Analysing and Reducing Costs of Deep Learning Compiler Auto-tuning

Damian Borowiec, BSc (Hons)

School of Computing and Communications

Lancaster University

This thesis is submitted for the degree of

Doctor of Philosophy

March, 2023

This page is left intentionally blank

I dedicate this thesis to my mother Celina.

This page is left intentionally blank

Analysing and Reducing Costs of Deep Learning Compiler Auto-tuning

School of Computing and Communications, Lancaster University

This thesis is submitted for the degree of *Doctor of Philosophy*. March, 2023

Abstract

Deep Learning (DL) is significantly impacting many industries, including automotive, retail and medicine, enabling autonomous driving, recommender systems and genomics modelling, amongst other applications. At the same time, demand for complex and fast DL models is continually growing. The most capable models tend to exhibit highest operational costs, primarily due to their large computational resource footprint and inefficient utilisation of computational resources employed by DL systems. In an attempt to tackle these problems, DL compilers and auto-tuners emerged, automating the traditionally manual task of DL model performance optimisation. While auto-tuning improves model inference speed, it is a costly process, which limits its wider adoption within DL deployment pipelines.

The high operational costs associated with DL auto-tuning have multiple causes. During operation, DL auto-tuners explore large search spaces consisting of billions of tensor programs, to propose potential candidates that improve DL model inference latency. Subsequently, DL auto-tuners measure candidate performance in isolation on the target-device, which constitutes the majority of auto-tuning compute-time. Sub-optimal candidate proposals, combined with their serial measurement in an isolated target-device lead to prolonged optimisation time and reduced resource availability, ultimately reducing cost-efficiency of the process.

In this thesis, we investigate the reasons behind prolonged DL auto-tuning and quantify their impact on the optimisation costs, revealing directions for improved DL

auto-tuner design. Based on these insights, we propose two complementary systems: Trimmer and DOPpler. Trimmer improves tensor program search efficacy by filtering out poorly performing candidates, and controls end-to-end auto-tuning using cost objectives, monitoring optimisation cost. Simultaneously, DOPpler breaks long-held assumptions about the serial candidate measurements by successfully parallelising them intra-device, with minimal penalty to optimisation quality. Through extensive experimental evaluation of both systems, we demonstrate that they significantly improve cost-efficiency of auto-tuning (up to 50.5%) across a plethora of tensor operators, DL models, auto-tuners and target-devices.

Acknowledgements

I would like to thank all those who supported me during my PhD journey, both academically and otherwise. I direct my first and foremost gratitude to my academic supervisors, Dr. Peter Garraghan and Professor Richard R. Harper for their endless help, academic and personal support as well as numerous opportunities they have offered me over the course of the past four years.

I would like to thank Peter for his continual reassurance in times where I doubted myself, and for his dedication and effort to help me understand that there are always options out there and I just need to step back and think. I look back fondly at the numerous late night calls, publication writing sessions and countless feedback on the drafts of this research work. Beyond research, I am grateful to Peter for his continual push towards improvements in my presentation, public speaking, teaching and leadership skills, which all have greatly flourished thanks to his guidance.

I am thankful to Richard for his guidance and supervision, on many occasions when I needed it the most. I would like to thank him for the opportunity of being part of the Material Social Futures Centre for Doctoral Training, which had an inestimable impact on my personal growth as a researcher, as well as many career, networking and academic opportunities he has offered me throughout my PhD journey. If I was to pick a single thing I learned from Richard, it would be to remain specific yet keep my mind open to the broader impact of my work.

For the mutual exchange of ideas and comradery I would like to thank all current and former members of the Experimental Distributed Systems Lab. I send my special thanks to Ging Fung Yeung for his ongoing guidance, close collaboration on various projects and publications as well as moral support. Thank you to Dominic Lindsay, William Hackett, Lewis Birch, Stefan Trawicki, James Bulman and Matthew Hodkin for their

efforts in the ongoing battle with the compute cluster downstairs and lab equipment as well as many fruitful conversations. I would like to thank Petter Terenius for our many collaborative and casual conversations throughout the years, and I truly believe your work is going to revolutionise the way we think about energy in datacentres. Thank you to Marta Adamska, Daria Smirnova, Sukhpal Singh Gill, Weija Li and John Hutchinson for making the PhD experience a more enriching one.

I would like to thank all current and former members of the Material Social Futures Centre for Doctoral Training for in-depth discussions on academic topics seemingly far outside my realm of research, yet in the end extremely eye-opening and useful - especially Christina Bremer, Ben Wood, Ariel Bernier and Sophie Au-yong. Thank you to all supervisors, speakers and the managerial board for sharing their multi-disciplinary knowledge in such an approachable way. Also, I would like to thank the MSF Centre for Doctoral Training and the Leverhulme Trust for providing me with the PhD studentship.

Thank you to Dr. Andrew Scott for an excellent supervision of my BSc FYP which has ultimately led to my interest in systems research and this thesis. I am grateful to my colleagues for their support and collaboration during my internships at Microsoft Research Cambridge and Huawei Research and Development Edinburgh. Both experiences have provided me with necessary industry and academic knowledge and confidence to complete this work, growing as a researcher and systems engineer.

Thank you to my friends - especially Eleanor Davies for keeping me sane. I want to thank my close family - brother Krzysztof, niece and nephews for their love and belief in my ability to complete my PhD, and especially my mother Celina - without whom I would not have been able to achieve this, and who has been there whenever I needed her. *Dziękuję za wszystko Mamo.* Finally, I would like to thank my partner, Niah who has accompanied me in the many ups and downs of my work and life and continues to help me see the bright side of things - with loving support.

Declaration

I declare that the work presented in this thesis is, to the best of my knowledge and belief, original and my own work. The material has not been submitted, either in whole or in part, for a degree at this, or any other university. This thesis does not exceed the maximum permitted word length of 80,000 words, including footnotes, but excluding the bibliography, appendices and the front matter. A rough estimate of the word count is: 53029

Damian Borowiec

March, 2023

This page is left intentionally blank

List of Publications

- **Borowiec, D.**, Yeung, G., Friday, A., Harper, R., Garraghan, P., 2023. DOPpler: Parallel Tuning for Deep Learning Tensor Programs. In: IEEE Transactions on Parallel and Distributed Systems (TPDS), Vol 34, No. 7, pp. 2208-2220.

  I designed, implemented and evaluated DOPpler. Ging Fung Yeung contributed towards the design of the alternative DOPpler policy (RLEARN), and provided guidance during design of the Calibrator module. Peter Garraghan contributed towards system design and paper writing. Adrian Friday and Richard Harper provided feedback, helped with revisions and improved the paper.

  The analysis of naïve parallel measurements during DL auto-tuning form the basis of Chapter 5 whilst Chapter 6 presents the design and evaluation of DOPpler.

- **Borowiec, D.**, Yeung, G., Friday, A., Harper, R., Garraghan, P., 2022. Trimmer: Cost-Efficient Deep Learning Auto-tuning for Cloud Datacenters. In: IEEE International Conference on Cloud Computing (CLOUD 22), pp. 374-384.

  I designed, implemented and evaluated Trimmer. The cold-candidate filtering model is a joint contribution with Ging Fung Yeung. Peter Garraghan contributed towards system design and paper writing. The paper was reviewed and improved by Adrian Friday and Richard Harper.

  The analysis of DL optimisation method costs presented within the paper, forms the basis of Chapter 3 while the design and experimental evaluation of Trimmer form the basis of Chapter 4.

- **Borowiec, D.**, Harper, R., Garraghan, P., 2022. Environmental Consequence of Deep Learning. In: ITNOW, Vol. 63, No. 4, pp. 10-11.

  I wrote the paper. Richard Harper and Peter Garraghan contributed towards initial idea discussions, provided feedback and helped revise the paper.

  This paper contributes to broader DL costs discussions in Chapters 1 and 7.

- Yeung, G. **Borowiec, D.**, Yang, R., Friday, A., Harper, R. and Garraghan, P., 2022. Horus: Interference-Aware and Prediction-Based Scheduling in Deep Learning Systems. In: IEEE Transactions on Parallel and Distributed Systems (TPDS), Vol. 33, No. 1, pp. 88-100.

  Ging Fung Yeung designed Horus and its evaluation methodology. I contributed towards the implementation of the profiling engine and evaluation scripts, implemented the Open Neural Network Exchange (ONNX) analyser, performed model co-location experiments and collected the associated data as well as helped with paper writing. Renyu Yang, Adrian Friday, Richard Harper and Peter Garraghan helped with paper writing, design discussions and revisions of the work.

  The ONNX analyser was partially re-used to characterise models used across Chapters 3, 4, 5 and 6, while the profiling engine was further extended to monitor compute platforms during evaluation experiments (Trimmer and DOPpler). Moreover, this paper explores the problem of interference when co-locating DL workloads, which feeds into the investigation performed as part of Chapter 5.

Contents

List of Figures

This page is left intentionally blank

List of Tables

List of Listings

List of Abbreviations

AaaS Auto-tuning as a Service

ALU Arithmetic Logic Unit

AMX Advanced Matrix Extensions

ANN Artificial Neural Network

API Application Programming Interface

ASIC Application Specific Integrated Circuit

AT AutoTVM

AVX Advanced Vector Extensions

BIC Binary Increase Congestion control

BN Batch Normalisation

CHP Combined Heat and Power

CI Computational Intensity

CLB Configurable Logic Block

CNN Convolution Neural Network

CPU Central Processing Unit

CRAC Computer Room Air Conditioning

CUDA Compute Unified Device Architecture

CUPTI CUDA Profiling Tools Interface

CV Computer Vision

DAG Directed Acyclic Graph

DDR Double Data Rate

DL Deep Learning

DNN Deep Neural Network

DRAM Dynamic Random Access Memory

DSL Domain Specific Language

DSP Digital Signal Processor

EDP Expand-Depth-wise-Project

EDPA Expand-Depth-wise-Project-Add

FC Fully-connected

FFNN Feed-forward Neural Network

FFT Fast Fourier Transform

FLOP Floating-point Operation

FMA Fused Multiply-Add

FMAC Fused Multiply-Accumulate

FP Floating-point

FPGA Field-Programmable Gate Array

FPU Floating-Point Unit

GA Genetic Algorithm

GAN Generative Adversarial Networks

GBT Gradient Boosted Trees

GDDR Graphics Double Data Rate

GEMM General Matrix Multiply

GFLOPs Giga Floating-point Operations

GHG Greenhouse Gas

GOPS Giga Operations

GPU Graphics Processing Unit

GR Grid Index

HBM High Bandwidth Memory

HDO Hardware-Dependent Optimisations

HLIR High-Level Intermediate Representation

HWCN (H)eight - (W)idth - (C)hannels - Batch-size(N)

IC Image Classification

IE Inference Engine

INT Integer

IPC Instructions per Cycle

IPU Image/Intelligence Processing Unit

IR Intermediate Representation

ISA Instruction Set Architecture

LLIR Low-Level Intermediate Representation

LLVM Low-Level Virtual Machine

LSTM Long-short Term Memory

LUT Lookup Table

MAC Multiply Accumulate

MAD Multiply Add

MIG Multi Instance GPU

ML Machine Learning

MLaaS Machine Learning as a Service

MPS Multi-process Service

MSR Model-Specific Register

NAS Neural Architecture Search

NCHW Batch-size(N) - (C)hannels - (H)eight - (W)idth

NHWC Batch-size(N) - (H)eight - (W)idth - (C)hannels

NLP Natural Language Processing

NNP Neural Network Processor

NPM Naïve Parallel Measurement

NPU Neural Processing Unit

NVML Nvidia Management Library

ONNX Open Neural Network Exchange

OOM Out-of-memory

OPS Operations

PE Processing Element

PPM Precise Parallel Measurer

PPO Proximal Policy Optimisation

PSU Power Supply Unit

RAPL Running Average Power Limit

RD Random Index

ReLU Rectified Linear Unit

RF Random Forest

RL Reinforcement Learning

RNN Recurrent Neural Network

RPC Remote Procedure Call

SA Simulated Annealing

SDP Spatial (Dataflow) Processor

SIMD Single-Instruction Multiple-Data

SIMT Single-Instruction Multiple-Threads

SM Simultaneous Multiprocessor

SMT Simultaneous Multi-threading

SOTA state-of-the-art

TC TensorComprehensions

TCP Transmission Control Protocol

TDP Thermal Design Power

TE Tensor Expression

TFLOPs Tera Floating-point Operations

TIP Temporal (Instruction) Processor

TOPI Tensor Operator Inventory

TOPS Tera Operations

TPU Tensor Processing Unit

TVM Tensor Virtual Machine

VPU Vision Processing Unit

Chapter 1

Introduction

1.1 Motivation

In the recent years, Deep Learning (DL) - an area of computer science, focused on solving pattern and feature extraction tasks such as image recognition, text generation or game-playing via the use of Deep Neural Networks (DNNs), has significantly impacted many industries such as autonomous transport, social media, finance or scientific applications, achieving impressive predictive performance [129, 205, 380, 95, 169]. With the proliferation of DL models, there is an ever-increasing demand for them to be accurate, generic and fast at performing inference, reflected by the growing access to on-demand compute infrastructures supporting DL workloads [131, 295, 55, 99]. Such success is in part enabled by developments in DL systems - ecosystems of software and hardware that support high-level DL model prototyping, training and deployment for inference. DL systems utilise computational capabilities of high-performance Central Processing Units (CPUs) and compute accelerators such as Graphics Processing Units (GPUs) to execute model training and inference [275].

Developing and operating high performance DL models and systems that support them, requires significant engineering effort, computational power and financial

investment [33, 303, 258, 204, 308]. Furthermore, the most generally applicable[1] and accurate models also tend to exhibit the highest operational costs due to their algorithmic complexity, need for prolonged training and large computational footprint during inference [332, 152]. However, operational costs are commonly a second-order concern in modern DL systems research and industrial developments, focusing foremost on improving model designs for increased accuracy and generality, whilst designing novel DL accelerator architectures such as GPUs, Tensor Processing Units (TPUs), Neural Processing Units (NPUs), and DL-specific Application Specific Integrated Circuits (ASICs) to perform more computation more quickly [67, 275, 324].

1.1.1 High Operational Costs in Deep Learning

While the algorithmic complexity of DL models is the primary cause for majority of the operational costs in DL deployments [65], inefficient utilisation of the compute resources (for example, GPUs) that underpin DL systems can vastly inflate these costs [368, 144, 123, 146]. DL engineers commonly design and train DL models using high-level DL frameworks such as Pytorch [1] or TensorFlow [2], whilst their deployment on high-performance hardware is facilitated using DL inference engines [327, 85, 240], internally supported by acceleration libraries [51, 232, 141]. At each of these levels of the software stack, great care is taken to provide efficient implementations for common DL model tensor operators[2], however, these efforts often fail to keep up with the development of novel DL model architectures (and their unique tensor operators), and continual developments in the area of high-performance, massively-parallel processors that underpin DL computation [39, 105]. This forces DL frameworks and inference engines to use sub-optimal tensor operator implementations for compatibility reasons, which leads to resource under-utilisation and inflated operational costs [39, 176, 173].

[1] Capable of performing multiple learning tasks such as image classification with object detection.

[2] Tensor operators are primary components within DL models that specify the mathematical operations to be performed as part of training and inference.

1.1.2 Optimising Model Inference Performance

A traditional approach to improve execution latency, compute resource utilisation and thus cost-efficiency of performing DL computation on high-performance hardware, is for the DL engineer to manually develop optimal *tensor programs* that implement individual tensor operators within a given DL model. With the current diversity of quickly changing operational DL environments, manually ensuring that each new DL model tensor operator is implemented optimally towards each of the progressively more powerful *target-devices* is infeasible, given the complexity of DL system deployments. Such manual optimisation of DL tensor operators must take into account a range of unique DL model architectures, multiple DL frameworks that can facilitate DL model development, tens to hundreds of unique tensor operators per DL model, a range of unique target-devices that can be leveraged to perform tensor computations of the DL models, and potentially billions of unique implementations (tensor program candidates) of a single tensor operator towards a single target-device.

1.1.3 Deep Learning Compilers

As a result, various DL compilers [39, 262, 18, 173, 307, 382] have gained prominence by enabling automated transformations of high-level DL model computation specifications (originating from DL frameworks), into sets of executable tensor programs (implementations of tensor operators) that execute on a variety of target-devices, including high-performance GPU accelerators. Effectively, DL compilers enable automated decoupling of the computation specification (for example, high-level mathematical descriptions) from the way in which such computations should be performed on target-devices (for example, loop nest structures, memory data layouts, thread parallelism). Furthermore, DL compilers facilitate DL model graph-level transformations that optimise the overall model architecture towards faster inference.

DL compilers relieve some of the engineering burden associated with developing high-performance DL model implementations, however, they require DL engineers to provide *schedules* - sets of program transformations that describe how a given DL tensor operator should leverage computational resources of a high-performance target-device. For example, within the TVM [39] DL compiler, such schedules take form of a configurable template of a tensor program, which accepts parameters that determine how the high-level tensor operator expression will map to low-level tensor program operations during compilation, including thread-level parallelism, data access, operation synchronisation and loop nest transformations. As such, while some of the complexity of implementing DL models is relieved by DL compilers, the process of determining schedule parameters that configure tensor programs remains manual[3] during the use of standalone DL compilers. As previously outlined, there can be in the order of billions of unique *candidate* schedules for any combination of a tensor operator and a target-device, that must be evaluated on the target-device to determine their performance.

1.1.4 Deep Learning Compiler Auto-tuning

In an attempt to further alleviate the engineering burden of discovering high-performance tensor program schedules, the concept of DL *auto-tuning* has been proposed [40]. Facilitated by DL compilers, DL auto-tuning automates discovery of optimised schedules via the use of cost models and search algorithms that help to traverse the enormous *schedule space*, followed by on target-device execution latency measurements of promising tensor program candidates, ultimately resulting in a schedule configuration proposal for a given tensor operator and target-device combination that results in reduced inference latency of the end-to-end DL model. Many DL auto-tuners have recently been proposed [40, 8, 385, 183, 377, 386, 105, 373], each utilising a different set of component variants

[3]The DL engineers must test different schedule parameters for combinations of tensor operator classes and target-devices

(for example, cost models, search strategies or schedule spaces), improving efficacy of DL optimisation by targeting increasingly more diverse ranges of tensor operator classes and target-devices, resulting in substantial DL model inference speedup [40, 385]. One component that nearly all state-of-the-art (SOTA) DL auto-tuners share in common is the measurement infrastructure, which manages candidate tensor program execution latency measurements. It is important to note that all such measurements are performed in isolation - that is one at a time, ensuring accuracy of measurement and reliability of the infrastructure, which when coupled with extensive schedule space exploration leads to prolonged auto-tuning time [8, 183, 385].

1.1.5 Costs of Optimising Deep Learning Model Inference

Through the use of high-level DL model graph optimisations and DL auto-tuners, both facilitated by DL compilers, DL engineers can substantially reduce end-to-end DL model inference latency when executing DL models on a given target-device [39, 18, 345, 388, 8, 385, 183]. However, the specific relationships between optimisation quality and time and energy costs of performing them, have not been previously explored nor quantified.

This is especially the case for DL auto-tuning. The process of optimisation via DL auto-tuning can be observably time and energy-expensive (taking several days in the case of large models [184]), yet the precise reasons why that is have not been studied. While auto-tuning times in the range of tens of hours per end-to-end DL model may be acceptable in sporadic, one-off usage patterns, prolonged auto-tuning of individual models at a Cloud cluster level, or when models have to be re-tuned towards different execution scenarios, begins to become a challenging problem. This is further amplified by the isolation of the target-device during DL auto-tuning, reducing device availability and increasing operational costs. As such, to start introducing any modifications to the existing DL auto-tuning infrastructures, it is necessary to first analyse and quantify the phenomena responsible for cost-inefficient DL auto-tuning.

1.2 Recent Challenges within DL Auto-tuning

DL auto-tuners exhibit several outstanding research challenges towards achieving cost-efficient DL inference performance optimisation.

1.2.1 Serial Candidate Latency Measurements

All SOTA DL auto-tuners utilise a similar approach to performing on-target-device latency measurements of candidate tensor programs [40, 385, 8]. This approach involves isolating the target-device and performing candidate tensor program latency measurements serially, in sequence, ensuring only a single kernel executes at the target-device at a time. This stems from a long-standing, held and practised assumption within the DL auto-tuning community that introducing any degree of parallelism will inevitably result in interference due to resource contention and unpredictability of GPU kernel scheduling. While unequivocally ensuring measurement reliability, such an approach significantly under-utilises the available compute resources, whilst reserving them for prolonged periods of time. Addressing this bottleneck could improve both the overall wall-clock auto-tuning time and the candidate measurement throughput, resulting in improved cost-efficiency of performing DL optimisations.

1.2.2 Sequential End-to-end Auto-tuning

SOTA DL auto-tuners that claim to support end-to-end DL model auto-tuning [40, 39, 8, 183], rely upon sequential optimisation of tensor operators. More specifically, given an end-to-end DL model architecture definition, DL auto-tuners decompose the architecture graph into individual tensor operators and optimise each of them from start to finish, subsequently moving onto the next queued tensor operator to be optimised.

Given the current design of DL auto-tuners, achieved (optimised) end-to-end DL model inference latency can only be measured once all of its individual tensor operators have been optimised. This prevents the model-level optimisation strategy from leveraging information about end-to-end inference latency of the model when performing optimisation of its individual tensor operators, for example, to avoid unnecessary auto-tuning whenever this would be appropriate and cost-efficient. There is a large potential in leveraging such intermediary information, however, existing DL auto-tuners do not provide these capabilities to their users.

1.2.3 Unfiltered Candidate Measurements

During DL auto-tuning, cost models and search algorithms traverse the schedule space to discover configurations exhibiting low-latency for each DL model tensor operator. Subsequently, batches of such candidate schedules are proposed for on-target-device measurements in isolation to ascertain their quality and guide the schedule search. Auto-tuners combine such online measurements with optimisation strategies such as Simulated Annealing [342] that provide eventual guarantees of optimal schedule discovery after a large number of evaluations, which can be time consuming.

These design assumptions result in prolonged auto-tuning that may eventually discover a high-quality schedules. Ahn et al. [8] find that candidate schedules can form clusters within the schedule space that exhibit similar execution footprints, and exploit this using statistical machine learning methods to propose higher quality candidates more often during auto-tuning. As such, there exists an opportunity to combine existing strategies with novel probabilistic methods to exploit candidate similarities further. For example, candidates with poor performance potential could be filtered out and their expensive on-device evaluations avoided, enabling the auto-tuner to dedicate time to potentially more favourable candidates.

1.3 Objective, Hypothesis and Research Questions

The core objective of this thesis:

Given the aforementioned limitations and prolonged duration of DL auto-tuning, the core objective of this work is to measurably improve cost-efficiency of optimising DL model inference via auto-tuning; by reducing the overall auto-tuning time, reducing incurred energy costs as a result of less complex computational footprint, or increasing the quality of optimisation given the same time or energy budgets. More specifically, this objective can be achieved by either delivering better auto-tuning optimisation for the same operational cost or equivalent optimisation at a reduced cost. To achieve this research objective, two new standalone yet complementary systems are proposed: Trimmer and DOPpler, both targeting different parts of the DL auto-tuning infrastructures. Trimmer increases cost-efficiency of DL auto-tuning by filtering out undesirable candidate schedules and proposes a novel approach to end-to-end DL auto-tuning termed *Survey tuning*. DOPpler complements these improvements by replacing the widely adopted, serial candidate measurement infrastructure, with novel, intra-device parallel approach to measuring candidate tensor programs reliably at a much higher throughput.

Within this thesis the following hypothesis is posited:

The design of SOTA DL auto-tuners is plagued by several performance bottlenecks that prevent them from achieving both high-quality and cost-efficient DL model inference optimisation. These bottlenecks result in prolonged auto-tuning time, increased energy costs and under-utilisation of available compute resources, which in turn raises the barrier to entry and limits their wider adoption in DL inference deployment pipelines. By utilising targeted DL-based, heuristic and reactive methods, it is possible to address these bottlenecks at different levels of the deep learning auto-tuner design architecture, resulting in significant improvements in overall auto-tuning time, optimisation energy costs, and

cost-efficiency without negatively impacting optimisation quality (i.e., achieving better optimisation for the same operational cost or equivalent optimisation whilst incurring smaller operational costs).

To validate this hypothesis, it is re-framed as specific research questions.

[RQ1] *How significant are the time and energy costs of applying high and low-level DL inference performance optimisations and where do they originate from, when analysed across individual tensor operators, end-to-end models and DL compiler / auto-tuner levels? Could such cost analysis identify key architectural reasons for prolonged and cost-inefficient DL auto-tuning, considering different optimisation scenarios?*

Answering this question would identify specific inefficiencies and bottlenecks within SOTA DL auto-tuners and quantify their operational cost impact, providing valuable design directions for targeted improvements to specific auto-tuner components and their operation, when attempting to improve their cost-efficiency.

[RQ2] *Is it possible to train a DL model to identify low-quality (high latency) tensor program candidates ahead of their latency measurements during DL auto-tuning? If so, could such a model be leveraged to reduce the negative impact of poor candidates on auto-tuning operational costs?*

To answer this question a new DL auto-tuner component will have to be designed, implemented and evaluated, comparing the modified infrastructure to existing SOTA DL auto-tuners in terms of optimisation quality and operational costs.

[RQ3] *Could the end-to-end DL model inference latency be measured and subsequently leveraged to control the auto-tuning process for cost-effectiveness trade-offs, whilst auto-tuning of the model operators is underway?*

Answering this question requires designing a new end-to-end DL model auto-tuning strategy that could facilitate partial operator optimisation and model compilation with these partially optimised tensor operators, whilst also leveraging end-to-end latency information to control cost-budget and/or objective-based auto-tuning.

[RQ4] *Could the long-standing requirement for serial, isolated candidate measurements during DL auto-tuning be avoided, instead enabling parallel candidate tensor program execution intra-target-device reliably? If so, what would be the optimisation quality and operational cost implications of such an approach, when applied across different classes of tensor operators, DL models, auto-tuners and target-devices?*

Answering this question could be achieved iteratively. Initially, a naïvely-parallel solution could be evaluated to ascertain and analyse potential drawbacks (for example, interference) that justify the widely-accepted requirement for measurement isolation intra-device. Informed by such an analysis, a comprehensive and reliable solution for performing intra-device parallel candidate measurements will need to be developed and evaluated in a variety of DL auto-tuning scenarios.

1.4 Broader Research Context

DL is increasingly becoming a significant part of our societies, industries and day-to-day lives. Until recently the energy and environmental costs associated with DL computation have been ignored when developing increasingly more complex models, even though the costs of manufacturing and operating high-performance compute infrastructures underpinning DL are enormous [303, 204, 308]. The increasingly energy-intensive DL computation begs questions whether these costs are proportional to the insights provided by models and their applications. As such, time, monetary, energy and environmental costs associated with DL computation should be judged alongside the value of their predictions and generative outputs.

However, defining and quantifying a generic measure of value provided by DL systems is challenging outside of the monetary realm, while quantifying precise material, environmental and monetary costs of manufacturing and operating DL systems is both difficult and often overlooked, including in research projects [291, 115]. From a broader research perspective, this thesis considers a question of how one might analyse and factor-in, operational costs (including time and energy), into the design and operation of DL systems, with a case study of DL inference performance optimisation via DL compilers and auto-tuners.

1.5 Core Research Contributions

1. An experimental analysis of time and energy costs associated with DL compiler optimisations, including a comprehensive analysis of DL auto-tuners, their operational cost and discovery of the major sources of inefficiencies within their design and procedural assumptions.

2. An auto-tuning system termed Trimmer, which includes a neural-network based candidate filter that reduces cold candidate occurrence and enables cost-efficient DL auto tuning at a DL model tensor operator level. Trimmer also introduces a single-model and multi-model level meta-tuning approach that improves cost-efficiency of end-to-end DL auto-tuning.

3. An empirical analysis of the source of inefficiencies within the conventional candidate measurement infrastructures of SOTA DL auto-tuners and autoschedulers - the serial, intra-device isolation of candidate executions during their latency measurement. Stemming from the analysis, a naïvely-parallel approach is proposed, to tackle the identified issues, and empirically guide further developments of a more reliable and comprehensive solution.

4. A modular and extensible intra and inter-device parallel candidate measurement infrastructure termed DOPpler, compatible with SOTA DL auto-tuners and autoschedulers that provides a sizeable reduction in auto-tuning costs, whilst enabling existing auto-tuners to maintain their optimisation quality.

5. A report and an analysis of the overall energy, environmental and financial costs incurred during the experimental evaluations performed as part of this thesis, discussing the importance of cost reporting within DL research.

The investigations, design and evaluations of Trimmer and DOPpler were sequential. Developing Trimmer helped to identify issues in SOTA DL auto-tuner design, which led to the further exploratory analysis and the design of DOPpler. Combined together, these two systems allowed my thesis to answer the aforementioned research questions.

1.6 Thesis Structure

Chapter 2 provides necessary information on Machine Learning (ML), DL, DL model life-cycle, specifics of DL computation and systems, characterisation of DL systems, DL compilers, DL compiler optimisations and the design characteristics of DL auto-tuners. Within Chapter 2, SOTA DL compilers and auto-tuners are also comprehensively studied and compared in an overview of the research space.

Chapter 3 presents a comprehensive experimental analysis of the time and energy costs associated with DL inference and DL optimisation via the use of DL compilers and auto-tuners. The chapter discusses the reasons for differential behaviour and operational costs of optimised and un-optimised DL models, as well as the costs of performing optimisations on said models - focusing in particular on DL auto-tuning. Importantly, this chapter provides a comprehensive analysis and discussion of one of the reasons for sub-optimal and inefficient DL auto-tuning - *cold candidates*.

