accuracy is maintained for all models until 95% pruning, at which point all models see a drop in accuracy at 99%. However, the drop in accuracy for MobileNetV1 and V2 is much more significant, likely because they have fewer total parameters.

We also observe this trend in the structured pruning (Figure A.1c), where VGG-16 and ResNet18 maintain their accuracy for longer than MobileNetV1 and V2. However, for all models the drop in accuracy is much sooner compared to weight pruning, with the elbow of

the graph appearing at 50% pruning for the MobileNet models, and 80% for VGG-16 and ResNet18. We also observe that after the elbow the drop is much more significant, to around 10% accuracy or equivalent to random guessing.

For our data-type quantization in Figure A.1e, we observe almost no change in accuracy for `float16` across the four models. The output is not bit-wise identical, which means that a very small number of images have a different classification, however at most this represents a 0.03% difference in top-1 accuracy. For uncalibrated `int8` quantization all models see a drop in accuracy, with MobileNets V1 and V2 seeing the highest drops, to 10.0% and 16.4% respectively. However, with calibration all models recover a large amount of their accuracy, with MobileNetV1 losing the most accuracy at 1.7%. We take the elbow points for each model using both pruning strategies to use in our inference experiments, with the elbow points chosen in Table A.2, and evaluate `float16` and `int8` in all cases.

### Inference – CPU (untuned)

The left and right columns of Figure A.2 show the untuned performance of our CIFAR-10 models when running on the CPUs of the HiKey and i7 platforms respectively, with varying compression strategies and convolutional primitives. The overall trends, including the fastest combination of parameters under different settings, are shown in Table A.3. Focusing on our dense `float32` models (the baseline used throughout our experiments), we observe that on the HiKey *GEMM* is the fastest algorithm for all models, however on the i7 *direct* is the fastest.

For our weight pruning models, sparse *GEMM* consistently outperforms the baseline, and in some cases gets the best performance for a given model on a given CPU. For example, MobileNetV2 on both CPUs, and MobileNetV1 and ResNet18 on the HiKey. For sparse *spatial pack*, on both CPUs only MobileNetV2 sees a speedup, and the most significant slowdowns are for ResNet18 and VGG-16. For sparse *direct*, on the HiKey both ResNet18 and VGG-16 see significant slowdowns, and MobileNetV1 only sees a small speedup. If we take the *best* baseline time for each model, we can compute an expected speedup given the compression ratio of each model optimization technique. For example, on the HiKey for MobileNetV1, with a pruning rate of 95% we would expect a 20× speedup. However, we only achieve a speedup of 2.6× for our best weight pruning algorithm (*GEMM*), i.e., 13.0% of our expected speedup. On average, for weight pruning, we achieve 11.5% and 21.8% of the expected speedup on the HiKey and i7 respectively.

For the channel pruning models, we again see that sparse *GEMM* consistently gets speedups, and is the best for all cases on both CPUs. *Direct* and *spatial pack* are less consistent. The elbow points for our models (see Table A.2) show that channel pruned models are less compressed than our weight pruning models, which means that we would expect the latter

to always be slower than the former. However, we observe several cases where a channel pruning model is *faster*, namely for all VGG-16 variants, except for *GEMM* on the HiKey, which is slightly slower. On average, for channel pruning, we achieve 77.9% and 83.9% of the expected speedup on the HiKey and i7 respectively.

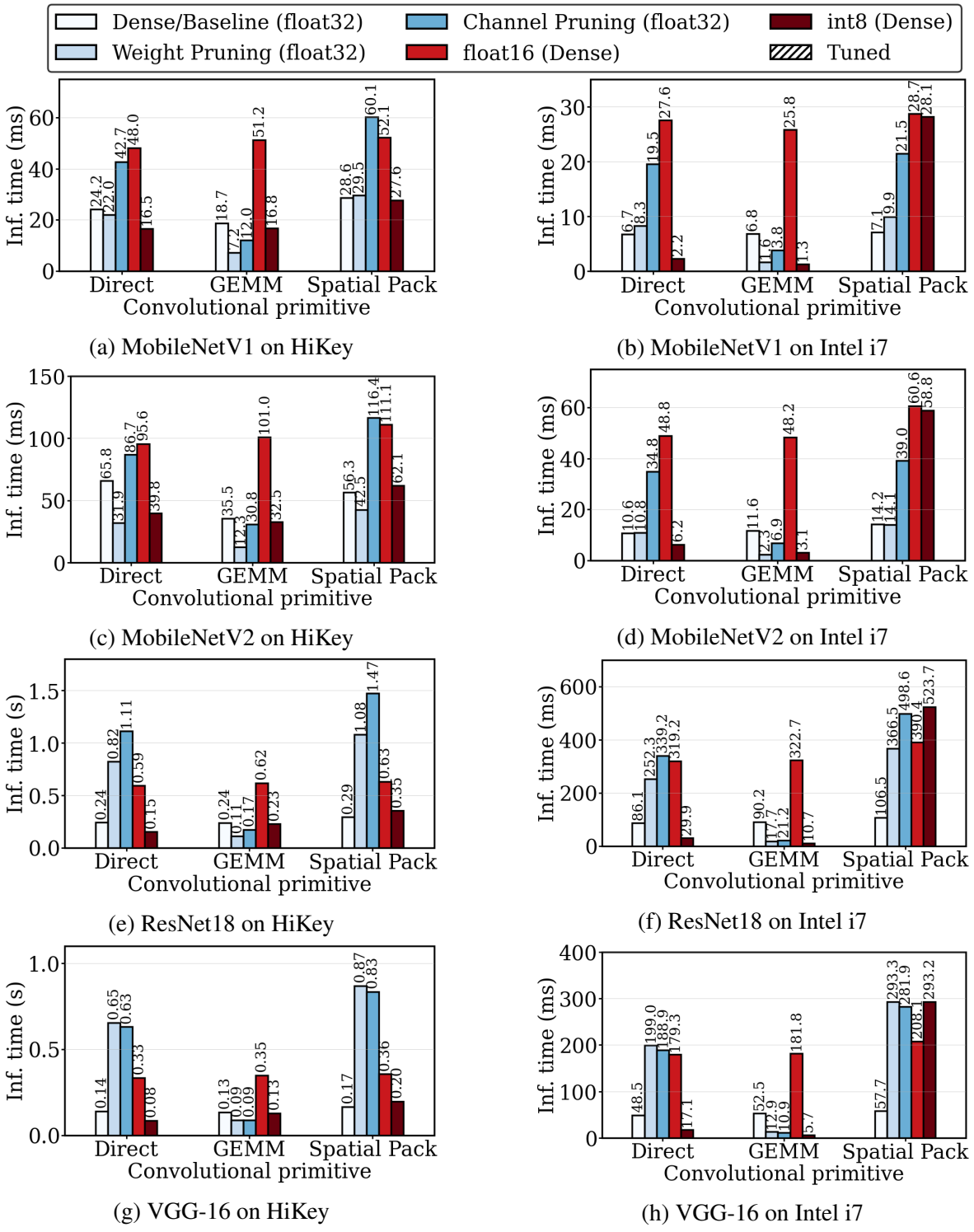
For our `float16` results, we observe a slowdown when compared to the `float32` baseline in every case, with the best algorithm varying between *direct* and *GEMM* between both models and CPUs. For our `int8` results, we generally observed a speedup relative to the baseline, except for the *spatial pack* implementation which is slower in all but one case (MobileNetV1 on the HiKey). In some cases, `int8` gives the best time overall, for example on VGG-16 for the *direct* and *GEMM* algorithms for the HiKey and i7 respectively. This trend of `int8` being fastest is not consistent, for example on MobileNetV2, where weight pruning with *GEMM* is faster than `int8` on both CPUs. This trend is not consistent between CPUs, for example for MobileNetV1, `int8` (with *GEMM*) is the fastest approach on the i7; whereas weight pruning (with *GEMM*) is the fastest on the HiKey. Compared to the baseline, we would naively expect a speedup of 4×, since we are reducing the number of bits from 32 to 8. On average, we achieve 33.6% and 157.2% of the expected speedup on the HiKey and i7 respectively.

Overall, on the HiKey *GEMM* convolution with weight pruning comes out as the best solution except for VGG-16, where *direct* with `int8` quantization is 1.05× faster. On the i7, *GEMM* with `int8` quantization is the fastest except for MobileNetV2, where *GEMM* with weight pruning is 1.35× faster.

### Inference – CPU (tuned)

Figure A.3 shows the tuned performance of our CIFAR-10 models when running on the HiKey and i7 CPUs, in the left and right columns respectively. Comparing to the untuned results in Section A.2.1, we observe that tuning has created some significant differences in the relative performance of our experiments, beyond reducing the inference time significantly, with overall trends shown in Table A.3. Focusing on our dense (baseline) models, we observe that *spatial pack* is now the best algorithm in every model on both CPUs, except for VGG-16 on the HiKey where *direct* is slightly faster.

For weight pruning, we observe that sparse *direct* is now the best weight pruning algorithm in every case on both CPUs. In the untuned case, this is only the case for MobileNets V1 and V2 on the HiKey. Sparse *direct* also outperforms the best dense algorithm (*spatial pack*) in almost every case, with equal performance on the i7 for MobileNetV2. This is to contrast against the sparse *GEMM* and *spatial pack* algorithms, which are consistently slower than their dense counterparts, reversing the trend for untuned sparse *GEMM*. On average, we




(a) MobileNetV1 on HiKey

(b) MobileNetV1 on Intel i7

(c) MobileNetV2 on HiKey

(d) MobileNetV2 on Intel i7

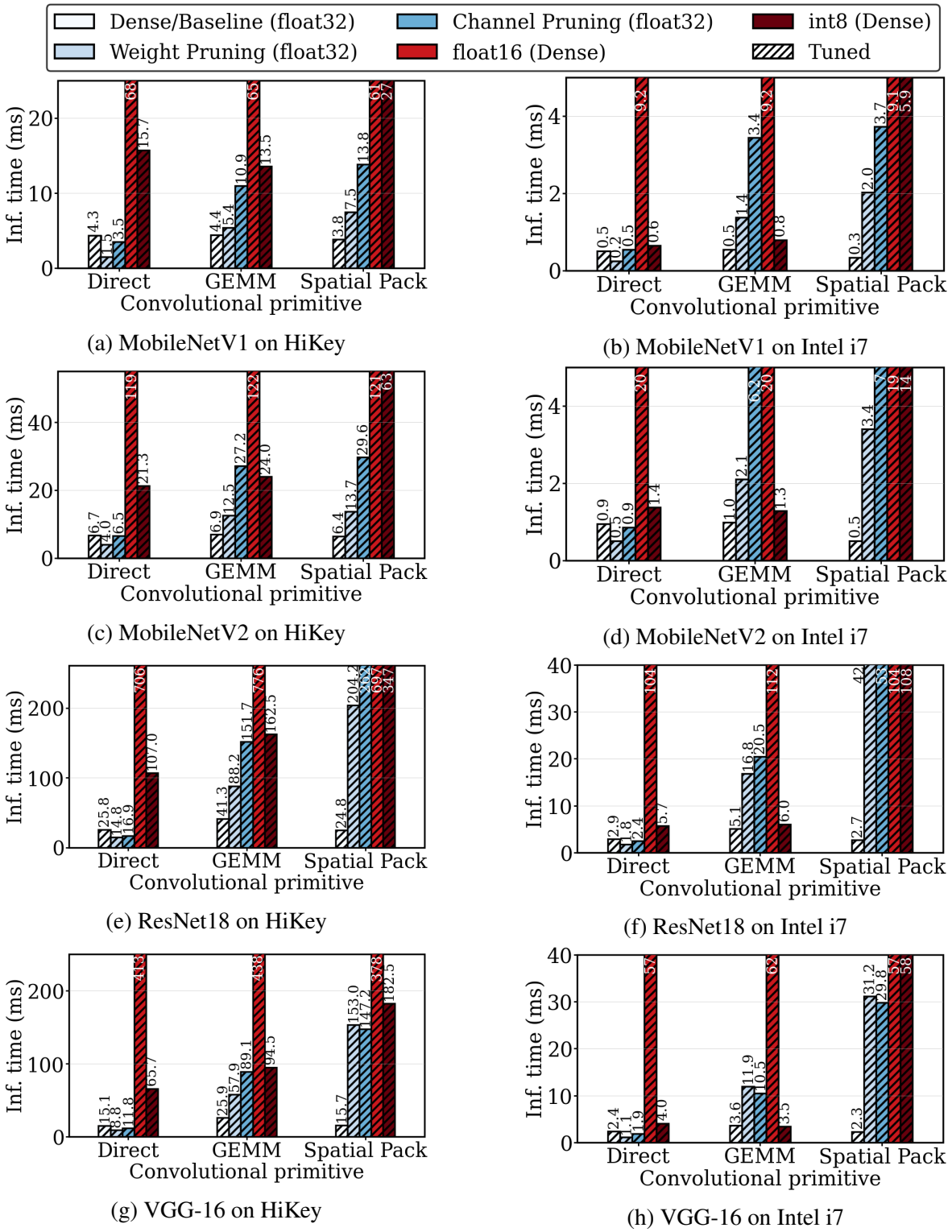
(e) ResNet18 on HiKey

(f) ResNet18 on Intel i7

(g) VGG-16 on HiKey

(h) VGG-16 on Intel i7

Figure A.2: Experiments comparing the compressed CIFAR-10 models chosen from obvious elbows of accuracy, with varying algorithmic primitives, benchmarked on the i7 and HiKey CPU platforms, without auto-scheduling.




(a) MobileNetV1 on HiKey

(b) MobileNetV1 on Intel i7

(c) MobileNetV2 on HiKey

(d) MobileNetV2 on Intel i7

(e) ResNet18 on HiKey

(f) ResNet18 on Intel i7

(g) VGG-16 on HiKey

(h) VGG-16 on Intel i7

Figure A.3: Experiments comparing the compressed CIFAR-10 models chosen from obvious elbows of accuracy, with varying algorithmic primitives, benchmarked on the i7 and HiKey CPU platforms, using auto-scheduling.

Table A.3: Analysis of CIFAR-10 CPU results for varying combinations of parameters, summarizing Figure A.2. The best inference times are shown in **bold**. Shortened names are used for brevity: WP (weight pruning), CP (channel pruning), i8 (`int8`), f16 (`float16`).

| Platform | Model | Fastest algorithm | | | | | Fastest compression technique | | | Overall fastest |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Dense | WP | CP | i8 | f16 | GEMM | Direct | Spatial (Pack) | |
| HiKey (untuned) | MobileNetV2 | GEMM | **GEMM** | GEMM | GEMM | Direct | **WP** | WP | WP | WP+GEMM |
| | ResNet18 | GEMM | **GEMM** | GEMM | Direct | Direct | **WP** | i8 | Dense | WP+GEMM |
| | VGG-16 | GEMM | GEMM | GEMM | **Direct** | Direct | WP | **i8** | Dense | i8+Direct |
| | MobileNetV1 | GEMM | **GEMM** | GEMM | Direct | Direct | **WP** | i8 | i8 | WP+GEMM |
| i7 (untuned) | MobileNetV2 | Direct | **GEMM** | GEMM | GEMM | GEMM | **WP** | i8 | WP | WP+GEMM |
| | ResNet18 | Direct | GEMM | GEMM | **GEMM** | Direct | **i8** | i8 | Dense | i8+GEMM |
| | VGG-16 | Direct | GEMM | GEMM | **GEMM** | Direct | **i8** | i8 | Dense | i8+GEMM |
| | MobileNetV1 | Direct | GEMM | GEMM | **GEMM** | GEMM | **i8** | i8 | Dense | i8+GEMM |
| HiKey (tuned) | MobileNetV2 | Spatial | **Direct** | Direct | Direct | Direct | Dense | **WP** | Dense | WP+Direct |
| | ResNet18 | Spatial | **Direct** | Direct | Direct | Spatial | Dense | **WP** | Dense | WP+Direct |
| | VGG-16 | Direct | **Direct** | Direct | Direct | Spatial | Dense | **WP** | Dense | WP+Direct |
| | MobileNetV1 | Spatial | **Direct** | Direct | GEMM | Spatial | Dense | **WP** | Dense | WP+Direct |
| i7 (tuned) | MobileNetV2 | **Spatial** | Direct | Direct | GEMM | Spatial | Dense | WP | **Dense** | Dense+Spatial |
| | ResNet18 | Spatial | **Direct** | Direct | Direct | Spatial | Dense | **WP** | Dense | WP+Direct |
| | VGG-16 | Spatial | **Direct** | Direct | GEMM | Direct | i8 | **WP** | Dense | WP+Direct |
| | MobileNetV1 | Spatial | **Direct** | Direct | Direct | Direct | Dense | **WP** | Dense | WP+Direct |

achieve 9.5% and 6.9% of the expected speedup on the HiKey and i7 respectively, significantly lower than the untuned proportion, which was 77.9% and 83.9% respectively.

For channel pruning, sparse *direct* is also the best algorithm, and we still observe that in the case of VGG-16 channel pruning is faster than weight pruning in most cases, despite in theory having more operations to compute. On average, we achieve 39.8% and 26.6% of the expected speedup on the HiKey and i7 respectively; again, lower than untuned.