Chapter 4 proposes Trimmer, a framework that utilises a neural-network based filter and a meta-tuning strategy that performs cost-efficient DL auto-tuning on DL models in a single-machine and multi-machine (Cloud cluster) setting. The chapter presents Trimmer's system design and an experimental evaluation of Trimmer against SOTA DL auto-tuners, concluding with a discussion on Trimmer's limitations, compatibility with auto-tuners and target-devices as well as different workloads.

Chapter 5 This chapter presents an experimental analysis of the issues and costs associated with isolated (serial) on-device candidate measurements in DL auto-tuning - an approach that is universally assumed and adopted within the existing SOTA DL auto-tuners and the broader DL compiler community. In response to these issues, a naïve approach is presented to enable parallel intra-device candidate latency measurements, followed by an experimental evaluation of the approach, which identifies several bottlenecks and potential design directions towards achieving high-quality, intra-device parallel candidate measurements during auto-tuning.

Chapter 6 proposes DOPpler, a DL auto-tuning measurement infrastructure capable of performing parallel measurements of candidate tensor programs intra and inter-device, greatly expediting DL auto-tuning, whilst maintaining high optimisation quality and providing plug-and-play compatibility with several existing auto-tuners. Informed by the findings presented in Chapter 5, DOPpler's design is described and evaluated experimentally against the serial measurement infrastructures adopted within SOTA DL auto-tuners. The chapter concludes with a discussion on compatibility considerations and limitations, including potential applications in other DL research areas.

Chapter 7 provides a summary of the contributions presented within this thesis, a discussion on the broader impact of this work with a focus on the energy costs, and is followed by thesis conclusions and future research directions. This chapter also includes an estimation and an analysis of the costs (time, energy, monetary) incurred as a result of the experiments performed as part of this work.

This page is left intentionally blank

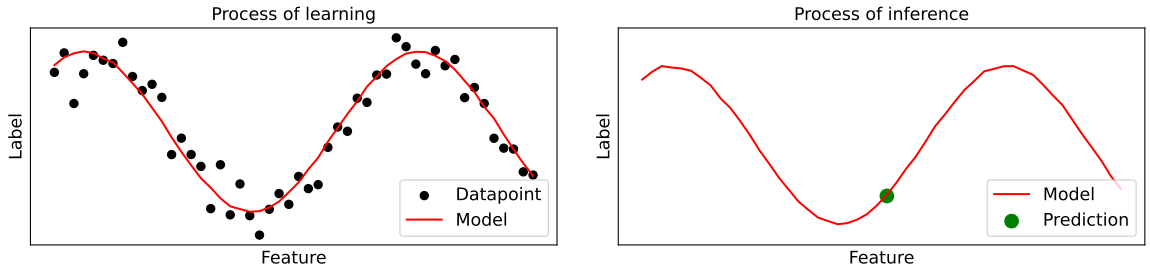
Chapter 2

Background and Related Work

This chapter provides broader context of DL systems, DL inference performance and energy costs, as well as methods for optimising DL model performance during inference. Within Section 2.1 topics related to ML are discussed, focusing on the different ways data is utilised in model training, including different learning paradigms, such as Supervised Learning. Section 2.2 focuses on DL and its associated constructs such as Artificial Neural Networks (ANNs) and DNNs, as well as different types of ANN layers and network types. Within Section 2.3 the concept of DL systems is outlined, encompassing their life-cycle, tensor data structures, DL tensor operators and programs, DL systems including software (DL frameworks, inference engines) and hardware (CPUs and GPUs). Section 2.4 characterises DL systems in terms of their performance and costs, focusing on multiple metrics such as accuracy, computational intensity and measures of energy efficiency used in academia and industry to compare DL systems. Within Section 2.5 DL compilers are introduced, discussing their features (for example, high and low-level optimisations), application specifics and criticality within modern DL inference pipelines, providing a comparison across prominent projects. Lastly, Section 2.6 discusses DL auto-tuning - an automated method for DL model tensor program candidate (schedule) discovery and performance evaluation.

2.1 Machine Learning (ML)

ML is the family of statistical approaches used to discover regularities and patterns in data, by proposing functions that estimate future data points given past value observations and *features* [212]. Features, also known as exploratory variables, are the observable attributes of the set of data—*dataset*, used by ML algorithms to discover correlations that are then used to predict previously unseen values.



(a) Fitting the model to the observed datapoints

(b) Model predicts a label, given an unseen feature

Figure 2.1: Learning and inference phases of Machine Learning

ML is an automated discovery of regularities in data x via a dynamically derived function f (in effect the *model*) that leverages parameters w to produce predictions y_{pred}, and more formally as $y_{pred} = f(x, w)$. This process is more commonly known as *inference* as the model infers predictions based on a priori information. As shown in Figure 2.1, in the process of learning (Figure 2.1a), an ML model is fitted to observed data *instances* that represent features and their corresponding *labels*. Subsequently, during inference, the model is used to propose a corresponding label for previously unseen values for features (Figure 2.1b). The space that is represented by n features is referred to as the *feature space* and can be n-dimensional. A single point in this space is represented by a set of values for each of the features, referred to as the *feature vector* - a single instance of data in the dataset.

2.1.1 Model Training

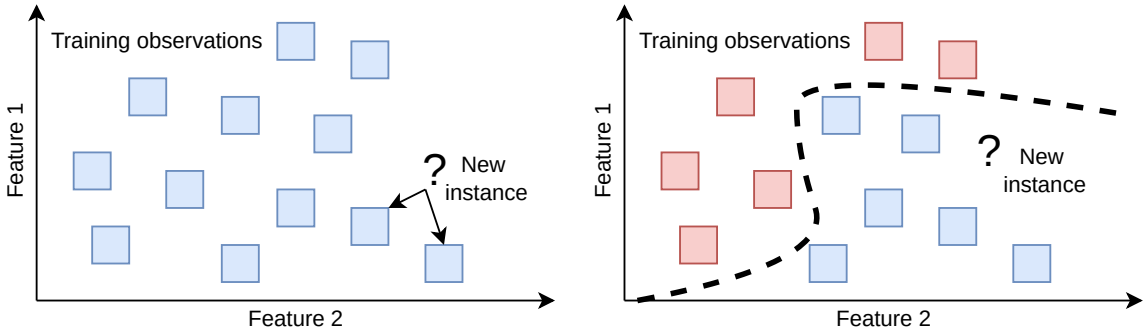
Raw data such as images, text or sound are too complex for an ML model to directly utilise in the process of learning and must be pre-processed into a numerical form. This process is referred to as feature extraction or feature engineering. The extracted features are then used in the process of *training*. During model training, the model f together with its parameters w is incrementally derived by observing available training data and adjusting w with discovered correlative patterns. To incrementally adjust parameters w of the trained model, the difference between the produced prediction and expected value must be determined. This is accomplished using a *loss function* [351], which is based on the measure of error such as $y_{pred} - y$ where y_{pred} is the predicted value and y is the expected value. Equation 2.1 depicts an example of a popular class of loss functions, the Mean Squared Error [37], from which other loss functions such as Root Mean Squared Error or Root Mean Squared Log Error are derived.

$$MSE = \frac{\sum_{i=1}^{N} (y_{pred_i} - y_i)^2}{N} \tag{2.1}$$

Different loss functions are used depending on the ML approach and the desired effects such as: outlier robustness (Mean Absolute Error) [356], ability to determine bias in the model (Mean Bias Error) or penalisation of false positives with large confidence (Cross-Entropy Loss) [381]. To validate the model accuracy and prediction robustness, model evaluation cannot be performed using the data instances that were used during training. A common practice is to divide the dataset into disjoint *training* and *testing* sets at random, where for a set of pairs of instance and label observations $S = (x_i, y_i), i = 1, \ldots, N$ and a probability $z \in [0, 1]$, the model is trained with $N \times (1-z)$ observation pairs picked at random and validated with $N * z$ pairs disjoint from the training set, using the difference between expected and predicted values [269].

To tune model parameters during training, a *validation* dataset is obtained by splitting the *training* set in the same manner as above. The validation set is used to cyclically evaluate the model, to adjust *hyperparameters* that guide the training process, for example, the *learning rate* at which the model updates its parameters [212].

2.1.2 Machine Learning Characterisation



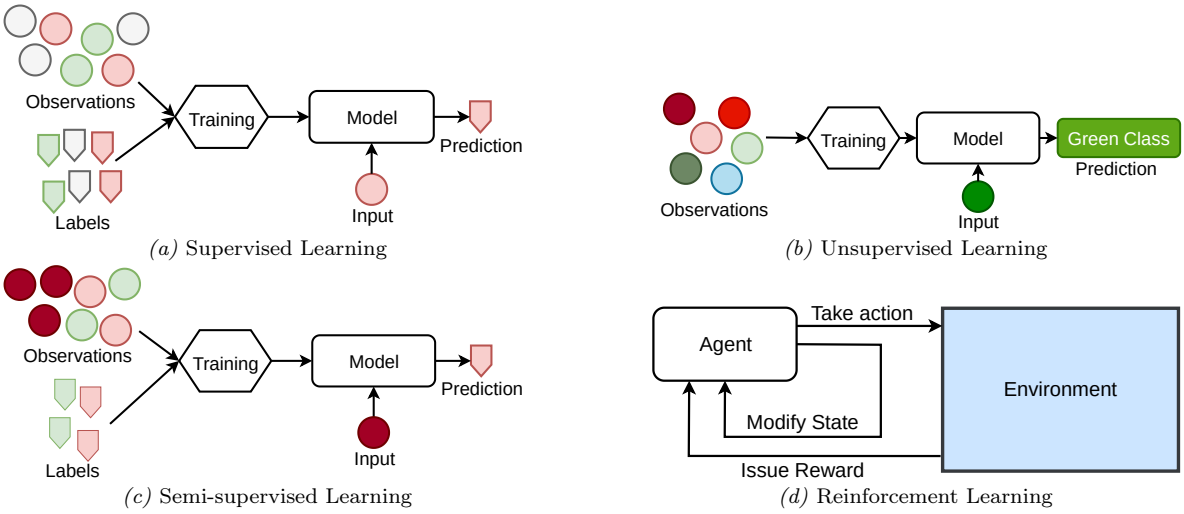
(a) Instance Learning *(b)* Model Learning

Figure 2.2: Differences between Instance Learning and Model Learning

Several criteria can be used to characterise ML approaches, comparing the manner in which the model utilises training data and the methods of learning used to train it. In **batch learning** [27], the training process assumes access to all available training data and performs *offline* training, where the model is first trained until convergence, and only deployed once a satisfactory accuracy is reached. The model does not update its parameters w unless explicitly re-trained with new observations. Conversely, in **online learning** [27], the model is trained within a production environment. Such training is incremental as new data arrives in the form of individual or *mini-batches* of observations. This type of training is useful for datasets too voluminous to fit within machine memory or when the system receives data as a stream of samples at different rates.

ML methods can be characterised by the manner in which they utilise available observations, as shown in Figures 2.2a and 2.2b. In **instance learning** [16], predictions

are performed by comparing ground-truth and unseen observations using a similarity measure such as euclidean distance [186]. Conversely in **model learning**, the ML approach builds a model based on the available observations such that the model generalises to new, unseen ones, as described in Section 2.1.1. The model then performs *inference* based on learned parameters (also known as *weights*). An ML model's learning process can also differ in the degree of interaction the model has with an outside environment during training [212]. This can manifest in querying the environment or performing actions within the environment — **active learning** or only observing the environment or dataset — **passive learning**, with no feedback influence on the environment or a set of data. ML operations can be divided into paradigms that describe overarching learning assumptions such as degree of human involvement.



(a) Supervised Learning *(b)* Unsupervised Learning *(c)* Semi-supervised Learning *(d)* Reinforcement Learning

Figure 2.3: Different ML paradigms

Supervised Learning: As depicted in Figure 2.3a, in Supervised Learning, the process of model training utilises pairs of input values x and corresponding ground-truth labels y to cyclically update its parameters w [212]. As the process continues, w correlates x to y with increasing accuracy, whilst generalising towards unseen inputs $\hat{x}$. During Supervised Learning it is important to consider *over-fitting* and *under-fitting*,

where the process of training causes the model to fit the training dataset too closely or not sufficiently enough [162, 140]. Continual adjustment of training parameters with the help of validation (as described in 2.1.1), aims to prevent both over and under-fitting. Supervised Learning approaches can be further subdivided into classification and regression [212], where in classification, the model predicts discrete classes such as different car models given an image of a vehicle, whereas in regression, the model predicts a non-discrete numeric value, for example, a stock price at a given time. There are many Supervised Learning approaches, including: Decision Trees [286], Random Forests [26], Linear/Logistic Regression [337, 42] and Neural Networks [216].

Unsupervised Learning: Conversely, in Unsupervised Learning the ML approach learns without relying upon labelled observations, instead discovering patterns in data, which are then used during inference [212], as depicted in Figure 2.3b. This paradigm is often applied when working with very large datasets, where labelling their observations would be costly. On the flip side, results of Unsupervised Learning can be less accurate than those produced by models trained under supervision and must be validated manually by experts before deployment in production [89]. Unsupervised Learning methods include techniques such as K-Means or Hierarchical Clustering [187, 149], Dimensionality Reduction such as Principal Component Analysis [358] or Anomaly Detection such as Isolation Forests [189], often used for dataset anomaly detection [35].

Semi-supervised Learning: As depicted in Figure 2.3c, Supervised Learning and Unsupervised Learning approaches can be hybridised into Semi-supervised Learning, where partially labelled data are used [212, 370], and can be particularly useful for obtaining ground-truth labels for Supervised Learning. A good example of Semi-supervised Learning are Deep Belief Networks [124] that utilise several unsupervised models based on Boltzmann Machines [389], which once trained, are fine tuned under supervision. Semi-supervised Learning can be used as an alternative to Supervised Learning in problems such as text classification [211] or speech analysis [371].

Reinforcement Learning: Fundamentally, Reinforcement Learning (RL) methods operate on a separate set of assumptions to Supervised or Unsupervised Learning methods. RL is built around a notion of an *agent* - an algorithm that exists within an *environment* it can observe and interact with [322], as depicted in Figure 2.3d. The RL environment provides a set of states the agent can transition into and out of by performing a set of *actions*. The environment either positively or negatively reinforces the agent (with *rewards*), given the actions taken by the agent and the states it occupies. In the process of navigating the environment, the agent builds a *policy* defining the actions the agent should perform given an occupied state and the state of the environment to acquire highest reward. Formally, the objective of an RL model is to build a policy p parameterised by w that is most optimal in terms of gained reward given the environment. During training, observations consisting of the current state, new state, an action taken and a reward received for performing that action are used. Examples of RL-based methods include the class of Monte Carlo methods [208], Policy Search methods [180] or Proximal Policy Optimisation (PPO) [290]. RL is also used in game-playing scenarios [145], customer behaviour analysis [299] or robotics [161], with many approaches being combined with Supervised Learning methods.

2.2 Deep Learning

Deep Learning (DL) is a subset of ML focusing on the design, training and operation of models based on ANNs [97]. More specifically, DL focuses on DNNs — ANNs with at least three logical layers, enabling ML that solves complex, non-linear problems such as image recognition [197], object detection [190] or language translation [305]. DL enables *feature learning* or *representation learning* — an automated discovery of representations required to extract features from raw data. DL is highly useful in problems such as Computer Vision (CV) (image-based learning). Image data is commonly represented

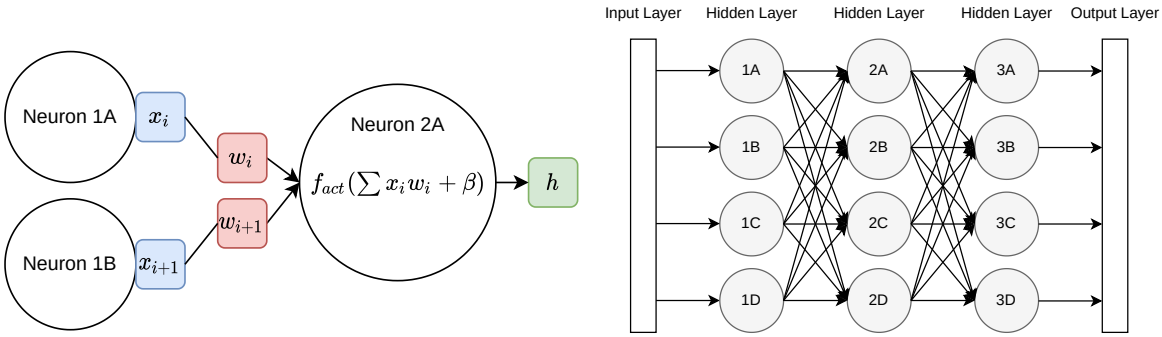
numerically as combinations of scalar values for each of the three colour channels, however manual discovery of features for machine learning in such diverse data is impractical. DL is also well suited for multi-dimensional data operations, exemplified by image-based or Natural Language Processing (NLP) [202] learning tasks.

As such, DL can be thought of as a subset of ML methods that both learn based on data features to provide predictions, but also automatically discover relevant features in data that enable effective learning. DL methods automate feature discovery and their subsequent exploitation for learning and inference, allowing engineers to shift their focus onto careful design of DL model architectures that best suit the application scenario [97]. The majority of DL methods are based on ANNs, an example of supervised, discriminative models inspired by the operation of neural networks found in animal brains. An unsupervised alternative to ANNs are the Deep Belief Networks [179] that act in a generative manner — i.e. can learn to reconstruct their inputs probabilistically as opposed to classify them as in supervised, discriminative ANNs. This thesis focuses on DNNs for the purposes of analysis and evaluation.

2.2.1 Artificial Neural Networks

ANNs are combinations of data and algorithms that operate on them, attempting to roughly model the operation of biological neurons to perform tasks such as learning [68]. An ANN is a structure consisting of artificial *neurons*, also referred to as *nodes* forming a directed, weighted graph, where the edges between the nodes are assigned *weights*. Each artificial neuron is conceptually inspired by biological neurons [109]. Figure 2.4a depicts an artificial neuron (a single node in the ANN graph). A neuron receives inputs x originating from the user or prior neurons in the network (as in Figure 2.4a), and performs a weighted sum between the input and its weights w_i found in the neighbour edges, adding a bias value β. Inputs that originate from the user (for example, during training) can be the feature values of training instances appropriately pre-processed

for the purposes of the network (for example, scaled or truncated). Before dispersing to neighbouring neurons, the outputs of weighted summation are passed through a differentiable *activation function* [336], which decides whether the neuron should activate or not, emitting output h. Neuron outputs are often referred to as *activations*.

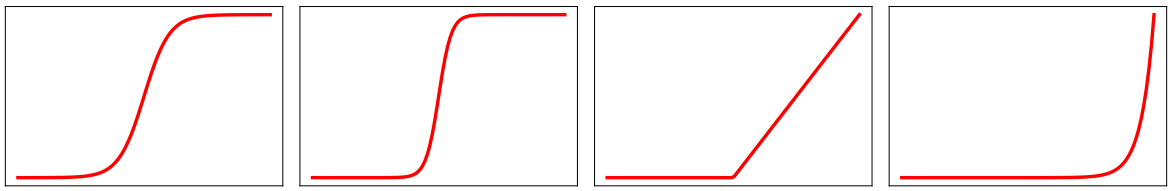


(a) Artificial Neuron — a single Perceptron

(b) Artificial Neural Network — Multi-layer Perceptron

Figure 2.4: A Perceptron and an Artificial Neural Network

Figures 2.4a and 2.4b depict an artificial neuron and a Multi-Layer Perceptron [264] - an ANN that is a minimum viable network to satisfy the definition of a DNN, which must have at least three layers [97]. All networks with less than 3 layers are *shallow*.



(a) Sigmoid *(b)* Hyperbolic Tangent *(c)* Rectified Linear Unit *(d)* Softmax

Figure 2.5: Depiction of popular activation functions

Activation Functions ϕ_{act} activate each neuron within the ANN and produce their outputs (activations). As ANN training requires differentiation to propagate loss backwards through the layers, activation functions must be differentiable [114] and introduce non-linearity, to enable the ANN to approximate patterns by stacking multiple network layers. There are several activation functions used by ANNs such as the Sigmoid [212], which produces outputs ranging between 0 and 1, as shown in

Figure 2.5a, the Hyperbolic Tangent [66] (Figure 2.5b) that produces outputs between -1 and 1, or Rectified Linear Unit (ReLU) [365] (Figure 2.5c) that produces semi-bounded output ranges between 0 and infinity. When weights w are initialised randomly between -1 and 1, the ReLU can prevent up to a half of neurons from activating, causing them to eventually cease partaking in inference. This can be fixed by a Leaky variation of the ReLU activation function [199], which uses very small activation values for negative inputs. Equations 2.2 to 2.6 depict several activation functions, where $\hat{h}$ is the neuron's weighted summation of input x and weights w.

$$\text{[Sigmoid] } f_{act}(\hat{h}) = \frac{1}{1 + e^{-\hat{h}}} \tag{2.2}$$

$$\text{[Hyperbolic Tangent] } f_{act}(\hat{h}) = \frac{e^{\hat{h}} - e^{-\hat{h}}}{e^{\hat{h}} + e{-\hat{h}}} \tag{2.3}$$

$$\text{[ReLU] } f_{act}(\hat{h}) = \begin{cases} 0, \text{ if } \hat{h} < 0 \\ \hat{h}, \text{otherwise} \end{cases} \tag{2.4}$$

$$\text{[Leaky ReLU] } f_{act}(\hat{h}) = \begin{cases} 0.001\hat{h}, \text{ if } \hat{h} < 0 \\ \hat{h}, \text{otherwise} \end{cases} \tag{2.5}$$

$$\text{[Softmax] } f_{act}(\hat{h})^n = \frac{e^{\hat{h}^{(n)}}}{\sum_{n=1}^{N} e^{\hat{h}^{(n)}}}, n = 1, \ldots, N \tag{2.6}$$

Problems such as multi-class classification require a probability distribution to be produced. Any time a representation of a probability distribution over a discrete variable with N possible values is required, the Softmax [97] activation function can be used. Softmax outputs a single value for each neuron in the layer, enabling each value to be interpreted as a class probability, all summing up to 1. Figure 2.5d and Equation 2.6 depict the Softmax function, where $\hat{h}$ are the ANN's last layer outputs.

Layers: As depicted in Figure 2.4b, an ANN structure is composed of layers, where each layer represents one or more neurons that activate simultaneously. Typically, ANN inputs and outputs are considered as separate layers of the network and referred to as the *input layer* and the *output layer* respectively. Any layers in-between the input and output are referred to as the *hidden layers*, and contain the bulk of ANN functionality. The larger the number of layers, the more complex each layer is and the more edges there are between individual nodes across neighbouring layers, the more complex representations can be learned by the network. When viewed from the perspective of multiple layers, an ANN can be conceptualised as a series of functions that depend on each other's outputs, starting from the input layer and ending on the output layer. Equation 2.7 depicts layers of an ANN represented as variations of a generic function $g()$ where N is the total number of layers in the network and w_n are the weights associated with each layer.

$$y_{pred} = g_n(g_{n-1}(g_{n-2}(\dots g_1(x, w_1)), w_{n-2}), w_{n-1}), \forall n \in N \tag{2.7}$$

Neural network training: The main goal of ANN training is to find the weight and bias values that minimise the loss function output (See 2.1.1) [97]. ANNs are typically trained using backpropagation [114], where weights w are set to random values, and subsequently, the observations are passed in batches through the network layers, resulting in output y_{pred}. Predictions y_{pred} are then compared to their ground-truth counterparts and a loss value is computed for the batch of observations using an appropriate loss function $L(y_{pred}, y)$, as described in Section 2.1.1. For example, in image classification, a cross-entropy loss function L [97] can be used, as follows:

$$L(y_{pred}, y) = -\sum_{i=1}^{N} y^i \log(y^i_{pred}) \tag{2.8}$$

where N denotes the number of classes used in prediction.

For every layer l in the network, the average error across the batch of observations B is calculated, as depicted in the following equation.

$$err_B = \frac{1}{B}\sum_{l=1}^{B} L(y^l_{pred}, y^l) \tag{2.9}$$

Subsequently, the loss values are used to update the weights w_l and biases β_l. α denotes the learning rate - a hyperparameter deciding the speed at which the network learns, and δ denotes the derivatives of the loss function w.r.t. w_l and β_l, as follows:

$$w_l = w_l - \alpha\frac{\delta err_B}{\delta w_l} \tag{2.10}$$

$$\beta_l = \beta_l - \alpha\frac{\delta err_B}{\delta \beta_l} \tag{2.11}$$

The derivation finds the slope of the loss function that minimises it, where training requires tweaking the weights and biases that contributed to the output of the ANN. To obtain derivatives of a multi-parameter function L, partial derivatives of each parameter are calculated and combined using the chain rule [101]. For example, for a single neuron n with parameter w_1, input x and bias value β_1 producing output y_{pred_1}, the loss function can be defined as $\frac{1}{2}\sum y - {y_{pred}}^2$ and the derivation of the loss function with respect to the weight w_1 and bias β_1 as follows:

$$\frac{\delta L}{\delta w_1} = \frac{\delta L}{\delta y_{pred}}\frac{\delta y_{pred}}{n}\frac{\delta n}{\delta w_1} \tag{2.12}$$

$$\frac{\delta L}{\delta \beta_1} = \frac{\delta L}{\delta y_{pred}}\frac{\delta y_{pred}}{n}\frac{\delta n}{\delta \beta_1} \tag{2.13}$$

Derivatives are used in the process of Gradient Descent [280], which optimises the ANN parameters to minimise loss. Due to the number of parameters in the ANN, it is often infeasible to calculate derivatives for every parameter. Stochastic Gradient Descent [29] alleviates this by performing derivations on subsets of parameters.

2.2.2 Deep Neural Networks and Their Layers

DNNs can be predominantly categorised into Feed-forward Neural Networks (FFNNs) and Recurrent Neural Networks (RNNs). Other types of DNNs have been proposed such as Autoencoders [165], Spiking Neural Networks [200], Encoder-decoders [17, 41] or Transformers [347]. The remainder of this section focuses on FFNNs and RNNs.

2.2.2.1 Feed-forward Neural Networks

As demonstrated in Figure 2.4b, FFNNs can be defined as ANNs that transform inputs by propagating them forward through the network's layers, forming a Directed Acyclic Graph (DAG). A simple form of an FFNN is the Multi-Layer Perceptron, a network consisting of at least three layers: input, output and at least one hidden layer (for details, see Section 2.2.1). Supervised FFNNs can be trained using backpropagation.

Fully-connected (FC) Layers layers (a.k.a. Dense or linear layers) are functionally equivalent to Multi-Layer Perceptrons as they connect every neuron of the layer to every activation in the preceding layer [97]. An FC layer is composed of a weighted matrix multiplication of the input x originating from the preceding layer, local weights w and commonly a bias term b expressed as a vector, which is added to the result of the multiplication. The resultant output is then passed through an activation function (see Section 2.2.1) to produce the activations. Equation 2.14 depicts computation of the output for an FC layer l.

$$y_j^l = f_{act}(\sum_{i=1}^{N} w_{ji}^l x_i^{l-1} + b_j) \tag{2.14}$$

FCs are sometimes referred to as General Matrix Multiply (GEMM) layers as they can be computed using matrix multiplication, or to increase computation efficiency, expressed as a dot-product operation: $y = f_{act}(w \cdot x + b)$.

2.2.2.2 Convolution Neural Networks (CNNs)

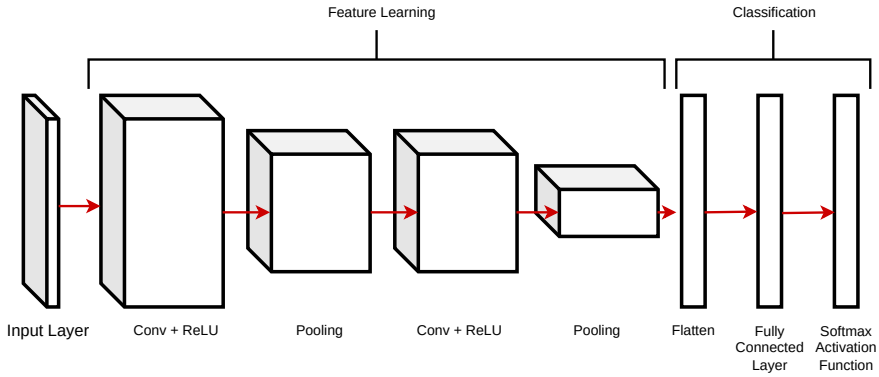


Figure 2.6: Convolution Neural Network

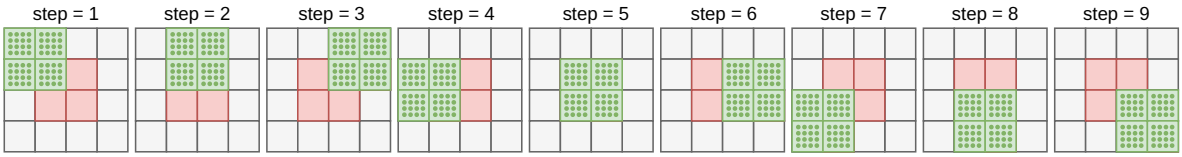


Figure 2.7: Convolution operation over 2×2 matrix (solid red) with padding $= 1$ (solid grey), stride $=$ 1 and a 2×2 filter (dotted green)

As depicted in Figure 2.6, Convolution Neural Networks (CNNs) are a class of FFNNs that facilitate solving spatial classification problems in domains such as image recognition or object detection [10]. CNNs combine FC layers with Convolution layers — specialised ANNs layers inspired in their design by the operation of the mammalian visual cortex, where neurons compartmentalise their inputs into regions that partially overlap and connect to individual neurons in the network, effectively building up a spatially networked structure of feature-recognising entities.