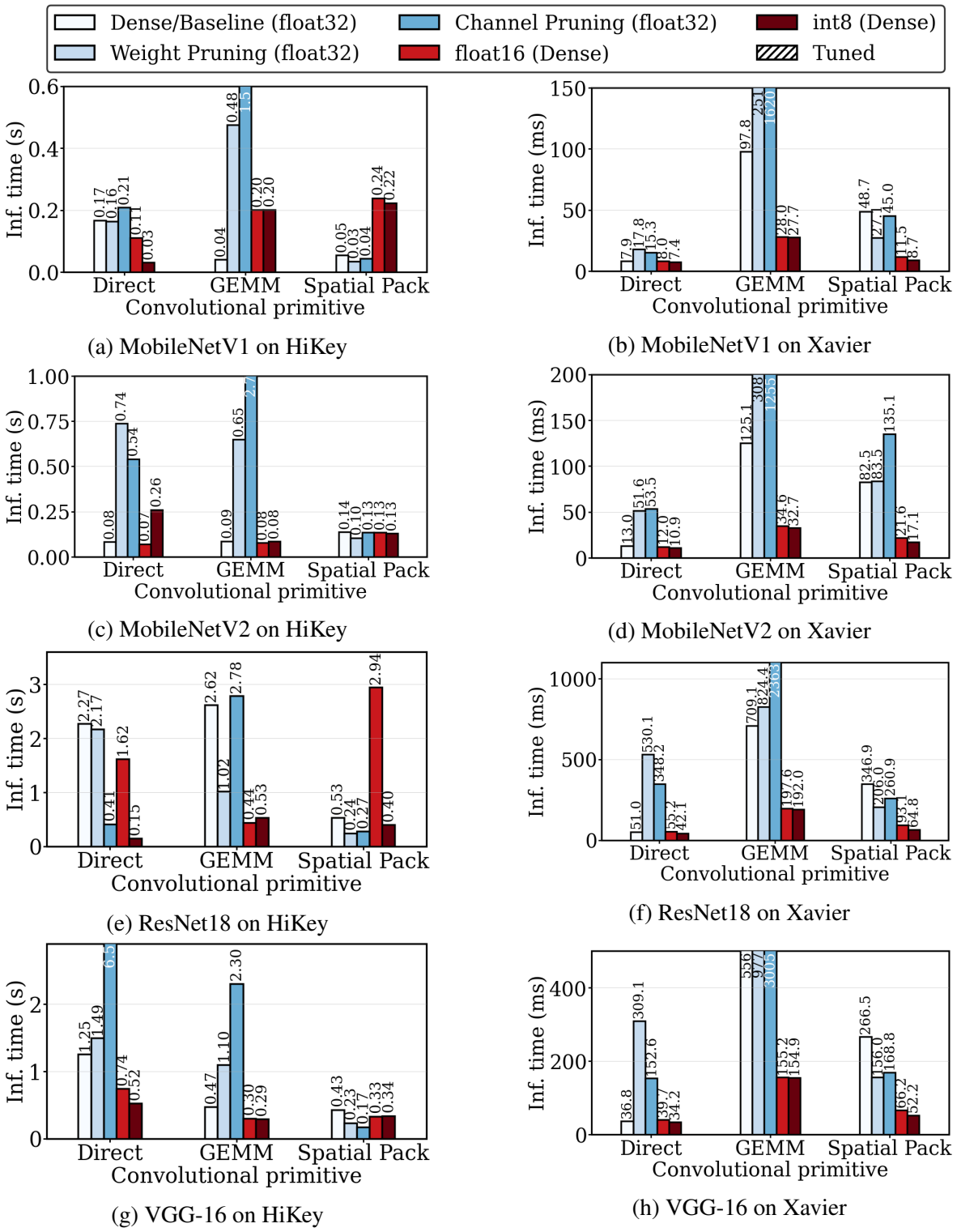
For our `float16` results, we observe that although there are speedups, we are still slower than the baseline in every case, and these differences are exacerbated when compared to the untuned results. The speedups of `int8` on the i7 have disappeared with tuning, with an average slowdown of 2.0×.

### Inference – GPU (untuned)

The left and right columns of Figure A.4 shows the untuned performance of our CIFAR-10 models when running on the HiKey and Xavier GPUs respectively, with overall trends shown in Table A.4. On the Xavier, dense *direct* convolution is consistently the fastest; whereas on the HiKey it is the best for MobileNetV2 and ResNet18, but *GEMM* and *spatial pack* are faster for MobileNetV1 and VGG-16 respectively. We also note that the HiKey's inference time on the GPU is much higher than on the CPU. For example, the baseline *direct* for MobileNetV1 is almost 7× slower when using the GPU.

For our pruned experiments, we see speedups most consistently using *spatial pack*, which is different from the CPU where we almost always saw a slowdown. On the HiKey, we see speedups relative to the baseline in multiple cases, and even the fastest possible variant in some cases, such as for channel pruning with *spatial pack* for VGG-16. In terms of our expected speedup, for the HiKey and Xavier respectively, for weight pruning we achieve 7.4% and 2.2%, and for channel pruning we achieve 41.1% and 26.0%.

For `float16`, unlike the CPU, we observe speedups for several cases when compared to our baseline `float32`. However, the behavior is not consistent across models, algorithms, or devices. For example, on the HiKey using *GEMM* with MobileNetV1 (Figure A.4a), `float16` provides a slowdown, whereas for other models we observe a speedup, as well as for the same case on the Xavier. Our most significant slowdowns for `float16` is for MobileNetV1 and ResNet18 on the HiKey (Figures A.4a and A.4e) using *spatial pack*. On the Xavier, `float16` provides a speedup in all cases except for *direct* convolution where we observe small slowdowns. On average `float16` achieves 51.9% and 49.4% of its potential speedup on the HiKey and Xavier respectively. The exception to this trend is MobileNetV2 using direct convolution, where we see a small speedup of 1.08×. For `int8` quantization we observe slowdowns for all instances of *direct* convolution, speedups for all instances of *GEMM*, and speedups for all cases of *spatial pack* except for MobileNetV1 on the HiKey.




(a) MobileNetV1 on HiKey

(b) MobileNetV1 on Xavier

(c) MobileNetV2 on HiKey

(d) MobileNetV2 on Xavier

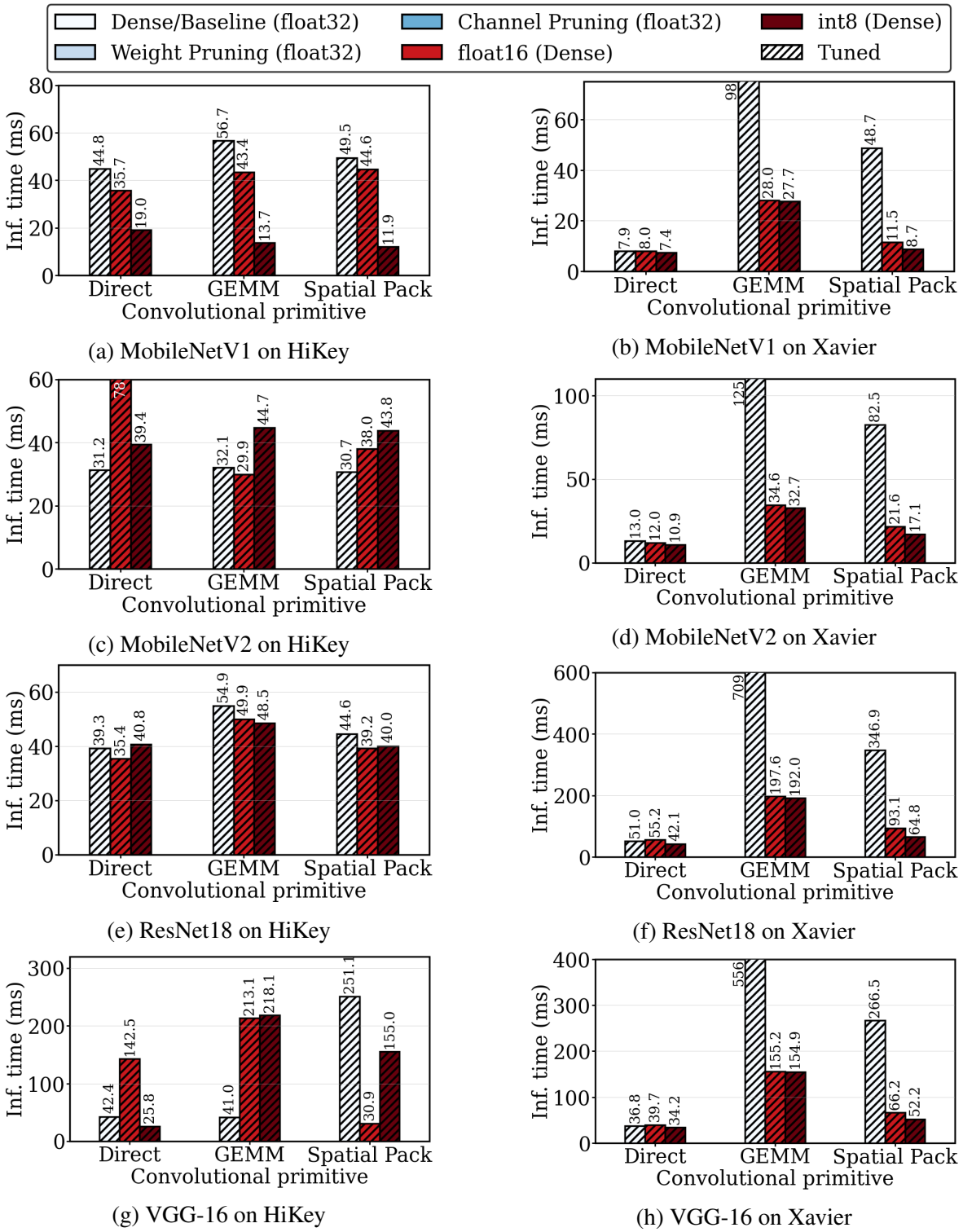
(e) ResNet18 on HiKey

(f) ResNet18 on Xavier

(g) VGG-16 on HiKey

(h) VGG-16 on Xavier

Figure A.4: Experiments comparing the compressed CIFAR-10 models chosen from obvious elbows of accuracy, with varying algorithmic primitives, benchmarked on the HiKey and Xavier GPU platforms, without auto-scheduling.




(a) MobileNetV1 on HiKey

(b) MobileNetV1 on Xavier

(c) MobileNetV2 on HiKey

(d) MobileNetV2 on Xavier

(e) ResNet18 on HiKey

(f) ResNet18 on Xavier

(g) VGG-16 on HiKey

(h) VGG-16 on Xavier

Figure A.5: Experiments comparing the compressed CIFAR-10 models chosen from obvious elbows of accuracy, with varying algorithmic primitives, benchmarked on the HiKey and Xavier GPU platforms, using auto-scheduling.

Table A.4: Analysis of CIFAR-10 GPU results for varying combinations of parameters, summarizing Figure A.4. The best inference times are shown in **bold**. Shortened names are used for brevity: WP (weight pruning), CP (channel pruning), i8 (`int8`), f16 (`float16`).

| Platform | Model | Fastest algorithm | | | | | Fastest compression technique | | | Overall fastest |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Dense | WP | CP | i8 | f16 | GEMM | Direct | Spatial (Pack) | |
| HiKey (untuned) | MobileNetV2 | GEMM | Spatial | Spatial | GEMM | **Direct** | f16 | **f16** | WP | f16+Direct |
| | ResNet18 | Spatial | Spatial | Spatial | **Direct** | GEMM | f16 | **i8** | CP | i8+Direct |
| | VGG-16 | Spatial | Spatial | **Spatial** | GEMM | GEMM | i8 | i8 | **CP** | CP+Spatial |
| | MobileNetV1 | GEMM | Spatial | Spatial | **Direct** | Direct | Dense | i8 | WP | i8+Direct |
| Xavier (untuned) | MobileNetV2 | Direct | Direct | Direct | **Direct** | Direct | i8 | **i8** | i8 | i8+Direct |
| | ResNet18 | Direct | Spatial | Spatial | **Direct** | Direct | i8 | **i8** | i8 | i8+Direct |
| | VGG-16 | Direct | Spatial | Direct | **Direct** | Direct | i8 | **i8** | i8 | i8+Direct |
| | MobileNetV1 | Direct | Direct | Direct | **Direct** | Direct | i8 | **i8** | i8 | i8+Direct |
| HiKey (tuned) | MobileNetV2 | Direct | - | - | Direct | **GEMM** | **f16** | Dense | f16 | f16+GEMM |
| | ResNet18 | Direct | - | - | Spatial | **Direct** | i8 | **f16** | f16 | f16+Direct |
| | VGG-16 | GEMM | - | - | **Direct** | Spatial | Dense | **i8** | f16 | i8+Direct |
| | MobileNetV1 | Direct | - | - | **Spatial** | Direct | i8 | i8 | **i8** | i8+Spatial |
| Xavier (tuned) | MobileNetV2 | Direct | - | - | **Direct** | Direct | i8 | **i8** | i8 | i8+Direct |
| | ResNet18 | Direct | - | - | **Direct** | Direct | i8 | **i8** | i8 | i8+Direct |
| | VGG-16 | Direct | - | - | **Direct** | Direct | i8 | **i8** | i8 | i8+Direct |
| | MobileNetV1 | Direct | - | - | **Direct** | Direct | i8 | **i8** | i8 | i8+Direct |

#### Inference – GPU (tuned)

Figure A.4 shows the tuned performance of our CIFAR-10 models on GPUs, with overall trends shown in Table A.4. As noted in Section A.1.4, we cannot provide tuned results for our sparse models on the GPUs. For `float32` inference on the HiKey we observe that the inference times for all algorithms are relatively similar, except for VGG-16 where *spatial pack* is significantly slower. If we take the normalized mean inference time, *direct* is the best for the HiKey. The HiKey GPU is also still slower than HiKey CPU (tuned), with the best baseline result being 3.1× slower on average. For the Xavier, *direct* convolution is the best performing algorithm in all cases. However, if we compare to the untuned case we see that we do not get any improvement when tuning the Xavier. We discuss this issue in Section 4.1.1. For quantization on the HiKey, taking the best result for each model, we achieve 58.9% and 44.4% of the expected speedup on average for `float16` and `int8` respectively; the Xavier achieves 49.0% and 28.4% of its expected speedups.

### A.2.2 ImageNet

#### Accuracy

For our four models, the baseline (dense) top-1 accuracy on ImageNet is shown in Table A.5. EfficientNetB0 has the highest accuracy, which may be surprising given it has fewer parameters than ResNet50 and DenseNet161. However, EfficientNetB0 is more recent and thus exploits a number of newer machine learning techniques to improve its parameter and training efficiency (see Section 3.2.1).

The top-1 accuracy on ImageNet of the models with varying levels of compression can be seen in the second column of Figure A.1. We observe a similar trend to our CIFAR-10 models, namely the smaller models (EfficientNetB0 and MobileNetV2) lose their accuracy more quickly than the larger models (DenseNet161 and ResNet50). We also observe that *all* models lose more accuracy earlier when compared to our CIFAR-10 pruning. This suggests that our CIFAR-10 models are more overparameterized.

For data-type quantization we observe a similar trend as CIFAR-10, namely a negligible difference in accuracy for `float16`. For ResNet50, we see a large drop in accuracy for uncalibrated `int8` quantization and recovering to around a 3.0% reduction in top-1 accuracy. For MobileNetV2, we observe a huge drop in accuracy for the uncalibrated model, down to around 0.09%, recovering to around a 6.6% reduction in top-1 accuracy. This drop in accuracy was much higher than we expected, especially compared to the CIFAR-10 version. Therefore, we also tried importing the Keras [Cho15] definition of MobileNetV2, however observed the same behavior.

For EfficientNetB0, we also observe a huge drop in accuracy to 0.08%, however the recovery is much smaller than MobileNetV2's, reaching only 0.43% accuracy. However, this is due to features of the architecture which make it less suitable for quantization which we discuss in Section 4.1.1. For DenseNet161, we cannot run the `int8` model in TVM due to an unsupported quantized operation. This excludes it from collection of uncalibrated accuracy and inference time results. However, we can still collect calibrated accuracy results in ONNXRuntime, where we lose only 1.9% of accuracy.

### Inference – CPU (untuned)

Figure A.6 shows the untuned performance of our ImageNet models when running on the i7 and HiKey CPUs, with overall trends shown in Table A.6. Focusing on our baseline dense models, we observe that in all cases on the HiKey *GEMM* gives the best performance, which matches its behavior as seen in CIFAR-10 (Section A.2.1). For the i7, *GEMM* is fastest for our large models (ResNet50 and DenseNet161), however *direct* is fastest for our small models (MobileNetV2 and EfficientNetB0); on CIFAR-10 *direct* was consistently the fastest on this CPU.

For weight pruning, we find that by taking our best performing variants as before, we achieve 30.3% and 41.8% for the HiKey and i7 respectively; significantly higher than for CIFAR-10. For channel pruning, this is 60.2% and 84.2% respectively, which is 16.7% less than CIFAR-10 for the HiKey, and 0.3% more for the i7. For both weight and channel pruning, we find that the only algorithm which consistently provides a speedup relative to dense is *GEMM*.