Convolution Layers abstract their inputs into *feature maps* or *activation maps* by convolving [97] a set of learnable *filters* [1] with regions of the input data, as shown in Figure 2.7. This enables Convolution layers to leverage local spatial correlations

[1]Filters are also referred to as *kernels* or *weights* (w) and are the learnable components within a Convolution layer

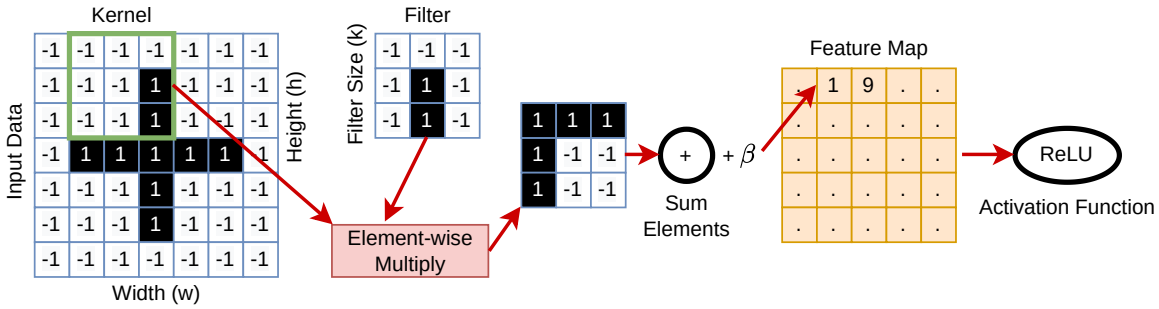


Figure 2.8: Convolution over image data representing a cross

between data features and substantially reduce the number of neuronal connections between two layers that must be computed (compared to FC layers). When filters are updated during training, they are transformed such that the layer produces activations that respond to specific spatial features (for example, edges or shapes) at specific spatial positions in the input. The fact that CNNs are FFNNs, enables early layers to recognise simplistic features such as lines or edges, while subsequent layers learn combinations of the simple features and recognise digits, faces, vehicles and more.

Figure 2.8 depicts a Convolution operation over a matrix representing an image of a cross with a single colour channel. For the purposes of 2D Convolution, data is specified as a tensor of dimensions $w \times h \times C_{input}$ where w and h are the width and height of the array and C_{input} is the number of channels (3rd dimension).

Convolution over input occurs in *strides*, defined by the engineer as a hyperparameter, with input commonly padded with zeroes to maintain desired layer dimensionality (see Figure 2.7). During Convolution, each filter moves across the input from left to right, top to bottom and across all channels by *stride* steps, performing element-wise multiplication with the portion of the input. The result is subsequently summed, to which a bias β is added, producing a single element of the feature map. All produced feature maps pass through an activation function such as ReLU to construct the layer output. Depending on the stride and padding, Convolution transforms the dimension of the output compared to its input. For example, a Convolution operation that accepts a

tensor of shape $w \times h \times C_{input}$, utilising F filters of size γ, with stride s and padding p, produces a tensor of shape $\hat{w} \times \hat{h} \times C_{output}$, where $\hat{w}$, $\hat{h}$ and C_{output} are defined as:

$$\hat{w} = \frac{w - \gamma + 2 * p}{s + 1} \tag{2.15}$$

$$\hat{h} = \frac{h - \gamma + 2 * p}{s + 1} \tag{2.16}$$

$$C_{output} = F \tag{2.17}$$

For a single element of a feature map (O) Convolution can be performed as follows, where k is the filter, x is the input and m and n are the feature map coordinates:

$$O_{mn} = \sum_{i}^{\gamma} \sum_{j}^{\gamma} k_{ij} x_{i+m,j+n} + \beta \tag{2.18}$$

Pooling Layers: Commonly placed directly after Convolution layers, the Pooling layer reduces dimensionality of Convolution outputs, yet preserves spatial invariance, downsampling the data [212]. Pooling layers traverse the inputs in strides using pre-defined windows and summarise values on each stride under the filtered range. Different types of pooling can be applied in CNNs such as: average pooling, global pooling or min pooling, with max pooling used to capture the most prominent features in the Convolution feature maps. Figure 2.6 depicts a simplified CNN, consisting of several Convolution and Pooling layers in the spatial, feature-learning portion and prediction-producing, FC layers during the latter portion of classification.

Normalisation Layers: Inputs to DNNs can vary in their distribution (for example, some contain negative values or have upper bounds of $1e^3$ or larger), where within deeper networks, such an arrangement may cause neuron instability and side-effects such as exploding gradients, reducing training effectiveness. Normalisation layers, with Batch Normalisation (BN) [138] being the most common kind, re-center and re-scale layer inputs (including those from prior layers) to normalise and stabilise the network.

Alternative normalisation methods using Layer Normalisation or Group Normalisation [320] can also be used. BN can cause interdependence between input samples within a batch and affect the learning rate. Techniques such as Gradient Clipping [31] help to alleviate these problems by reducing sensitivity of the network to its inputs.

2.2.2.3 Recurrent and Long-short Term Memory Neural Networks (LSTM)

While CNNs are suited for independent, grid-like inputs, Recurrent Neural Networks (RNNs) process sequences of related samples, such as text [209] and are designed to scale to variably-long sequences, thanks to parameter sharing. RNNs also maintain additional internal state that combines current information with prior information from the sequence when performing inference. RNNs are discussed in more detail in Appendix A. Stemming from RNNs, Long-short Term Memory (LSTM) networks emerge, utilising recurrent cells that control what information is stored within the internal state, what information is forgotten and what information can be passed onto the next recurrent cell — via gates. LSTM networks are particularly successful in the area of Natural Language Processing (NLP) and text-related problems such as language translation or text generation [353]. LSTMs are discussed in more detail in Appendix A.

RNN models are typically implemented by RNN cells, while LSTM cells introduce gates that improve upon the vanishing or exploding [252, 116] gradient problems in conventional RNN cells. Superseding RNNs and LSTMs, DNNs such as the Transformer [347] utilise Attention layers [221] that capture how neighbouring layer outputs influence one another, also performing distant associations between input tokens, capturing context.

2.2.3 Systems for DNN Computation

The ability of DL models to deliver rapid predictions during inference given input data - for example, to decide how to classify an image in less than 1ms, is possible thanks to

complex DL systems that underpin and enable DL computation.

2.3 Deep Learning Systems & Computation

This section describes how DL models are transformed into sets of individual programs that operate on tensors and how DL systems - ecosystems of different software such as DL frameworks, inference engines and compilers are used during training and deployment for inference, across high-performance processors and accelerators, to deliver DL model outputs in a timely and efficient manner.

2.3.1 Tensors and Tensor Operators

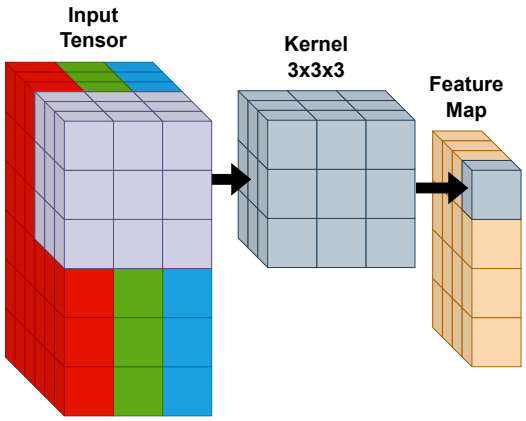


Figure 2.9: Applying 3×3×3 Convolution kernel to a 6×6×3 input tensor (stride=1, padding=0). Input Tensor contains three channels of 6×6 matrices forming a 6×6×3 tensor in the HWC layout.

Tensors are generalisations of vectors and matrices[2] capable of representing data in multiple dimensions [97], supporting many of the operations performed on vectors and matrices such as multiplication or dot-product. Tensors with dimensionality higher than 2D are common in DL (for example, 3D, 4D, 5D tensors), often referred to as the *rank* of a tensor (for example, a 3×3, 2D tensor has a rank of two). In DL, tensors are the default

[2]Scalars, vectors and matrices are zero, one and two-dimensional tensors respectively.

data structures that contain DNN inputs, outputs, weights, inter-layer activations and are used for intermediary storage within layers [7]. For example, CNN networks accept input tensors with data layouts representing the batch size (N), width/height of the image (WH) and number of channels (C), to represent the RGB image components.

Figure 2.9 depicts a set of tensors used during the Convolution operation, where the input layer has an NCHW layout with the outermost and innermost dimensions of N and W. The data layout determines how the data will be laid out in memory, thus influencing the memory access patterns to these memory regions by the layer's operations. Tensor data layouts and transformations are described in Section 2.5.3.

2.3.1.1 Tensor Operators and their types

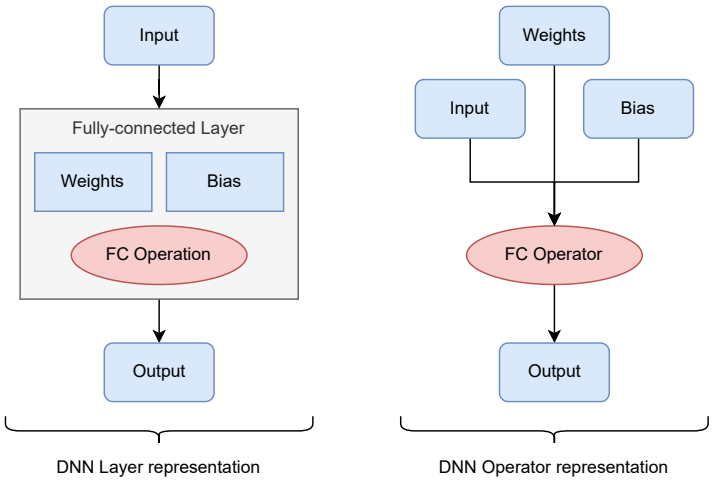


Figure 2.10: Difference between a DNN layer and a DNN tensor operator. The operator representation decouples data from computation when expressing DNN topologies.

As described in Section 2.2.2, DNNs are composed of layers such as FC or Convolution, with the operations performed within them usually represented by *tensor operators* [97]. The core difference between a DNN layer and a tensor operator lies in the decoupling of computation from data, as depicted in Figure 2.10. Early DL frameworks such as Caffe [147], combined operations with the weights and bias data. Modern DL frameworks

implement tensor operators as separate subroutines with multiple inputs and one or more outputs, decoupling data from computation.

DNNs are predominantly described as graphs, where the nodes represent the tensor operators and the edges represent the data flow between them in the form of encoded tensors, managed by the DL framework. When performing inference or training, the DL framework matches tensor operators to their internal implementations that accept inputs, perform computation and produce outputs, feeding the produced outputs to subsequent tensor operators. A multitude of tensor operators exist[3], where each performs a unique DNN or ML operation and can be grouped into the following types:

Algebraic Operators include operators such as lograrithm (*log*), exponential (*exp*), power (*pow*), cosine (*cos*), sine (*sin*) or tangent (*tan*), performed on a tensor element-wise, or *add*, which adds two or more tensors element-wise, amongst others.

DNN Layer Operators perform operations of DNN layers, such as the FC, which computes the FC layer. *Conv* performs variations of 1D/2D/3D Convolution given input, weights, kernel and bias. *BatchNorm* performs BN, while Max/Min/AvgPool performs pooling of an input tensor and *Sigmoid* applies the Sigmoid activation function to a tensor element-wise (see Equation 2.2 in Section 2.2.1).

Tensor Transformation Operators transform the dimensionality of tensors. The *reshape* operator changes tensor dimensions to those provided as a parameter, for example, given parameter $\{5, 3, 4\}$, transforms tensor shaped $3 \times 4 \times 5$ into $5 \times 3 \times 4$. *Concat* operator concatenates tensors, provided their dimensions match, for example, $4 \times 3 \times 3$ tensor with $2 \times 3 \times 3$ tensor producing a $6 \times 3 \times 3$ tensor. Other operators exist, such as *Transpose* or *Flatten*, which flattens a tensor (for example, $3 \times 3 \times 3$ into 1×27).

Broadcast Operators transform tensors such that their shapes are matching. For example, when performing an addition between a matrix and a vector, a *BroadcastAdd* operator will replicate (broadcast) the vector enough times to match its shape to the

[3]TensorFlow [2] implements more than 1000 unique tensor operators

shape of the matrix, such that element-wise addition can be performed. Broadcast operators are usually combined with other operators such as *Add* or *Mul* (Multiply).

Reduction Operators reduce the number of elements within the tensor, including: *Sum* (sums elements of a tensor), *Min/Max/Avg* (retrieve min/max/mean value of the tensor) or *ArgMax/ArgMin* (retrieve index of the max/min value within the tensor). More complex operators, often used within DNN layers (MinPool, AvgPool) are also reduction operators as they reduce the number of elements within the input tensor.

Control Flow and Boolean Operators: Operators such as *If*, *Else* or *Loop* enable control flow for data-dependent conditional execution in architectures such as RNNs; and paradigms such as RL. Control flow operators accept as a parameter the condition and alternative subgraphs of the model to be followed when the condition is false. Boolean operators such as *Or*, *And*, *Not* enable element-wise application of boolean logic to tensor data, defining dynamic condition variables for control-flow operators.

2.3.2 Deep Learning Systems

To enable effective architecting, training and deployment of DL models, complex DL systems are used by engineers and designers in various ways. A DL system can be designed as a single machine or a set of multiple machines equipped with necessary hardware and software, that support DL operations such as training or inference. DL systems are composed of multiple parts that work together to achieve these goals.

2.3.2.1 Models and their applications

Conceived in the 1960s [139], DNNs did not gain prominence until models such as LeNet (1989) [177] were introduced, limited by the amount of available data and compute capabilities. DL model development was gradual until AlexNet (2010) [168] proposed utilising GPUs as the core DNN computation processors, greatly expediting training and inference. In the last decade, adoption of novel DNN architectures into various areas

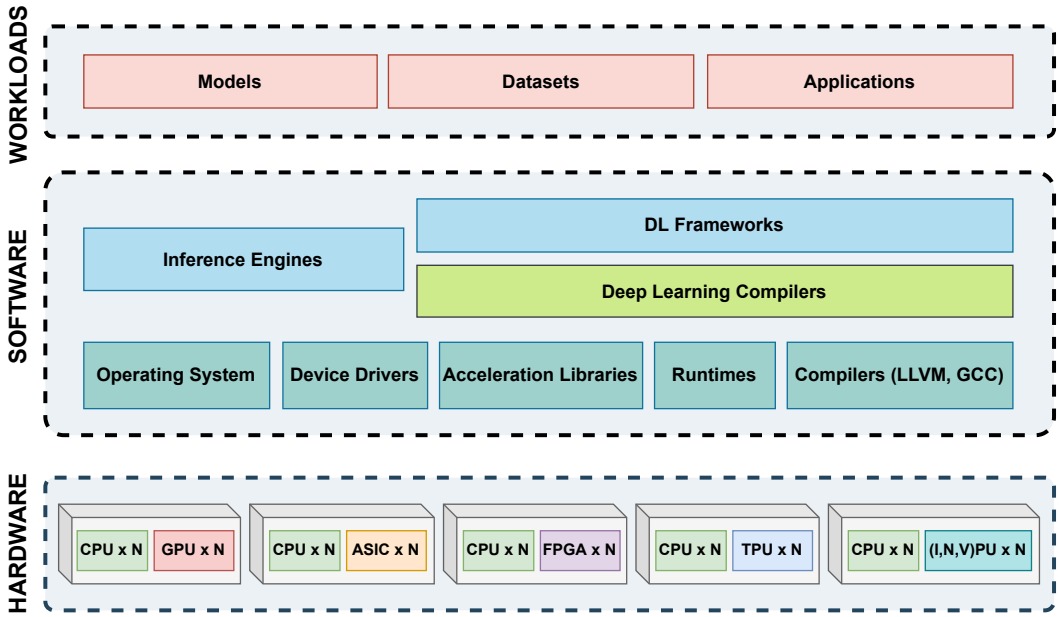


Figure 2.11: Components and abstraction layers of DL systems.

of science and industry has accelerated and naturally divided into sub-areas based on intended applications. The most popular sub-domains include CV which enables image classification, object detection or image generation; NLP, enabling sentiment analysis, Q&A systems or Neural Machine Translation of text; Recommender Systems, which recommend items to users based on their prior interactions and; RL, with its primary uses in gaming and robotics. This thesis focuses on the CV domain of DL models due to its prominence and scale of adoption in various applications. The CV domain can be further divided into DL models that support specific CV tasks, as follows:

Image Classification (IC) is one of the primary tasks within the domain of CV. IC models learn features from a set of images to then recognise and classify unseen images. Well-known examples of IC models include the AlexNet [168], MobileNets [121, 288, 120], DenseNets [125], ResNets [113] or the ConvNeXt model [194]. This thesis focuses on IC DL models due to their prominence within academia and industry. Details about the explored models can be found in Appendices D.1 to D.9.

Image Generation: During Image Generation, model learns the image feature distribution of some domain (for example, animals), to then generate representative examples given identifiers such as dog breeds. The Generative Adversarial Networks such as DCGAN [260], 3DGAN [362] or StackedGAN [126], as well as Generative Pre-Trained Transformers such as DALL-E [265] are popular examples of such networks.

Object Detection models discover object positions within an image and in some cases, classify them. Prominent object detection models include SPP-nets [112], Single Shot Detectors [191], R-CNNs [94] or the YOLO network [268].

2.3.2.2 Datasets

To perform supervised training of DNNs for IC, the DNNs use example images labelled with correct classes, such that the DNN learns the associations between the classes and discovered features. There are several prominent IC datasets, including: MNIST (70,000 $28\times28\times1$ images of handwritten digits) [178], CIFAR-10/100 (60,000 $32\times32\times3$ images of entities such as cats, dogs, cars) [167], SVHN (600,000 $32\times32\times3$ images of house numbers) [219] and ImageNet [282] - the most widely used IC dataset, containing over 14m $224\times224\times3$ images labelled with 1000 classes of various entities. ImageNet gave rise to the ILSVRC challenges where IC model designers compete at classifying the ImageNet test dataset with the highest achieved accuracy, leading to many new developments in the CV domain. Many other IC datasets exist, varying in the number of examples, variability of classes and image size and quality. COCO [188] or the KITTI [91] dataset used in vision for robotics are examples of datasets for CV tasks such as Object Detection, with Image Generation tasks often reusing IC datasets.

2.3.2.3 DL Frameworks

With the emergence of various DL model architectures, it became critical to reduce the effort to develop DL training and deployment pipelines. As a result, multiple DL

frameworks have been proposed, facilitating model design using high-level programming languages such as Python and abstractions to DL operations. The two most prominent DL frameworks are Tensorflow [328, 2] and Pytorch [1, 249], and both support high-level DL model design, DAG-based architecture specification, training and inference. Pytorch enables dynamic, high-level graph composition and is often leveraged for experimentation with novel training pipelines. Conversely, TensorFlow utilises a static computation graph, enabling more aggressive performance optimisations for inference and training. The two frameworks compete in terms of their features (distributed execution, model serving, device support), with equally large number of DL projects leveraging them. Other frameworks such as Apache MXNet [79], PaddlePaddle [243] or Keras [331] fill in feature gaps such as ease of prototyping, facilitate end-to-end product development or focus on parallel execution. Open Neural Network Exchange (ONNX) [82] is a cross-compatible representation format for DL models originating from different DL frameworks. Through ONNX, engineers can export and import models between DL frameworks to leverage their features. Many frameworks have been discontinued, whilst others merged into more prominent projects [335, 147, 292, 9]. This work utilises several DL frameworks including Pytorch, Apache MXNet and ONNX due to their prominence.

2.3.2.4 Hardware

There are several types of processors that support DL computation. Whilst CPUs are capable of executing any type of workload (including DL computation) thanks to their rich instruction sets and versatile Arithmetic Logic Units, other processors flourish in DL due to their supreme abilities in parallelising computation, and the DL computation being highly parallelisable. The embarrassingly parallel DL computation does not require the versatile compute capabilities provided by the CPU (being primarily matrix or tensor computations), and as such, dedicated accelerator processors provide higher compute efficiency and performance-per-watt ratio. Processors used in DL computation

can be generally grouped into Temporal (Instruction) Processors (TIPs) and Spatial (Dataflow) Processors (SDPs) (for example, ASICs). The following paragraphs describe TIPs such as CPUs and GPUs, while SDPs are outlined in Appendix B.

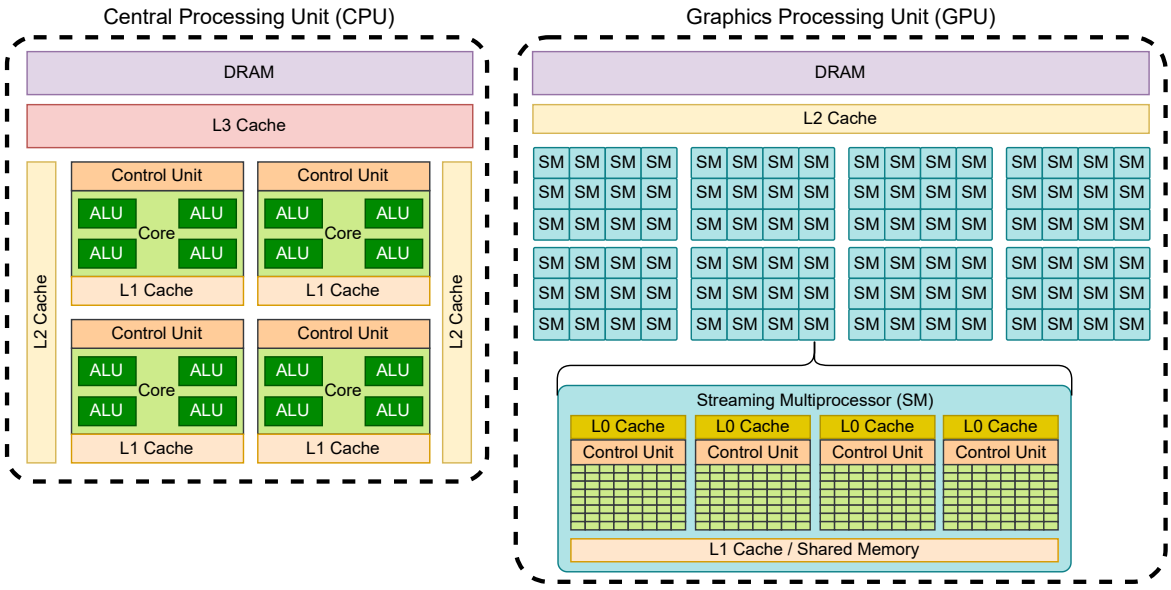


Figure 2.12: Differences between CPUs and GPUs. SM stands for Streaming Multiprocessor

Temporal (Instruction) Processors (TIPs) compute guided by a monotonic clock, signalling to perform actions such as loading or storing data within the memory or executing instructions [245]. CPUs and GPUs are primary examples of TIPs, as shown in Figure 2.12. TIPs leverage the clock to control a number of Arithmetic Logic Units (ALU) that perform arithmetic and logical operations and do not otherwise communicate between each other, instructed by Instruction Set Architectures (ISAs). CPUs typically implement one of two ISAs - the Complex Instruction Set Computer (instructions require multiple clock cycles), or Reduced Instruction Set Computer (multiple simple instructions per clock cycle). Parallelism is achieved via multi-core and multi-thread designs in TIPs, implementing paradigms such as Single-Instruction Multiple-Data (SIMD) for CPUs and Single-Instruction Multiple-Threads (SIMT) for GPUs. DL workloads often leverage these parallelisation capabilities to accelerate computation.

Central Processing Units (CPUs) consist of execution elements referred to as *cores* that contain Arithmetic Logic Units, Control Units, and low-level cache memory, as depicted in Figure 2.12. CPU cores are more versatile and complex compared to processing elements found in SDPs. The Arithmetic Logic Units within cores work independently and may be occupied by different execution threads concurrently, driven by high-frequency clocks. CPUs provide most flexibility in terms of programming compared to other processors, at the expense of more complex and energy-costly instruction-based computation. In terms of DL, CPUs are widely used for inference, and less so during training due to their more serial architectures, compared to other processors such as GPUs. Novel CPU designs include increasingly higher number of parallel cores[4] as well as dedicated silicon for tensor and vector operations enabled via ISA extensions[5], together with pipeline parallelism, whereby multiple instructions execute during the same clock cycle on the same core, enabling Simultaneous Multi-threading (SMT), which effectively doubles the number of cores.

Graphics Processing Units (GPUs): Similar to a CPU, GPU contains execution elements referred to as *cores* or *threads*, Control Units, low-level cache and Dynamic Random Access Memory (DRAM), as depicted in Figure 2.12. Unlike the CPUs, GPUs are designed for embarrassingly parallel and energy-efficient but less versatile computation such as vector or matrix arithmetic. GPU cores cannot decode and execute instructions independently like CPUs do, where instead they process instructions in groups of cores called *warps*[6]. Such primitive cores can sometimes be referred to as Processing Units such as a Floating-Point Unit (FPU) or Integer Unit since they perform a limited set of one or few simultaneous primitive operations on every clock cycle, unlike the more complex CPU cores. Specifically in Nvidia GPUs, these primitive cores are contained within Simultaneous Multiprocessors (SMs) - collections of cores that contain

[4]Modern Intel Xeon and AMD EPYC CPUs have up to 128 virtual cores [52, 4]
[5]Advanced Matrix Extensions (AMX) [53] and Advanced Vector Extensions (AVX) [46, 48]
[6]Nvidia: *warps*, Intel: *thread-groups*, AMD: *wavefronts*

the Processing Units, along with other resources, such as shared memory, caches, and special function units that perform sine, cosine, reciprocal, and square root functions. DL tensor operators such as Convolution can be accelerated on GPUs since they involve repeated matrix arithmetic, and can be easily parallelised across the thousands of available cores. Newer Nvidia GPUs (Volta onwards) include *tensor cores* - dedicated silicon for one-shot matrix arithmetic [237].

2.3.2.5 Acceleration Libraries

With many different processors, ensuring high-performance execution of DL models is a challenging task, especially for designers without prior systems programming expertise. Acceleration libraries expedite adoption of various processors, their ISAs and one-shot instructions for DL model execution, providing high-performance linear algebra, vector and tensor mathematics or DNN layer implementations (e.g Convolution and BN). Several prominent acceleration libraries are widely used for DL acceleration. Intel oneMKL [47] and oneDNN [51], provide high-performance implementations of operators such as compute-intensive Convolutions or memory-bound BNs. Nvidia Compute Unified Device Architecture (CUDA) [231] and AMD's equivalent ROCm [5], provide high-level programming interfaces to GPUs, enabling parallelisation of DL workloads, while cuDNN [232] implements DL-specific routines and operators via CUDA. DL engineers can also leverage Basic Linear Algebra Subprograms libraries to accelerate tensor computation, such as cuBLAS [230], rocBLAS [6], OpenBLAS [363] and Eigen [141]. The disadvantage of relying upon vendor-provided acceleration libraries is that their ongoing development fails to keep up with development of novel DL operators.

2.3.2.6 Inference Engines

Inference Engines (IEs) are either bespoke systems or lean versions of existing DL frameworks that focus purely on DL inference with already trained models across select

platforms, including applying high-level optimisations to models that target the chosen hardware platform. IEs analyse and optimise DL model graphs (TensorFlow Grappler optimiser [170], Pytorch Just-in-Time optimiser/runtime [84]) to dispatch individual tensor operators to platform and device-specific implementations from acceleration libraries or the IE for a specific device. Development of IEs is growing, where prominent ones are: TensorFlow-Serving [327] and TorchServe [85] for large-scale (cluster-level) deployment of models for inference; Nvidia Tensor RT [240] - an inference toolset leveraging Nvidia GPUs and acceleration libraries, with support for Pytorch, TensorFlow and ONNX; TensorFlow Lite [329] and TensorFlow.js [330], enabling DL inference at the Edge (Android, iOS) and in web browsers; ONNX Runtime [57], supporting inference with ONNX models across a variety of platforms; and AWS Neuron [297] that supports model execution on Amazon Inferentia.

Much like DL frameworks, IEs focus on a limited number of target-devices, DL frameworks and tensor operator types, each adopting different strategies for accessing platform's resources, ingesting models, performance optimisation and computation scheduling. This degree of variance requires DL engineers to either limit their choice of target-devices, platforms, frameworks and models, manually extend IEs to suit their specific use-case, or utilise multiple IEs and maintain multiple model inference pipelines.

2.3.2.7 Deep Learning Compilers

As outlined above, the engineering approach of developing high-performance implementations for combinations of M models, N DL frameworks, T tensor operators within each model, P target devices, I (billions) variably-optimal implementations per tensor operator and device[7], results in engineering complexity of $O(MNTPI)$. This limits adoption and performance of processors, tensor operators and their novel implementations. To alleviate these problems, DL compilers were proposed, providing

[7]Implementation specifics depend on the shape of inputs to the operator

unified interfaces for specifying logical representations of DL model graphs and tensor operators. DL compilers ingest trained models from DL frameworks and apply various high and low-level optimisations such as *auto-tuning* and *autoscheduling*. Finally, DL compilers compile the resultant model towards devices such as CPUs, GPUs, ASICs and Field-Programmable Gate Arrays (FPGAs).

DL compilers are an emerging technology with a large potential to substantially improve DL inference performance, however, applying optimisations facilitated by DL compilers can be time-consuming, leading to their limited adoption. This thesis focuses on DL compilers, and more specifically on their strategies for high and low-level optimisations, auto-tuning and the time and energy costs associated with applying them, with the broader aim to reduce these costs and the aforementioned barrier to entry. DL compilers are covered in more depth in Section 2.5.