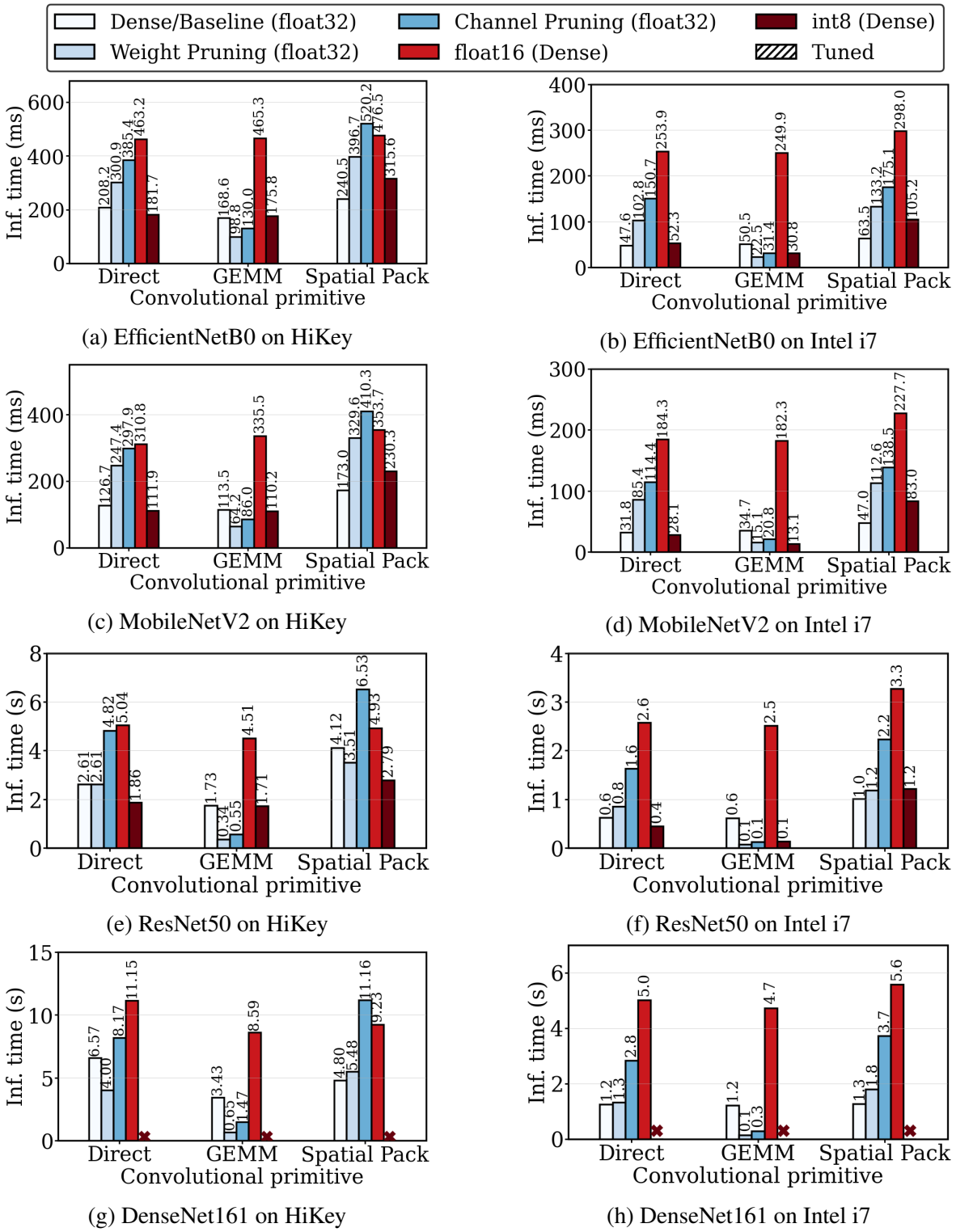
For quantization we see similar trends to CIFAR-10, namely a slowdown using `float16`, and a speedup using `int8`. For `int8`, we achieve 25.0% and 73.0% of the expected speedup on the i7 and HiKey respectively, lower CIFAR-10.

### Inference – CPU (tuned)

Figure A.7 shows the tuned performance of our ImageNet models when running on the i7 CPU, with overall trends shown in Table A.6. We note that tuning on the HiKey CPU (and GPU) was not practical, since the two variants we attempted took over 140 hours each, so we do not include any of the 57 variants required for each device. For the dense case on the i7, we see that *spatial pack* is consistently the best, matching our observed trends on tuned CIFAR-10. For our pruned models, we do not see any cases where pruned models are faster than a dense `float32` implementation. This is contrasted with CIFAR-10, where we observe this behavior in every case.

Table A.5: ImageNet models, including baseline accuracy (Top1), and our chosen compression ratios and corresponding accuracies.

| Model | Params | MACs | Top1 | Model Optimization Accuracy (& Compression Ratio) | | | |
|---|---|---|---|---|---|---|---|
| | | | | Weight Pruning | Channel Pruning | `float16` | `int8` |
| MobileNetV2 | 3.5M | 327M | 71.9% | 58.4% (80%) | 49.9% (50%) | 71.9% (50%) | 65.3% (75%) |
| ResNet50 | 25.6M | 4.1G | 76.1% | 67.3% (95%) | 46.6 (80%) | 76.1% (50%) | 73.1% (75%) |
| DenseNet161 | 27.7M | 7.8G | 77.1% | 74.3% (95%) | 62.7 (80%) | 77.1% (50%) | 75.2% (75%) |
| EfficientNetB0 | 5.3M | 415M | 77.7% | 65.3% (80%) | 54.9 (50%) | 77.6% (50%) | 0.4% (75%) |




(a) EfficientNetB0 on HiKey

(b) EfficientNetB0 on Intel i7

(c) MobileNetV2 on HiKey

(d) MobileNetV2 on Intel i7

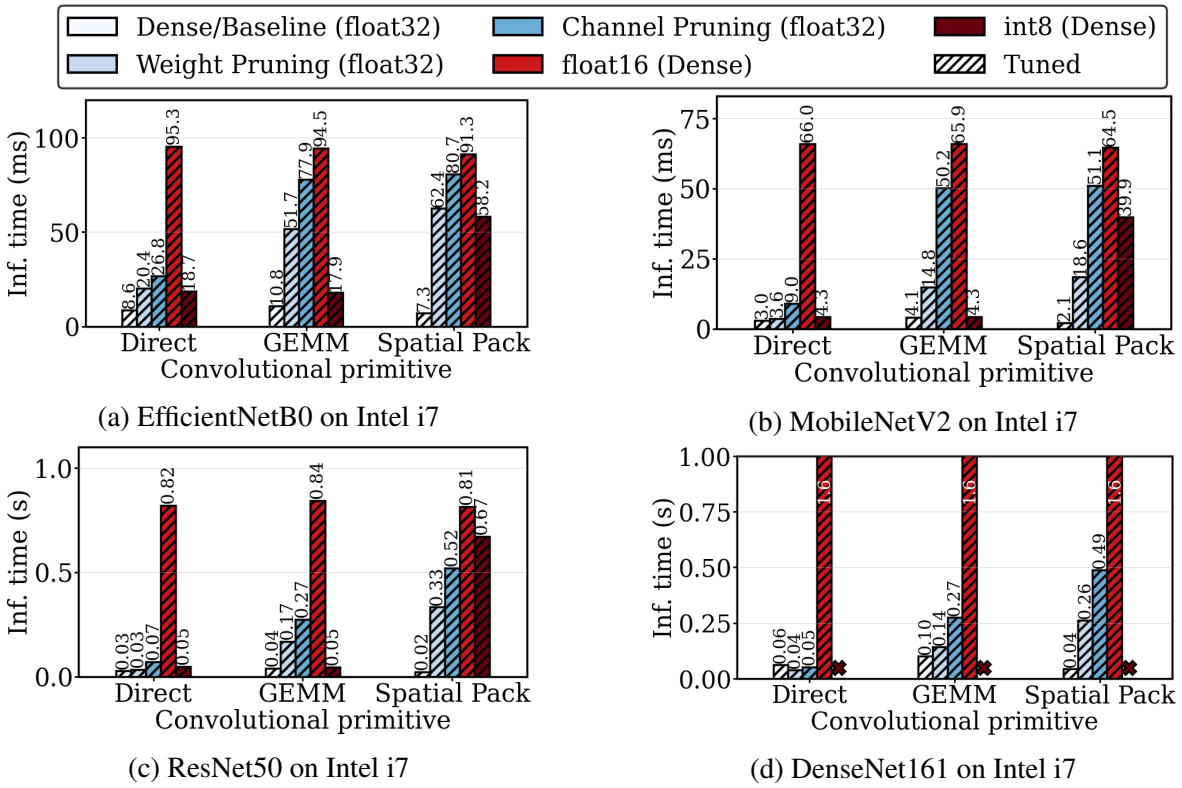
(e) ResNet50 on HiKey

(f) ResNet50 on Intel i7

(g) DenseNet161 on HiKey

(h) DenseNet161 on Intel i7

Figure A.6: Experiments comparing the compressed ImageNet models chosen from obvious elbows of accuracy, with varying algorithmic primitives, benchmarked on the i7 and HiKey CPU platforms, without auto-scheduling.




(a) EfficientNetB0 on Intel i7

(b) MobileNetV2 on Intel i7

(c) ResNet50 on Intel i7

(d) DenseNet161 on Intel i7

Figure A.7: Experiments comparing the compressed ImageNet models chosen from obvious elbows of accuracy, with varying algorithmic primitives, benchmarked on the i7 CPU platform, using auto-scheduling. HiKey results are not included due to impractically high tuning time.

Table A.6: Analysis of ImageNet results for varying combinations of parameters, summarizing Figures A.6 and A.8. The best inference times are shown in **bold**. Shortened names are used for brevity: WP (weight pruning), CP (channel pruning), i8 (`int8`), f16 (`float16`).

| Platform | Model | Fastest algorithm | | | | | Fastest compression technique | | | Overall fastest |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Dense | WP | CP | i8 | f16 | GEMM | Direct | Spatial (Pack) | |
| HiKey CPU (untuned) | MobileNetV2 | GEMM | **GEMM** | GEMM | GEMM | Direct | **WP** | i8 | Dense | WP+GEMM |
| | ResNet50 | GEMM | **GEMM** | GEMM | GEMM | GEMM | **WP** | i8 | i8 | WP+GEMM |
| | DenseNet161 | GEMM | **GEMM** | GEMM | - | GEMM | **WP** | WP | Dense | WP+GEMM |
| | EfficientNetB0 | GEMM | **GEMM** | GEMM | GEMM | Direct | **WP** | i8 | Dense | WP+GEMM |
| i7 (untuned) | MobileNetV2 | Direct | GEMM | GEMM | **GEMM** | GEMM | **i8** | i8 | Dense | i8+GEMM |
| | ResNet50 | GEMM | **GEMM** | GEMM | GEMM | GEMM | **WP** | i8 | Dense | WP+GEMM |
| | DenseNet161 | GEMM | **GEMM** | GEMM | - | GEMM | **WP** | Dense | Dense | WP+GEMM |
| | EfficientNetB0 | Direct | **GEMM** | GEMM | GEMM | GEMM | **WP** | Dense | Dense | WP+GEMM |
| i7 (tuned) | MobileNetV2 | **Spatial** | Direct | Direct | Direct | Spatial | Dense | Dense | **Dense** | Dense+Spatial |
| | ResNet50 | **Spatial** | Direct | Direct | GEMM | Spatial | Dense | Dense | **Dense** | Dense+Spatial |
| | DenseNet161 | Spatial | **Direct** | Direct | - | Spatial | Dense | **WP** | Dense | WP+Direct |
| | EfficientNetB0 | **Spatial** | Direct | Direct | GEMM | Spatial | Dense | Dense | **Dense** | Dense+Spatial |
| HiKey GPU (untuned) | MobileNetV2 | GEMM | Spatial | Direct | Direct | **GEMM** | **f16** | i8 | i8 | f16+GEMM |
| | ResNet50 | Direct | Spatial | Spatial | Direct | Direct | WP | i8 | **WP** | WP+Spatial |
| | DenseNet161 | Direct | **Spatial** | Spatial | - | GEMM | WP | WP | **WP** | WP+Spatial |
| | EfficientNetB0 | GEMM | Spatial | Spatial | **Direct** | GEMM | f16 | **i8** | i8 | i8+Direct |
| Xavier (untuned) | MobileNetV2 | Direct | Spatial | Spatial | Direct | **Direct** | f16 | **f16** | f16 | f16+Direct |
| | ResNet50 | Spatial | Spatial | Spatial | **Direct** | Direct | i8 | **i8** | i8 | i8+Direct |
| | DenseNet161 | Spatial | **Spatial** | Spatial | - | Direct | WP | f16 | **WP** | WP+Spatial |
| | EfficientNetB0 | Direct | Spatial | Spatial | Direct | **Direct** | f16 | **f16** | f16 | f16+Direct |

### Inference – GPU (untuned)

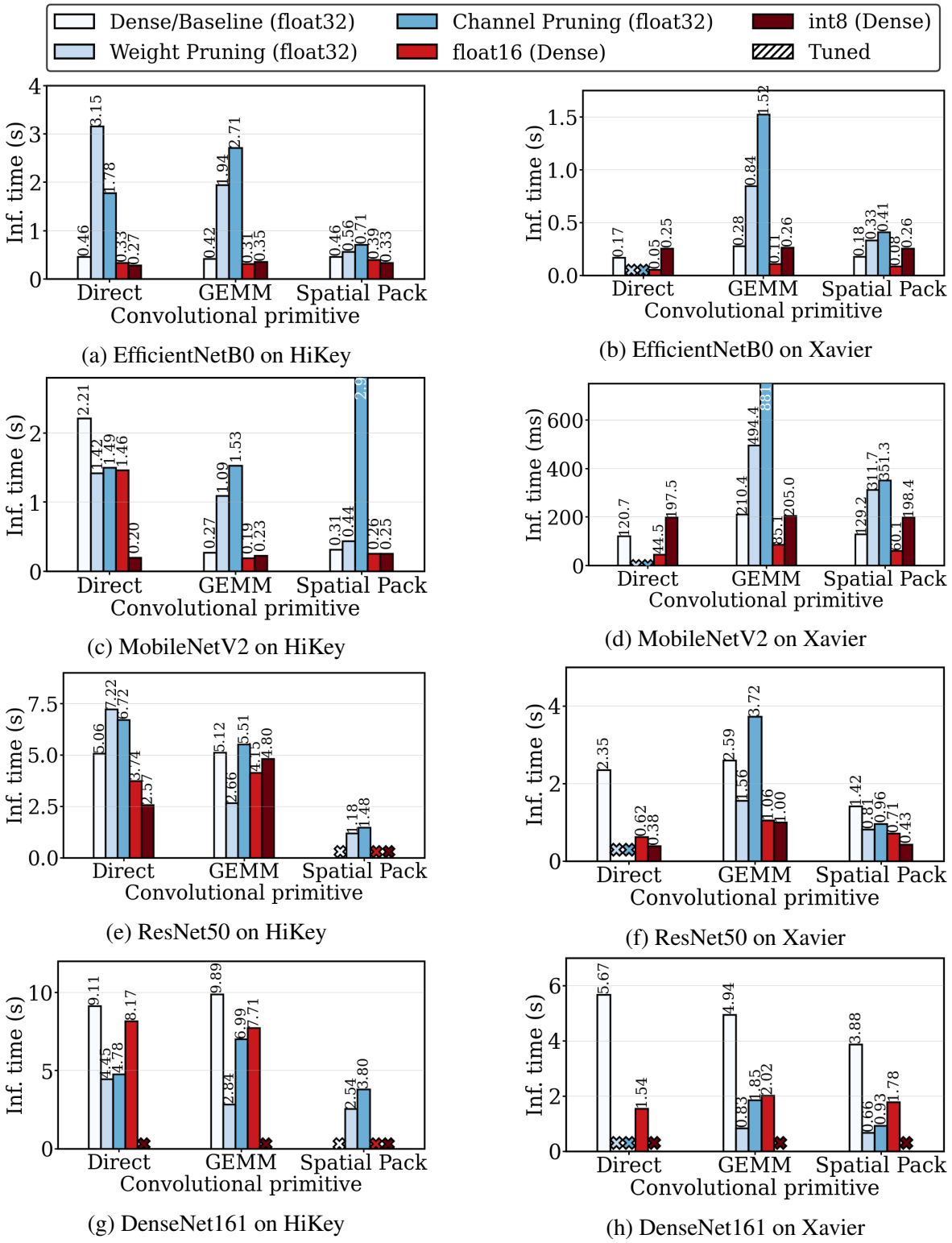
Figure A.8 shows the untuned performance of our ImageNet models on the GPUs, with overall trends shown in Table A.6. On the Xavier for the dense `float32` case, we observe that *spatial pack* is the best algorithm for the larger models (ResNet50 and DenseNet161), and *direct* is the best for the smaller models (MobileNetV2 and EfficientNetB0). On the Hikey, for the smaller models *GEMM* was the best, and for the larger models *direct* was the best. However, on the HiKey dense ResNet50 and DenseNet161 experiments using *spatial pack* crashed with the error `CL_INVALID_WORK_GROUP_SIZE`. This means that in one or more layers of these models TVM is exceeding the number of work items our GPU can support (see the OpenCL specification for more details [SGS10]). If we run auto-tuning, TVM can configure the work group size and other GPU parameters, which could avoid this issue. However, we did not observe this on the Xavier, which has more hardware resources available.

Sparse *spatial pack* using weight pruning was consistently the best pruned model across both GPUs, however only outperformed the baseline in one case, ResNet50 on the Xavier. On the Xavier, the sparse *direct* experiments did not halt, even allowing hours for a single run. GPU memory utilization was at its maximum, suggesting some inefficiency in this algorithm/hardware combination. Again, auto-tuning may make this variant viable, however we cannot tune sparse models on GPUs using current versions of TVM (see Section A.1.4).

Quantization tends to give speedups compared to dense `float32`. Overall, the `int8` quantized model performs the best on the Xavier, with the *direct* algorithm being the fastest in this case. On the Hikey, `float16` *GEMM* is marginally better for EfficientNetB0 and MobileNetV2, and weight pruning *spatial pack* is best for ResNet50 and DenseNet161.

### Inference – GPU (tuned)

As discussed in Section A.2.2, collecting tuned results for the HiKey GPU was not practical. In addition, we again observed no speedup on the Xavier when tuning, hence we do not include the graphs, since they are identical to the Xavier results shown in Figure A.8.




(a) EfficientNetB0 on HiKey

(b) EfficientNetB0 on Xavier

(c) MobileNetV2 on HiKey

(d) MobileNetV2 on Xavier

(e) ResNet50 on HiKey

(f) ResNet50 on Xavier

(g) DenseNet161 on HiKey

(h) DenseNet161 on Xavier

Figure A.8: Experiments comparing the compressed ImageNet models chosen from obvious elbows of accuracy, with varying algorithmic primitives, benchmarked on the HiKey and Xavier GPU platforms, without auto-scheduling.

# Glossary

**accuracy** Performance on a given task (e.g., number of correctly identified images). 2, 3, 5, 7, 13, 15, 16, 18, 19, 23–26, 46–52, 66, 69, 70, 73, 76–78, 86, 89, 112, 122, 125, 129, 133, 135, 137–139, 141, 145, 146, 148–152, 155

**activation function** Non-linear differentiable applied to the outputs of some DNN layers. 18, 19, 29, 30, 52, 59, 102, 128

**Ansor** An auto-scheduling system within the Apache TVM tensor compiler. 8, 38, 39, 57, 59–61, 84, 94–96, 100, 104, 105, 108, 111, 113–116, 118, 120, 123, 124, 134

**auto-scheduling** The process of automatically generating an optimized schedule for a given kernel. 1, 8, 55–61, 64, 73, 74, 84, 94–100, 102, 112, 114, 115, 120, 121, 123–128, 130, 134, 135, 141, 142, 145, 146, 151, 152, 155

**big.LITTLE** Heterogeneous CPU architecture from Arm, coupling relatively lower power slower processor cores (LITTLE) with relatively more powerful and power-hungry ones (big). 41, 75, 86, 127, 135

**CIFAR-10** A popular small-scale image classification dataset. 4, 12, 13, 44, 78, 85, 132, 133, 135, 137–142, 144–146, 148, 149

**co-design** Developing a solution which combines techniques from two or more domains. 1, 2, 4, 46, 52, 54, 63, 66, 67, 76, 130

**grouped convolution** A special case of convolutional layers, which reduces the number of MACs and parameters required. 7, 8, 10, 24, 25, 47, 50, 77–81, 85, 87, 89, 91, 92, 121–123, 125–128

**im2col** Rearranges blocks of data into columns, potentially replicating data, to make it more amenable to matrix-multiplication. Often used in GEMM convolution. 27, 28, 53, 54, 72

**ImageNet** A popular image classification dataset. 4, 12, 13, 44, 47, 85, 89, 102, 104, 111, 114, 115, 117, 132, 133, 135, 138, 148–152, 154, 155

**inference** Mode of operation of a neural network, where we pass it some input and get an output. 2, 3, 5, 7, 8, 11, 15, 16, 22, 24, 27, 35, 40, 45, 48, 50, 52, 55, 58, 60, 63, 65, 66, 69, 72–78, 80–82, 84–86, 89–98, 100, 104–109, 112, 113, 118, 120–123, 125, 128, 129, 133, 135, 139, 140, 144, 148, 149

**kernel** A sub-program that implements a given algorithm. 33, 37, 38, 57, 61, 65, 94–96, 98–100, 102, 104–108, 110, 113, 115, 117, 118, 120, 127–129

**kernel library** A library of optimized kernels for a given set of tasks and hardware platforms. 33, 34, 36, 47, 54–56, 58, 129

**knowledge distillation** Using a pre-trained larger model to train a more compressed model. 4, 25, 47

**learning rate** The size of the parameter changes when training a neural network, usually adjusted dynamically, trending to smaller step sizes. 16, 23, 70, 86, 132, 133

**loop unrolling** A compiler optimization which replaces loops with the body repeated `N` times. 31, 32, 82, 99, 123

**ONNX** An open source format for AI models, often used as an interchange format between DNN frameworks. 29, 35, 65, 133, 149

**quantization** Reducing the range of values that a datum can take. 4, 22–24, 49, 51, 52, 56, 57, 125, 133, 138–140, 144, 148, 149

**recursion** Defining a problem in terms of itself. 158

**ReLU** A common activation function, which sets all negative values to zero. 18, 29, 30, 35, 52, 73, 102, 104, 110, 128

**roofline model** A visuzalization of the peak performance we might expect of a kernel given the memory and compute limits of the hardware. 5, 6, 39, 128

**supervised learning** A learning task where we have example input data and output data, and we must learn the mapping between them. 12, 15

**tensor** A multi-dimensional array. 26

**tensor compiler** A domain-specific compiler for tensor programs. 1, 3, 7–9, 11, 34–38, 50, 52, 55–61, 64–66, 68, 71, 80, 121, 125, 127–131

**tensor program** A class of software designed for manipulation and calculation with tensors. 1, 13, 36, 56, 57, 60, 82, 94, 96, 98, 122, 124, 126, 128

**training** Mode of operation of a neural network, where we run a learning algorithm with training data to update the parameters of the model to better complete some task. 3, 12, 15–17, 20, 22, 23, 26, 40, 45, 46, 49, 52, 58, 65, 86, 123

**Transformer** A neural architecture characterized by the self-attention mechanism. 4, 20, 21, 46, 47, 50, 67, 75, 107, 108, 111

**Winograd convolution** A class of algorithms implementing convolutional layers, which compute in Fourier space. 4, 53, 54, 75, 127

# Acronyms

**AI** Artificial Intelligence 1, 40, 63, 65, 66, 98,

**AOT** ahead-of-time 32, 52, 81, 99, 115,

**ASIC** Application-specific Integrated Circuit 4, 39, 63,

**CNN** Convolutional Neural Network 4, 21, 24, 26, 27, 29, 30, 46, 47, 49, 50, 53, 55, 67, 78, 111, 112, 117,

**CPU** Central Processing Unit 4, 8, 33, 36, 37, 39–43, 54, 55, 57, 58, 62, 64, 66, 67, 72, 74, 75, 77, 78, 80, 82, 86, 90, 92, 93, 95, 98, 100, 104, 111–116, 119, 120, 123, 124, 126, 130, 134–136, 139–142, 144, 148, 149, 151, 152,

**CSE** common sub-expression elimination 31, 32, 35, 82,

**CSR** compressed sparse row 4, 27, 28, 73, 134,

**DAG** Directed Acyclic Graph 17, 96,

**DCE** dead code elimination 31, 32, 35,

**DLAS** Deep Learning Acceleration Stack 2–10, 12, 13, 29, 39, 40, 43, 44, 46, 54, 56, 62, 64, 65, 70–72, 74–76, 121, 122, 124, 125, 127–130, 132,

**DNN** Deep Neural Network 1–4, 6–8, 11–19, 21–24, 26, 29, 33–36, 39–56, 58–74, 76–78, 85, 86, 89–96, 98, 99, 102, 104, 105, 108, 109, 111, 112, 114, 117, 121–130, 134,

**DSE** design space exploration 7, 8, 11, 64–67, 69, 72–74, 76, 122–125, 128, 129,

**FPGA** Field Programmable Gate Array 4, 57, 63, 64, 67,

**GEMM** general matrix multiplication 4, 27, 33, 35, 40, 50, 53, 54, 62, 72, 74, 78, 85, 100, 101, 104, 114, 127

**GPU** Graphics Processing Unit 2, 4, 33, 36, 37, 39–43, 48, 51, 54, 55, 57, 58, 62, 64, 66, 72, 74, 75, 85, 86, 122, 123, 125–127, 130, 134, 135, 144–146, 148, 149, 154, 155,

**GSPC** Grouped Spatial Pack Convolution 77–86, 89–93, 123, 126, 127, 130,

**IoT** Internet of Things 40, 126, 127,

**IR** Intermediate Representation 30, 31, 36, 57, 58,

**ISA** Instruction Set Architecture 39, 40, 51, 86,

**JIT** Just-In-Time 33, 53, 57, 65,

**MAC** Multiply-Accumulate 7, 24, 25, 35, 50, 51, 78, 79, 82, 85, 86, 88, 89, 92, 121, 122,

**NAS** Neural Architecture Search 17, 18, 38, 46–48, 63, 76, 123, 125, 128,

**NLP** Natural Language Processing 13, 20, 21, 44, 47, 112,

**SECDA** SystemC Enabled Co-design of DNN Accelerators 66, 67,

**SGD** Stochastic Gradient Descent 16, 86, 132, 133,

**SIMD** Single instruction, multiple data 4, 27, 36, 42, 58, 62, 65, 82, 96, 100, 134,

**TPU** Tensor Processing Unit 4, 33, 39, 40, 51, 58, 63,

# Bibliography

[Aba+16] Martín Abadi et al. "TensorFlow: A System for Large-Scale Machine Learning". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. USA: USENIX Association, Nov. 2016, pp. 265–283. ISBN: 978-1-931971-33-1.

[AG18] Andrew Anderson and David Gregg. "Optimal DNN Primitive Selection with Partitioned Boolean Quadratic Programming". In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. CGO 2018. New York, NY, USA: ACM, 2018, pp. 340–351. ISBN: 978-1-4503-5617-6. DOI: 10.1145/3168805.

[Agg18] Charu C. Aggarwal. *Neural Networks and Deep Learning: A Textbook*. Cham: Springer International Publishing, 2018. DOI: 10.1007/978-3-319-94463-0.

[AH18] Dario Amodei and Danny Hernandez. *AI and Compute*. May 2018. URL: https://openai.com/research/ai-and-compute.

[Ahn+20] Byung Hoon Ahn et al. "Chameleon: Adaptive Code Optimization for Expedited Deep Neural Network Compilation". In: *Eighth International Conference on Learning Representations*. Apr. 2020. URL: https://iclr.cc/virtual_2020/poster_rygG4AVFvH.html.

[AKA22] Peter Ahrens et al. "Autoscheduling for Sparse Tensor Algebra with an Asymptotic Cost Model". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2022. New York, NY, USA: Association for Computing Machinery, June 2022, pp. 269–285. ISBN: 978-1-4503-9265-5. DOI: 10.1145/3519939.3523442.

[Alb+16] Jorge Albericio et al. "Cnvlutin: Ineffectual-neuron-free Deep Neural Network Computing". In: *Proceedings of the 43rd International Symposium on Computer Architecture*. ISCA '16. Seoul, Republic of Korea: IEEE Press, June

2016, pp. 1–13. ISBN: 978-1-4673-8947-1. DOI: 10.1109/ISCA.2016.11.

[Amd67] Gene M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). New York, NY, USA: Association for Computing Machinery, Apr. 1967, pp. 483–485. ISBN: 978-1-4503-7895-6. DOI: 10.1145/1465482.1465560.

[And+16] Renzo Andri et al. "YodaNN: An Ultra-Low Power Convolutional Neural Network Accelerator Based on Binary Weights". In: *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. July 2016, pp. 236–241. DOI: 10.1109/ISVLSI.2016.111.

[And+17a] Andrew Anderson et al. *triNNity*. 2017. URL: https://bitbucket.org/STG-TCD/trinnity.

[And+17b] Kota Ando et al. "BRein Memory: A 13-Layer 4.2 K Neuron/0.8 M Synapse Binary/Ternary Reconfigurable in-Memory Deep Neural Network Accelerator in 65 Nm CMOS". In: *2017 Symposium on VLSI Circuits*. June 2017, pp. C24–C25. DOI: 10.23919/VLSIC.2017.8008533.

[Arm17] Arm. *Compute Library*. Arm Software. Apr. 2017. URL: https://github.com/ARM-software/ComputeLibrary.

[ASA16] Seher Acer et al. "Improving Performance of Sparse Matrix Dense Matrix Multiplication on Large-scale Parallel Systems". In: *Parallel Computing*. Theory and Practice of Irregular Applications 59 (Nov. 2016), pp. 71–96. ISSN: 0167-8191. DOI: 10.1016/j.parco.2016.10.001.

[Ash+17] Amir H. Ashouri et al. "MiCOMP: Mitigating the Compiler Phase-Ordering Problem Using Optimization Sub-Sequences and Machine Learning". In: *ACM Transactions on Architecture and Code Optimization* 14.3 (Sept. 2017), 29:1–29:28. ISSN: 1544-3566. DOI: 10.1145/3124452.

[Ash+18] Amir H. Ashouri et al. "A Survey on Compiler Autotuning Using Machine Learning". In: *ACM Computing Surveys* 51.5 (Sept. 2018), 96:1–96:42. ISSN: 0360-0300. DOI: 10.1145/3197978.

[Asi+23] Kazi Asifuzzaman et al. "A Survey on Processing-in-memory Techniques: Advances and Challenges". In: *Memories - Materials, Devices, Circuits and Systems* 4 (July 2023), p. 100022. ISSN: 2773-0646. DOI: 10.1016/j.memori.2022.100022.

[Ask+23] MohammadHossein AskariHemmat et al. “Quark: An Integer RISC-V Vector Processor for Sub-Byte Quantized DNN Inference”. In: *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*. May 2023, pp. 1–5. DOI: 10.1109/ISCAS46773.2023.10181985.

[Ass22] Bonseyes Association. *Bonseyes AI Asset Container Generator*. 2022. URL: https://gitlab.com/bonseyes/artifacts/assets/aiasset_container_generator.

[Bag+19] 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)*. Feb. 2019, pp. 193–205. DOI: 10.1109/CGO.2019.8661197.

[Bag+21] Riyadh Baghdadi et al. “A Deep Learning Based Cost Model for Automatic Code Optimization”. In: *Proceedings of Machine Learning and Systems* 3 (Mar. 2021), pp. 181–193. URL: https://proceedings.mlsys.org/paper/2021/hash/3def184ad8f4755ff269862ea77393dd-Abstract.html.

[Bak+22] Bowen Baker et al. *Video PreTraining (VPT): Learning to Act by Watching Unlabeled Online Videos*. June 2022. DOI: 10.48550/arXiv.2206.11795. arXiv: 2206.11795 [cs].

[Bal+91] Wolfgang Balzer et al. “Weight Quantization in Boltzmann Machines”. In: *Neural Networks* 4.3 (Jan. 1991), pp. 405–409. ISSN: 0893-6080. DOI: 10.1016/0893-6080(91)90077-I.

[BDS19] Andrew Brock et al. *Large Scale GAN Training for High Fidelity Natural Image Synthesis*. Feb. 2019. DOI: 10.48550/arXiv.1809.11096. arXiv: 1809.11096 [cs, stat].

[Bel+18] Guillaume Bellec et al. “Deep Rewiring: Training Very Sparse Deep Networks”. In: *International Conference on Learning Representations*. 2018. URL: https://openreview.net/forum?id=BJ_wN01C-.

[Ben+10] Mohamed-Walid Benabderrahmane et al. “The Polyhedral Model Is More Widely Applicable Than You Think”. In: *Compiler Construction*. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 283–303. ISBN: 978-3-642-11970-5. DOI: 10.1007/978-3-642-11970-5_16.

[BI19] Paul Barham and Michael Isard. “Machine Learning Systems Are Stuck in a Rut”. In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. HotOS ’19. Bertinoro, Italy: Association for Computing Machinery, May

2019, pp. 177–183. ISBN: 978-1-4503-6727-1. DOI: 10.1145/3317550.3321441.

[Bik+22] Aart Bik et al. "Compiler Support for Sparse Tensor Computations in MLIR". In: *ACM Transactions on Architecture and Code Optimization* 19.4 (Sept. 2022), 50:1–50:25. ISSN: 1544-3566. DOI: 10.1145/3544559.

[Bla+20] Davis Blalock et al. "What Is the State of Neural Network Pruning?" In: *Proceedings of Machine Learning and Systems*. Vol. 2. 2020, pp. 129–146. URL: https://proceedings.mlsys.org/paper/2020/file/d2ddea18f00665ce8623e36bd4e3c7c5-Paper.pdf.

[BLZ+19] Junjie Bai et al. "ONNX: Open Neural Network Exchange". In: *GitHub repository* (2019). URL: https://github.com/onnx/onnx.

[Bou+22] Wadii Boulila et al. "Weight Initialization Techniques for Deep Learning Algorithms in Remote Sensing: Recent Trends and Future Perspectives". In: *Advances on Smart and Soft Computing*. Advances in Intelligent Systems and Computing. Singapore: Springer, 2022, pp. 477–484. ISBN: 9789811655593. DOI: 10.1007/978-981-16-5559-3_39.

[Bro+16] Greg Brockman et al. "OpenAI Gym". In: *arXiv:1606.01540 [cs]* (June 2016). arXiv: 1606.01540 [cs]. URL: http://arxiv.org/abs/1606.01540.

[Bro+17] Andrew Brock et al. "SMASH: One-Shot Model Architecture Search through HyperNetworks". In: *arXiv:1708.05344* (Aug. 2017). arXiv: 1708.05344.