2.3.3 Tensor Programs and Their Execution

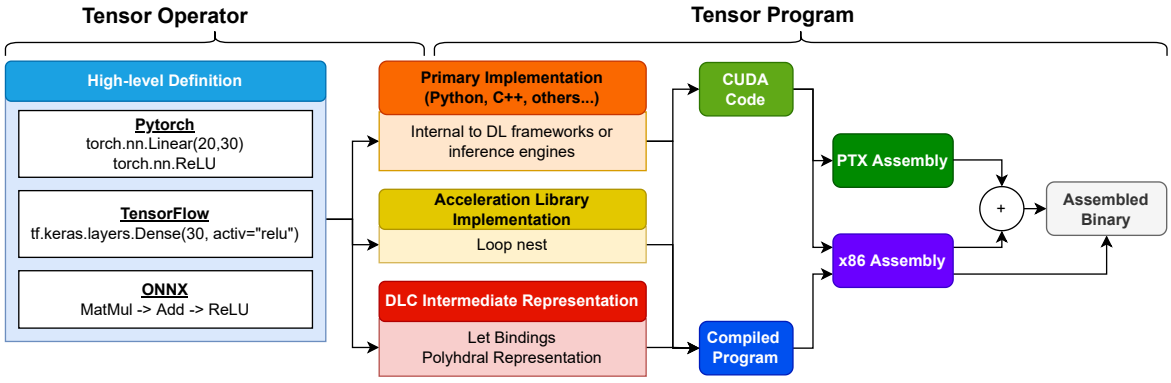


Figure 2.13: Evolution of a tensor operator into a tensor program

As outlined in Section 2.3.1, DL tensor operators provide high-level, logical representations of DL model layer computation. Tensor operators implemented towards a given execution environment (CPU, GPU, specific runtimes/libraries - for example, CUDA) are referred to as *tensor programs*. In DL compiler nomenclature, a tensor operator

specification may be referred to as *tensor expression* or a *primitive function*. Each such tensor operator has operands consisting of tensors inputs, operation(s) expressed as a function of the operands and an output in the form of a tensor(s). A specific implementation of the tensor operator, expressed in low-level DL compiler Intermediate Representation (IR) is often referred to as a *schedule*. Figure 2.13 depicts an FC layer as a tensor operator and program across DL frameworks and a DL compiler.

2.3.3.1 Tensor Program Computation

To achieve high DL performance on CPUs and GPUs that enable hardware parallelism via SIMD and SIMT, tensor operators such as Convolution or FC are often implemented as matrix multiplications, that can leverage parallel Multiply Accumulate (MAC) operations. For example, an FC layer tensor operator can be represented as a matrix vector multiply (+accumulate) or matrix matrix multiply (+accumulate) by rearranging the inputs and the weights for matching dimensions. Recalling Equation 2.14 in Section 2.2.2, an FC layer's operation applies a dot-product between the input and layer's weights and adds a bias term. To compute the FC tensor operator using matrix vector multiplication, the 3D input (Inp) tensor with shape $C \times H \times W$ must be rearranged into a vector $(CHW) \times 1$, and the $M \times C \times H \times W$ 4D weights (W) tensor re-arranged into a matrix $M \times (CHW)$. The FC operator can then be performed as follows:

$$O^{1 \times M} = W^{M \times (CHW)} \times Inp^{(CHW) \times 1} \tag{2.19}$$

Likewise, to utilise matrix matrix multiplication for batched FC tensor operator computation, the input tensor of shape $N \times C \times H \times W$ containing N inputs would be rearranged as a matrix of shape $(CHW) \times N$ to then produce output O as follows:

$$O^{M \times N} = W^{M \times (CHW)} \times Inp^{(CHW) \times N} \tag{2.20}$$

Convolution as well as other tensor operators can also be represented as matrix or vector computation. This is possible using Toeplitz matrices [316, 346], with further reductions to the necessary computations achieved via algorithmic transforms [174] such as: the Strassen's matrix multiplication algorithm [317], the Winograd transforms [357] or Fast Fourier Transforms (FFTs) [344]. The FFT approach performs best with weight tensors of shapes 5×5 and above, and Winograd for shapes 3×3 and below. When performed within DL frameworks, different Convolution implementations may be used via different routines present within DL acceleration libraries, optimised for specific Convolution shapes and processor characteristics.

```
// Convolution
// O = output tensor
// I = previous layer's activations
// W = current layer's weights
for (n = 0; n < N; n++) {
  for (oc = 0; oc < O_C; oc++) {
    for (ic = 0; ic < I_C; ic++) {
      for (oh = 0; oh < O_H; oh++) {
        for (ow = 0; ow < O_W; ow++) {
          for (fh = 0; fh < F_H; fh++) {
            for (fw = 0; fw < F_W; fw++) {
              O[n][oc][oh][ow] = (I[n][ic][oh+fh-1][ow+fw-1]) * W[oc][ic][fh][fw] + O[n][oc][oh][ow]
            }
          }
        }
      }
    }
  }
}
```

Listing 2.1: Naïve, deep loop nest implementation of Convolution

Naïvely, Convolution can be implemented as a seven-layer deep loop nest, as shown in Listing 2.1, where I is the input tensor, W is the weights tensor, O is the output tensor, N is the batch size, O_C is the number of output channels / number of filters, I_C is the number of input channels, O_H and O_W are the height and width dimensions of the output tensor and; F_H and F_W are the height and width of the filters. Such an implementation is sub-optimal given its frequent memory accesses. Efficient memory access is a major aspect of developing high performance DL tensor program implementations, since

accessing memory is orders of magnitude slower compared to arithmetic operations [118, 323]. Many DL acceleration libraries design Convolution implementations to better fit within the memory and cache hierarchies of CPUs or GPUs, for example, by using tiling [354]. Tiling and other loop nest transformations that improve runtime performance of tensor programs are described in Section 2.5.5.

2.3.3.2 Parallelism during DL inference

Since many of the tensor operators found in DL models can be expressed as tensor programs relying upon MAC operations, such tensor programs can be parallelised in various ways across the CPU and GPU.

Model-level parallelism: Multiple DL models can be executed simultaneously by scheduling their tensor programs to separate CPU cores. Individual tensor programs tend to execute serially, one after another to maintain logical data dependency, since outputs of one layer are inputs to another. Processes executing tensor programs can execute their data-parallel instructions (for example, matrix arithmetic) on the GPU via calls to libraries such as CUDA. In Nvidia's GPUs, calls from processes reside within separate CUDA Streams [271] and are serialised, unless modules such as Nvidia Multi-process Service (MPS) [228] are enabled. When considering a single model, provided there is sufficient resource capacity, parallelism can be achieved by executing multiple tensor programs simultaneously. For example, the ResNet-18 [113] architecture (see Appendix D.7) contains shortcut Convolution layers within its residual blocks that can execute in parallel without affecting model correctness. Each such tensor program could be scheduled as a separate thread or process, and a separate CUDA Stream on the GPU. Multiple tensor programs executing within the same process and variations of the above are often leveraged in DL IEs (see Section 2.3.2.6) to achieve low DL inference latency.

Tensor program-level parallelism: Within a single tensor program, (for example, one implementing the Convolution operator), parallelism can be achieved by leveraging

parallel execution units such as CPU or GPU cores and instruction pipelines via the SIMD or SIMT models, as described in Section 2.3.2.4. The SIMD paradigm implemented by modern CPUs as a specialised set of Arithmetic Logic Units and control circuitry, or dedicated silicon, enables a single instruction (for example, Advanced Vector Extensions (AVX)) to instruct computation across a range of values in large registers simultaneously. For example, an AVX-256 instruction can perform element-wise multiplication on two 256-bit registers, each containing eight, 32-bit *float32* values. DL acceleration libraries take advantage of this circuitry and implement specific DL tensor programs that replace inner loop nests with singular AVX instructions.

```
__global__ void matrix_multiply_kernel(int *a, int *b, int *c, int m, int n, int k){
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int sum = 0;
    if( col < k && row < m){
        for(int i = 0; i < n; i++){
            sum += a[row * n + i] * b[i * k + col];
        }
        c[row * k + col] = sum;
    }
}


int main(int argc, char const *argv[]){
    ...
    int *A_dev, *B_dev, *C_dev;
    cudaMalloc((void **) &A_dev, sizeof(int) * M * N);
    cudaMalloc((void **) &B_dev, sizeof(int) * N * K);
    cudaMalloc((void **) &C_dev, sizeof(int) * M * K);
    cudaMemcpy(A_dev, A, sizeof(int) * M * N, cudaMemcpyHostToDevice);
    cudaMemcpy(B_dev, B, sizeof(int) * N * K, cudaMemcpyHostToDevice);
    dim3 dimGrid((k + B_size - 1) / B_size, (m + B_size - 1) / B_size);
    dim3 dimBlock(B_size, B_size);
    // Kernel call
    matrix_multiply_kernel<<<dimGrid, dimBlock>>>(A_dev, B_dev, C_dev, M, N, K);
    cudaMemcpy(C, C_dev, sizeof(int) * M * K, cudaMemcpyDeviceToHost);
    cudaThreadSynchronize();
}
```

Listing 2.2: Simple matrix multiply implementation in CUDA

DL compilers also perform such mappings automatically for different tensor operator input shapes and processors via *vectorisation*, as described in Section 2.5.5. The GPUs SIMT paradigm also enables a single instruction to be performed across multiple threads (a 32/64-thread warp) (see Section 2.3.2.4, with the hierarchy of threads specified via the CUDA kernel definition [231]. The GPU internal scheduler analyses this thread hierarchy, and assigns threads to the processing units in accordance with internal scheduling policies. Unlike CPUs, GPUs allow threads within warps to simultaneously utilise shared scratchpad caches to perform load coalescing, which reduces memory access bottlenecks and accelerates computation.

2.3.3.3 Thread Hierarchies and Parallel Execution in Nvidia GPUs:

Nvidia CUDA [231] is a parallel programming model along with C++ language extensions that allow engineers to develop functions (kernels) that execute on the GPU and take advantage of their massively parallel computation [239]. Within CUDA, each kernel is a sub-program that executes N times in parallel across N threads. Listing 2.2 depicts implementation of matrix multiplication ($C = A \times B$) within CUDA. Threads in CUDA are grouped into one, two or three-dimensional *thread blocks* containing a maximum of 1024 threads, representing computation across less than or equal to 1024 elements of data. When scheduled, all threads within a single thread block execute across the same SM (see Section 2.3.2.4), grouped into a one, two or three-dimensional *grid*. Each thread block executes independently and its execution can occur in any order across any SM or core within the GPU, however, threads can share local memory.

Thus, an optimised thread hierarchy arrangement and kernel implementation can take advantage of memory locality and data reuse to account for global memory bottlenecks and improve performance. During kernel launch, the thread blocks are distributed across SMs according to compute resource availability, where one SM concurrently executes one or more thread blocks. When GPU performs scheduling, threads are re-grouped

into warps, where each warp executes a single instruction at a time to achieve higher instruction-level parallelism. As such, high utilisation occurs when there is as little data-dependent divergence in thread execution paths as possible, achievable via either manual optimisations or automatically using a DL compiler.

2.3.4 Deep Learning Model Life Cycle

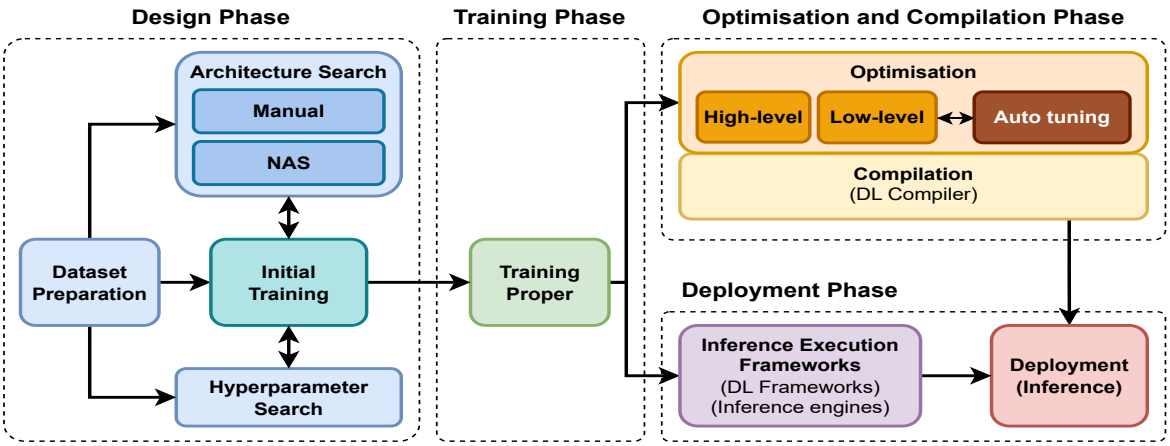


Figure 2.14: Life cycle of a DL model

DL models are computational artefacts with a complex life cycle, encompassing several phases as depicted in Figure 2.14. Early-phase decisions made by DL engineers, often have drastic impact on model performance and characteristics, and transitively on its footprint such as operational costs during the latter stages. For example, modifying the tensor data layout during the design phase can diametrically change model behaviour due to memory access pattern differences during deployment.

2.3.4.1 Design Phase

During the design phase, DL engineers iteratively produce a training pipeline that constructs a DL model from a dataset and a model architecture [88], evaluating multiple potential architectures to discover ones that are best suited towards the problem area

and ones that achieve highest initial accuracy. Likewise, the dataset is pre-processed and split into subsets (see. Section 2.1.1) to serve during model training. Typically a large number of architectures are evaluated either manually or by using Neural Architecture Search (NAS) techniques [392, 72] that automate this process, each time partially training the model to determine performance. Such experimental training is also used to determine optimal training hyperparameters such as the *learning rate* [24]. With a large number of experimental training sessions, the design phase is often resource-costly.

2.3.4.2 Training Phase

Once an architecture and optimal hyperparameters are established, the training phase can commence. During the training phase, the model is trained until it achieves satisfactory accuracy [97]. Training is facilitated by DL frameworks that leverage CPUs or other processors for computation. Initially trained on CPUs or singular GPUs [314], contemporary training pipelines utilise multiple GPUs [263, 167, 168] to train increasingly complex models that require extensive computation to achieve competitive inference performance. The growing model complexity and scale, continually translates into demand for increasingly complex training pipelines (for example, distributed training [63, 36]), as well as motivates development of more powerful processors and compute clusters to contain large models [69]. The increasing number of more powerful processors and growing amount of time necessary to train DL models results in large DL operational costs (time and energy) [318], further exacerbated by frequent model re-training towards new applications or datasets [244, 391].

2.3.4.3 Optimisation and Compilation Phase

Trained models are deployed in various environments such as the Cloud or Edge to perform inference. Whilst DL frameworks offer several deployment options (see Sections 2.3.2.3 and 2.3.2.6), they are limited to specific tensor program implementations

optimised for a select class of devices. As outlined in Section 2.3.2.7, DL compilers provide a solution to the problem of efficient, high-performance compilation of trained DL models towards various hardware platforms, including facilitation of a plethora of optimisations during this process [184]. DL model implementations can be specialised towards a target-device (for example, a GPU) via a lengthy process of auto-tuning, which involves repeated testing of thousands of potential tensor programs that can implement the model towards the device. Since each test requires full target-device isolation, the process greatly under-utilises the device and the host platform both within a given time instant and across time, resulting in high energy-disproportionality. The optimisation and compilation life cycle stage is the focus of this thesis. More specifically, the thesis investigates the efficacy of various DL model optimisation methods and explores the costs associated with performing them, focusing primarily on auto-tuning.

2.3.4.4 Deployment Phase

Once a model has been trained (and optimised), it is deployed for inference using DL frameworks, serving platforms [327, 85] and IEs, or as part of an application such as a mobile app, leveraging DL compilers for code generation. Compared to the other phases, performing inference using deployed models is the most expensive phase of the DL model life cycle. This is because once deployed, DL models serve inference requests for an unpredictable period of time, at unpredictable rates, consuming energy within every environment they are deployed [65]. For example, Facebook reports that models deployed within their data-centre scale compute clusters serve trillions of inference requests every day [111], utilising high-performance processors [248, 360]. Precisely estimating complete operational cost footprints of DL models is therefore extremely challenging, with many works relying on metrics and observations instead. From a DL engineer point of view, a feasible method to reduce operational costs of deployed DL models is to optimise their runtime performance, such that the value they provide is energy-proportional and

adequate to the inference task at hand [291, 310]. This thesis focuses on analysing and improving DL model optimisation strategies, with primary objective of improving the understanding of the costs associated with DL model optimisation and compilation, in the hopes of lowering the barrier to entry for DL inference optimisation.

2.3.5 Machine Learning as a Service (MLaaS)

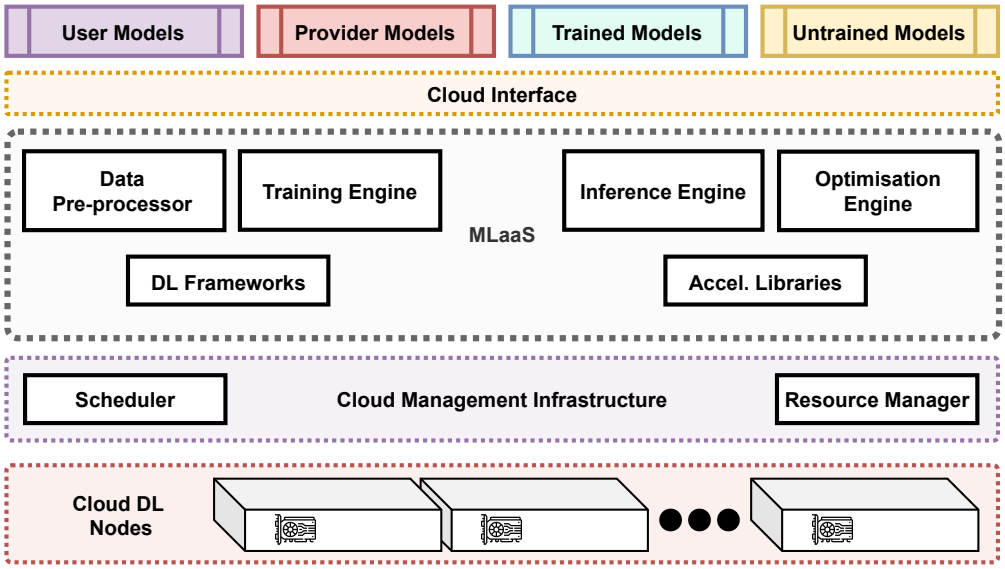


Figure 2.15: Machine Learning as a Service (MLaaS) system architecture

Underpinned by complex ecosystems of software and hardware, ML and DL applications can be challenging to deploy and operate for organisations without prior DL engineering experience and necessary compute infrastructures know-how in place. Machine Learning as a Service (MLaaS) is a Cloud computing paradigm that facilitates development, deployment and provisioning of ML and DL applications [273, 253]. Figure 2.15 depicts a high-level MLaaS system architecture. Within the MLaaS paradigm, users rely upon remotely-accessible compute infrastructures and middleware such as training or inference engines, made available by the Cloud providers. Such MLaaS platforms also provide DL engineers with a variety of tools such as web interfaces for

specifying, deploying and monitoring DL workloads. Many MLaaS platforms facilitate Cloud-based data and model management services, including data collection, verification, pre-processing, as well as model validation, testing and versioning. The MLaaS paradigm underpins Cloud service models such as Data Science as a Service (DSaaS) [256], that focus on providing higher-level abstractions over conventional ML applications such as statistical models, clustering methods or ANN-based models. Large Cloud providers such as Microsoft (Azure Machine Learning [54]), Google (Cloud Platform [98]) or Amazon (AWS & SageMaker [298, 296]) are increasingly adopting the MLaaS service model to facilitate ML and DL services to their users.

2.4 Characterising Deep Learning Systems

There are many factors that affect DL model performance and computational costs, including time and energy consumption. To better understand these factors, it is important to outline and cross-associate the relevant metrics that are used both in industry and academia to characterise DL models and hardware during inference.

2.4.1 Characterising Models

Accuracy, Error, Recall, Precision and F1 Score measure inference performance of DL models [326]. More specifically, accuracy is a ratio of correct answers to all answers provided by the model within some number of trials, as depicted in Equation 2.21, where T_{pos} is the number of true-positives, F_{pos} is the number of false-positives, T_{neg} is the number of false-positives and F_{neg} is the number of false-negatives. Directly from the accuracy metric, the prediction error can be derived (Equation 2.22). Recall, determines what proportion of true-positive predictions were identified correctly, as shown in Equation 2.23. For example, recall of 0.75 indicates that 75% of some N predictions were true-positives. The precision metric provides a ratio of true-positives to

all correct predictions, especially important when inference results are used in use-cases such as self-driving vehicles. Equation 2.24 depicts the precision metric. Achieving both high precision and high recall is challenging. F1 score (depicted in Equation 2.25) captures both of these characteristics, measuring efficacy of the model as a single metric.

$$Acc = \frac{T_{pos} + F_{neg}}{T_{pos} + T_{neg} + F_{pos} + F_{neg}} \tag{2.21}$$

$$Err = 1 - Acc \tag{2.22}$$

$$Recall = \frac{T_{pos}}{T_{pos} + F_{neg}} \tag{2.23}$$

$$Precision = \frac{T_{pos}}{T_{pos} + F_{pos}} \tag{2.24}$$

$$F1_{score} = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{2.25}$$

Achieving high accuracy, precision and recall requires both effective training methods and a large-enough model composed of complex operators to sufficiently capture the training dataset features, also generalising to unseen data. As such, DL models are often characterised by their architectural features.

Number of and size of Layers and Operators: As described in Section 2.2.2, DL models are composed of multiple layers, each implemented by one or more tensor operators such as Convolution or BN [97]. Conventionally, only complex layers such as Convolution or FC are counted towards the model *depth* measure. When considering the complexity of the model, individual layers are also characterised as *wide* or *thin* by observing the shapes of their input tensors, as outlined in Section 2.3.1.

Number of trainable parameters and output activations: Memory required to execute a DL model is dependent on the number of parameters and intermediary activations used and produced by each layer [323]. During training, both parameters and activations are usually represented as Floating-point (FP) values (typically *float32*) and as such, each require four bytes of memory. The number of parameters and activations is

related to the shape of layer inputs, outputs, kernels (for example, Convolution filters), and settings such as the stride or padding. The following equations demonstrate how to calculate the number of parameters and activations for a 2D Convolution:

$$Num_{params} = (C * K^2 * N) + B \tag{2.26}$$

$$Out_H = \frac{(H + 2 * P) - K + 1}{S} \tag{2.27}$$

$$Out_W = \frac{(W + 2 * P) - K + 1}{S} \tag{2.28}$$

$$Out_C = N \tag{2.29}$$

$$Act_{shape} = \{Out_H, Out_W, Out_C\} \tag{2.30}$$

$$Num_{activations} = Out_H \times Out_W \times Out_C \tag{2.31}$$

where H, W and C are the height, width and channels of the input tensor, N is the number of kernels, K is the kernel height/width, B is the number of biases (usually $B = N$), performed with stride S and padding P. As such, a Convolution layer containing five filters of shape 10×10 and an input with size $224 \times 224 \times 3$, with $S = 1$ and $P = 0$, would contain 1505 trainable parameters, and output an activation tensor of shape $215 \times 215 \times 5$ containing 231,125 activations. In FC layers, weights represent connections of every neuron to every other neuron, with the total number of parameters: $N \times M$, where N is the input size and M is the output size. Characterising an entire network, the AlexNet model (see architecture details in Appendix D.1) contains 61.1m parameters across all its Convolution and FC layers.

Required number of MACs/MADs/FLOPs: DL model computational complexity can be characterised using the measure of MAC or Multiply Add (MAD) operations necessary to perform the model's tensor operator computations [275]. MACs frequently describe model complexity because it is common for DL tensor operators to use multiplier-accumulator (MAC) processing units found in CPUs and GPUs, that

perform $a \leftarrow a + (b \times c)$ operation as a single-cycle instruction (Fused Multiply-Add (FMA) or Fused Multiply-Accumulate (FMAC)). Dot-product or GEMM operations can be easily implemented as a set of FMA operations, improving throughput compared to multiplication and accumulation performed as consecutive instructions. Simultaneous FMA operations are achieved using multiple MAC units, with modern CPUs relying on SIMD AVX instructions [48, 53] and GPUs relying on tensor cores [237].

At a fundamental level, each MAC represents two logical Floating-point Operations (FLOPs) - multiplication and accumulation (addition) [275, 323]. FLOPs and MACs are often confused when reporting model complexity, since modern hardware implements MACs as one-shot operations completing within a single clock cycle. As such, when representing complexity, a single MAC is often counted as a singular FLOP. The total required number of MAC operations for a given tensor operator (for example, 2D Convolution) in the forward pass of the network [275], can be estimated as:

$$Num_{MAC} = K^2 * In_C * Num_{activations} + Num_{activations} \tag{2.32}$$

where In_C is the number of input channels, K is the kernel size, and $Num_{activations}$ is calculated as per Equation 2.30, with the additional $Num_{activations}$ accounting for the bias term. For example, a $224 \times 224 \times 3$ 2D Convolution with five, 10×10 kernels, stride = 1, padding = 0 and included bias term, requires 69.57m MACs. Likewise, the AlexNet model with 61.1m parameters, requires 715.56m MACs.

2.4.2 Characterising Hardware and its Performance

Number of cores and core capabilities: CPUs consist of several (1 - 64) complex cores capable of executing one or two threads (via SMT), where each thread executes one or more instructions within a clock cycle [275]. A DL model could be parallelised on a CPU by performing simultaneous MAC operations across the available threads.

Contrastingly, GPUs contain a large number of more primitive cores, such as Integer (INT)/FP32/64, performing one or more 32/64-bit integer and floating-point arithmetic operations on every clock cycle [275, 106]. For example, the Nvidia Ampere architecture [229] contains 6912 FP32 CUDA cores, grouped into SMs, where each SM concurrently executes up to 2048 threads scheduled as warps, as described in Section 2.3.2, which enables massively-parallel computation. To leverage these capabilities, DL tensor operators should off-load heavy FP arithmetic to the GPU, and tensor programs should be designed such that operand shapes and sizes match appropriately to the available scheduling primitives (threads, warps) and memory hierarchies. However, both CPUs and GPUs are limited in terms of achievable speedup due to parallelism, as described by Amdahl's law [12] (Equation 2.33) for N independent compute units in parallel, and P designating a portion of the program that can be parallelised.

$$MaxSpeedup = \frac{1}{1 - P + (P/N)} \tag{2.33}$$

Clock frequency: CPUs and GPUs execute instructions dictated by the clock [106, 275]. The clock frequency (F) determines how fast the processor loads instructions, fills up execution pipelines and executes instruction computations. Typical CPU has a clock frequency ranging between 2 and 5GHz, whilst GPUs operate at clock speeds under 2GHz. High clock frequency and number of cores deliver high computation throughput at the expense of higher energy consumption, as high F increases circuit power.

Instructions and Peak OPS: TIP can be characterised in terms of their instruction processing speed. Given a single-core scalar processor with cycle period C_T, execution latency L_{exec} of a program can be estimated as: $L_{exec} = C_T \times \sum_i^N CPI_i$ [106, 323], where CPI_i is the number of cycles required per instruction $i \in I$ and N are the instructions of the program. Other measures such as Instructions per Cycle (IPC) also exist and are widely used. Alternatively, the processor's performance can be characterised by the

maximum number of operations each core can perform within a period of time. This measure is often referred to as peak OPS or peak operation throughput, as shown in Equation 2.34, where CPO_o stands for Cycles per Operation for operation o, while Equation 2.35 depicts device-wide peak throughput estimation, where $Core_{count}$ is the number of cores per device and $Core_{util}$ is the aggregate utilisation of all device cores.

$$Core_{PeakThroughput} = (\frac{1}{CPO_o} \times F) \tag{2.34}$$

$$OPS_{peak} = Core_{PeakThroughput} \times Core_{count} \times Core_{util} \tag{2.35}$$

Newest Intel Xeon CPUs are capable of up to 168 Giga Floating-point Operations (GFLOPs) and 293 integer Giga Operations (GOPS) [52], whereas Nvidia Ampere A100 GPU delivers 19.5(FP64), 156(FP32), 624(INT8) Tera Floating-point Operations (TFLOPs)/Tera Operations (TOPS) of peak throughput [229].

Memory Hierarchy: Ahead of instruction execution, CPUs and GPUs must fetch data from memory into local registers, which can take more than 100 times the time it takes to complete an FP32 arithmetic multiplication [107]. For example, Nvidia A100 achieves 156 TFLOPs OPS_{peak} for FP32 operations and has a maximum memory throughput of 2,039GB/s, and for it to achieve full compute utilisation, the A100 would have to perform 77 operations for every byte fetched. As such, CPUs and GPUs employ multi-level caches to leverage data locality and store frequently accessed data closer to the compute unit ahead of computation [323]. Memory can be characterised by its capacity and peak bandwidth (Bw^{peak}), calculated as follows:

$$Bw^{peak} = F \times N_{iface} \times TPC \times W_{bus} \tag{2.36}$$

where F is the memory frequency (400MHz for Double Data Rate (DDR)4 DRAM), N_{iface} is the number of memory interfaces, TPC is the number of transfers per clock-

cycle and W_{bus} is the bus width in bits (for example, 64 for DDR). CPUs rely upon external DDR memory whilst GPUs utilise built-in Graphics Double Data Rate (GDDR) or High Bandwidth Memory (HBM) with large bus width (1024-bit) and larger number of channels (for example, in Nvidia A100 80GB HBM2 [229]). CPUs and GPUs use caches to avoid memory bottlenecks, with CPUs employing three levels of cache and GPUs two levels, where level one cache has the highest speed, lowest capacity and is the most expensive to produce. Some GPUs enable modifying low-level cache contents directly by configuring them as scratchpads (for example, shared memory in Nvidia GPUs), which is sometimes leveraged within DL compiler optimisations [80, 262].