[Bro+20] Tom B. Brown et al. "Language Models Are Few-shot Learners". In: *Proceedings of the 34th International Conference on Neural Information Processing Systems*. NIPS'20. Red Hook, NY, USA: Curran Associates Inc., Dec. 2020, pp. 1877–1901. ISBN: 978-1-71382-954-6.

[BTS21] Jakob Božič et al. "Mixed Supervision for Surface-Defect Detection: From Weakly to Fully Supervised Learning". In: *Computers in Industry* 129 (Aug. 2021), p. 103459. ISSN: 0166-3615. DOI: 10.1016/j.compind.2021.103459.

[Cae+20] Holger Caesar et al. "nuScenes: A Multimodal Dataset for Autonomous Driving". In: *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2020, pp. 11618–11628. DOI: 10.1109/CVPR42600.2020.01164.

[Cai+20] Han Cai et al. “Once-for-All: Train One Network and Specialize It for Efficient Deployment”. In: *Eighth International Conference on Learning Representations*. Apr. 2020. URL: https://iclr.cc/virtual_2020/poster_HylxE1HKwS.html.

[Cav+07] J. Cavazos et al. “Rapidly Selecting Good Compiler Optimizations Using Performance Counters”. In: *International Symposium on Code Generation and Optimization (CGO'07)*. Mar. 2007, pp. 185–197. DOI: 10.1109/CGO.2007.32.

[CB17] Lukas Cavigelli and Luca Benini. “Origami: A 803-GOp/s/W Convolutional Network Accelerator”. In: *IEEE Transactions on Circuits and Systems for Video Technology* 27.11 (Nov. 2017), pp. 2461–2475. ISSN: 1558-2205. DOI: 10.1109/TCSVT.2016.2592330.

[CBD15] Matthieu Courbariaux et al. “BinaryConnect: Training Deep Neural Networks with Binary Weights during Propagations”. In: *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*. NIPS'15. Cambridge, MA, USA: MIT Press, Dec. 2015, pp. 3123–3131.

[CG16] Tianqi Chen and Carlos Guestrin. “XGBoost: A Scalable Tree Boosting System”. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. New York, NY, USA: Association for Computing Machinery, Aug. 2016, pp. 785–794. ISBN: 978-1-4503-4232-2. DOI: 10.1145/2939672.2939785.

[CGS18] Elliot J. Crowley et al. “Moonshine: Distilling with Cheap Convolutions”. In: *Advances in Neural Information Processing Systems 31*. Curran Associates, Inc., 2018, pp. 2888–2898. DOI: 10.5555/3327144.3327212.

[Cha+18] Anirban Chakraborty et al. *Adversarial Attacks and Defences: A Survey*. Sept. 2018. DOI: 10.48550/arXiv.1810.00069. arXiv: 1810.00069 [cs, stat].

[Che+14a] Yunji Chen et al. “DaDianNao: A Machine-Learning Supercomputer”. In: *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. Dec. 2014, pp. 609–622. DOI: 10.1109/MICRO.2014.58.

[Che+14b] Sharan Chetlur et al. “cuDNN: Efficient Primitives for Deep Learning”. In: *arXiv:1410.0759 [cs]* (Oct. 2014). arXiv: 1410.0759 [cs]. URL: http://arxiv.org/abs/1410.0759.

[Che+15] Tianqi Chen et al. “MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems”. In: *Neural Information Processing Systems, Workshop on Machine Learning Systems*. 2015.

[Che+17] Y. Chen et al. “Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks”. In: *IEEE Journal of Solid-State Circuits* 52.1 (Jan. 2017), pp. 127–138. ISSN: 0018-9200. DOI: 10.1109/JSSC.2016.2616357.

[Che+18a] Tianqi Chen et al. “Learning to Optimize Tensor Programs”. In: *Advances in Neural Information Processing Systems 31*. Curran Associates, Inc., 2018, pp. 3393–3404. DOI: 10.5555/3327144.3327258.

[Che+18b] Tianqi Chen et al. “TVM: An Automated End-to-End Optimizing Compiler for Deep Learning”. In: *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. OSDI’18. USA: USENIX Association, Oct. 2018, pp. 579–594. ISBN: 978-1-931971-47-8.

[Che+18c] Yu Cheng et al. “Model Compression and Acceleration for Deep Neural Networks: The Principles, Progress, and Challenges”. In: *IEEE Signal Processing Magazine* 35.1 (Jan. 2018), pp. 126–136. ISSN: 1558-0792. DOI: 10.1109/MSP.2017.2765695.

[Che+19] Yu-Hsin Chen et al. “Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices”. In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9.2 (June 2019), pp. 292–308. ISSN: 2156-3365. DOI: 10.1109/JETCAS.2019.2910232.

[Che+20] Yanjiao Chen et al. “Deep Learning on Mobile and Embedded Devices: State-of-the-art, Challenges, and Future Directions”. In: *ACM Computing Surveys* 53.4 (Aug. 2020), 84:1–84:37. ISSN: 0360-0300. DOI: 10.1145/3398209.

[Chi+17] Soumith Chintala et al. *DLPack: Open In Memory Tensor Structure*. Distributed (Deep) Machine Learning Community. June 2017. URL: https://github.com/dmlc/dlpack.

[Cho+20a] Yunjey Choi et al. “StarGAN v2: Diverse Image Synthesis for Multiple Domains”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 8188–8197.

[Cho+20b] Tejalal Choudhary et al. “A Comprehensive Survey on Model Compression and Acceleration”. In: *Artificial Intelligence Review* 53.7 (Oct. 2020), pp. 5113–5155. ISSN: 1573-7462. DOI: 10.1007/s10462-020-09816-7.

[Cho+23] Krzysztof Marcin Choromanski et al. “Rethinking Attention with Performers”. In: *International Conference on Learning Representations*. Apr. 2023. URL: https://openreview.net/forum?id=Ua6zuk0WRH.

[Cho15] François Chollet. *Keras*. 2015. URL: https://keras.io/.

[Cho17] Francois Chollet. *Deep Learning with Python*. 1st. USA: Manning Publications Co., 2017. ISBN: 978-1-61729-443-3.

[CKA18] Stephen Chou et al. "Format Abstraction for Sparse Tensor Algebra Compilers". In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (Oct. 2018), 123:1–123:30. DOI: 10.1145/3276493.

[CKA20] Stephen Chou et al. "Automatic Generation of Efficient Sparse Tensor Format Conversion Routines". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. New York, NY, USA: Association for Computing Machinery, June 2020, pp. 823–838. ISBN: 978-1-4503-7613-6. DOI: 10.1145/3385412.3385963.

[Coo+05] Keith D. Cooper et al. "ACME: Adaptive Compilation Made Efficient". In: *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES '05. New York, NY, USA: Association for Computing Machinery, June 2005, pp. 69–77. ISBN: 978-1-59593-018-7. DOI: 10.1145/1065910.1065921.

[CPC17] Alfredo Canziani et al. *An Analysis of Deep Neural Network Models for Practical Applications*. Apr. 2017. DOI: 10.48550/arXiv.1605.07678. arXiv: 1605.07678 [cs].

[CPD01] R. Clint Whaley et al. "Automated Empirical Optimizations of Software and the ATLAS Project". In: *Parallel Computing*. New Trends in High Performance Computing 27.1 (Jan. 2001), pp. 3–35. ISSN: 0167-8191. DOI: 10.1016/S0167-8191(00)00087-9.

[CPJ21] Lizhong Chen et al. *AI for Computer Architecture: Principles, Practice, and Prospects*. Synthesis Lectures on Computer Architecture. Cham: Springer International Publishing, 2021. DOI: 10.1007/978-3-031-01770-4.

[CR19] Jiasi Chen and Xukan Ran. "Deep Learning With Edge Computing: A Review". In: *Proceedings of the IEEE* 107.8 (Aug. 2019), pp. 1655–1674. ISSN: 1558-2256. DOI: 10.1109/JPROC.2019.2921977.

[CSO22] Nihat Mert Cicek et al. "Energy Efficient Boosting of GEMM Accelerators for DNN via Reuse". In: *ACM Transactions on Design Automation of Electronic Systems* 27.5 (June 2022), 43:1–43:26. ISSN: 1084-4309. DOI: 10.1145/3503469.

[Cum+22] Chris Cummins et al. "CompilerGym: Robust, Performant Compiler Optimization Environments for AI Research". In: *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '22.

Virtual Event, Republic of Korea: IEEE Press, May 2022, pp. 92–105. ISBN: 978-1-66540-584-3. DOI: 10.1109/CGO53902.2022.9741258.

[CWC21] W. Chen et al. "Towards Mixed-precision Quantization of Neural Networks via Constrained Optimization". In: *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*. Los Alamitos, CA, USA: IEEE Computer Society, Oct. 2021, pp. 5330–5339. DOI: 10.1109/ICCV48922.2021.00530.

[CX14] Jason Cong and Bingjun Xiao. "Minimizing Computation in Convolutional Neural Networks". In: *Artificial Neural Networks and Machine Learning – ICANN 2014*. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014, pp. 281–290. ISBN: 978-3-319-11179-7. DOI: 10.1007/978-3-319-11179-7_36.

[Cyp+18] Scott Cyphers et al. *Intel nGraph: An Intermediate Representation, Compiler, and Executor for Deep Learning*. Jan. 2018. DOI: 10.48550/arXiv.1801.08058. arXiv: 1801.08058 [cs].

[Den+09] Jia Deng et al. "ImageNet: A Large-scale Hierarchical Image Database". In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. June 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.

[Den+20] Lei Deng et al. "Model Compression and Hardware Acceleration for Neural Networks: A Comprehensive Survey". In: *Proceedings of the IEEE* 108.4 (Apr. 2020), pp. 485–532. ISSN: 1558-2256. DOI: 10.1109/JPROC.2020.2976475.

[Den+74] R.H. Dennard et al. "Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions". In: *IEEE Journal of Solid-State Circuits* 9.5 (Oct. 1974), pp. 256–268. ISSN: 1558-173X. DOI: 10.1109/JSSC.1974.1050511.

[Den12] Li Deng. "The MNIST Database of Handwritten Digit Images for Machine Learning Research". In: *IEEE Signal Processing Magazine* 29.6 (Nov. 2012), pp. 141–142. ISSN: 1558-0792. DOI: 10.1109/MSP.2012.2211477.

[Dev+19] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423.

[Dhi+22] Pudi Dhilleswararao et al. “Efficient Hardware Architectures for Accelerating Deep Neural Networks: Survey”. In: *IEEE Access* 10 (2022), pp. 131788–131828. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2022.3229767.

[DHS11] John Duchi et al. “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization”. In: *Journal of Machine Learning Research* 12.61 (2011), pp. 2121–2159. ISSN: 1533-7928. URL: http://jmlr.org/papers/v12/duchi11a.html.

[Don+21a] Shi Dong et al. “Spartan: A Sparsity-Adaptive Framework to Accelerate Deep Neural Network Training on GPUs”. In: *IEEE Transactions on Parallel and Distributed Systems* 32.10 (Oct. 2021), pp. 2448–2463. ISSN: 1558-2183. DOI: 10.1109/TPDS.2021.3067825.

[Don+21b] Zhen Dong et al. “HAO: Hardware-aware Neural Architecture Optimization for Efficient Inference”. In: *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. May 2021, pp. 50–59. DOI: 10.1109/FCCM51124.2021.00014.

[Don+21c] Zheng Dong et al. “Location-Aware Single Image Reflection Removal”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. IEEE Computer Society, 2021, pp. 5017–5026. DOI: 10.1109/ICCV48922.2021.00497.

[Dos+17] Alexey Dosovitskiy et al. “CARLA: An Open Urban Driving Simulator”. In: *Proceedings of the 1st Annual Conference on Robot Learning*. 2017, pp. 1–16.

[Du+15] Zidong Du et al. “ShiDianNao: Shifting Vision Processing Closer to the Sensor”. In: *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. June 2015, pp. 92–104. DOI: 10.1145/2749469.2750389.

[DV18] Vincent Dumoulin and Francesco Visin. *A Guide to Convolution Arithmetic for Deep Learning*. Jan. 2018. DOI: 10.48550/arXiv.1603.07285. arXiv: 1603.07285 [cs, stat].

[DWA21] Shi Dong et al. “A Survey on Deep Learning and Its Applications”. In: *Computer Science Review* 40 (May 2021), p. 100379. ISSN: 1574-0137. DOI: 10.1016/j.cosrev.2021.100379.

[Ett+21] Scott Ettinger et al. “Large Scale Interactive Motion Forecasting for Autonomous Driving: The Waymo Open Motion Dataset”. In: *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*. Oct. 2021, pp. 9690–9699. DOI: 10.1109/ICCV48922.2021.00957.

[Evc+20] Utku Evci et al. “Rigging the Lottery: Making All Tickets Winners”. In: *Proceedings of the 37th International Conference on Machine Learning*. Vol. 119. ICML’20. JMLR.org, July 2020, pp. 2943–2952.

[Fai+20] Susan Fairley et al. “The International Genome Sample Resource (IGSR) Collection of Open Human Genomic Variation Resources”. In: *Nucleic Acids Research* 48.D1 (Jan. 2020), pp. D941–D947. ISSN: 0305-1048. DOI: 10.1093/nar/gkz836.

[Fan+20] Jingzhi Fang et al. “Optimizing DNN Computation Graph Using Graph Substitutions”. In: *Proceedings of the VLDB Endowment* 13.12 (July 2020), pp. 2734–2746. ISSN: 2150-8097. DOI: 10.14778/3407790.3407857.

[FC19] Jonathan Frankle and Michael Carbin. “The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks”. In: *International Conference on Learning Representations*. 2019. URL: https://openreview.net/forum?id=rJl-b3RcF7.

[FCC90] Emile Fiesler et al. “Weight Discretization Paradigm for Optical Neural Networks”. In: *The Hague ’90, 12-16 April*. The Hague, Netherlands, Aug. 1990, p. 164. DOI: 10.1117/12.20700.

[FJL18] Roy Frostig et al. “Compiling Machine Learning Programs via High-level Tracing”. In: *Systems for Machine Learning* 4.9 (2018).

[FT19] William Falcon and The PyTorch Lightning team. *PyTorch Lightning*. Mar. 2019. DOI: 10.5281/zenodo.3828935.

[FW19] Adi Fuchs and David Wentzlaff. “The Accelerator Wall: Limits of Chip Specialization”. In: *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2019, pp. 1–14. DOI: 10.1109/HPCA.2019.00023.

[Gal+20] Trevor Gale et al. “Sparse GPU Kernels for Deep Learning”. In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. Nov. 2020, pp. 1–14. DOI: 10.1109/SC41405.2020.00021.

[Gao+23] Jianhua Gao et al. “A Systematic Survey of General Sparse Matrix-matrix Multiplication”. In: *ACM Computing Surveys* 55.12 (Mar. 2023), 244:1–244:36. ISSN: 0360-0300. DOI: 10.1145/3571157.

[Gar+93] John S. Garofolo et al. *TIMIT Acoustic-Phonetic Continuous Speech Corpus*. 1993. DOI: 10.35111/17GK-BN40.

[GBC16] Ian Goodfellow et al. *Deep Learning*. MIT Press, 2016. ISBN: 978-0-262-03561-3.

[GC20] Perry Gibson and José Cano. "Orpheus: A New Deep Learning Framework for Easy Deployment and Evaluation of Edge Inference". In: *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2020, pp. 229–230. DOI: 10.1109/ISPASS48437.2020.00042.

[GC22] Perry Gibson and José Cano. "Productive Reproducible Workflows for DNNs: A Case Study for Industrial Defect Detection". In: *4th Workshop on Accelerated Machine Learning (AccML)*. 2022. DOI: 10.48550/arXiv.2206.09359.

[GC23] Perry Gibson and José Cano. "Transfer-Tuning: Reusing Auto-Schedules for Efficient Tensor Program Code Generation". In: *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. PACT '22. New York, NY, USA: Association for Computing Machinery, Jan. 2023, pp. 28–39. ISBN: 978-1-4503-9868-8. DOI: 10.1145/3559009.3569682.

[Geo+18] Evangelos Georganas et al. "Anatomy of High-performance Deep Learning Convolutions on SIMD Architectures". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. SC '18. Piscataway, NJ, USA: IEEE Press, 2018, 66:1–66:12. DOI: 10.1109/SC.2018.00069.

[Gib+20] Perry Gibson et al. "Optimizing Grouped Convolutions on Edge Devices". In: *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 2020, pp. 189–196. DOI: 10.1109/ASAP49362.2020.00039.

[GMG16] Philipp Gysel et al. "Hardware-Oriented Approximation of Convolutional Neural Networks". In: *International Conference on Learning Representations (ICLR)*. 2016.

[Gon+19] Ashish Gondimalla et al. "SparTen: A Sparse Tensor Accelerator for Convolutional Neural Networks". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '52. New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 151–165. ISBN: 978-1-4503-6938-1. DOI: 10.1145/3352460.3358291.

[Goo+20] Ian Goodfellow et al. "Generative Adversarial Networks". In: *Communications of the ACM* 63.11 (2020), pp. 139–144.

[Goo19] Google. *TensorFlow Lite*. 2019. URL: https://www.tensorflow.org/lite/.

[Gua+17] Yijin Guan et al. “FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates”. In: *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Apr. 2017, pp. 152–159. DOI: 10.1109/FCCM.2017.25.