TDP and Energy Consumption: To execute instructions, CPUs and GPUs switch transistor state between high and low, and set/unset bits in registers and cache [106]. Transistor switching dissipates electrical energy as heat as their capacitance is charged/discharged, known as dynamic power dissipation and estimated as follows:

$$P_{dyn} = \alpha \times C \times V^2 \times F \tag{2.37}$$

where F is the chip's clock frequency, V is the operating voltage, C is the total capacitance of all transistors and α is the activity factor which describes the active portion of transistors within a time period (for example, one second). A portion of energy is also dissipated via static power dissipation P_{static}, due to slow current leakage when transistors are in their off state, derived as: $P_{static} = I_{leak} \times V$ where I_{leak} is the amount of leakage in amperes. In total, a chip dissipates $P_{total} = P_{dyn} + P_{static}$ of instantaneous power, where energy consumed within some time T is $E = P_{total} \times T$. P_{total} is often inflated by another 10 - 15% due to brief but unavoidable short-circuit periods. Commercially, chips are characterised by their maximum instantaneous power draw, also known as Thermal Design Power (TDP). Server CPUs have TDP of 150 - 250W, whereas GPUs such as Nvidia H100 peak at 800W TDP.

Given perfect knowledge of silicon behaviour, the energy consumption (measured in Joules: O_{joules}) of individual operations (for example, FP multiplication), could be approximated by applying Equation 2.37, replacing α and C with values appropriate for the silicon implementing the operation [324]. Derived from O_{joules}, one could also estimate the number of operations that can be performed for one Joule of energy - $J_{OPS} = \frac{OPS}{Joule}$. Horowitz [118] details that on a 45nm chip, FP32 multiplication consumes 37 times more energy than INT32 addition and that DRAM accesses can be up to 173 times more energy expensive than arithmetic operations.

However, such modelling is often infeasible in DL systems due to ever-changing, black-box designs of chips, lack of precise measurement tools, and the complexity of program analysis for different platforms, DL models and their tensor programs. Instead, energy analysis more often relies on system-wide measurements or estimations, reported as a function of peak performance - for example, TOPS/Watt, deemed adequate to compare effects of different DL model execution patterns [60]. Modern Intel and AMD CPUs include current sensors for energy consumption estimation and expose a Running Average Power Limit (RAPL) Model-Specific Register (MSR) that can be queried to obtain energy data [61]. Likewise, Nvidia GPUs report coarsely sampled (0.5 - 1s), package-wide power usage via the Nvidia Management Library (NVML) library [234].

2.4.3 Characterising Model Execution and Efficiency

Several metrics, which encompass both workloads and hardware, are important to consider when deploying DL models across different systems for inference, namely:

Inference Latency measures the time between an input is supplied to the model and the time when the model's output is generated, when executing on a particular device. Latency encapsulates data loading, individual tensor program launch procedures, data transfers from host DRAM memory to GPU DRAM, execution of GPU kernels and transfer of results to the host. Given perfect knowledge of all operations conducted

as part of model's execution, inference latency ($ILat$) could be estimated as follows:

$$ILat = max(\frac{P_{OPS}}{OPS_{peak}}, \frac{P_{bytes}}{Bw^{peak}}) \tag{2.38}$$

where P_{OPS} is the total number of operations and P_{bytes} is the total number of memory accesses in bytes performed during inference. Due to hardware and software complexity, latency is usually measured using high-precision hardware clocks. CPU-level measurements utilise instructions that retrieve values from CPU Real Time Clock, while GPU timings (for example, of kernel latency) are retrieved via Application Programming Interfaces (APIs) such as Nvidia Activity API [234] directly from the device driver.

Inference Throughput measures the number of model inferences performed within a time period. Peak inference throughput ($IThr$) can be estimated as follows:

$$IThr = OPS_{peak} \times \frac{1}{O_{inf}} \tag{2.39}$$

where O_{inf} is the total number of operations associated with inference, or by observing the number of performed inferences in a time period.

Memory vs. Compute: Proposed by Williams et al. [355], the Roofline model calculates the maximum theoretical performance a program can achieve on a particular processor, and has previously been used to evaluate DL inference performance [59]. The model includes peak operation throughput (OPS_{peak}), peak memory bandwidth (B_{peak}) and the program's Computational Intensity (CI), which is a ratio of the program's operations to the number of memory bytes accessed. High CI indicates a compute-bound program and low CI a memory-bound program, where increasing processor compute would not result in execution speedup. For a matrix multiply operator with input $Inp_{a \times b}$,

weights $W_{b\times c}$ and output $Out_{a\times c}$, where all values are four-byte floats, the CI is:

$$N_{ops} = 2^{(add/mul)} \times a \times b \times c \tag{2.40}$$

$$N_{bytes} = 4^{(float32)} \times ((2^{rd/wr} \times a \times c) + (1^{rd} \times a \times b) + (1^{rd} \times b \times c)) \tag{2.41}$$

$$CI = \frac{N_{ops}}{N_{bytes}} \tag{2.42}$$

When no caching is performed, the processor must fetch and store values to DRAM during each multiply-add operation, which would result in substantial slowdown since DRAM reads/writes cost around 200 cycles each. Caches circumvent this bottleneck, enabling data locality and reuse close to compute (L1 cache read is around four cycles). The Roofline model determines CI and maximum attainable OPS for a given program and processor, which guides the design of DL tensor programs and optimisations.

Energy Efficiency of DL inference can be measured in $\frac{Joules}{Inference}$ ($JInf$) or $\frac{Inferences}{Joule}$ ($InfJ$) capturing the costs of delivering predictions. $JInf$ and $InfJ$ can be estimated by observing the associated compute and memory access operations as follows:

$$JInf = O_{joules} \times O_{inf} \tag{2.43}$$

$$InfJ = J_{OPS} \times \frac{1}{O_{inf}} \tag{2.44}$$

Development of energy-efficient DL models is challenging. Different DL operations exhibit different energy consumption patterns, influenced by memory and cache access characteristics and the program's ability to utilise compute resources effectively (for example, via parallelism). Energy-efficiency of DL is important to consider due to operational costs, especially within large-scale DL inference serving deployments such as those at Facebook, where DL inference workloads run *"tens-of-trillions of times per day"* [111] and result in high power dissipation [360, 248, 104]. As such, it is important to explore optimisation strategies for improving DL inference performance and efficiency.

2.5 Deep Learning Compilers

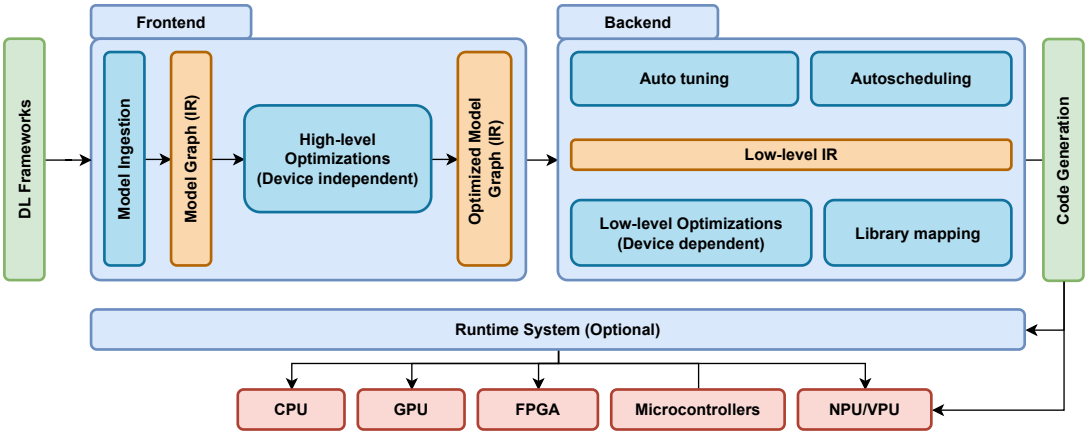


Figure 2.16: Deep Learning Compiler Design

There exists a great diversity of DL models that leverage Edge devices as well as Cloud servers equipped in powerful CPUs and GPUs. Such diversity extends to individual deployment scenarios, where executing DL models on different types of Mobile/Cloud CPUs or GPUs (Intel, AMD, Nvidia, ARM, Google), requires bespoke implementations to achieve maximum performance. Manually adapting a high-level model definition specified in a DL framework towards the various processors, requires systems expertise and large engineering effort. This issue becomes compounded by the ever-growing number of increasingly more complex DL model tensor operators designed as part of novel model architectures. Each such operator requires additional engineering and optimisation to enable deployment on different processors.

As a result DL compilers have gained prominence in the recent years. The core purpose of a DL compiler is to transform a high-level DL model computation specification (for example, ones produced by DL frameworks), into a set of outputs necessary for a DL model to be executed on a given device. Such outputs involve a set of routines for computation of model operators, device-host memory management and APIs to

Table 2.1: Characterisation and support matrix of popular deep learning compilers
✓= Supported, × = Unsupported, ? = Limited support / support unclear

		TVM [39]	PlaidML [255]	nGraph [58]	Halide [262]	MLIR [173]	TF XLA [176]	Glow [278]	TC [345]	Tiramisu [18]	TACO [159]	Hummingbird [218]	Rammer [198]	Roller [390]	Astra [307]	Apollo [382]	FusionStitch. [388]	AStitch [387]
Supported Languages	Python	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	C++	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	✓	✓	×	✓	✓	✓
	Java	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
	Go	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
	Rust	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
	WebAssembly	✓	×	×	✓	×	×	×	×	×	×	×	×	×	×	×	×	×
Supported Frameworks	TF/Keras [331]	✓	✓	✓	?	✓	✓	×	×	×	×	×	✓	✓	✓	×	✓	✓
	Pytorch [1]	✓	?	?	?	✓	✓	✓	✓	✓	×	✓	✓	✓	✓	×	×	×
	MXNet [79]	✓	✓	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	×
	ONNX [82]	✓	✓	✓	×	✓	?	✓	×	×	×	✓	✓	✓	×	×	×	×
	Caffe(2) [147]	✓	×	×	×	×	×	×	✓	×	×	×	×	×	×	×	×	×
	CoreML [133]	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
	DarkNet [267]	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
	OneFlow [372]	✓	×	×	×	✓	×	×	×	×	×	×	×	×	×	×	×	×
	PaddlePaddle [243]	✓	✓	✓	×	×	×	×	×	×	×	×	×	×	×	×	×	×
	Mindspore [127]	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	×	×
Supported Libraries	oneMKL [47]	✓	✓	✓	×	×	✓	×	×	×	×	×	×	×	×	×	×	×
	oneDNN [51]	✓	✓	✓	×	×	✓	×	×	×	×	×	×	×	×	×	×	×
	cuDNN [232]	✓	×	×	×	×	✓	×	✓	×	×	×	×	×	✓	✓	✓	✓
	cuBLAS [230]	✓	×	×	×	×	×	×	✓	?	×	×	×	×	×	×	×	×
	MIOpen [156]	✓	×	×	×	×	×	?	×	×	×	×	×	×	×	×	×	×
	rocBlas [6]	✓	×	×	×	×	×	?	×	×	×	×	×	×	×	×	×	×
Supported Target Devices	AMD GPU	✓	✓	✓	✓	✓	✓	✓	×	×	×	?	✓	✓	?	×	×	×
	Nvidia GPU	✓	✓	✓	✓	✓	✓	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	ARM GPU	✓	?	?	?	✓	?	×	×	×	×	×	✓	✓	×	×	×	×
	Qualcomm GPU	✓	×	×	?	×	×	×	×	×	×	×	×	×	×	×	×	×
	ARM CPU	✓	?	?	✓	✓	✓	×	×	×	×	?	×	×	×	✓	×	×
	x86 / amd64 CPU	✓	✓	✓	✓	✓	✓	✓	×	✓	✓	✓	✓	✓	×	×	×	×
	RISC-V CPU	✓	×	×	✓	✓	×	×	×	×	×	×	×	×	×	×	×	×
	FPGA	✓	×	×	×	✓	×	×	×	✓	×	×	×	×	×	×	×	×
	TPU	✓	×	×	×	×	✓	×	×	×	×	×	×	×	×	×	×	×
	NPU/VPU/IPU	×	✓	✓	×	×	×	✓	×	×	×	×	✓	✓	×	✓	×	×
Supported Optimisations	Graph-level	✓	✓	✓	×	✓	✓	✓	?	×	×	?	✓	✓	✓	✓	✓	✓
	Low-level	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	?	✓	✓	✓	✓	✓	✓
	Auto-tuning	✓	×	×	✓	×	×	×	✓	✓	×	×	✓	✓	×	✓	✓	✓

interact with the application that leverages the model. Another important goal of a DL compiler is to optimise the model implementation towards a given target device. DL compilers utilise multiple levels of Intermediate Representation (IR) to represent the model graph, operations and data, and apply various optimisations whilst translating the model definitions to their executable form.

DL compilers differ from traditional compilers that accept program source code and output binaries that execute on the *target device*. Instead, they act as an intermediary compiler that focuses on optimising the high-level model definitions specified in DL frameworks, and transforming them into a set of low-level IR representations of optimised DL *tensor programs* - a process referred to as *lowering*. Once the lowering is complete, the DL compiler utilises conventional compilers such as LLVM [172] for code generation-proper. As shown in Figure 2.16, akin to traditional compilers, DL compilers include a *frontend* component, a single/multi-level IR and a *backend* component. The DL compiler backend is responsible for producing low-level IR, compatible with more generic code generators such as LLVM [172], or device-specific compilers such as Nvidia NVCC [225] targeted towards Nvidia GPUs, which then compile the optimised model towards the target-device. Some DL compilers generate executable binaries or other specialised accelerator instructions directly, for example, to support esoteric FPGA or ASIC architectures - see the VTA module in TVM [213].

2.5.1 Existing Deep Learning Compilers

Recently, several DL compilers have been introduced, proposing different methods for lowering DL models to high-performance tensor programs that execute on variety of devices. Table 2.1 outlines differences in support for target-devices, DL frameworks, libraries and optimisation approaches across recently proposed DL compilers.

TensorComprehensions (TC) [345] is a DL compiler with a polyhedral tensor operator expression language, used for operator lowering towards CUDA kernels, whilst performing

low-level optimisations and program compilation. Tiramisu [18] combines a polyhedral approach with scheduling (see Section 2.5.4.1) within its four-level IR, to generate high-performance image processing pipelines and DL model tensor programs.

TACO [159] is a tensor algebra compiler focused on efficiently generating low-latency tensor programs that accept both dense and sparse tensor inputs. Rammer [198] targets both spatial and temporal low-level characteristics of accelerators such as GPUs and Image/Intelligence Processing Units (IPUs) by providing an abstraction layer over parallel computation, producing and then pruning (based on predicted performance) a large number of tensor programs for each tensor operator. Roller [390] also abstracts spatio-temporal scheduling for parallel tensor algebra computation by introducing tiling and a micro-performance cost model to evaluate candidate tensor programs.

Glow [278] lowers DNN graphs into two-phase IR to perform memory allocation, quantisation and instruction scheduling optimisations towards CPUs and GPUs, leveraging hardware-specific features. Hummingbird [218] unifies expressing both DL and conventional ML operations as tensor computations, enabling existing DL libraries and frameworks to support high-performance execution of ML primitives on various target-devices. Triton [334] proposes an LLVM-based IR for representing tensor operation primitives, as tiled (multi-dimensional sub-array) computations, accelerating them within environments where acceleration libraries cannot be used.

Intel nGraph [58] is an IR, DL compiler and an inference engine that supports several third-party DL frameworks, and most notably, one of the first DL compilers to support ASIC-based Neural Network Processors (NNPs) when generating optimised tensor programs. nGraph has undergone cross-integration with other DL compilers such as PlaidML [255], which was initially focused on tensor program optimisation towards resource-constrained Edge devices. Recently, MLIR [173] and openVINO [49] integrated PlaidML and nGraph within their compilation ecosystems as sub-modules.

Astra [307] is a DL compiler and a DL training engine that performs multi-version

compilation whilst the training is underway. Astra replaces logically-equivalent tensor programs during model execution to establish high-performance operator combinations, whilst allowing the training to continue unaffected. TensorFlow XLA [176] is a compilation module closely integrated with the TensorFlow DL framework, which performs both high-level graph optimisations and selection of low-level tensor program implementations for CPU and GPU backends, leveraging the LLVM code generator.

Apollo [382] compiles DNNs just-in-time, whilst enabling computation scheduling optimisations such as tiling to influence high-level optimisation decisions (for example, operator fusion) made at the model graph-level. FusionStitching [388] and AStitch [387] optimise memory intensive tensor operators via fusion into larger operators and perform fine tuning using predictive cost models to improve program performance.

Originally developed as a compiler for image processing pipelines[8], Halide [262] aims to increase flexibility of specifying tensor operations and implementing fast tensor programs. Halide was one of the first tensor algebra compilers to decouple computation specification (for example, tensor expressions) from their schedules (implementations), enabling broad variety of tensor programs to be compiled towards different target-devices.

The Versatile DL Compiler: Inspired by the decoupled compute and schedule and schedule transformations in Halide, TVM [39] optimises and compiles a wide array of DL operations and end-to-end models towards a various target-devices such as CPUs, through GPUs and FPGAs. The core advantage of TVM over the other discussed DL compilers are its rich integration APIs for DL frameworks, DL acceleration libraries, several supported programming languages and external code generators such as LLVM. TVM generates DL tensor programs for different processors, while performing high and low-level optimisations via graph and schedule transformations and auto-tuning. This thesis leverages TVM as the core DL compiler during experimentation, due to its prominence within the field of DL performance optimisation, versatility and third-party

[8]Halide was later adapted to support DL tensor algebraic computation

project support. TVM is a foundational project for many prominent DL auto-tuners and is increasingly adopted within large-scale DL deployment pipelines.

2.5.2 Deep Learning Compiler Frontend

DL compilers ingest DL model definitions from DL frameworks via their frontend component[9]. The DL compiler frontend transforms model definitions into a computational graph representation, also known as the High-Level Intermediate Representation (HLIR) or graph IR. Depending on compiler design, the HLIR is part of the same multi-level IR as the backend, or a separate single-level IR enabling only high-level transformations. As each DL framework specifies the DL model using a different format and/or programming language, achieving compatibility with the compiler frontend is challenging. Many DL compilers have limited support for frameworks (for example, Tensorflow XLA [176]), or focus on supporting common model specifications (for example, ONNX [82]). As shown in Table 2.1, only several DL compilers (TVM [39], Intel nGraph + PlaidML [255, 58] support multiple languages and frameworks.

2.5.2.1 High-level Intermediate Representation (HLIR)

The DL compiler HLIR represents the DL model tensor operators and data dependencies between them, enabling device-independent graph-level optimisations. Existing DL compilers implement several types of HLIRs.

Graph-based HLIRs organise computation as a set of vertices and edges to form a DAG, encoding DL operators such as Convolution as the vertices and their operands and outputs (tensors) as the edges, facilitating graph-based optimisations. Many DL frameworks utilise this format to represent models, and as such, Graph-based HLIRs are commonly leveraged in DL compilers. As shown in Figure 2.17, the Graph-based HLIR

[9]In DL compiler terminology, "frontend" is not related to web-development

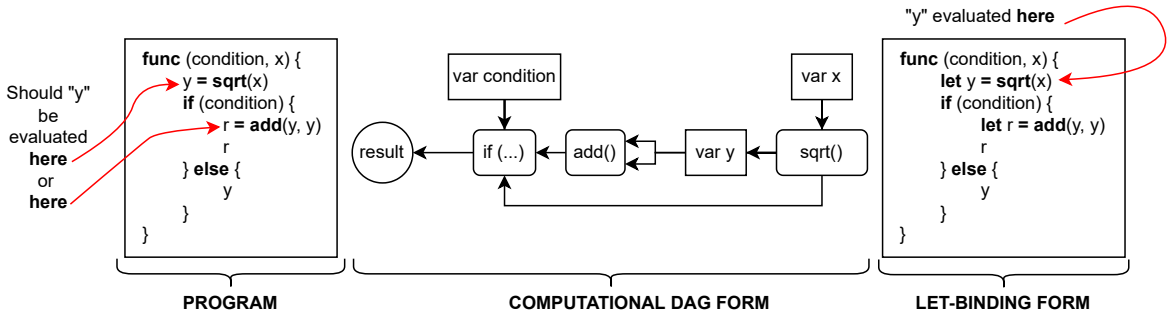


Figure 2.17: Graph-based vs. Let-binding HLIR

does not specify the location and scope of each expression, which can be disadvantageous in later stages of compilation where certainty about computation scope is necessary.

Let-binding HLIRs improve upon this semantic ambiguity of location and order of operations, being a more strict computation representation, where each expression is bound to a variable with a *"let"* keyword pointing to the variable and the operation. Variables can then be indentified within a variable map as the program is analysed. Conversely, in a Graph-based HLIR, the compiler must perform recursive descent to evaluate each node in the DAG by first evaluating all children nodes.

Other HLIRs have also been proposed. The Glow [278] and Tensorflow XLA [176] compilers adopt a functional HLIR, whereas TVM adopts Relay [276], a custom HLIR that combines both Graph and Let-binding specifications to enable a wider array of optimisations at the expense of specification complexity.

2.5.3 High-level Optimisations

The DL compiler HLIR facilitates application of multiple high-level optimisations to the ingested DL model, with majority being device-independent. Such optimisations involve traversal of the computation graph / let-node bindings to rewrite them more optimally, leveraging input / output tensor shape and size information. Following are the common high-level optimisations applied in many recent DLs compilers:

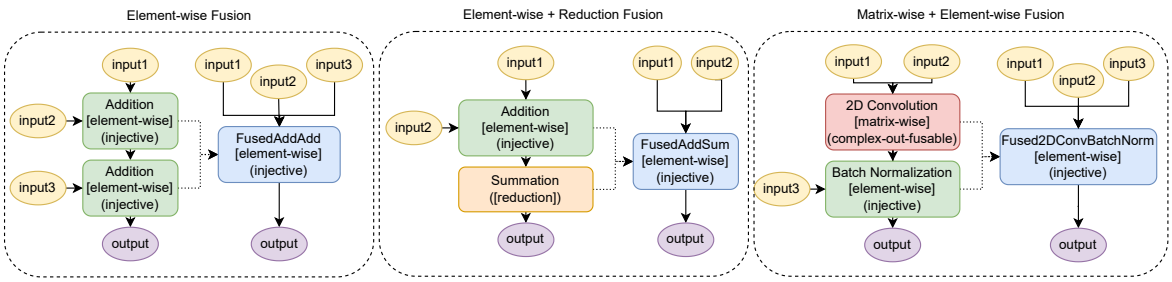


Figure 2.18: Different rules of Operator Fusion

Operator Fusion: An un-optimised DL model graph contains operators as defined by the DL engineer during model construction. Forward pass of such a model, involves execution of sometimes hundreds of individual operators, including their launch procedures and memory management [22, 248]. During operator fusion, two or more compatible operators are fused into a single routine, combining intermediate memory and launch procedures, and thus reducing overheads [220, 184]. Fusion is especially important for GPU execution as the memory allocation/access is fused within the scope of a single kernel, enabling greater reuse of fast registers or scratchpads, and avoids costly data transfers between operator launches [361]. Element-wise operators (for example, ReLU) can also be fused with complex Convolutions, enabling data reuse whilst those still reside within the scratchpads or cache. Operator fusion is possible only for certain patterns of consecutive operators, and is guided by rules that include: (1) fusing element-wise operators (*injective* in TVM) together (for example, two additions), (2) fusing an element-wise operator such as addition with a reduction operator such as summation and (3) fusing a matrix-wise operator such as Convolution with an element-wise operator (*complex-out-fusable* + *injective* fusion in TVM), as shown in Figure 2.18. These rules permit fusion of operators such as Convolution + Bias + Activation function or BN + Activation function, which are common patterns in CNNs.

Algebraic Identification & Strength Reduction optimisations simplify DL model graphs. For example, the distributivity characteristics of GEMM, enable matrix

transposes to be removed when multiplying two matrices, while consecutive transposes can be eliminated via the involution property: $(A^T)^T = A$. Reducing mathematical complexity of the model graph reduces the computational complexity of the compiled model [287, 45]. The following equations depict common algebraic optimisations:

$$a \times c + b \times c \Rightarrow (a + b) \times c \tag{2.45}$$

$$a - a \Rightarrow 0 \tag{2.46}$$

$$(X^T Y^T)^T \Rightarrow XY \tag{2.47}$$

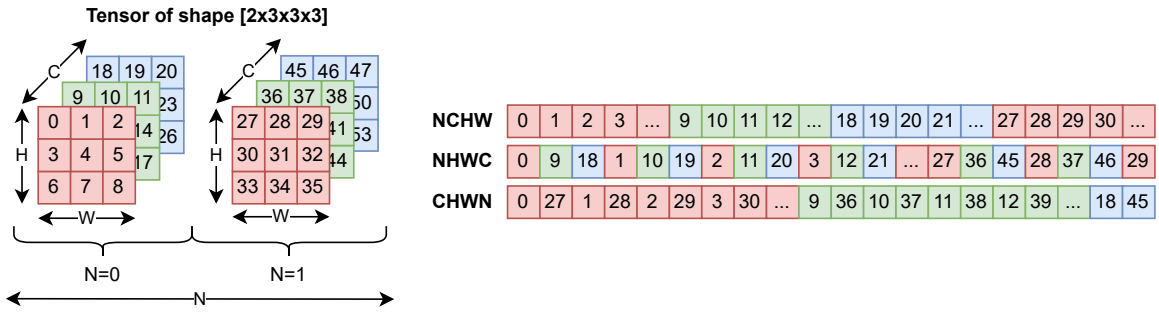


Figure 2.19: Different types of tensor data layouts given the same tensor

Layout Transformations: Patterns of memory access in DL operators heavily influence load/store latency of data from memory via multi-level caches. Given the same operator (for example, Convolution), data access time differs depending on its operand tensor layout in memory. Some DL tensor operator implementations within libraries such as cuDNN or oneDNN, may require data layouts to match certain specifications. As such, DL compilers apply layout transformations to ensure library compatibility during later-stage compilation at the backend. During layout transformation, the compiler modifies the shape information attached to each tensor so that during low-level optimisations, these transformations on data can be applied. Figure 2.19 presents different data layouts for the same $3 \times 3 \times 3$ tensor. The Batch-size(N) - (C)hannels - (H)eight - (W)idth

(NCHW) layout achieves reliably good performance on GPUs due to the specifics of their memory and cache organisation, where NHWC or (H)eight - (W)idth - (C)hannels - Batch-size(N) (HWCN) layouts do not perform as well [192, 182]. Consideration must be made for the computational footprint of applying layout transformations as they can be costly during runtime (for example, introduce additional transpose operators).

Constant Folding is an optimisation that replaces constant expressions and their uses with actual values as the code is being compiled. If a constant expression involves arithmetic on other constants or inline values, constant folding evaluates these expressions at compile time to avoid doing so during every inference [214]. For DL tensors, the compiler infers tensor values whenever possible by folding suitable graph nodes.

Dead Code / Node Elimination: Often caused by prior transformations, dead code are expressions or nodes within the graph, of which results are never used and can be completely removed, reducing computation burden [160]. For example, an operator which regularly accepts two inputs, however, only one is available (due to prior rewriting), can be completely eliminated. Also, if one of tensor value dimensions is zero, the expression can be replaced with the other non-zero operand, simplifying the graph.

Common Sub-expression Elimination: A common sub-expression is a repeating expression within the DL model definition that has previously been computed and which does not require re-computation to satisfy the computation logic. Common sub-expressions can be computed once and later reused, reducing computation [257].

Expression Inlining involves moving the operations of the called function into the body of the calling function, eliminating function call overheads such as stack operations at the expense of increased instruction cache load [143].

Buffer Reuse: Some memory buffers can be re-used. For example, the same memory area can be used for input and output of a DL operator, provided no other operators depend on the input further within the data-flow cycle. As this is a compile-time optimisation, complex memory planning algorithms can be used [155].

2.5.4 Deep Learning Compiler Backend

The DL compiler backend is a set of components responsible for optimising and lowering the DL model representation closer to the intended target-device implementation, often leveraging generic compilers such as GCC [311], LLVM [172] or NVCC [225] for the executable binary generation. Once the model is ingested by the DL compiler and the HLIR optimisations are applied, the resultant IR is lowered to Low-Level Intermediate Representation (LLIR). The LLIR facilitates optimisation of individual operators and model sub-graphs (if fused), transforming the computationally-invariant linear algebra representation into tensor programs. The DL compiler backends commonly rely on existing compiler tool-chains (LLVM, NVCC) for compilation-proper, to take advantage of general-purpose code optimisations. DL compilers can also generate custom low-level code (by progressively transforming the LLIR) and optimisation passes, to leverage domain-specific knowledge in DL models and processors [184].

2.5.4.1 Low-level Intermediate Representation (LLIR)

```
// Matrix-matrix multiplication (GEMM)
// C = output tensor, A & B = input tensors
for (n = 0; n < N; n++) {
    for (m = 0; m < M; m++) {
        for (k = 0; k < K; k++) {
            C[n][m] = C[n][m] + (A[n][k] * B[k][m])
        }
    }
}
```

Listing 2.3: GEMM loop nest example

The unique LLIRs support code transformations under the compiler's purview (for example, support for a subset of accelerators or specific optimisations requiring support within the IR). At the same time, granularity of LLIRs must ensure hardware characteristics (for example, cache, memory layout, scheduling patterns) information

can be represented, yet also be generic enough to convert a substantial library of DL operators from linear algebra towards their compute representation for specific target-devices. A common representation of operators such as Convolution involves a set of loop nests, that execute one or more arithmetic operations within the innermost loop - as shown in Listing 2.3. Especially in the case of multi-core processors, the LLIR should support generating high-performance tensor programs that leverage parallelisation, taking advantage of multiple cores and vector instructions, as well as enable optimisation of tensor programs in terms of data storage, reuse and locality, since memory accesses are orders of magnitude slower than arithmetic operations [118], for example, through loop tiling [210]. Broadly, DL compiler LLIRs can be grouped into the following categories:

Polyhedral-based LLIRs transform loop nests based on the Polyhedral model [76] - a mathematical framework that represents loops as polyhedra, which determine data dependencies between compute statements contained within loop bodies. The Polyhedral model transforms loop variables (tensor indices) linearly, to reorganise loop nests towards some objective, given a set of data dependency constraints. The loop nest generated when HLIR is lowered to a LLIR is referred to as the initial *schedule*. Each polyhedral transformation generates a new schedule that represents the transformation, where after several transformations, the final schedule is produced. One of the major use-cases of the polyhedral representation are the loop parallelisation, cache-locality and memory access optimisations [302]. As all transformations within this model are discoverable via linear programming, the discovery cost is independent of the program complexity. The Polyhedral model can also optimise programs with cyclic data dependencies (for example, LSTM cells) due to its initial stage of data dependency and validity analysis, whilst other LLIRs may struggle to support such scenarios. DL compilers such as TC [345], PlaidML [374] or Tiramisu [18] support the Polyhedral model within their LLIRs.