[Gus78] Fred G. Gustavson. “Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition”. In: *ACM Transactions on Mathematical Software* 4.3 (Sept. 1978), pp. 250–269. ISSN: 0098-3500. DOI: 10.1145/355791.355796.

[Had+19] Ramyad Hadidi et al. “Characterizing the Deployment of Deep Neural Networks on Commercial Edge Devices”. In: *2019 IEEE International Symposium on Workload Characterization (IISWC)*. Nov. 2019, pp. 35–48. DOI: 10.1109/IISWC47752.2019.9041955.

[Hag+20] Bastian Hagedorn et al. “Achieving High-performance the Functional Way: A Functional Pearl on Expressing High-performance Optimizations as Rewrite Strategies”. In: *Proceedings of the ACM on Programming Languages* 4.ICFP (Aug. 2020), 92:1–92:29. DOI: 10.1145/3408974.

[Han+15] Song Han et al. “Learning Both Weights and Connections for Efficient Neural Networks”. In: *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*. NeurIPS’15. Montreal, Canada: MIT Press, Dec. 2015, pp. 1135–1143.

[Hao+19] Cong Hao et al. “FPGA/DNN Co-Design: An Efficient Design Methodology for IoT Intelligence on the Edge”. In: *Proceedings of the 56th Annual Design Automation Conference 2019*. DAC ’19. Las Vegas, NV, USA: Association for Computing Machinery, June 2019, pp. 1–6. ISBN: 978-1-4503-6725-7. DOI: 10.1145/3316781.3317829.

[Har+21] Jude Haris et al. “SECDA: Efficient Hardware/Software Co-Design of FPGA-based DNN Accelerators for Edge Inference”. In: *IEEE 33rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. Oct. 2021, pp. 33–43. DOI: 10.1109/SBAC-PAD53543.2021.00015.

[Har+23] Jude Haris et al. “SECDA-TFLite: A Toolkit for Efficient Development of FPGA-based DNN Accelerators for Edge Inference”. In: *Journal of Parallel and Distributed Computing* 173 (Mar. 2023), pp. 140–151. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2022.11.005.

[HB20] Danny Hernandez and Tom B. Brown. "Measuring the Algorithmic Efficiency of Neural Networks". In: *arXiv:2005.04305 [cs, stat]* (May 2020). arXiv: 2005.04305 [cs, stat]. URL: http://arxiv.org/abs/2005.04305.

[He+16] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[HGC23] Wenhao Hu et al. "ICE-Pick: Iterative Cost-Efficient Pruning for DNNs". In: *Neural Compression: From Information Theory to Applications – Workshop @ ICML*. 2023.

[HKV19] Frank Hutter et al., eds. *Automated Machine Learning: Methods, Systems, Challenges*. The Springer Series on Challenges in Machine Learning. Springer International Publishing, 2019. ISBN: 978-3-030-05317-8. DOI: 10.1007/978-3-030-05318-5.

[HMD16] Song Han et al. "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization, and Huffman Coding". In: *International Conference on Learning Representations (ICLR)* (2016).

[Hoe+21] Torsten Hoefler et al. "Sparsity in Deep Learning: Pruning and Growth for Efficient Inference and Training in Neural Networks". In: *The Journal of Machine Learning Research* 22.1 (Jan. 2021), 241:10882–241:11005. ISSN: 1532-4435.

[Hoo21] Sara Hooker. "The Hardware Lottery". In: *Communications of the ACM* 64.12 (Nov. 2021), pp. 58–65. ISSN: 0001-0782. DOI: 10.1145/3467017.

[Hor91] Kurt Hornik. "Approximation Capabilities of Multilayer Feedforward Networks". In: *Neural Networks* 4.2 (Jan. 1991), pp. 251–257. ISSN: 0893-6080. DOI: 10.1016/0893-6080(91)90009-T.

[How+17] Andrew G. Howard et al. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". In: *arXiv:1704.04861 [cs]* (Apr. 2017). arXiv: 1704.04861 [cs]. URL: http://arxiv.org/abs/1704.04861.

[HP18] John L. Hennessy and David A. Patterson. "A New Golden Age for Computer Architecture: Domain-specific Hardware/Software Co-design, Enhanced Security, Open Instruction Sets, and Agile Chip Development". In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. June 2018, pp. 27–29. DOI: 10.1109/ISCA.2018.00011.

[Hua+17] Gao Huang et al. “Densely Connected Convolutional Networks”. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. July 2017, pp. 2261–2269. DOI: 10.1109/CVPR.2017.243.

[Hua+18] Gao Huang et al. “CondenseNet: An Efficient DenseNet Using Learned Group Convolutions”. In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. June 2018, pp. 2752–2761. DOI: 10.1109/CVPR.2018.00291.

[Hua+19] Yanping Huang et al. “GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism”. In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. 10. Red Hook, NY, USA: Curran Associates Inc., Dec. 2019, pp. 103–112.

[Hub+16] Itay Hubara et al. “Binarized Neural Networks”. In: *Advances in Neural Information Processing Systems 29*. Curran Associates, Inc., 2016, pp. 4107–4115. URL: http://papers.nips.cc/paper/6573-binarized-neural-networks.pdf.

[IEE12] IEEE. “IEEE Standard for Standard SystemC Language Reference Manual”. In: *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)* (Jan. 2012), pp. 1–638. DOI: 10.1109/IEEESTD.2012.6134619.

[Int20] Intel. *oneDNN*. oneAPI. June 2020. URL: https://github.com/oneapi-src/oneDNN.

[Ioa+17] Yani Ioannou et al. “Deep Roots: Improving CNN Efficiency with Hierarchical Filter Groups”. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. July 2017, pp. 5977–5986. DOI: 10.1109/CVPR.2017.633.

[IS15] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. Mar. 2015. DOI: 10.48550/arXiv.1502.03167. arXiv: 1502.03167 [cs].

[Jac+18] Benoit Jacob et al. “Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference”. In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. June 2018, pp. 2704–2713. DOI: 10.1109/CVPR.2018.00286.

[Jal+10] Aamer Jaleel et al. “High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP)”. In: *Proceedings of the 37th Annual International Symposium on Computer Architecture*. ISCA ’10. New York, NY, USA: Association for Computing Machinery, June 2010, pp. 60–71. ISBN: 978-1-4503-0053-7. DOI: 10.1145/1815961.1815971.

[Jeo+23] Byungsoo Jeon et al. “Collage: Seamless Integration of Deep Learning Backends with Automatic Placement”. In: *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. PACT ’22. New York, NY, USA: Association for Computing Machinery, Jan. 2023, pp. 517–529. ISBN: 978-1-4503-9868-8. DOI: 10.1145/3559009.3569651.

[Jia+19] 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*. SOSP ’19. New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 47–62. ISBN: 978-1-4503-6873-5. DOI: 10.1145/3341301.3359630.

[Jia+20] Weiwen Jiang et al. “Standing on the Shoulders of Giants: Hardware and Neural Architecture Co-Search With Hot Start”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.11 (Nov. 2020), pp. 4154–4165. ISSN: 1937-4151. DOI: 10.1109/TCAD.2020.3012863.

[Jou+17] 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 ’17. Toronto, ON, Canada: Association for Computing Machinery, June 2017, pp. 1–12. ISBN: 978-1-4503-4892-8. DOI: 10.1145/3079856.3080246.

[JPR22] Charles Jin et al. “Neural Architecture Search Using Property Guided Synthesis”. In: *Proceedings of the ACM on Programming Languages* 6.OOPSLA2 (Oct. 2022), 166:1150–166:1179. DOI: 10.1145/3563329.

[Kan+17] Yiping Kang et al. “Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge”. In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’17. New York, NY, USA: ACM, 2017, pp. 615–629. ISBN: 978-1-4503-4465-4. DOI: 10.1145/3037697.3037698.

[Kar+20a] Tero Karras et al. “Analyzing and Improving the Image Quality of StyleGAN”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. IEEE Computer Society, 2020, pp. 8110–8119. DOI: 10.1109/CVPR42600.2020.00813.

[Kar+20b] Tero Karras et al. “Training Generative Adversarial Networks with Limited Data”. In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., 2020, pp. 12104–12114. DOI: 10.5555/3495724.3496739.

[Kat+20] Angelos Katharopoulos et al. “Transformers Are RNNs: Fast Autoregressive Transformers with Linear Attention”. In: *Proceedings of the 37th International Conference on Machine Learning*. Vol. 119. ICML’20. JMLR.org, July 2020, pp. 5156–5165. DOI: 10.5555/3524938.3525416.

[KB15] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. 2015.

[kG23] kuangliu and Perry Gibson. *PyTorch Lightning CIFAR10*. Mar. 2023. URL: https://github.com/Wheest/pytorch-lightning-cifar.

[KH91] Anders Krogh and John A. Hertz. “A Simple Weight Decay Can Improve Generalization”. In: *Proceedings of the 4th International Conference on Neural Information Processing Systems*. NIPS’91. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Dec. 1991, pp. 950–957. ISBN: 978-1-55860-222-9.

[Kha+19] Jehandad Khan et al. *MIOpen: An Open Source Library For Deep Learning Primitives*. Sept. 2019. DOI: 10.48550/arXiv.1910.00078. arXiv: 1910.00078 [cs, stat].

[Kjo+17] Fredrik Kjolstad et al. “The Tensor Algebra Compiler”. In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (Oct. 2017), 77:1–77:29. DOI: 10.1145/3133901.

[KLA19] Tero Karras et al. “A Style-Based Generator Architecture for Generative Adversarial Networks”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. IEEE Computer Society, 2019, pp. 4401–4410. DOI: 10.1109/TPAMI.2020.2970919.

[Koa+16] Penporn Koanantakool et al. “Communication-Avoiding Parallel Sparse-Dense Matrix-Matrix Multiplication”. In: *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. May 2016, pp. 842–853. DOI: 10.1109/IPDPS.2016.117.

[Kra+17] Ivan Krasin et al. “OpenImages: A Public Dataset for Large-Scale Multi-Label and Multi-Class Image Classification.” In: (2017). DOI: https://storage.googleapis.com/openimages/web/index.html.

[Kri09] Alex Krizhevsky. “Learning Multiple Layers of Features from Tiny Images”. In: (2009), p. 60.

[KSH12] Alex Krizhevsky et al. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Communications of the ACM* 60.6 (2012), pp. 84–90. ISSN: 00010782. DOI: 10.1145/3065386.

[KSK18] Hyoukjun Kwon et al. “MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects”. In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’18. Williamsburg, VA, USA: Association for Computing Machinery, Mar. 2018, pp. 461–475. ISBN: 978-1-4503-4911-6. DOI: 10.1145/3173162.3173176.

[KTZ19] Sachin Kumar et al. “Internet of Things Is a Revolutionary Approach for Future Technology Enhancement: A Review”. In: *Journal of Big Data* 6.1 (Dec. 2019), p. 111. ISSN: 2196-1115. DOI: 10.1186/s40537-019-0268-2.

[Lat+21] Chris Lattner et al. “MLIR: Scaling Compiler Infrastructure for Domain Specific Computation”. In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. Feb. 2021, pp. 2–14. DOI: 10.1109/CGO51591.2021.9370308.

[Lat02] Chris Lattner. “LLVM: An Infrastructure for Multi-Stage Optimization”. PhD thesis. Urbana, IL: Computer Science Dept., University of Illinois at Urbana-Champaign, Dec. 2002.

[LAT19] Namhoon Lee et al. “SNIP: Single-shot Network Pruning Based on Connection Sensitivity”. In: *International Conference on Learning Representations (ICLR)*. 2019. URL: https://openreview.net/forum?id=B1VZqjAcYX.

[Lat21] Chris Lattner. “The Golden Age of Compiler Design in an Era of HW/SW Co-design”. In: *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2021.

[LC20] Hugh Leather and Chris Cummins. “Machine Learning in Compilers: Past, Present, and Future”. In: *2020 Forum for Specification and Design Languages (FDL)*. Sept. 2020, pp. 1–8. DOI: 10.1109/FDL50818.2020.9232934.

[LCO18] Manolis Loukadakis et al. “Accelerating Deep Neural Networks on Low Power Heterogeneous Architectures”. In: *Eleventh International Workshop on Programmability and Architectures for Heterogeneous Multicores (MULTIPROG-2018)*. Manchester, UK, Jan. 2018. URL: http://eprints.gla.ac.uk/183819/.

[Lec+98] Y. Lecun et al. “Gradient-Based Learning Applied to Document Recognition”. In: *Proceedings of the IEEE* 86.11 (Nov. 1998), pp. 2278–2324. ISSN: 1558-2256. DOI: 10.1109/5.726791.

[Lee+17] Edward H. Lee et al. “LogNet: Energy-efficient Neural Networks Using Logarithmic Computation”. In: *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. Mar. 2017, pp. 5900–5904. DOI: 10.1109/ICASSP.2017.7953288.

[Lee+19] W. Lee et al. “White-Box Program Tuning”. In: *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. Feb. 2019, pp. 122–135. DOI: 10.1109/CGO.2019.8661177.

[LG16] Andrew Lavin and Scott Gray. “Fast Algorithms for Convolutional Neural Networks”. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016, pp. 4013–4021. DOI: 10.1109/CVPR.2016.435.

[LH17] Ilya Loshchilov and Frank Hutter. “SGDR: Stochastic Gradient Descent with Warm Restarts”. In: *International Conference on Learning Representations (ICLR)*. 2017. URL: https://openreview.net/forum?id=Skq89Scxx.

[Li+20a] Tian Li et al. “Federated Learning: Challenges, Methods, and Future Directions”. In: *IEEE Signal Processing Magazine* 37.3 (May 2020), pp. 50–60. ISSN: 1558-0792. DOI: 10.1109/MSP.2020.2975749.

[Li+20b] Yuhong Li et al. “EDD: Efficient Differentiable DNN Architecture and Implementation Co-search for Embedded AI Solutions”. In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. July 2020, pp. 1–6. DOI: 10.1109/DAC18072.2020.9218749.

[Li+21] Mingzhen Li et al. “The Deep Learning Compiler: A Comprehensive Survey”. In: *IEEE Transactions on Parallel and Distributed Systems* 32.3 (Mar. 2021), pp. 708–727. ISSN: 1558-2183. DOI: 10.1109/TPDS.2020.3030548.

[Li+22a] Fengfu Li et al. *Ternary Weight Networks*. Nov. 2022. DOI: 10.48550/arXiv.1605.04711. arXiv: 1605.04711 [cs].

[Li+22b] Shengzhao Li et al. “An Efficient CNN Accelerator Using Inter-Frame Data Reuse of Videos on FPGAs”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 30.11 (Nov. 2022), pp. 1587–1600. ISSN: 1557-9999. DOI: 10.1109/TVLSI.2022.3151788.

[Li+22c] Wei Li et al. *SepViT: Separable Vision Transformer*. May 2022. DOI: 10.48550/arXiv.2203.15380. arXiv: 2203.15380 [cs].

[Lia+21] Tailin Liang et al. “Pruning and Quantization for Deep Neural Network Acceleration: A Survey”. In: *Neurocomputing* 461 (Oct. 2021), pp. 370–403. ISSN: 0925-2312. DOI: 10.1016/j.neucom.2021.07.045.

[Lin+14] Tsung-Yi Lin et al. “Microsoft COCO: Common Objects in Context”. In: *Computer Vision – ECCV 2014*. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014, pp. 740–755. ISBN: 978-3-319-10602-1. DOI: 10.1007/978-3-319-10602-1_48.

[Lin+20] Ji Lin et al. “MCUNet: Tiny Deep Learning on IoT Devices”. In: *Proceedings of the 34th International Conference on Neural Information Processing Systems*. NIPS’20. Red Hook, NY, USA: Curran Associates Inc., Dec. 2020, pp. 11711–11722. ISBN: 978-1-71382-954-6.

[Liu+15] Ziwei Liu et al. “Deep Learning Face Attributes in the Wild”. In: *2015 IEEE International Conference on Computer Vision (ICCV)*. Dec. 2015, pp. 3730–3738. DOI: 10.1109/ICCV.2015.425.

[Liu+19] Zhuang Liu et al. “Rethinking the Value of Network Pruning”. In: *International Conference on Learning Representations*. 2019. URL: https://openreview.net/forum?id=rJlnB3C5Ym.

[Liu+22] Hsin-I Cindy Liu et al. “TinyIREE: An ML Execution Environment for Embedded Systems From Compilation to Deployment”. In: *IEEE Micro* 42.5 (Sept. 2022), pp. 9–16. ISSN: 0272-1732. DOI: 10.1109/MM.2022.3178068.

[Liu20] Renjie Liu. *Higher Accuracy on Vision Models with EfficientNet-Lite*. Mar. 2020. URL: https://blog.tensorflow.org/2020/03/higher-accuracy-on-vision-models-with-efficientnet-lite.html.

[Lle+17] Tim Llewellynn et al. “BONSEYES: Platform for Open Development of Systems of Artificial Intelligence: Invited Paper”. In: ACM Press, 2017, pp. 299–304. ISBN: 978-1-4503-4487-6. DOI: 10.1145/3075564.3076259.

[Lon+18] Guoping Long et al. *FusionStitching: Deep Fusion and Code Generation for Tensorflow Computations on GPUs*. Nov. 2018. DOI: 10.48550/arXiv.1811.05213. arXiv: 1811.05213 [cs].