Halide/Schedule-based LLIRs: Halide [262, 261] - a Domain Specific Language (DSL), was initially designed to optimise image processing pipelines, by reorganising

computation to maximise parallelism and cache locality. As image processing pipelines and DL tensor arithmetic share many similarities, subsequent Halide LLIR versions were adapted to support DL compilation [185]. Whilst polyhedral-based LLIRs guarantee data dependency correctness, they are limited in the scope of supported operations.

Halide-based LLIRs address these limitations by simplifying the representation of computation via the use of interval (loop iterator extent) arithmetic rather than polyhedra linear arithmetic, and to further relax expression representation and increase versatility, propose the concept of decoupled *compute* and *schedule*[10]. Each schedule is a series of *primitives* such as *tile*, *vectorise* or *parallel* that determine how the loop nest should be transformed and which portion of computation should be parallelised or computed using SIMD/SIMT instructions. Whilst more versatile and easily-applicable to DL tensor program compilation, Halide-based LLIRs lack correctness guarantees and require the DL engineer to produce tensor program schedules for each target-device. Halide-based scheduling is partially adopted in the TVM DL compiler [39].

Composite / Custom LLIRs Some projects (for example, TVM) combine Polyhedral and Halide-based LLIRs to increase schedule specification flexibility for DL tensor operators. Other DL compilers implement completely custom approaches, not relying upon Halide or the Polyhedral model, including the HLO IR used in Tensorflow XLA [176] or Glow [278], with its strongly typed LLIR. Standalone Intel nGraph [58] utilises a combined HLIR and LLIR to ingest DL model definitions and transform them into calls to high-performance DL operator implementations within libraries such as Nvidia cuDNN [232] or Intel's oneDNN [47]. The choice of the LLIR largely determines the trade-off between low-level optimisations, IR expressivity, difficulty of program development and the level of support for DL operators and target devices.

[10]The *compute* defines the logical algorithm, whilst *schedule* determines where and how the *compute* will be executed on the device (for example, in what order an by how many threads)

2.5.5 Low-level Optimisations

The backend component of a DL compiler applies low-level optimisations by transforming the LLIR, whilst managing the trade-off between computation parallelism, data locality and computation redundancy. Low-level optimisations navigate this trade-off space to discover LLIR transformations that produce high-performance tensor programs. Prominent examples of low-level DL compiler optimisations include:

2.5.5.1 Operator Stacking

Notably adopted in Glow [278], operator stacking combines subsequent operators that perform element-wise operations (for example, addition or multiplication), such that they are computed within the same memory location during each loop iteration. For example, when Multiply directly follows Add, their sequential computation requires the CPU/GPU to load intermediate values into memory each time, invalidating caches. Operator stacking optimises data locality by removing intermediate stages, and while similar to operator fusion, it does not require bespoke schedules for resultant operators.

2.5.5.2 Loop optimisations

Loop optimisations are loop nest transformations that modify data access patterns, aligning them towards better cache, memory and core/thread use, leveraging specific parallelism patterns of the target-device. This improves data locality, reduces cache invalidation and thus computation latency. In Halide-based DL compilers, loop optimisations are usually implemented as schedule primitives, whilst Polyhedral LLIRs naturally arrive at equivalent loop nests as a result of affine transformations.

Loop permutation: Also referred to as loop reordering or loop interchange, loop permutation is an optimisation where the order of inner and outer loops within loop nests is permuted [206]. This optimisation is often selectively applied to better match

Original loop nest

```
for (a = 0; a < M; a++) {
    for (b = 0; b < N; b++) {
        W[a][b] = X[a][b] + Z[a][b]
    }
}
```

Transformed loop nest

```
for (a = 0; a < N; a++) {
    for (b = 0; b < M; b++) {
        W[a][b] = X[a][b] + Z[a][b]
    }
}
```

Listing 2.4: Loop permutation example

Unfused loops

```
// Loop A = matrix-vector multiplication
for (a = 0; a < N; a++) {
    for (b = 0; b < N; b++) {
        X[a][b] = Y[a][b] * Z[b]
    }
}
// Loop B = addition
for (a = 0; a < N; a++) {
    for (b = 0; b < N; b++) {
        W[a][b] = X[a][b] + Z[a][b]
    }
}
```

Fused loop

```
// Fused Loop AB
for (a = 0; a < N; a++) {
    for (b = 0; b < N; b++) {
        X[a][b] = Y[a][b] * Z[a][b]
        W[a][b] = X[a][b] + Z[a][b]
    }
}
```

Listing 2.5: Loop fusion example

Original loop nest

```
for (a = 0; a < M; a++) {
    W[a] = X[a] + Z[a]
}
```

Transformed loop nest

```
// (K: unroll factor = 3)
for (a = 0; a < M; a += K) {
    W[a] = X[a] + Z[a]
    W[a] = X[a + 1] + Z[a + 1]
    W[a] = X[a + 2] + Z[a + 2]
}
```

Listing 2.6: Loop unrolling example

Original loop nest

```
for (a = 0; a < M; a++) {
    W[a] = X[a] + Z[a]
}
```

Transformed loop nests

```
// (K: spliting factor = 2)
for (a = 0; a < M / K; a++) {
    W[a] = X[a] + Z[a]
}
for (b = M / K; b < M; b++) {
    W[b] = X[b] + Z[b]
}
```

Listing 2.7: Loop splitting example

the cache and register patterns of the target-device, reducing cache invalidation and promoting data locality. Listing 2.4 depicts an example of loop permutation.

Loop Fusion is a data locality optimisation that fuses two or more loops, as long as they share extent boundaries by combining their loop bodies together. Listing 2.5 depicts an example of loop fusion. Halide-based TVM [39] utilises schedule primitives for loop fusion, while Tiramisu [18] or PlaidML [374] rely on polyhedral transformations. FusionStitching [388] utilises pre-defined computation patterns and implements fusion based on predicted performance potential (via a cost model).

Loop splitting: During loop splitting, a loop is split into K loops, where each operates on a portion of the original data [34, 74]. This transformation optimises the loop nest towards data parallelism, where each of the loops may be executed by a separate thread, better leveraging multi-core architectures, as well as enabling other optimisations such as vectorisation. Listing 2.7 depicts an example of loop splitting.

Loop unrolling is a loop nest transformation where the loop body is repeated K times, whilst loop extent is modified to iterate in K increments, reducing total number of iterations necessary [154, 289]. Loop unrolling reduces computation associated with loop condition checks and facilitates the compiler to introduce aggressive instruction parallelism optimisations, leveraging processor instruction pipelines [62, 250].

Original loop nest

```
for (a = 0; a < N; a++) {
    for (b = 0; b < N; b++) {
        W[a] = X[a][b] * Z[b]
    }
}
```

Transformed loop nest

```
// (K: tiling factor = 2)
// Producing <2x2> tiles
for (a = 0; a < N; a += K) {
    for (b = 0; b < N; b += K) {
        for (c = a; c < min(a + K, N); c++) {
            for (d = b; d < min(b + K, N); d++) {
                W[c] = X[c][d] * Z[d]
            }
        }
    }
}
```

Listing 2.8: Loop tiling example

Loop tiling, sometimes referred to as *loop blocking*, exploits locality of data accesses in loops by reorganising loop nests to access data in chunks/blocks, also referred to as *tiles* [359]. Loop tiling transforms a loop nest into an inner (intra-tile) and outer (inter-tile) loop, where the inner loop iterates over a single tile. Tile size is adjusted to better match the cache hierarchy available on the target-device. Multi-level tiling can improve access to multiple levels of caches and the DRAM memory for data locality [246]. Listing 2.8 depicts an example of loop tiling with the *tiling factor* of two, producing 2×2 tiles. The *tiling factor* decides the tile size and is chosen based on cache size / organisation in the target-device, often determined via auto-tuning.

Sliding windows: The default method of computation with multiple sequential loops containing data dependent operations is to perform computation breadth-first and element-wise, initially performing the first operation across all elements of the tensor and only then performing the following operation, requiring intermediate results storage. Thus, each operation can be fully parallelised, provided there are sufficient compute resources (for example, threads), however, reducing data locality and register/cache reuse. Alternatively, the loops can be fused to combine the two operations within a single loop body, increasing data reuse, however, reducing data locality as the data necessary for both operations may not fit within low-level caches, invalidate them frequently. Sliding windows computes intermediate values as needed (similar to loop fusion), yet also retains them until no longer needed, increasing possibilities for data reuse. Most notably, the sliding windows is used in the Halide DL compiler [262].

2.5.5.3 Hardware-dependent Optimisations (HDOs)

During Hardware-Dependent Optimisationss (HDOs), the DL compiler modifies implementations of tensor operators based on target-device features and domain knowledge (for example, high-level compute definitions of operators). Alternatively, the DL compiler may leverage existing compiler stacks such as LLVM, to perform HDOs such

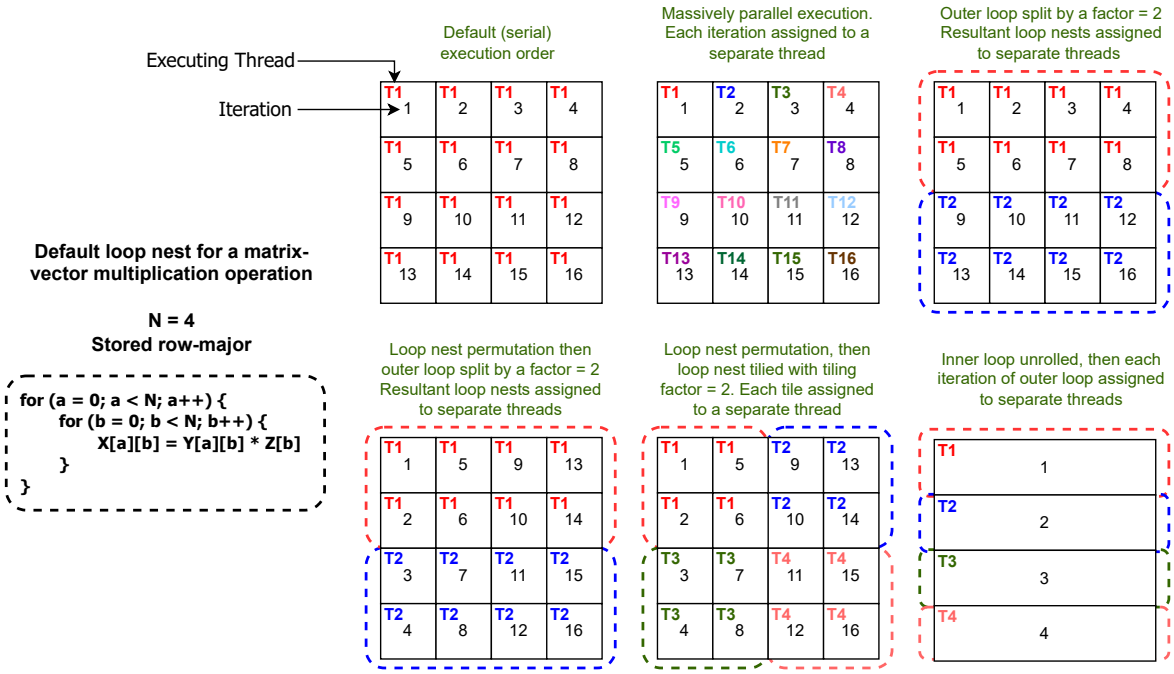


Figure 2.20: Different parallelisation schemes enabled by loop nest transformations

as *vectorisation*. HDOs typically make use of device parallelism capabilities to maximally utilise resources and improve program execution latency. Many of the aforementioned loop optimisations (for example, tiling) improve data locality, but also facilitate other, parallelism-focused optimisations. During HDOs, the trade-offs between data locality, reuse and parallelism, must be carefully balanced to produce high-performance tensor program implementations. Following, several HDOs are described:

Parallelisation is an optimisation used by DL compilers to improve runtime performance of tensor programs, by exploiting features of modern CPUs and GPUs such as: multi/hyper-threading, thousands of primitive cores and complex thread hierarchies [154]. Halide [262] and TVM [39] both utilise schedule primitives such as *parallel*, to recognise computation patterns that can be split across multiple threads of the CPU, and automatically perform loop splitting or tiling to assign threads to the blocked loops, while specific loop nest levels are assigned to GPU thread blocks and threads using the *bind* schedule primitive. Stripe [374], the LLIR used by the PlaidML introduces a polyhedral

block, which recognises opportunities for thread-level parallelisation within the loop nest levels and assigns threads to appropriate loop extent portions after polyhedral transformations. Figure 2.20 depicts different approaches to parallelisation of the loop nest at a single operator level.

Original loop nest

```
// W, X, Y, Z are vectors
// containing 16x float32 values
for (a = 0; a < N; a++) {
    W[a] = X[a] * Y[a] + Z[a]
}
```

Vectorised loop nest

```
// Using Intel's AVX-512 intrinsics
W = _mm512_fmadd_ps(X, Y, Z);
```

Listing 2.9: Vectorisation example

Vectorisation: SIMD [78, 13] vector units and their associated instruction sets enable one-shot arithmetic on vectors rather than scalars, which can be advantageous in highly data-parallel DL tensor computations. DL compilers replace loops with extents of size N with singular vector instructions that support vector operands of size $\leq N$ [171, 25] on a given target-device. Listing 2.9 depicts an example of vectorisation using Intel's *intrinsic*[11] C API functions, compatible with the AVX instruction set [195, 46]. Alternatively, the DL compiler can rely on third-party toolchains such as LLVM [172] to detect computation patterns susceptible to vectorisation [270], with the Glow [278] DL compiler being a prominent example such an approach.

Tensorisation and Hardware Intrinsics: To leverage processing elements such as Nvidia GPU tensor cores [203] during optimisations, DL compilers must support embedding of device-specific instructions within operator loop nests - *hardware intrinsics*. TVM [39] solves this problem using *tensorisation* (via the *tensorize* schedule primitive), which separates the intrinsic function bahaviour declaration from the mapping rule that determines how the call to intrinsic functions should be lowered together with the

[11] Intrinsic functions are function calls of which the DL compiler has internal knowledge, often used to provide parallelisation or vectorisation functionality given specific target-device architecture support

tensor program implementation. The *tensorise* primitive also facilitates embedding micro-kernels within tensor programs to improve their performance, for example, by introducing quantisation micro-kernels [108] during LLIR transformations.

External tensor algebra libraries such as cuDNN [232] or Intel oneDNN [47], contain high-performance implementations for different combinations of processors and tensor operator variants. DL compilers such as TVM can automatically link calls to these libraries within the compiled tensor program. For example, when compiling a specific variant of the Convolution operator, TVM can link to cuDNN and take advantage of a high-performance matrix-multiply routine dedicated for a specific GPU [39]. However, DL compilers have no means to inspect, analyse or further optimise such external routines, which can sometimes degrade runtime performance compared to code generated directly via the DL compiler, or obtained through auto-tuning. To take advantage of these routines, the DL compiler must ensure data layouts and operand shapes match the library API specifications, oftentimes prohibiting the compiler from applying other optimisations such as operator fusion.

Memory-related optimisations: Before the DL tensor programs can access the tensor data, an appropriate amount of DRAM memory must be allocated ahead of the program launch. Accelerators such as GPUs also contain multiple types and levels of memory and cache, including fast shared memory space or thread-local memory space that is slower but has a larger capacity [207, 379]. Ability to effectively access allocated memory can speed up tensor program computation on CPUs and GPUs [92, 312, 103]. As such, some DL compilers introduce separate memory related optimisations, such as the explicit memory scopes implemented as schedule primitives in TVM, which can pin select compute expressions to shared or local memory spaces, inserting specific memory allocation instructions within the generated tensor program [184].

Access to DRAM memory during cache misses can be substantially more time and energy consuming than singular arithmetic instruction executions [118]. As such, careful

synchronisation of threads and their memory accesses must be considered to achieve high program performance, especially when considering memory shared amongst multiple threads. TVM addresses this by introducing virtual threads and associated multi-level thread hierarchy as a form of a schedule primitive [201, 39]. Once lowered during LLIR transformations, memory barriers and thread control statements are introduced to ensure synchronisation of threads and interleaving of shared memory accesses, hiding the memory access latency.

2.5.6 Code Generation

Once all low-level optimisations have been applied, the LLIR is lowered towards the target-device, either via conventional compilers (for example, LLVM [172]), or by generating binaries directly within the DL compiler. Lowering using external compilers enables generic code optimisations to be applied to the already domain-optimised tensor operator implementation, further improving performance. The result of the code generation stage produces several artefacts. Firstly, the optimised model graph is produced, often in human-readable format such as JSON. IEs follow this structure and map high-performance operator implementations to the individual nodes within the model DAG during inference. This artefact also commonly contains references to model data (for example, weights), encoded as tensors. Secondly, compiled operator implementations are produced, usually in the form of individual binary files or static library bundles, that enable various runtimes to call the exposed program interfaces during model execution.

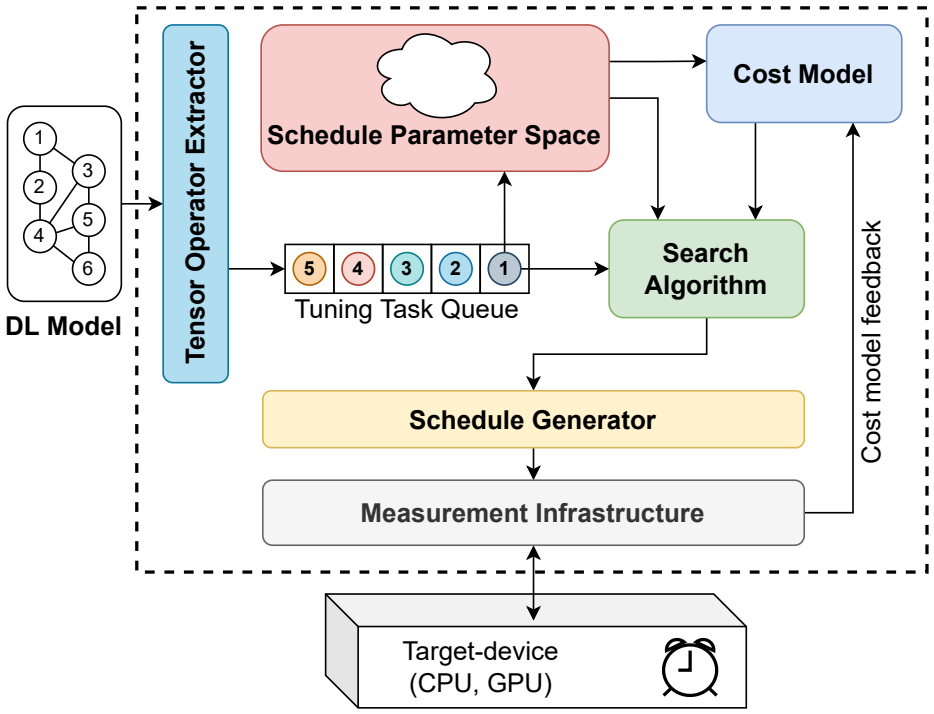


Figure 2.21: Components and operation of a typical DL auto-tuner

2.6 Deep Learning Auto-Tuning & Autoscheduling

2.6.1 Overview

As outlined in Section 2.3.2.7, manual engineering and optimisation of DL model tensor programs is extremely challenging considering the multitude of DL models, operator shapes and varied characteristics of target-devices. To help alleviate this problem, DL compilers facilitate applying high and low-level optimisations such as loop nest transformations to DL models during compilation.

There are millions of potential loop nest transformations (i.e. schedules) for a single unique tensor operator (see Section 2.5.5), where only a small subset is computationally valid, and an even smaller subset exhibits low latency on the specific target-device [40, 8, 183]. For example, within the schedule spaces produced by TVM [39], there

Table 2.2: Characterisation of prominent DL compiler auto-tuners and autoschedulers

Auto-tuner	**Compiler**	**Scheduling**	**Cost Model**	**Search**	**Device**
AutoTVM [40]	TVM	Templates	Gradient Boosting, TreeGRU	Simulated Annealing	CPU,GPU
Chameleon [8]	TVM	Templates	Gradient Boosting	Reinforcement Learning	GPU
AdaTune [183]	TVM	Templates	Random Forest	Simulated Annealing	CPU,GPU
OneShot Tuner [283]	TVM	Templates	Transformer	N/A	CPU,GPU
DynaTune [377]	TVM	Templates	Gradient Boosting	Simulated Annealing	CPU,GPU
ALT [373]	TVM	Templates	Gradient Boosting	Active Learning	GPU
NoSE [90]	TVM	Templates	K-Nearest Neighbour	Evolutionary	GPU
Transfer-Tuning [93]	TVM	Custom Templates	N/A	N/A	CPU
Ansor [385]	TVM	Hybrid	Gradient Boosting	Evolutionary	CPU,GPU
Moses [383]	TVM	Hybrid	Transferrable	Evolutionary	GPU
FamilySeer [378]	TVM	Hybrid	Multiple Gradient Boosting models	Gradient Descent	CPU,GPU
FlexTensor [386]	TVM	Generative	Reinforcement Learning	Heuristic, Simulated Annealing	CPU,GPU
DietCode [384]	TVM	Custom	Linear Regression	Decision Tree	GPU
Bolt [364]	TVM	Custom	N/A	Custom	GPU
1st Gen Autoscheduler [215]	Halide	Generative	N/A	Brute-force	CPU,GPU
2nd Gen Autoscheduler [3]	Halide	Generative	FC ANN + Embeddings	Tree Search	CPU,GPU
3rd Gen Autoscheduler [15]	Halide	Generative	FC ANN + Embeddings	Beam Search	CPU,GPU
ProTuner [105]	Halide	Generative	FC ANN + Embeddings	Monte Carlo Tree Search, Beam Search	CPU
TC Auto-tuner [345]	TC	Custom	N/A	Genetic	GPU
PET [349]	Multiple	Custom	N/A	Genetic	GPU

are over 10.45m unique tensor program schedules for a 2D Convolution with input shape of $1\times512\times7\times7$ and kernel shape of 3×3. Each such *candidate* tensor program exhibits different performance characteristic (execution latency, target-device utilisation), however, these characteristics are unknown until the program is compiled and executed. Manually determining an optimal schedule for a tensor operator and target-device combination is challenging as it requires in-depth understanding of the operator and target-device characteristics, and can only be guided heuristically.

In response to these challenges, DL auto-tuners - optimisation frameworks that automate search over the enormous schedule space of candidate tensor programs, started to gain prominence. A DL auto-tuner automatically determines a near-optimal schedule for a given tensor program by parameterising a schedule template (auto-tuning) or generating the schedule based on different rules and steps (autoscheduling). An auto-tuner traverses the schedule space using cost models and search algorithms, generates and compiles promising tensor programs, and measures their latency on target-device to then utilise these measurement results to further guide schedule search. During search, thousands of candidate tensor programs are evaluated on target-device for every unique tensor operator, causing DL auto-tuning to be a lengthy process. Figure 2.21 depicts a high-level design of a DL auto-tuner while Table 2.2 depicts their characterisation.

As DL auto-tuners substantially improve inference speed, there is a market for automating DL model optimisations. Both established DL MLaaS providers such as Amazon AWS [296], Alibaba [102], Huawei [128], and startup companies such as Ampere Computing [44], NeuralMagic [136] or OctoML [241] are now providing DL model optimisation via auto-tuning as part of their Cloud MLaaS service.

2.6.2 DL Auto-tuner Operation and Components

DL auto-tuners are composed of several modules that co-operate with one another to optimise tensor operators towards target-devices. Auto-tuners solve an optimisation

problem, where $o_i \in O$ is a set of tensor operators to be optimised and T_o a set of candidate tensor programs for operator o. The problem can be formulated as follows:

$$\arg\min_{t \in T_o} m(c(o,t)) \tag{2.48}$$

where $c()$ represents the compilation process for a given operator and its unique tensor program, and $m()$ is the program latency measurement on the target-device.

2.6.2.1 Schedules and Schedule Parameter Space

The space of all potential tensor programs (schedules) T_o, for a tensor operator $o \in O$, is different across DL compilers and can be different across DL auto-tuners utilising the same DL compiler. The schedule space typically consists of schedule parameters which configure a schedule that constructs a tensor program from a tensor operator expression, where a schedule is a set of transformation steps.

The TVM [39] DL compiler introduces the concept of schedule templates (explained in more detail in Section 2.6.3.1), which provide a set of steps such as *tile*, *parallelise*, *split* or *reorder*, that modify the program loop nest in some way, enabling modular application of low-level DL compiler optimisations as explained in Section 2.5.5. In schedule templates, parameters configure schedule steps to modify transformation effects such as the number of loop nest tiles (see Table F.1 in Appendix F).

DL auto-tuners such as the AutoTVM [40], rely on schedule templates to construct a space of all possible schedule parameters, and search this space to discover *configurations* that produce high-performance schedules. Different DL auto-tuners rely on different schedule spaces and different parameter representations. For example, Ansor [385], formulates each schedule as a series of incremental generation rules that construct a tensor program step by step, without the use of templates.

2.6.2.2 Cost Models

As outlined in Section 2.6.1, schedule spaces for even rudimentary tensor operators can be very large (millions to billions of unique schedules). To feasibly traverse these spaces and discover high-performance schedules, DL auto-tuners often utilise cost models that propose potentially optimal schedules without the need to perform brute-force traversal. Guided by latency measurements, cost models propose progressively better candidate schedules - a concept referred to as "learning to optimise" [40].

Different cost models have been used by different DL auto-tuners, such as Gradient Boosted Trees (GBT) [87] leveraging the XGBoost library [38] or TreeGRU [325], used within AutoTVM [40, 39] and Ansor [385]. AdaTune [183] replaces the GBT-based model in the AutoTVM infrastructure with a Random Forest (RF) regression model, whilst the Halide autoscheduler [215, 3] utilises a cost model based ANN embeddings to predict schedule performance given its characteristics and target-device features as inputs, further extended by [15] to include more architectural features as model inputs. FlexTensor [386] utilises a Q-Learning RL cost model [352], whilst DeepCuts [151] or DietCode [384] rely on the Roofline model [355] (see Section 2.4) and its derivatives, to estimate schedule or sub-schedule CI and prune inefficient ones.

Alternatively, the user can specify their own cost model that is best suited for a given DL auto-tuning scenario - an approach chosen by the ATF auto-tuner [266], or to create a transferable cost models that work across devices or operators, leveraging transfer learning (Transfer-tuning [93]), schedule similarity (FamilySeer [378]), or the Lottery Ticket Hypothesis approach [86] (Moses [383]).

2.6.2.3 Search Algorithms

DL auto-tuners utilise search algorithms to traverse the schedule space and propose program candidates, guided by cost-models as their energy functions.

AutoTVM [40] utilises Simulated Annealing (SA) [158] in combination with the cost model to traverse the space towards a global optimum, by accepting sub-optimal solutions with a decreasing probability, which avoids local minima. FlexTensor [386] leverages an RL-based search strategy, while Chameleon [8] replaces SA with an RL-based Proximal Policy Optimisation (PPO) [290] optimiser and a K-means clustering [110] adaptive sampler, proactively filtering candidates ahead of measurement.

AdaTune [183], modifies the SA-based search by dynamically balancing exploration vs. exploitation decisions during search, using a contextual, tensor operator-dependent factor, while AutoTVM [40] utilises a fixed hyperparameter for this purpose. AdaTune's [183] and Chameleon's [8] filtering and sampling also inspired the design of ALT [373], which balances candidate diversity and performance uncertainty.

The NoSE auto-tuner [90] proposes a unified Evolutionary search strategy and a K-Nearest-Neighbour [11] regression model to select tensor programs with highest achieved FLOPs, whilst ProTuner [105] leverages Monte Carlo Tree Search [32] to traverse the search space. Unified algorithm-model search is also explored in [350], whilst Evolutionary search is used by the Ansor [385] autoscheduler. The PET auto-tuner [349] utilises a multi-linear schedule mutator that modifies existing loop nest transformations and corrects ones with broken functional equivalency, ensuring schedule coherence. The Halide autoschedulers [15, 3, 215], utilise Beam search [73] and Tree Search with variations of hierarchical sampling or candidate pruning, based on the cost model predictions. Similarly to AutoTVM, the Tensor Comprehensions [345] DL compiler auto-tuner utilises Genetic Algorithm (GA) to traverse the candidate space.

Non-DL kernel auto-tuners such as the Kernel Tuner [343] have also been proposed. Kernel Tuner implements multiple search strategies such as SA and GA, Basin Hopping [348], Differential Evolution [315], Particle Swarm Optimisation [153] and others to optimise generic kernels / programs. In ATF [266] kernel auto-tuner, the auto-tuner functionality is determined by the user, with ATF providing the necessary infrastructure.