[Lou+20] Qian Lou et al. “AutoQ: Automated Kernel-Wise Neural Network Quantization”. In: *International Conference on Learning Representations*. Mar. 2020. URL: https://openreview.net/forum?id=rygfnn4twS.

[Lou+22] Nikolaos Louloudakis et al. “Assessing Robustness of Image Recognition Models to Changes in the Computational Environment”. In: *NeurIPS ML Safety Workshop*. Dec. 2022. URL: https://openreview.net/forum?id=-7DjNGvdpx.

[Lou+23] Nick Louloudakis et al. “Fault Localization for Buggy Deep Learning Framework Conversions in Image Recognition”. In: *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*. ASE ’23. New York, NY, USA: Association for Computing Machinery, Sept. 2023, pp. 1–5.

[LPT17] Sheng Li et al. *Enabling Sparse Winograd Convolution by Native Pruning*. Oct. 2017. DOI: 10.48550/arXiv.1702.08597. arXiv: 1702.08597 [cs].

[LSC18] Liangzhen Lai et al. “Not All Ops Are Created Equal!” In: *SysML Conference*. SysML Conference, Jan. 2018.

[Lu+17a] Wenyan Lu et al. “FlexFlow: A Flexible Dataflow Accelerator Architecture for Convolutional Neural Networks”. In: *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2017, pp. 553–564. DOI: 10.1109/HPCA.2017.29.

[Lu+17b] Zhou Lu et al. “The Expressive Power of Neural Networks: A View from the Width”. In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS’17. Red Hook, NY, USA: Curran Associates Inc., Dec. 2017, pp. 6232–6240. ISBN: 978-1-5108-6096-4.

[Luo+20] X. Luo et al. “EdgeNAS: Discovering Efficient Neural Architectures for Edge Systems”. In: *2020 IEEE 38th International Conference on Computer Design (ICCD)*. Oct. 2020, pp. 288–295. DOI: 10.1109/ICCD50377.2020.00056.

[LYL19] Guoping Long et al. *FusionStitching: Boosting Execution Efficiency of Memory Intensive Computations for DL Workloads*. Nov. 2019. DOI: 10.48550/arXiv.1911.11576. arXiv: 1911.11576 [cs].

[Ma22] Enze Ma. “DLGR: A Rule-Based Approach to Graph Replacement for Deep Learning”. In: *2022 26th International Conference on Engineering of Complex Computer Systems (ICECCS)*. Mar. 2022, pp. 183–188. DOI: 10.1109/ICECCS54210.2022.00030.

[Maa+11] Andrew L. Maas et al. “Learning Word Vectors for Sentiment Analysis”. In: *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies - Volume 1*. HLT ’11. USA: Association for Computational Linguistics, June 2011, pp. 142–150. ISBN: 978-1-932432-87-9.

[Mar+16] Luiz G. A. Martins et al. “Clustering-Based Selection for the Exploration of Compiler Optimization Sequences”. In: *ACM Transactions on Architecture and Code Optimization* 13.1 (Mar. 2016), 8:1–8:28. ISSN: 1544-3566. DOI: 10.1145/2883614.

[Mar+18] Stefano Markidis et al. “NVIDIA Tensor Core Programmability, Performance & Precision”. In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. May 2018, pp. 522–531. DOI: 10.1109/IPDPSW.2018.00091.

[mc16] TorchVision maintainers and contributors. *TorchVision: PyTorch’s Computer Vision Library*. Nov. 2016. URL: https://github.com/pytorch/vision.

[MHL14] Michael Mathieu et al. *Fast Training of Convolutional Networks through FFTs*. Mar. 2014. DOI: 10.48550/arXiv.1312.5851. arXiv: 1312.5851 [cs].

[MLM16] Daisuke Miyashita et al. “Convolutional Neural Networks Using Logarithmic Data Representation”. In: *arXiv:1603.01025 [cs]* (Mar. 2016). arXiv: 1603.01025 [cs]. URL: http://arxiv.org/abs/1603.01025.

[Mol+19] P. Molchanov et al. “Importance Estimation for Neural Network Pruning”. In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, June 2019, pp. 11256–11264. DOI: 10.1109/CVPR.2019.01152.

[Mor+19] Thierry Moreau et al. “A Hardware-Software Blueprint for Flexible Deep Learning Specialization”. In: *arXiv:1807.04188 [cs, stat]* (Apr. 2019). arXiv: 1807.04188 [cs, stat].

[MP69] Marvin Minsky and Seymour Papert. *Perceptrons*. Cambridge, MA: MIT Press, 1969. ISBN: 0-262-13043-2.

[Muñ+21] Francisco Muñoz-Martínez et al. “STONNE: Enabling Cycle-Level Microarchitectural Simulation for DNN Inference Accelerators”. In: *2021 IEEE International Symposium on Workload Characterization (IISWC)*. Nov. 2021, pp. 201–213. DOI: 10.1109/IISWC53511.2021.00028.

[NCZ17] Arsha Nagrani et al. “VoxCeleb: A Large-Scale Speaker Identification Dataset”. In: *Interspeech 2017*. Aug. 2017, pp. 2616–2620. DOI: 10.21437/Interspeech.2017-950. arXiv: 1706.08612 [cs].

[Net+11] Yuval Netzer et al. “Reading Digits in Natural Images with Unsupervised Feature Learning”. In: *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*. 2011.

[NH10] Vinod Nair and Geoffrey E. Hinton. "Rectified Linear Units Improve Restricted Boltzmann Machines". In: *Proceedings of the 27th International Conference on International Conference on Machine Learning*. ICML'10. Madison, WI, USA: Omnipress, June 2010, pp. 807–814. ISBN: 978-1-60558-907-7.

[Nic+08] John Nickolls et al. "Scalable Parallel Programming with CUDA: Is CUDA the Parallel Programming Model That Application Developers Have Been Waiting For?" In: *Queue* 6.2 (Mar. 2008), pp. 40–53. ISSN: 1542-7730. DOI: 10.1145/1365490.1365500.

[Nug18] Cedric Nugteren. "CLBlast: A Tuned OpenCL BLAS Library". In: *Proceedings of the International Workshop on OpenCL*. IWOCL '18. New York, NY, USA: ACM, 2018, 5:1–5:10. ISBN: 978-1-4503-6439-3. DOI: 10.1145/3204919.3204924.

[Nvi16] Nvidia. *TensorRT: Programmable Inference Accelerator*. Apr. 2016. URL: https://developer.nvidia.com/tensorrt.

[ONN18] ONNX Runtime developers. *ONNX Runtime*. Nov. 2018. URL: https://github.com/microsoft/onnxruntime.

[Ope23] OpenAI. *GPT-4 Technical Report*. Mar. 2023. DOI: 10.48550/arXiv.2303.08774. arXiv: 2303.08774 [cs].

[Ott+20] Gianmarco Ottavi et al. "A Mixed-Precision RISC-V Processor for Extreme-Edge DNN Inference". In: *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. July 2020, pp. 512–517. DOI: 10.1109/ISVLSI49217.2020.000-5.

[Pan+15] 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)*. Apr. 2015, pp. 5206–5210. DOI: 10.1109/ICASSP.2015.7178964.

[Pap23] Papers with Code. *Papers with Code - CIFAR-10 Benchmark (Image Classification)*. Mar. 2023. URL: https://paperswithcode.com/sota/image-classification-on-cifar-10.

[Par+17a] Angshuman Parashar et al. "SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. Toronto ON Canada: ACM, June 2017, pp. 27–40. ISBN: 978-1-4503-4892-8. DOI: 10.1145/3079856.3080254.

[Par+17b] Jongsoo Park et al. “Faster CNNs with Direct Sparse Convolutions and Guided Pruning”. In: *International Conference on Learning Representations*. 2017. URL: https://openreview.net/forum?id=rJPcZ3txx.

[Pas+19] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., Dec. 2019, pp. 8026–8037.

[Pat+21] David Patterson et al. *Carbon Emissions and Large Neural Network Training*. Apr. 2021. DOI: 10.48550/arXiv.2104.10350. arXiv: 2104.10350 [cs].

[PB14] A. J. Peña and P. Balaji. “Toward the Efficient Use of Multiple Explicitly Managed Memory Subsystems”. In: *2014 IEEE International Conference on Cluster Computing (CLUSTER)*. Sept. 2014, pp. 123–131. DOI: 10.1109/CLUSTER.2014.6968756.

[Pha+18] Hieu Pham et al. “Efficient Neural Architecture Search via Parameters Sharing”. In: *International Conference on Machine Learning*. PMLR, July 2018, pp. 4095–4104. URL: http://proceedings.mlr.press/v80/pham18a.html.

[Pm+19] Wyder Pm et al. *Autonomous Drone Hunter Operating by Deep Learning and All-Onboard Computations in GPS-denied Environments*. Nov. 2019. DOI: 10.1371/journal.pone.0225092.

[PPB19] Miguel de Prado et al. “Learning to Infer: RL-based Search for DNN Primitive Selection on Heterogeneous Embedded Systems”. In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Mar. 2019, pp. 1409–1414. DOI: 10.23919/DATE.2019.8714959.

[Qin+20] E. Qin et al. “SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training”. In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2020, pp. 58–70. DOI: 10.1109/HPCA47549.2020.00015.

[Rad+18] Alec Radford et al. “Improving Language Understanding by Generative Pre-training”. In: *OpenAI blog* (2018), p. 12.

[Rad+19] Alec Radford et al. “Language Models Are Unsupervised Multitask Learners”. In: *OpenAI blog* (2019), p. 24.

[Rag+17] Jonathan Ragan-Kelley et al. “Halide: Decoupling Algorithms from Schedules for High-Performance Image Processing”. In: *Communications of the ACM* 61.1 (Dec. 2017), pp. 106–115. ISSN: 0001-0782. DOI: 10.1145/3150211.

[RAG18] M. Riera et al. “Computation Reuse in DNNs by Exploiting Input Similarity”. In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. June 2018, pp. 57–68. DOI: 10.1109/ISCA.2018.00016.

[Raj+16] Pranav Rajpurkar et al. “SQuAD: 100,000+ Questions for Machine Comprehension of Text”. In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Austin, Texas: Association for Computational Linguistics, Nov. 2016, pp. 2383–2392. DOI: 10.18653/v1/D16-1264.

[Ras+16] Mohammad Rastegari et al. “XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks”. In: *Computer Vision – ECCV 2016*. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 525–542. ISBN: 978-3-319-46493-0. DOI: 10.1007/978-3-319-46493-0_32.

[Ras+21] Ari Rasch et al. “Efficient Auto-Tuning of Parallel Programs with Interdependent Tuning Parameters via Auto-Tuning Framework (ATF)”. In: *ACM Transactions on Architecture and Code Optimization* 18.1 (Jan. 2021), 1:1–1:26. ISSN: 1544-3566. DOI: 10.1145/3427093.

[Red+16] Joseph Redmon et al. “You Only Look Once: Unified, Real-Time Object Detection”. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016, pp. 779–788. DOI: 10.1109/CVPR.2016.91.

[RF17] Joseph Redmon and Ali Farhadi. “YOLO9000: Better, Faster, Stronger”. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. July 2017, pp. 6517–6525. DOI: 10.1109/CVPR.2017.690.

[RF18] Joseph Redmon and Ali Farhadi. *YOLOv3: An Incremental Improvement*. Apr. 2018. DOI: 10.48550/arXiv.1804.02767. arXiv: 1804.02767 [cs].

[Rhu+18] Minsoo Rhu et al. “Compressing DMA Engine: Leveraging Activation Sparsity for Training Deep Neural Networks”. In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2018, pp. 78–91. DOI: 10.1109/HPCA.2018.00017.

[RJL18] Pranav Rajpurkar et al. “Know What You Don’t Know: Unanswerable Questions for SQuAD”. In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Melbourne, Australia: Association for Computational Linguistics, July 2018, pp. 784–789. DOI: 10.18653/v1/P18-2124.

[Rom+22] Robin Rombach et al. “High-Resolution Image Synthesis with Latent Diffusion Models”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2022, pp. 10684–10695.

[Rot+19] Nadav Rotem et al. *Glow: Graph Lowering Compiler Techniques for Neural Networks*. Apr. 2019. DOI: 10.48550/arXiv.1805.00907. arXiv: 1805.00907 [cs].

[RPS22] Jaehun Ryu et al. “One-Shot Tuner for Deep Learning Compilers”. In: *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*. CC 2022. New York, NY, USA: Association for Computing Machinery, Mar. 2022, pp. 89–103. ISBN: 978-1-4503-9183-2. DOI: 10.1145/3497776.3517774.

[RZL17] Prajit Ramachandran et al. *Searching for Activation Functions*. Oct. 2017. DOI: 10.48550/arXiv.1710.05941. arXiv: 1710.05941 [cs].

[SA20] Sidak Pal Singh and Dan Alistarh. “WoodFisher: Efficient Second-order Approximation for Neural Network Compression”. In: *Proceedings of the 34th International Conference on Neural Information Processing Systems*. NIPS’20. Red Hook, NY, USA: Curran Associates Inc., Dec. 2020, pp. 18098–18109. ISBN: 978-1-71382-954-6.

[Sam+19] Ananda Samajdar et al. “SCALE-Sim: Systolic CNN Accelerator Simulator”. In: *arXiv:1811.02883 [cs]* (Feb. 2019). arXiv: 1811.02883 [cs]. URL: http://arxiv.org/abs/1811.02883.

[Sam+20] Ananda Samajdar et al. “A Systematic Methodology for Characterizing Scalability of DNN Accelerators Using SCALE-Sim”. In: *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Aug. 2020, pp. 58–68. DOI: 10.1109/ISPASS48437.2020.00016.

[San+18] M. Sandler et al. “MobileNetV2: Inverted Residuals and Linear Bottlenecks”. In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, June 2018, pp. 4510–4520. DOI: 10.1109/CVPR.2018.00474.

[SC10] A. Sanches and J. M. P. Cardoso. “On Identifying Patterns in Code Repositories to Assist the Generation of Hardware Templates”. In: *2010 International Conference on Field Programmable Logic and Applications*. Aug. 2010, pp. 267–270. DOI: 10.1109/FPL.2010.62.

[SFH17] Sara Sabour et al. “Dynamic Routing Between Capsules”. In: *Advances in Neural Information Processing Systems 30*. Curran Associates, Inc., 2017, pp. 3856–3866. DOI: 10.5555/3294996.3295142.

[SGC22] Axel Stjerngren et al. “Bifrost: End-to-End Evaluation and Optimization of Reconfigurable DNN Accelerators”. In: *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. May 2022, pp. 288–299. DOI: 10.1109/ISPASS55109.2022.00042.

[SGS10] John E. Stone et al. “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems”. In: *Computing in Science & Engineering* 12.3 (May 2010), pp. 66–73. ISSN: 1558-366X. DOI: 10.1109/MCSE.2010.69.

[Sha+16] Yakun Sophia Shao et al. “Co-Designing Accelerators and SoC Interfaces Using Gem5-Aladdin”. In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Oct. 2016, pp. 1–12. DOI: 10.1109/MICRO.2016.7783751.

[Sha+22] Junru Shao et al. “Tensor Program Optimization with Probabilistic Programs”. In: *Advances in Neural Information Processing Systems*. Oct. 2022. URL: https://openreview.net/forum?id=nyCr6-0hinG.

[She+18] Tao Sheng et al. “A Quantization-Friendly Separable Convolution for MobileNets”. In: *2018 1st Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*. IEEE Computer Society, Mar. 2018, pp. 14–18. ISBN: 978-1-5386-7367-6. DOI: 10.1109/EMC2.2018.00011.

[Sif14] Laurent Sifre. “Rigid-Motion Scattering for Image Classification”. PhD thesis. Ecole Polytechnique, CMAP, 2014. URL: http://www.cmapx.polytechnique.fr/~sifre/research/phd_sifre.pdf.

[Sil+16] David Silver et al. “Mastering the Game of Go with Deep Neural Networks and Tree Search”. In: *Nature* 529.7587 (Jan. 2016), pp. 484–489. ISSN: 1476-4687. DOI: 10.1038/nature16961.

[So+21] David So et al. “Searching for Efficient Transformers for Language Modeling”. In: *Advances in Neural Information Processing Systems*. Vol. 34. Curran Associates, Inc., 2021, pp. 6010–6022. URL: https://proceedings.neurips.cc/paper/2021/hash/2f3c6a4cd8af177f6456e7e51a916ff3-Abstract.html.

[Soh+15] Jascha Sohl-Dickstein et al. “Deep Unsupervised Learning Using Nonequilibrium Thermodynamics”. In: *Proceedings of the 32nd International Conference on Machine Learning*. PMLR, June 2015, pp. 2256–2265. DOI: 10.5555/3045118.3045358.

[SRD17] Michel Steuwer et al. “LIFT: A Functional Data-Parallel IR for High-Performance GPU Code Generation”. In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. Feb. 2017, pp. 74–85. DOI: 10.1109/CGO.2017.7863730.

[Sri+14] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. ISSN: 1533-7928. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[ST18] Leslie N. Smith and Nicholay Topin. *Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates*. May 2018. DOI: 10.48550/arXiv.1708.07120. arXiv: 1708.07120 [cs, stat].

[Str69] Volker Strassen. “Gaussian Elimination Is Not Optimal”. In: *Numerische Mathematik* 13.4 (Aug. 1969), pp. 354–356. ISSN: 0945-3245. DOI: 10.1007/BF02165411.

[Sun+19] Yifan Sun et al. “MGPUSim: Enabling Multi-GPU Performance Modeling and Optimization”. In: *Proceedings of the 46th International Symposium on Computer Architecture*. ISCA ’19. New York, NY, USA: Association for Computing Machinery, June 2019, pp. 197–209. ISBN: 978-1-4503-6669-4. DOI: 10.1145/3307650.3322230.

[Sun+20a] Pei Sun et al. “Scalability in Perception for Autonomous Driving: Waymo Open Dataset”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. IEEE Computer Society, 2020, pp. 2446–2454. DOI: 10.1109/CVPR42600.2020.00252.

[Sun+20b] Zhiqing Sun et al. “MobileBERT: A Compact Task-Agnostic BERT for Resource-Limited Devices”. In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Online: Association for Computational Linguistics, July 2020, pp. 2158–2170. DOI: 10.18653/v1/2020.acl-main.195.

[SV16] Shreyas Saxena and Jakob Verbeek. “Convolutional Neural Fabrics”. In: *Proceedings of the 30th International Conference on Neural Information Processing Systems*. NIPS’16. Red Hook, NY, USA: Curran Associates Inc., Dec. 2016, pp. 4060–4068. ISBN: 978-1-5108-3881-9.

[SZ14] Karen Simonyan and Andrew Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: *arXiv:1409.1556 [cs]* (Sept. 2014). arXiv: 1409.1556 [cs]. URL: http://arxiv.org/abs/1409.1556.

[Sze+15] Christian Szegedy et al. “Going Deeper with Convolutions”. In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2015, pp. 1–9. DOI: 10.1109/CVPR.2015.7298594.

[Sze+16] Christian Szegedy et al. “Rethinking the Inception Architecture for Computer Vision”. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016.

[Sze+20] Vivienne Sze et al. *Efficient Processing of Deep Neural Networks*. Synthesis Lectures on Computer Architecture. Cham: Springer International Publishing, 2020. DOI: 10.1007/978-3-031-01766-7.

[Tab+19] Domen Tabernik et al. “Segmentation-Based Deep-Learning Approach for Surface-Defect Detection”. In: *Journal of Intelligent Manufacturing* (May 2019). ISSN: 1572-8145. DOI: 10.1007/s10845-019-01476-x.

[Tan+13] Lingjia Tang et al. “Optimizing Google’s Warehouse Scale Computers: The NUMA Experience”. In: *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2013, pp. 188–197. DOI: 10.1109/HPCA.2013.6522318.

[Tan+19] Mingxing Tan et al. “MnasNet: Platform-Aware Neural Architecture Search for Mobile”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. IEEE Computer Society, 2019, pp. 2815–2823. DOI: 10.1109/CVPR.2019.00293.

[Tan+22] Muhammad Tanvir et al. “Towards Performance Portability of AI Models Using SYCL-DNN”. In: *International Workshop on OpenCL*. IWOCL’22. New York, NY, USA: Association for Computing Machinery, May 2022, pp. 1–3. ISBN: 978-1-4503-9658-5. DOI: 10.1145/3529538.3529999.

[Tav+21] Sanket Tavarageri et al. “PolyDL: Polyhedral Optimizations for Creation of High-performance DL Primitives”. In: *ACM Transactions on Architecture and Code Optimization* 18.1 (Jan. 2021), 11:1–11:27. ISSN: 1544-3566. DOI: 10.1145/3433103.

[TCO21] Jack Turner et al. “Neural Architecture Search as Program Transformation Exploration”. In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’21. New York, NY, USA: Association for Computing Machinery, Apr. 2021, pp. 915–927. ISBN: 978-1-4503-8317-2. DOI: 10.1145/3445814.3446753.

[Tea17] TensorFlow Development Team. *XLA - TensorFlow, Compiled*. Mar. 2017. URL: https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html.

[Ter+10] Dan Terpstra et al. "Collecting Performance Data with PAPI-C". In: *Tools for High Performance Computing 2009*. Berlin, Heidelberg: Springer, 2010, pp. 157–173. ISBN: 978-3-642-11261-4. DOI: 10.1007/978-3-642-11261-4_11.

[The19] The IREE Authors. *IREE*. Sept. 2019. URL: https://github.com/openxla/iree.

[TL19] Mingxing Tan and Quoc Le. "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks". In: *International Conference on Machine Learning*. PMLR, May 2019, pp. 6105–6114. URL: http://proceedings.mlr.press/v97/tan19a.html.

[Tol+23] Nicolas Tollenaere et al. "Autotuning Convolutions Is Easier Than You Think". In: *ACM Transactions on Architecture and Code Optimization* 20.2 (Mar. 2023), 20:1–20:24. ISSN: 1544-3566. DOI: 10.1145/3570641.

[Tur+18a] Jack Turner et al. "Characterising Across-Stack Optimisations for Deep Convolutional Neural Networks". In: *2018 IEEE International Symposium on Workload Characterization (IISWC)*. Sept. 2018, pp. 101–110. DOI: 10.1109/IISWC.2018.8573503.

[Tur+18b] Jack Turner et al. "Distilling with Performance Enhanced Students". In: *arXiv:1810.10460 [cs, stat]* (Oct. 2018). arXiv: 1810.10460 [cs, stat]. URL: http://arxiv.org/abs/1810.10460.

[Tur+20] Jack Turner et al. "BlockSwap: Fisher-guided Block Substitution for Network Compression on a Budget". In: *Proceedings to the International Conference on Learning Representations 2020*. Jan. 2020. URL: https://openreview.net/forum?id=SklkDkSFPB.

[Uhl+20] Stefan Uhlich et al. "Mixed Precision DNNs: All You Need Is a Good Parametrization". In: *Eighth International Conference on Learning Representations*. Apr. 2020. URL: https://iclr.cc/virtual_2020/poster_Hyx0slrFvH.html.

[Umu+17] Yaman Umuroglu et al. "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '17. New York, NY, USA: Association for Computing Machinery, Feb. 2017, pp. 65–74. ISBN: 978-1-4503-4354-1. DOI: 10.1145/3020078.3021744.

[VAG17] Aravind Vasudevan et al. “Parallel Multi Channel Convolution Using General Matrix Multiplication”. In: *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. July 2017, pp. 19–24. DOI: 10.1109/ASAP.2017.7995254.

[Var+22] Mihaly Varadi et al. “AlphaFold Protein Structure Database: Massively Expanding the Structural Coverage of Protein-Sequence Space with High-Accuracy Models”. In: *Nucleic Acids Research* 50.D1 (Jan. 2022), pp. D439–D444. ISSN: 0305-1048. DOI: 10.1093/nar/gkab1061.

[Vas+14] Nicolas Vasilache et al. “Fast Convolutional Nets With Fbfft: A GPU Performance Evaluation”. In: *arXiv:1412.7580 [cs]* (Dec. 2014). arXiv: 1412.7580 [cs]. URL: http://arxiv.org/abs/1412.7580.

[Vas+17] Ashish Vaswani et al. “Attention Is All You Need”. In: *Advances in Neural Information Processing Systems 30*. Curran Associates, Inc., 2017, pp. 5998–6008. DOI: 10.5555/3295222.3295349.

[Vas+18] Nicolas Vasilache et al. “Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions”. In: *arXiv:1802.04730 [cs]* (June 2018). arXiv: 1802.04730 [cs]. URL: http://arxiv.org/abs/1802.04730.

[VBC06] Nicolas Vasilache et al. “Polyhedral Code Generation in the Real World”. In: *Compiler Construction*. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 185–201. ISBN: 978-3-540-33051-6. DOI: 10.1007/11688839_16.

[Vee16] Fjodor Van Veen. *The Neural Network Zoo*. 2016. URL: https://www.asimovinstitute.org/neural-network-zoo/.

[Ven+19] Anand Venkat et al. “SWIRL: High-performance Many-core CPU Code Generation for Deep Neural Networks”. In: *The International Journal of High Performance Computing Applications* 33.6 (Nov. 2019), pp. 1275–1289. ISSN: 1094-3420. DOI: 10.1177/1094342019866247.

[VSM11] Vincent Vanhoucke et al. “Improving the Speed of Neural Networks on CPUs”. In: *Deep Learning and Unsupervised Feature Learning Workshop, NIPS*. 2011.

[Wan+17] Xiaosong Wang et al. “ChestX-ray8: Hospital-Scale Chest X-Ray Database and Benchmarks on Weakly-Supervised Classification and Localization of Common Thorax Diseases”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. IEEE Computer Society, 2017, pp. 2097–2106. DOI: 10.1109/CVPR.2017.369.

[Wan+19a] Alex Wang et al. “GLUE: A Multi-task Benchmark and Analysis Platform for Natural Language Understanding”. In: *International Conference on Learning Representations*. 2019. URL: https://openreview.net/forum?id=rJ4km2R5t7.

[Wan+19b] Kuan Wang et al. “HAQ: Hardware-Aware Automated Quantization With Mixed Precision”. In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2019, pp. 8604–8612. DOI: 10.1109/CVPR.2019.00881.

[Wan+20] Siqi Wang et al. “High-Throughput CNN Inference on Embedded ARM Big.LITTLE Multicore Processors”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.10 (Oct. 2020), pp. 2254–2267. ISSN: 1937-4151. DOI: 10.1109/TCAD.2019.2944584.

[War18] Pete Warden. *Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition*. Apr. 2018. DOI: 10.48550/arXiv.1804.03209. arXiv: 1804.03209 [cs].

[Wei+17] Xuechao Wei et al. “Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs”. In: *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. June 2017, pp. 1–6. DOI: 10.1145/3061639.3062207.

[Wen+19] Yuan Wen et al. “POSTER: Space and Time Optimal DNN Primitive Selection with Integer Linear Programming”. In: *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Sept. 2019, pp. 489–490. DOI: 10.1109/PACT.2019.00059.

[Wen+20] Yuan Wen et al. “TASO: Time and Space Optimization for Memory-Constrained DNN Inference”. In: *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. Sept. 2020, pp. 199–208. DOI: 10.1109/SBAC-PAD49847.2020.00036.

[WSS16] Daniel Weimer et al. “Design of Deep Convolutional Neural Network Architectures for Automated Feature Extraction in Industrial Inspection”. In: *CIRP Annals* 65.1 (Jan. 2016), pp. 417–420. ISSN: 0007-8506. DOI: 10.1016/j.cirp.2016.04.072.

[Wu+19a] Bichen Wu et al. “FBNet: Hardware-Aware Efficient ConvNet Design via Differentiable Neural Architecture Search”. In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society,

June 2019, pp. 10726–10734. ISBN: 978-1-72813-293-8. DOI: 10.1109/CVPR.2019.01099.

[Wu+19b] Carole-Jean Wu et al. "Machine Learning at Facebook: Understanding Inference at the Edge". In: *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Washington, DC, USA: IEEE, Feb. 2019, pp. 331–344. ISBN: 978-1-72811-444-6. DOI: 10.1109/HPCA.2019.00048.

[Xi+20] Sam (Likun) Xi et al. "SMAUG: End-to-End Full-Stack Simulation Infrastructure for Deep Learning Workloads". In: *ACM Transactions on Architecture and Code Optimization* 17.4 (Nov. 2020), 39:1–39:26. ISSN: 1544-3566. DOI: 10.1145/3424669.

[Xia+18] Gui-Song Xia et al. "DOTA: A Large-Scale Dataset for Object Detection in Aerial Images". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. IEEE Computer Society, 2018, pp. 3974–3983. DOI: 10.1109/CVPR.2018.00418.

[XQY12] Zhang Xianyi et al. "Model-Driven Level 3 BLAS Performance Optimization on Loongson 3A Processor". In: *2012 IEEE 18th International Conference on Parallel and Distributed Systems*. Dec. 2012, pp. 684–691. DOI: 10.1109/ICPADS.2012.97.

[XR19] Yiting Xie and David Richmond. "Pre-Training on Grayscale ImageNet Improves Medical Image Classification". In: *Computer Vision – ECCV 2018 Workshops*. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 476–484. ISBN: 978-3-030-11024-6. DOI: 10.1007/978-3-030-11024-6_37.

[XRV17] Han Xiao et al. *Fashion-MNIST: A Novel Image Dataset for Benchmarking Machine Learning Algorithms*. Sept. 2017. DOI: 10.48550/arXiv.1708.07747. arXiv: 1708.07747 [cs, stat].

[Xu+20] Sheng Xu et al. "Convolutional Neural Network Pruning: A Survey". In: *2020 39th Chinese Control Conference (CCC)*. July 2020, pp. 7458–7463. DOI: 10.23919/CCC50068.2020.9189610.

[Yan+22] Xiangli Yang et al. "A Survey on Deep Semi-Supervised Learning". In: *IEEE Transactions on Knowledge and Data Engineering* (2022), pp. 1–20. ISSN: 1558-2191. DOI: 10.1109/TKDE.2022.3220219.

[Yao+22] Hongyi Yao et al. “RAPQ: Rescuing Accuracy for Power-of-Two Low-bit Post-training Quantization”. In: *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*. ijcai.org, 2022, pp. 1573–1579. DOI: 10.24963/ijcai.2022/219.

[Ye+23] Zihao Ye et al. “SparseTIR: Composable Abstractions for Sparse Compilation in Deep Learning”. In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, Mar. 2023, pp. 660–678. ISBN: 978-1-4503-9918-0. DOI: 10.1145/3582016.3582047.

[ZB19] Tim Zerrell and Jeremy Bruestle. “Stripe: Tensor Compilation via the Nested Polyhedral Model”. In: *arXiv:1903.06498 [cs]* (Mar. 2019). arXiv: 1903.06498 [cs]. URL: http://arxiv.org/abs/1903.06498.

[ZC18] Lanmin Zheng and Tianqi Chen. “Optimizing Deep Learning Workloads on ARM GPU with TVM”. In: *Proceedings of the 1st on Reproducible Quality-Efficient Systems Tournament on Co-designing Pareto-efficient Deep Learning*. ReQuEST ’18. New York, NY, USA: ACM, 2018. ISBN: 978-1-4503-5923-8. DOI: 10.1145/3229762.3229764.

[ZFL18] Jiyuan Zhang et al. “High Performance Zero-Memory Overhead Direct Convolutions”. In: *International Conference on Machine Learning*. July 2018. Chap. Machine Learning, pp. 5776–5785. URL: http://proceedings.mlr.press/v80/zhang18d.html.

[ZG17] Michael Zhu and Suyog Gupta. *To Prune, or Not to Prune: Exploring the Efficacy of Pruning for Model Compression*. Nov. 2017. DOI: 10.48550/arXiv.1710.01878. arXiv: 1710.01878 [cs, stat].

[Zha+16] Shijin Zhang et al. “Cambricon-X: An Accelerator for Sparse Neural Networks”. In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Oct. 2016, pp. 1–12. DOI: 10.1109/MICRO.2016.7783723.

[Zha+18] Xiaofan Zhang et al. “DNNBuilder: An Automated Tool for Building High-Performance DNN Hardware Accelerators for FPGAs”. In: *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. Nov. 2018, pp. 1–8. DOI: 10.1145/3240765.3240801.

[Zha+20] Ruizhe Zhao et al. “On the Challenges in Programming Mixed-Precision Deep Neural Networks”. In: *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. MAPL 2020. New York, NY, USA: Association for Computing Machinery, June 2020, pp. 20–28. ISBN: 978-1-4503-7996-0. DOI: 10.1145/3394450.3397468.

[Zhe+20a] 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)*. 2020, pp. 863–879. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/zheng.

[Zhe+20b] Size Zheng et al. “FlexTensor: An Automatic Schedule Exploration and Optimization Framework for Tensor Computation on Heterogeneous System”. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’20. Lausanne, Switzerland: Association for Computing Machinery, Mar. 2020, pp. 859–873. ISBN: 978-1-4503-7102-5. DOI: 10.1145/3373376.3378508.

[Zhe+22] Bojian Zheng et al. “DietCode: Automatic Optimization for Dynamic Tensor Programs”. In: *Proceedings of Machine Learning and Systems* 4 (Apr. 2022), pp. 848–863. URL: https://proceedings.mlsys.org/paper/2022/hash/fa7cdfad1a5aaf8370ebeda47a1ff1c3-Abstract.html.

[Zhu+15] Yukun Zhu et al. “Aligning Books and Movies: Towards Story-Like Visual Explanations by Watching Movies and Reading Books”. In: *2015 IEEE International Conference on Computer Vision (ICCV)*. IEEE Computer Society, Dec. 2015, pp. 19–27. ISBN: 978-1-4673-8391-2. DOI: 10.1109/ICCV.2015.11.

[ZK16] Sergey Zagoruyko and Nikos Komodakis. “Wide Residual Networks”. In: *British Machine Vision Conference (BMVC)*. 2016.

[ZK17] Sergey Zagoruyko and Nikos Komodakis. “Paying More Attention to Attention: Improving the Performance of Convolutional Neural Networks via Attention Transfer”. In: *International Conference on Learning Representations (ICLR)*. 2017. URL: https://openreview.net/forum?id=Sks9_ajex.

[ZL17] Barret Zoph and Quoc Le. “Neural Architecture Search with Reinforcement Learning”. In: *International Conference on Learning Representations (ICLR)*. 2017. URL: https://openreview.net/forum?id=r1Ue8Hcxg.

[Zop+18] Barret Zoph et al. “Learning Transferable Architectures for Scalable Image Recognition”. In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, June 2018, pp. 8697–8710. ISBN: 978-1-5386-6420-9. DOI: 10.1109/CVPR.2018.00907.