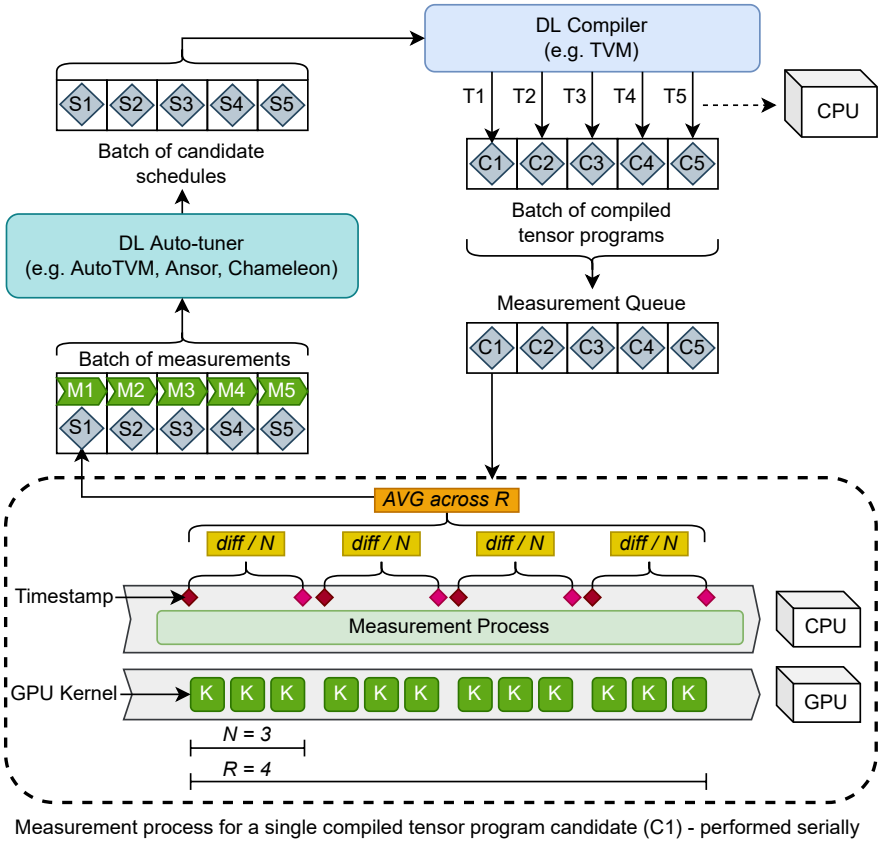


Figure 2.22: DL Auto-tuner Measurement Infrastructure

2.6.2.4 Measurement Infrastructure

Majority of DL auto-tuners leverage a mixture of online-trained cost models, search algorithms and tensor program latency measurements performed on the target-device during auto-tuning, balancing optimisation speed and quality of discovered schedules. Several approaches claim that avoiding (online) measurements and relying purely on a priori trained (offline) cost models is sufficient to produce high-quality schedules [284, 313, 306]. However, offline cost models require training datasets containing millions of pre-measured schedule examples [313], that cover a limited subset of tensor operators and target-devices, necessitating frequent re-training when any of these factors change. As such, tensor program latency measurements remain crucial for DL auto-tuning.

The candidate measurement infrastructure cross-cuts the auto-tuner and DL compiler component domains as it relies on the compiler to generate proposed tensor programs, and relies on execution runtimes to execute them, also often provided by DL compilers. A notable measurement infrastructure centers around the TVM DL compiler, utilised by auto-tuners such as AutoTVM [40], Chameleon [8], AdaTune [183], FlexTensor [386] or Ansor [385]. Similarly, auto-tuners derived from the Halide [262] DL compiler, leverage a common measurement infrastructure [15, 3, 215].

Despite minor differences, majority of DL auto-tuners utilise similar measurement infrastructures, whereby the auto-tuner provides a batch of candidate schedules that are compiled by the DL compiler and measured in isolation (serially) on the target-device, producing a series of measurements. These results are fed back to the auto-tuner as online training samples, for example, to update the cost model.

To measure candidates on variety of target-devices, auto-tuners such as AutoTVM [40] execute them remotely via Remote Procedure Call (RPC), leveraging small, self-contained execution runtimes available within the TVM [80] codebase. In the TVM compiler ecosystem, RPC enables multi-target-device measurements, where separate RPC servers are instantiated per remote or local device. For example, given two remote platforms, each containing four GPUs, the auto-tuner can measure up to eight candidate tensor programs simultaneously and maintain measurement isolation. Figure 2.22 depicts a host-local, single-device candidate measurement infrastructure.

2.6.2.5 Anatomy of a Measurement

As depicted in Figure 2.22, during measurement, each candidate tensor program is executed more than once to measure latency accurately. More specifically, a batch of candidate schedules S is compiled using the DL compiler[12] to produce a batch of compiled tensor programs C. The batch C is transferred to a measurement queue, where

[12]Usually in parallel, for example, by pinning compilation processes to individual CPU cores

each candidate is executed serially for measurement, producing a batch of measurements M that are fed back to the auto-tuner upon batch completion.

During each candidate measurement, the program is executed $N \times R$ times, where N is the number of executions within each repeat round R. Execution latency is measured using start (s_t) and end (e_t) timestamps collected at the CPU process level, at the start and end of each repeat $r \in R$, with a single candidate-wide average calculated across R repeats as $\frac{e_t - s_t}{N}$, producing measurement $m_i \in M$ of candidate schedule s_i. Not all candidate compilations and measurements succeed, primarily due to invalid schedules, Out-of-memory (OOM) errors or timeouts[13], with any failed candidates reported to the auto-tuner as measurement errors.

2.6.3 DL Auto-tuner Types

Whilst DL auto-tuners utilise different search strategies, cost models and other components (for example, samplers or filters), fundamentally they can be categorised by their methods for generating schedules from tensor operator expressions.

2.6.3.1 Template-based Auto-tuning

In template-based auto-tuners such as AutoTVM [40], schedule templates are used to construct the candidate space. For each tensor operator, a series of loop nest optimisation transformations are applied (for example, reordering, splitting, tiling), where a template is a pre-defined series of transformations with placeholder parameters that configure the transformation. For example, given a matrix multiplication operator, its schedule template describes that it can be implemented as a loop nest, transformed by tiling and reordering consecutively. Parameters within the template correspond to values used within each transformation, for example, the loop extents within tiling iterations or the axis over which to reorder the computation. A candidate schedule space consists of all

[13]OOMs and timeouts occur when valid schedules breach target-device capability limits

possible combinations of parameter values applicable to the given template. In template-based auto-tuning, the user must select a pre-defined template for a given tensor operator and target-device pair (for example, Convolution or FC towards CUDA-enabled GPUs), or develop a bespoke template themselves for more esoteric combinations.

Multiple DL auto-tuners rely on schedule templates, such as: AutoTVM [40], Chameleon [8], AdaTune [183], NoSE [90], CNN tuner for integrated GPUs [350], ALT [373], DynaTune [377] and OneShot Tuner [283]. Bolt [364] has a different approach to templated auto-tuning by leveraging parameterised acceleration libraries such as Nvidia CUTLASS [227], where it discovers optimal library call parameters to generate function calls for different combinations of tensor operators and Nvidia GPUs.

2.6.3.2 Generative Autoscheduling

Autoscheduling is an alternative approach to template-based auto-tuning, where optimal schedules are generated by applying the different transformations and their parameters based on pre-defined rules, without the need for templates. This approach is more opinionated compared to template-based auto-tuning as it relies on a priori rules defined by the autoscheduler designer, where rules are related to different tensor operator and target-device combinations. Notably, this approach has been used in the first generation Halide autoscheduler [215] for image processing pipelines. FlexTensor [386] also demonstrated generative autoscheduling as a viable method for optimising DL workloads, leveraging the TVM scheduling infrastructure to apply schedule transformations iteratively, and build up high-performance schedules.

2.6.3.3 Hybrid and Other Approaches

Recent works hybridise generative and template-based auto-tuning by initially generating a set of schedules using rules related to tensor operators and device classes, followed by automated schedule parameterisation to fine-tune them. Hybrid approaches provide the

best of both worlds in flexibility and versatility, however, increase the schedule space size (including invalid schedules), necessitating more complex search strategies. Notable hybrid autoschedulers are: the second and third generation Halide autoschedulers [3, 15], ProTuner [105] for Halide, and the Ansor [385] autoscheduler for TVM, while PET [349] performs hybridised autoscheduling and fine-tuning without an external DL compiler. Other projects perform auto-tuning as part of compilation rather than as a separate process, where the TC auto-tuner [345] maps a pre-defined CUDA kernel to a *tensor comprehension* (see paper for details), while TensorFlow XLA [176] selects best-performing cuDNN [232] or cuBLAS [230] kernels during compilation.

2.7 Chapter Summary

This Chapter discussed Machine Learning (ML), Deep Learning (DL), Deep Neural Networks (DNNs) and their layers, DL systems, including DL models, datasets, frameworks, inference engines, acceleration libraries and hardware that performs DL computation. Furthermore, topics such as parallelism in DL computation, different phases of the DL model life cycle and methods for characterisation of DL models and high-performance hardware were also discussed. The Chapter then focused on DL compilers and their relationship with achieving fast DL inference via both high and low-level optimisations, before diving into DL compiler auto-tuning - an automated method for discovering fast tensor programs that implement DL model computation.

DL auto-tuners and autoschedulers deliver promising tensor program and end-to-end DL model inference performance improvements, by leveraging plethora of advanced cost models, search algorithms and online target-device measurements. Design and implementation of DL auto-tuners often revolves around specific DL compilers, to take advantage of existing optimisation primitives, tensor program scheduling methodologies and program compilation capabilities.

However, the reliance of DL auto-tuners on combined cost model and measurement-based candidate tensor program evaluations, causes the process to become computationally expensive, as indicated in prior work [183, 8, 283, 386, 385, 377, 93, 383]. This thesis focuses on analysing, evaluating and reducing the high computational, time and energy costs associated with DL auto-tuning and autoscheduling, to reduce the barrier to entry for automated DL optimisation via DL compilers for DL practitioners.

The following Chapter explores and quantifies the costs associated with applying both high and low-level optimisations to tensor programs and model architectures. It also identifies specific inefficiencies within DL auto-tuner designs, which when rectified, could significantly reduce DL optimisation operational costs.

This page is left intentionally blank

Chapter 3

Cost of Deep Learning Optimisation

Optimising DL models using auto-tuning and autoscheduling is typically a prolonged process that consumes a lot of energy and isolates computational resources. Even in scenarios of optimising relatively small models such as ResNet-18 [113], it can take more than ten hours to achieve sizeable latency improvements when auto-tuning them towards a single target-device such as a unique Nvidia GPU [8, 39]. The extended time and energy costs stem from the heavy load imposed on the platform CPU and under-utilisation of the target-device (GPU) for prolonged periods of time during auto-tuning. Initially, the CPU is engaged with traversing the large candidate schedule space, leveraging cost models and search algorithms to discover template parameterisations (auto-tuning) or applying scheduling rules and steps (autoscheduling) to generate high-performance tensor program candidates. Once a portion of high-performance schedules are discovered, they must be individually lowered and compiled to programs that can execute on the target-device. Each compiled tensor program is executed in isolation on the target-device to ascertain its execution latency reliably. Results of these tensor program execution latency measurements are then fed back to the cost model to improve the schedule space traversal in the next search round. The auto-tuning procedure is described in more detail in Sections 2.6.1 and 2.6.2.4.

The implementation (schedule) space for each DL tensor operator within a model, often consists of a large number ($> 10^{11}$ [386]) of possible schedules, with the high-performance candidate schedules typically non-linearly distributed across the schedule space [183, 386, 8, 40]. To achieve significant performance improvement, effective search strategies propose a subset (typically hundreds to thousands) of schedules with predicted low-latency potential. Once proposed, their execution latency is measured on the target-device to ascertain their performance. This, coupled with an increasing number of DL models containing tens to hundreds of complex operators, causes DL auto-tuning to engage the host machine and target-device for a prolonged period of time, as it requires an isolated execution environment to ensure accurate latency measurements. Furthermore, any modification performed to any of the DL model operators, with respect to their structure, data layout or algebraic logic, would require for the auto-tuning process to be repeated, as these modifications invalidate the results of candidate latency measurements for the particular operator and target-device combination.

DL optimisation is beginning to become adopted by large DL Cloud providers within their MLaaS deployments, enabling automatic optimisation of their users' DL models via services such as Amazon SageMaker Neo [296], Huawei Tensor Boost Engine [128] or Alibaba MNN [148]. The cost concerns associated with local, small-scale auto-tuning of a few models across several target-devices are inherently amplified when considering a Cloud setting at large scale. The issues of high host platform load, low target-device utilisation and low system availability caused by device isolation, cause reduced system throughput, increasing operational costs. Whilst intuitively assumed, the costs of DL auto-tuning have not yet been comprehensively studied. The objective of this Chapter is to provide a thorough analysis of DL optimisation (particularly auto-tuning) time and energy costs, and discover phenomena that when adequately exploited, could provide improvements to the cost-efficiency of DL optimisation at small and large scale, reducing barrier to entry and improving sustainability of DL systems.

3.1 Study Setup

To analyse the impact of various optimisations on DL model performance (execution latency) and execution costs on GPUs, and in particular the costs of performing optimisations, an experimental study was performed, involving several DL models, frameworks, platforms, high-level optimisations and auto-tuning approaches.

Table 3.1: Details of DL models used during the study

Model	Parameters	FLOPs	Input Size	Top-1/5 Acc.	Details
MobileNet-V2 [288]	3.4M	314M	1x3x224x224	71.88% / 90.29%	Appendix D.4
DenseNet-121 [125]	8M	3B	1x3x224x224	74.65% / 92.17%	Appendix D.6
ResNet-18 [113]	11.5M	8B	1x3x224x224	69.76% / 89.08%	Appendix D.7
VGG-16 [304]	138.4M	19.6B	1x3x224x224	71.59% / 90.38%	Appendix D.8
VGG-19 [304]	143.7M	20B	1x3x224x224	72.38% / 90.88%	Appendix D.9

Table 3.2: Details of hardware platforms used during the study

	Host		Target-device (GPU)		
Abrv.	CPU	DRAM	Model	Arch.	DRAM
A	64-cores 2x Intel Xeon 5218, 2.3Ghz	196GB	Nvidia V100	Volta	32GB
B	12-cores Intel i7-8700K, 3.7GHz	16GB	Nvidia GTX2080	Turing	8GB
C	24-cores AMD 1920X, 3.5GHz	128GB	Nvidia GTX2080	Turing	8GB
D	12-cores Intel i7-6850K, 3.8GHz	32GB	Nvidia GTX1080	Pascal	8GB

3.1.1 DL models

The study analyses the cost of optimising prominent CNN models as described in Table 3.1. The studied model architectures range in terms of structure, number of parameters (3.4m - 143m) and computational complexity (314m - 20b FLOP). Each of the studied models contained weights trained on the ImageNet dataset [64], acquired from online repositories and their DL framework implementations [83, 163], as well as models made available as part of the TVM [39] testing infrastructure. Each of the models receives an input tensor for inference prediction of shape $1 \times 3 \times 224 \times 224$, with tensor layout of NCHW, and outputs a 1×1000 tensor containing class predictions. Further information

about the individual model architectures is provided within Appendices D.4, D.6 - D.9. During auto-tuning, only tensor operators supported by the available schedule templates within TVM's Tensor Operator Inventory (TOPI) were optimised.

3.1.2 Hardware Platforms

During all performed transformations and model optimisations, model compilation was performed using TVM towards GPUs as the target devices, situated within different host platforms as described in Table 3.2. The four platforms differ with respect to CPU compute performance (virtual cores / clock speed), available host DRAM memory and the compute performance of the GPU target-device.

Table 3.3: Details of middleware and software used during the study

Type	Specification	Version
Operating System	Ubuntu	20.04
GPU Driver / Compute Lib.	Nvidia CUDA [231] / Driver (Linux)	11.3.1 / 465.31
DL / Codegen Compiler	Apache TVM / LLVM [172]	0.7-dev1 / 11.0
DL Frameworks	Pytorch [1] / Apache MXNet [79]	1.6.0 / 1.6.0

3.1.3 Software

Each platform was provisioned with an identical Operating System and necessary middleware (drivers, compute libraries, compiler and DL frameworks), as described within Table 3.3. Most recent software versions were used at the time of experimentation. The TVM [39] DL compiler was selected to perform model compilation and optimisations due to its versatility, level of support for different DL frameworks and the number of available high and low-level optimisations (for more information and comparison of DL compilers, see Table 2.1 within Section 2.5). DL model definitions originated from two popular frameworks: Pytorch [1] and Apache MXNet [79]. For the purposes of the study, each of the models was converted from the respective frameworks into the TVM Relay IR using TVM's frontend modules for Pytorch and MXNet.

3.1.4 High-level Optimisations

Once transformed into the TVM Relay IR, the graph optimisation level applied to the DL model Relay IR was varied from 0 to 4 (all available levels). The optimisations associated with each level are described in Table E.1 of Appendix E. For more information about the operations associated with the fundamental high-level optimisation transformations, see Section 2.5.3. Optimisation levels 0 to 3 do not modify the control flow or computation logic of the DL model operators, whilst level 4 introduces passes such as FastMath that may affect the model accuracy as algebraic computation may be selectively approximated. Other than during experimentation involving varying graph optimisation levels, the level was affixed to 3 - preserving original model integrity.

3.1.5 Low-level Optimisations - Auto-tuning

Experiments with low-level optimisations performed via auto-tuning involved using several auto-tuners compatible with the TVM DL compiler, focusing on evaluating four template-based auto-tuners: Grid Index (GR), Random Index (RD), Genetic Algorithm (GA) search and AutoTVM (AT), each implemented within the mainline TVM auto-tuner sub-package AutoTVM [40]. The auto-tuning infrastructure provided by AutoTVM involves parameterising schedule templates (described in Section 2.6.3.1), with the schedule template parameters constituting the auto-tuning search space outlined in Table F.1 of Appendix F. During each auto-tuning experiment, all suitable[1] operators of a DL model are auto-tuned until 500 hardware measurements are performed, with all other parameters kept to default as per AutoTVM codebase and tutorials [80] at the time of the study being performed. The GR auto-tuner explores the template parameter space sequentially, the RD auto-tuner selects parameters from the space at random, the

[1]Whilst majority of computationally complex operators (for example, Convolutions, FC) are supported by most auto-tuners, more esoteric operators may lack suitable schedule templates for the particular device class, disqualifying them from optimisation via template-based auto-tuning.

GA search auto-tuner leverages a genetic search algorithm to more efficiently explore the schedule space and finally, the AT auto-tuner leverages a Gradient Boosted Trees (GBT) cost model and a search method based on SA to find parameters that generate high-performance schedules - as described in more detail in Sections 2.6.2.2 and 2.6.2.3. Details on the auto-tuner configurations used during the study can be found in Tables F.2 - F.5 in Appendix F. During this study, DL auto-tuning was performed locally whereby the machine executing the optimisation process contained the target-device.

3.1.6 Collected Metrics

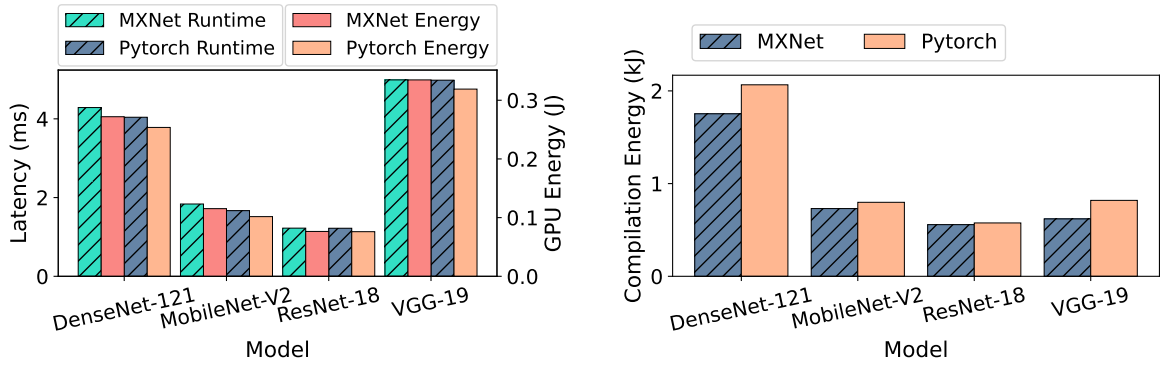
The core metrics collected during the study were: inference latency (time taken to perform single forward pass of the model on a single input sample), energy cost incurred during inference, the energy costs of performing model transformations from DL frameworks to Relay IR, high-level optimisations and auto-tuning - measured in Joules. All measurements were performed in isolation with no other significant processes executing on the host platform. GPU and CPU energy consumption costs incurred during experiment computations were collected using a custom profiling suite that leverages Nvidia's NVML [234] and the RAPL MSR interfaces for both Intel and AMD-based platforms [48]. The profiling suite was developed in Python and C++ with Cython [21] acting as the binding layer, and integrated within the TVM codebase.

3.2 High-level Transformations and Optimisations

3.2.1 The choice of a DL Framework

Each DL Framework has a unique method for specifying the model graph structure. Whilst tensor operators have equivalent functionality across frameworks such as Pytorch or Apache MXNet, the IR formats adopted by these frameworks and the ability of the

DL compiler to ingest these representations into its own IR (for example, Relay in TVM), may differ across DL frameworks and thus impact performance of the ingested models when performing inference on the same target-device. At the same time, the energy and time costs incurred during parsing, ingestion and compilation of DL framework model representations may differ across frameworks given the same DL compiler.



(a) Inference latency (ms) and GPU energy cost (joules) of the converted model.

(b) Energy costs (kilojoules) incurred by conversion and compilation.

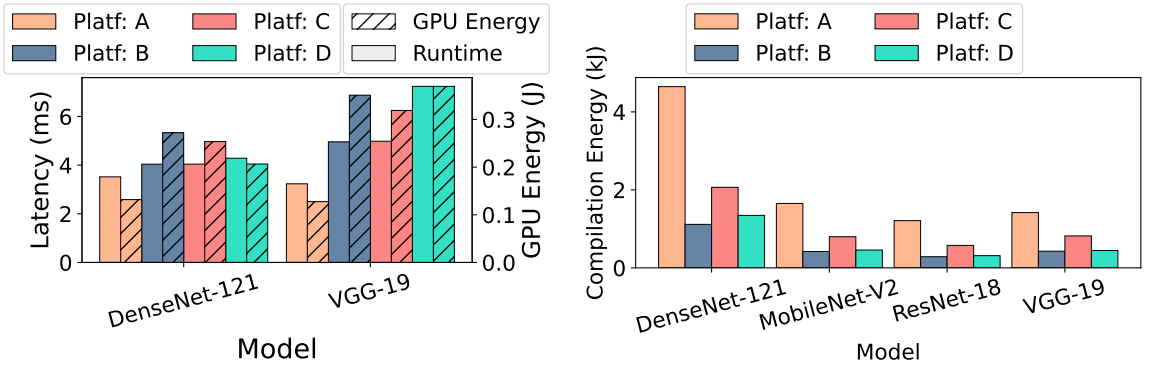
Figure 3.1: Conversion of four DL models from Pytorch and MXNet into TVM. Models are compiled towards the GPU in Platform A with graph optimisation level = 3, and executed for inference.

As depicted in Figure 3.1a, majority of the studied models, when converted from Pytorch to TVM Relay IR and compiled to the Platform A GPU, perform on average 1.05× better compared to the same models converted from Apache MXNet. These differences stem from the degree of support within the TVM frontend component for model conversion from Pytorch and Apache MXNet, including effective mapping to equivalent tensor operator definitions within the TVM's Relay IR.

At the same time, ingesting and compiling models originating from Pytorch, incurs on average 1.11× more energy costs compared to the MXNet models (see Figure 3.1b. Whilst the TVM frontend is more effective at ingesting Pytorch model graphs compared to MXNet graphs, and as a result produces slightly faster model implementations, the careful mapping of more adequate operator definitions in Relay results in increased energy consumption during compilation. It is important to note the magnitudes of scale

with respect to these differences in compilation cost and inference cost.

Whilst the compilation cost may be greater for Pytorch-sourced models, doing so produces slightly faster models that use less energy during inference - an activity performed in far greater numbers than compilation (up to tens of times for compilation and thousands or millions for inference). Additionally, an observation can be made about the strong positive correlation between resultant DL model inference latency and the energy consumed by the GPU as a result of the associated computation (Pearson: 0.9987). This suggests that energy consumption is a strong indicator of both time cost and costs stemming from computational complexity of model implementation as a composite metric, as also observed in prior work [65, 281].



(a) Inference latency (ms) and GPU energy cost (Joules) of the converted model.

(b) Energy costs (kilojoules) incurred by conversion and compilation.

Figure 3.2: Conversion of two/four DL models from Pytorch into TVM. Each model is then compiled towards the GPU target-device in Platforms A, B, C and D with graph optimisation level = 3, and executed for inference on the respective target-device of each platform.

3.2.2 Impact of the Hardware Platform

Whilst the complexity (measured in FLOPs required to be performed by each tensor program) of each DL model operator remains largely the same across platforms and target devices that execute the operator tensor programs, the choice of hardware has an impact on computational capability requirements and the associated time and energy

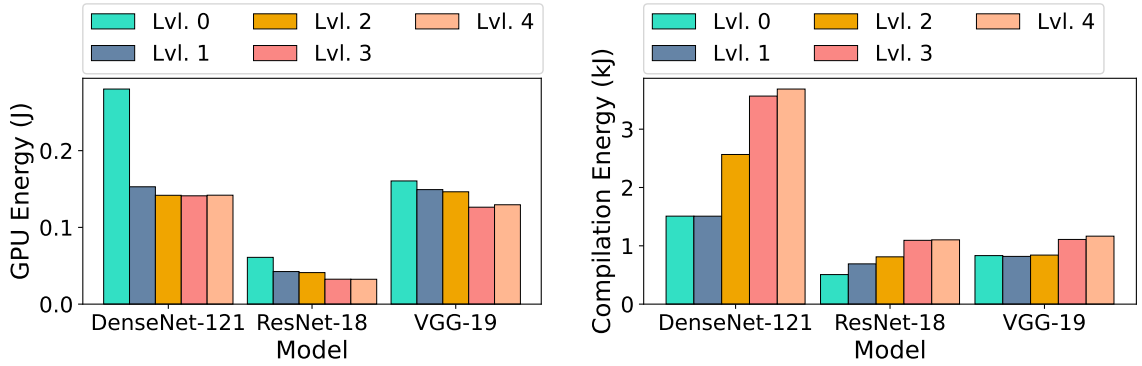
costs when performing these operations. Figures 3.2a and 3.2b depict the effects of performing framework-to-DL compiler transformations, high-level optimisations (level 3) and a compilation of two DL models towards GPUs contained within platforms A - D.

The choice of target-device and platform that hosts it, has measurable impact on the model execution costs, inference performance (latency) and the costs incurred during model compilation. Notably, it costs 4.16× more energy to compile the DenseNet-121 model towards platforms A and C compared to B and D. At the same time, despite platform A containing a more modern CPU, with higher number of cores than other platforms, the compilation under-utilises these cores and wastes energy. Likewise, for both models, platform C's GPU consumes less energy during model execution, despite being identical to that of platform B, which indicates that the CPU may be a bottleneck in that platform - being a more modern CPU with higher number of cores.

As such, when designing deployment pipelines that involve model transformations, optimisations and compilation, the DL engineer must consider both the target-device and the host environment where the model will operate. Much like in the case of differences between DL frameworks that supply the model definition, it is also necessary to consider the long-term costs of compilation and model deployment for inference. With the deployment being a substantially longer endeavour, it may be more cost-efficient to disregard compilation costs when trade-offs with end-to-end model performance exist. However, for scenarios such as Neural Architecture Search (NAS) [392, 72], where thousands of similar ANN architectures are partially trained and evaluated to determine their suitability and performance, this assumption may no longer hold and the aforementioned costs may need to be reconsidered when designing such DL pipelines.

3.2.3 Impact of High-level Optimisations

As outlined in Section 2.5.3, high-level optimisations transform the DL model graph to reduce inference latency. Application of different graph optimisation approaches to



(a) Energy cost (joules) of executing the optimised model on the GPU during inference.

(b) Energy costs (kilojoules) incurred by conversion and compilation with the applied graph optimisation level.

Figure 3.3: Conversion of three DL models from Pytorch into TVM. Each model is then compiled towards the GPU target-device in Platform A with varying graph optimisation levels between 0 and 4. The model is then executed for inference on the target-device.

several DL models was evaluated to understand their impact on the resultant model performance and inference cost, as well as the cost of applying these optimisations during compilation. The TVM DL compiler groups high-level optimisations into levels of optimisation passes (0 to 4) as described within Table E.1 of Appendix E.

As it can be observed in Figure 3.3a, as the optimisation level increases, the performance of the model improves and its inference energy costs reduce by 25 - 50%. Achieving this improvement however, incurs increasingly higher compilation costs as the level increases (by 29 - 60%) as shown in Figure 3.3b. This is intuitive as the higher optimisation levels perform more sophisticated transformations such as combining parallel convolution operations, which incurs higher computational cost.

It can also be observed that different models respond differently to graph level optimisations. For example, applying level 4 graph optimisations, improved inference latency of VGG-19 by 22.5%, whilst a 50.1% improvement can be observed for DenseNet-121. This is an example of a relationship between model structure, the number and complexity of its layers and the effects of applying high-level optimisations. DenseNet-121 contains 95% less parameters than VGG-19 (8m vs 143m) with 75% less required

FLOPs to perform a forward pass of the network. However, DenseNet-121 is composed of 121 complex layers making it more susceptible to optimisations such as operator fusion than VGG-19, resulting in a more significant, relative performance improvement.

An overarching observation is that applying a progressively more complex model graph-level optimisations brings substantial performance improvements and execution cost reductions, however, at a cost of increased energy consumption during model compilation. It can also be observed that performance gains resulting from these optimisations do not scale proportionally with the energy consumed to apply them. For example, DenseNet-121 experiences the largest improvement in execution energy costs after applying optimisation level 1, incurring a small increase in compilation cost. In the case of DenseNet-121, pplying further optimisation levels brings negligible inference cost reduction, with level 4 pass actually increasing the overall execution energy costs of the model whilst at the same time, doubling the compilation costs.

These non-linear cost vs. benefit differences can, however, be disregarded in cases where it is known ahead of time that the model will be performing inference for a prolonged period of time (for example, being deployed in production). Again, in the case of DenseNet-121, the doubled one-off cost of applying level 4 optimisations, compared to level 0 is negligible, compared to the nearly halved costs of performing inference using the optimised model. In other words, it would take just over 1500 inferences using the optimised model to amortise the costs of graph-level optimisation at level 4, with any inferences beyond that being 50% more energy efficient.

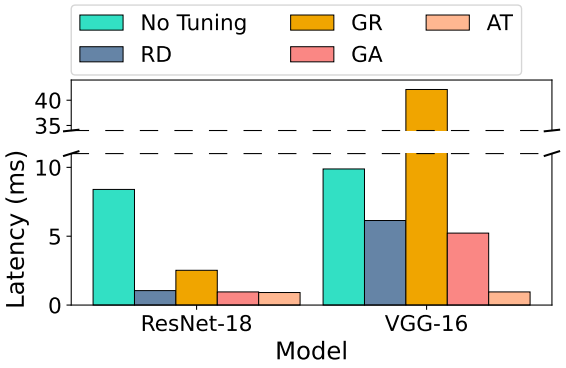
3.3 Tensor Operator Auto-tuning

As outlined in Section 2.6, DL auto-tuning is an automated method of generating high-performance tensor program schedules (tensor operator implementations) via an exploration of the space of all possible program transformations. To determine the

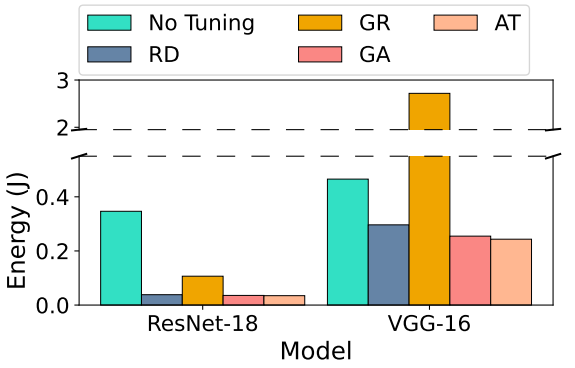
efficacy and costs associated with performing DL auto-tuning, several DL models were optimised using four auto-tuners, measuring the resultant model inference latency, execution energy costs and auto-tuning energy and time costs. Energy measurements were performed both at the CPU and GPU levels as described in Section 3.1.

3.3.1 Performance and Runtime Energy Costs

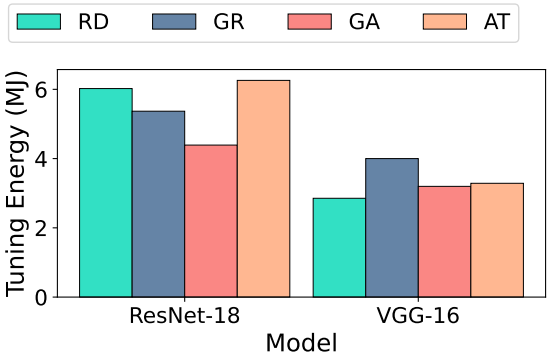
It can be observed that auto-tuning significantly reduces inference latency of the DL model, with some auto-tuners being more effective than others. More sophisticated



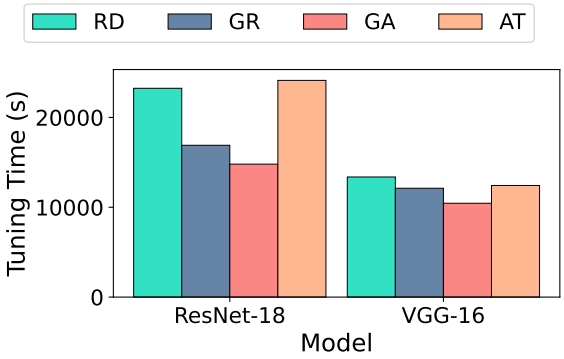
(a) Resultant model inference latency (ms).



(b) Runtime energy costs (joules) of the model during execution for inference.



(c) Energy cost (megajoules) incurred from auto-tuning.



(d) Wall-clock time cost of auto-tuning (seconds).

Figure 3.4: Impact of performing auto-tuning using four auto-tuners for two models towards the GPU in Platform A. Auto-tuning was performed until 500 hardware measurements were performed for each operator of each of the models. Prior to auto-tuning, level 3 graph optimisations were applied to each model. Resultant model inference latency is then tested by executing it on the target-device. "No tuning" corresponds to a model compiled using default schedules for each DL model operator.

auto-tuners such as GA search or AT were able to improve model latency by 9.22 - 10.35× compared to an un-optimised model relying on default schedules, whilst reducing execution energy consumption by 1.91 - 9.98×, as shown in Figures 3.4a and 3.4b.

Less sophisticated auto-tuners such as the brute-force GR search or RD search were less successful compared to their more sophisticated counterparts, however, still brought significant performance improvements and inference energy cost reductions to the optimised models. In some instances (GR for VGG-16), they caused a substantial performance degradation and increase in inference costs. This can be attributed to the duration of auto-tuning (up to 500 hardware measurements). GR explores the schedule space sequentially and it may so be that in the case of VGG-16, the first 500 candidate schedules within the space exhibited very low performance. Considering some of the most computationally complex operator shapes in VGG-16 are repeating across the model (see Appendix D.8), finding poorly performing schedules will negatively impact the overall achieved model inference latency, since an identical, best-found schedule will be applied to operators of the same input/output shapes.

3.3.2 Costs Across Different Auto-tuners

Much akin to the correlation that exists between DL model latency and its runtime energy costs, there exists a strong correlation between the overall wall-clock auto-tuning time and the energy costs incurred during optimisation (Pearson: 0.9880). However, as shown in Figures 3.4d and 3.4, different auto-tuners exhibit different time and energy cost patterns for different models. For example, when auto-tuning ResNet-18 with AT towards platform A, the resultant model exhibited a low inference latency of 0.89ms, however, the auto-tuning process costed additional 7200 - 8300s and incurred additional 240 - 1800kJ of energy costs compared to the GA search auto-tuner, despite small relative improvement in achieved model latency. The additional energy costs stem from the more complex operation of AT compared to the other studied auto-tuners. AT

must query its cost model and utilise an SA-based search strategy to propose candidate schedules ready to be measured on the target-device - a much more complex endeavour compared to a sequential (GR) or random (RD) brute-force methods.

A surprising phenomenon are the increased time and energy costs of auto-tuning with RD (+23,250s of time spent and +6,020kJ consumed), compared to sequential exploration (GR) or the GA approach. Given the similar computational complexity of both RD and GR approaches, they would be expected to exhibit similar time and energy cost patterns. Upon closer observation of the auto-tuning procedure, it can be observed that the procedure constitutes several stages: the schedule space search, candidate compilation and candidate measurement. Due to its stochasticity, the RD approach proposes candidates that may be erroneous (invalid schedule transformations that cannot be compiled or valid transformations that compile, yet fail to execute on the target-device due to computational or memory limits). Validity of such candidate schedules may not always be deterministic ahead of compilation or execution, and as such, they consume additional energy during their failed measurement attempts.

Furthermore, brute-force approaches propose measurement candidates without any substantial prior analysis of their potential performance, and as such, valid schedules that may exhibit very high latency are evaluated on the target-device, occupying the isolated environment for longer periods of time compared to other candidates. To understand the impact of erroneous and slow candidates on the overall costs of DL auto-tuning, it is necessary to examine their time and energy cost profiles more closely.

3.4 Sources of Inefficiencies in DL Auto-tuning

To determine the impact of slow and erroneous candidates on the costs of DL auto-tuning, the exhibited execution latency patterns of successful candidates and those that were erroneous were analysed at the level of individual operators.

3.4.1 Erroneous Candidate Schedules

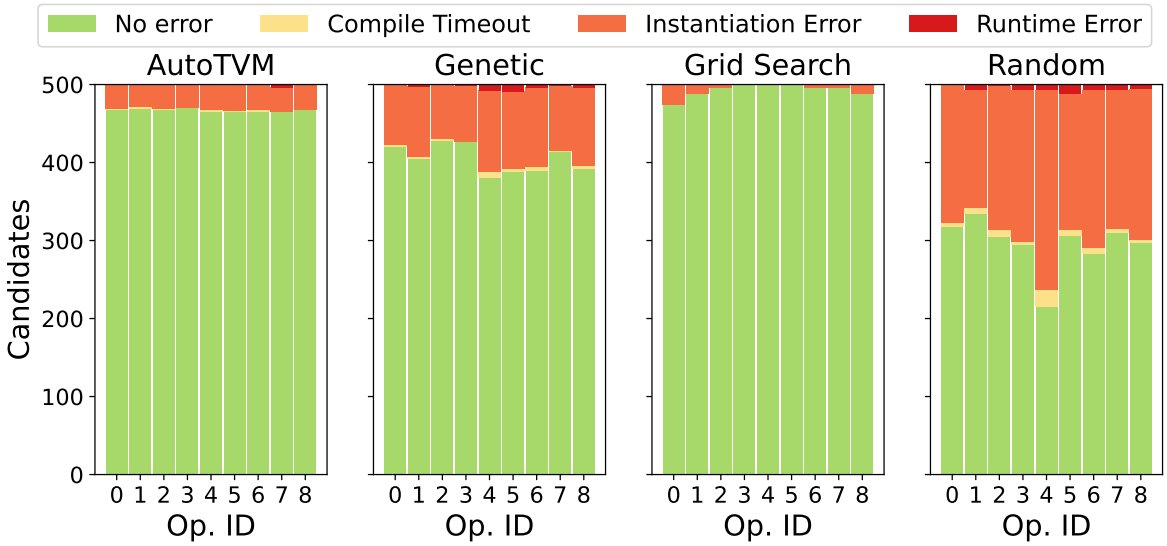


Figure 3.5: Erroneous candidate occurrences during auto-tuning of ResNet-18 towards platform A across four different auto-tuners. Auto-tuning until 500 hardware measurements are performed.

It can be observed that all auto-tuners propose at least some candidate schedules that are erroneous, as shown in Figure 3.5. Out of all studied auto-tuners, RD proposed the highest number of erroneous candidates, caused by its stochastic exploration of the schedule space, which does not guarantee, nor limit the occurrence of invalid schedules. Figure 3.5 depicts the number of candidates that produced timeouts during compilation (*Compile Timeout*). *Compile Timeouts* occurred, for example, due to schedule transformations that involved valid but cyclic operations, that are unable to complete in reasonable time (set by default to three seconds by the TVM infrastructure). Schedules unable to successfully compile were also observed, stemming from invalid loop nest transformations (*Instantiation Error*). Schedules that successfully compiled, yet could not successfully execute during their measurement due to compute / memory limit violations on the target-device (*Runtime Error*) were also observed.

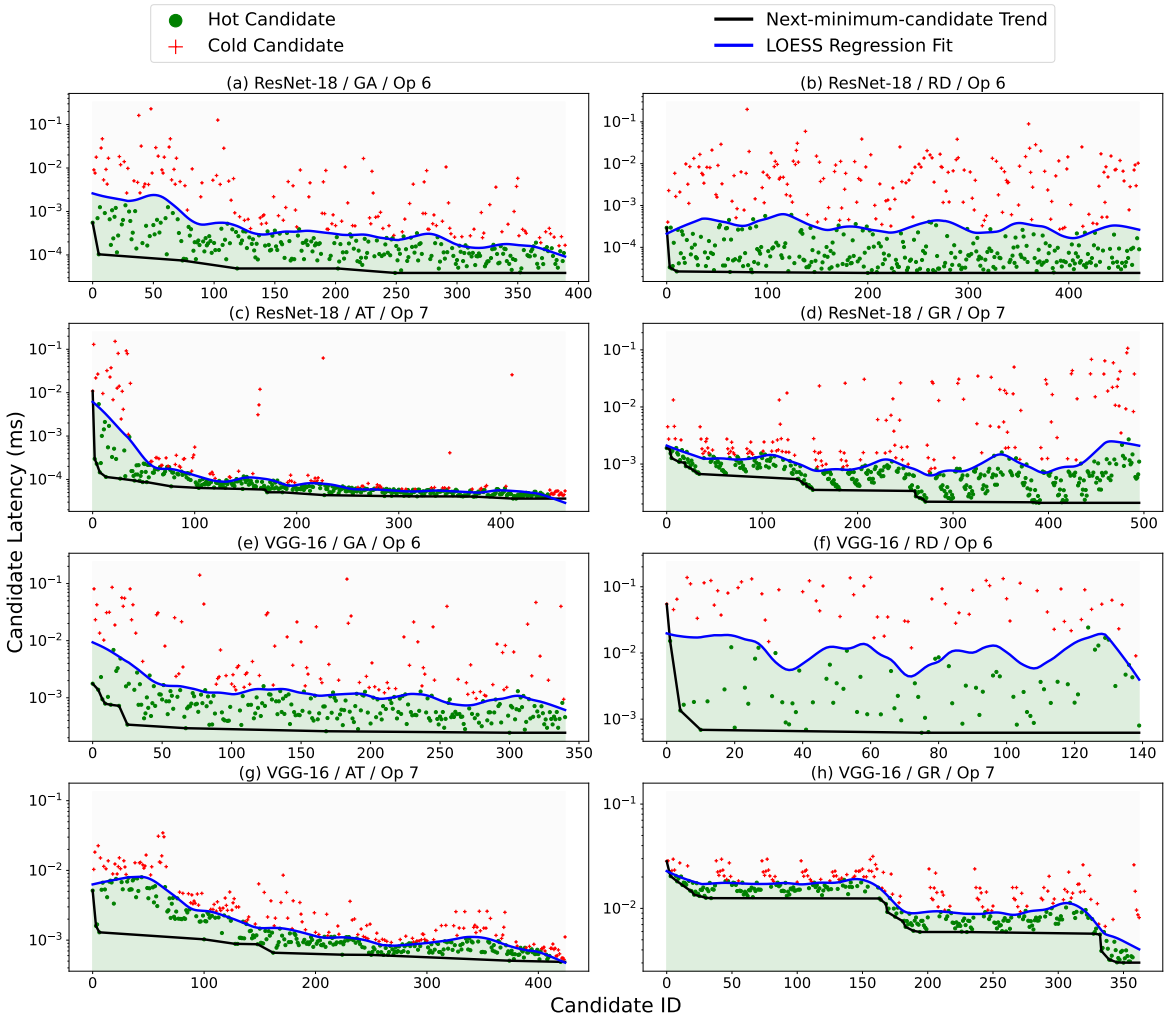


Figure 3.6: Runtime latency improvement trends when auto-tuning several ResNet-18 and VGG-16 operators using four different auto-tuners until 500 hardware measurements are performed. The figure depicts measured runtime latencies of successfully executed candidate schedules for each operator. LOESS regression is used to differentiate candidates that positively contribute towards optimisation progress (*hot* candidates) and those that were proposed for measurement by the search strategy despite their low performance (*cold* candidates).

It was found that *Instantiation Errors* were the most common reason for erroneous candidates across all auto-tuners, caused in part by the stochastic or cost-model-based schedule space exploration approaches of examined auto-tuners, particularly prominently

in GA (15% of all candidates proposed) and RD (36%). Another 1.9% and 2.6% of erroneous candidates were attributed to *Runtime Errors* and *Compile Timeouts* respectively. An important insight from this analysis is that existing auto-tuners determine when to stop the optimisation by observing the total number of hardware measurements that have been completed, including successful and erroneous ones. With erroneous candidates consuming additional 11 - 18% of energy, not only do they increase the cost of auto-tuning but also decrease opportunity for other (potentially desirable) candidates to be proposed and measured, reducing efficacy of the entire process. As such, it is important to minimise the occurrence of erroneous candidates.

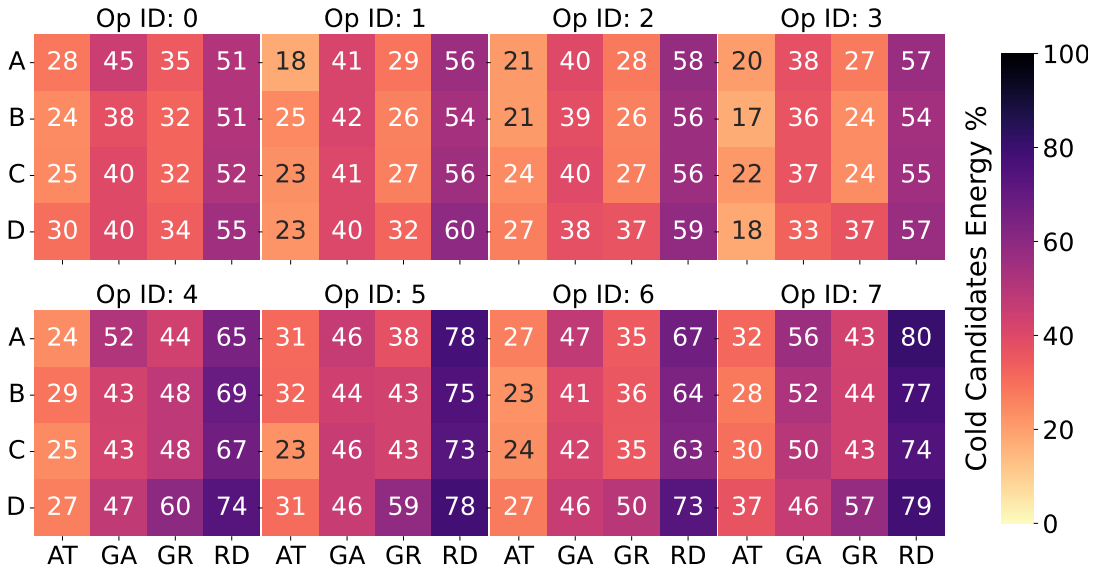


Figure 3.7: Percentage of energy costs attributed to *cold* candidates when auto-tuning eight operators of the ResNet-18 model using four different auto-tuners across four platforms.

3.4.2 Identifying Cold Candidates

An observation of the individual operator candidate latency profiles shows that a significant portion of the total proposed (and thus measured) candidates, exhibit comparatively high execution latency relative to their time of occurrence within the

auto-tuning period - i.e. it would be expected for the auto-tuning process to identify high-performance schedules more frequently as the auto-tuning progresses further. As such, these candidates do not meaningfully contribute to the optimisation progress, yet cost time and energy to evaluate. These candidates were labelled as *cold* candidates.

Table 3.4: Cold candidate impact on time and energy costs of auto-tuning across all studied platforms for ResNet-18 and VGG-16. Values calculated as an average of cold candidates across all operators per model. Auto-tuning until 500 hardware measurements were performed for each operator.

Model	**Platform**	**Number of cold candidates (out of 500)**	**Percentage of time spent on evaluating cold candidates**	**Energy spent on evaluating cold candidates (in kilojoules <kJ>)**
Resnet-18	A	133 ± 21	26.74 ± 4.3	52.1 ± 10.4
	B	134 ± 18	26.86 ± 2.8	36.7 ± 7.9
	C	130 ± 11	26.13 ± 2.2	37.1 ± 4.2
	D	139 ± 14	27.92 ± 2.9	9.0 ± 4.6
VGG-16	A	202 ± 29	40.51 ± 5.9	76.7 ± 2.0
	B	201 ± 28	40.24 ± 5.6	49.3 ± 2.4
	C	200 ± 29	40.16 ± 5.9	49.1 ± 4.2
	D	216 ± 25	43.33 ± 5.0	41.2 ± 3.1

To differentiate *cold* candidates from those that meaningfully contribute towards auto-tuning (*hot* candidates), LOESS regression [43] is applied, based on the latency of all successfully measured candidates within a DL model tensor operator auto-tuning session. The result of applying this method can be seen in Figure 3.6 for the traces of eight auto-tuning sessions using GA, RD, AT and GR auto-tuners for two operators of ResNet-18 (top row) and two operators of VGG-16 (bottom row) each.

It can be observed that different auto-tuners exhibit different patterns of *cold* candidate occurrences, as shown in Figure 3.6. For example, AT produces the highest number of *cold* candidates early-on, when the cost model is not yet initialised, whereas GR and RD produce more diffused *cold* candidate proposal patterns as they explore the schedule space in a brute-force manner.

3.4.3 Impact of cold candidates

Apart from restricting promising candidate proposals, *cold* candidates also contribute to almost 50.5% of total auto-tuning energy costs, as shown in Table 3.4 and Figure 3.7. This varies across auto-tuners, especially between AT and the other studied auto-tuners, where AT proposes fewer cold candidates in the latter stages of auto-tuning. Overall, *cold* candidates originating from AT constituted 17 - 38% of total auto-tuning energy costs, whereas for RD, *cold* candidates constituted nearly 80% of all costs.

This impact also varies depending on the operator. As shown in Figure 3.7, the *cold* candidates proposed for the first few (less computationally complex) operators of ResNet-18 have lesser energetic impact compared to operators 6 or 7 (more computationally complex). This is because a low-performance implementation for an operator with a higher number of FLOPs required during execution, is likely to consume more energy than a low-performance implementation for less a complex operator would. For more information on ResNet-18 architecture composition, see Appendix D.7. In terms of the choice of platform and target-device, intuitively, *cold* candidate impact was proportional to the platform capabilities, as shown in Table 3.4.

3.4.4 Converging Onto Hot Candidates

Observing candidate proposals for a single tensor operator across different auto-tuners, it becomes apparent they exhibit different convergence patterns as the auto-tuning progresses, as depicted in Figure 3.6 - signified by the *next-minimum-candidate Trend*. More complex auto-tuners (AT, GA) tend to discover satisfactory candidates relatively quickly within the auto-tuning process, whilst the brute-force approaches (GR, RD) discover the next-best candidate with some degree of probability that is dependent on the organisation of the schedule space rather than the auto-tuner's effectiveness.

It can be observed that all studied auto-tuners propose progressively better candidates

as the auto-tuning progresses, whether discovering them by chance or actively performing complex space exploration. Interestingly, a subset of low-latency candidates are usually discovered early on during auto-tuning, especially when using more complex auto-tuners (GA, AT). These candidates are hereafter referred to as *hot* candidates. A good example of hot candidate occurrence is the auto-tuning trace of AT for operator 7 of ResNet-18, depicted in Figure 3.6, where a high-performance candidate was discovered within the first 87 measurements out of 500 total. This candidate schedule exhibited performance only three to five percent slower than the globally best candidate (100ns difference), suggesting that an earlier selection of near-locally-optimal schedule, could have avoided 82.6% of the costs associated with this auto-tuning session.

3.5 Findings and Design Directions

The study examined both high-level and low-level (auto-tuning) DL inference optimisation methods identified several important design directions, that if explored, could lead to significant improvements in the efficiency of DL optimisation.

Insight 1 - *Understanding diversity of optimisation cost*: The interaction between compute platforms, target-devices, DL frameworks, models, high-level optimisations and low-level auto-tuner approaches, lead to unique optimisation cost profiles and resultant model performance profiles once optimised, whilst existing work focuses on reducing model inference costs on specific target devices [39, 8]. Additionally, it can be observed that there exists no single auto-tuner that can guarantee significant and cost-efficient model inference latency reduction across all models, platforms and target-devices. These insights can be advantageous when *determining model, target-device and auto-tuner combinations that successfully produce DL models with low inference latency.*

Insight 2 - *Avoiding erroneous and cold candidates*: All studied auto-tuners produced a number of erroneous candidates (due to invalid schedules or schedules that break

compute capability limits of the target-device). These erroneous candidates do not positively contribute to auto-tuning progress since they count towards the measurement count total that determines the duration of auto-tuning. This in turn prevents more favourable candidates from being explored instead, reducing the overall auto-tuning efficacy. Additionally, it can be observed that *cold* candidates exhibit high execution latency relative to their time of occurrence during auto-tuning[2]. *Cold* candidates do not positively contribute towards the auto-tuning progress, furthermore incurring additional time and energy costs during their selection, compilation and measurements. As such, a design direction for future auto-tuner development is to *focus on avoiding both cold and erroneous candidates, thus increasing cost-efficiency of auto-tuning, whilst maintaining efficacy and quality of proposed schedules.*

Insight 3 - *Taking advantage of hot candidates*: During analysis, it has been discovered that candidates exhibiting runtime latencies similar to that of a globally fastest candidate (for that operator), could be found relatively early during the candidate search. Within this theses, such candidates are referred to as *hot* candidates. It is generally accepted that the more extensive the candidate exploration, the higher the chance of discovering progressively better candidates. However, analysis suggests that continuing auto-tuning for prolonged periods of time, only to discover marginally better candidates further into the exploration, may not be cost-proportional. This causes auto-tuning to become less cost-efficient the longer it continues. As such, *it is possibly more cost-efficient to select a well-performing, hot candidate discovered early during auto-tuning.* Additionally, *the hot candidate quality and frequency of their occurrence could help to establish auto-tuning efficacy across different approaches*, especially that quality is influenced by the combination of the model characteristics, target-device capabilities and the chosen auto-tuner, as outlined in **Insight 1**.

[2]It is intuitive that candidates manifesting lower latency should be found more often as the auto-tuning progresses, as opposed to continuing to find high-latency candidates.

This page is left intentionally blank

Chapter 4

Trimmer: Cost-Efficient DL Auto-tuning

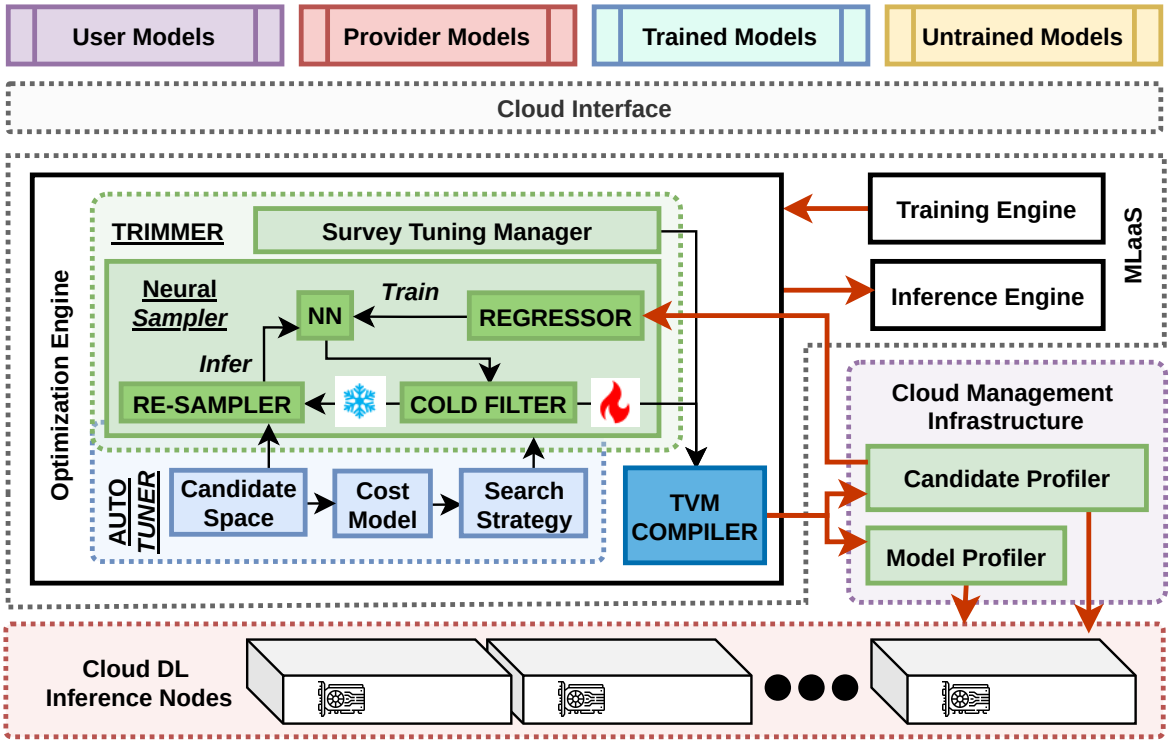


Figure 4.1: Trimmer system architecture

From the insights and design directions established within Chapter 3, there emerged the design of Trimmer - an auto-tuning framework focused on achieving cost-efficient DL model optimisation, by reducing the total time and energy costs of performing end-to-end DL model auto-tuning, whilst maintaining high optimisation quality. At the level of tensor operators and end-to-end DL models, Trimmer enhances the operation of the search algorithm and the multi-operator auto-tuning strategy. At the level of a single DL model layer tensor operator, Trimmer utilises an ANN to predict and exclude sub-optimal *cold* candidates. This enables the auto-tuner to explore the schedule space for more optimal *hot* candidates sooner during auto-tuning, accelerating optimisation convergence. At the end-to-end DL model level, Trimmer performs *Survey-tuning* to enable early auto-tuning completion based on real-time monitoring of model-level inference latency improvements. Overall, Trimmer is designed as a component within the MLaaS ecosystem, that increases cost-efficiency of optimising trained DL models to be deployed for inference by Cloud providers, as shown in Figure 4.1.

4.1 System Design and Implementation

4.1.1 Trimmer's Objective

Trimmer's core objective is to minimise DL model inference latency on a particular target-device. Given DL model inference latency f_t, Trimmer performs auto-tuning on each tensor operator $\{o_i \in O | i = 1 \cdots K\}$, where K is the number of unique tensor operators in the DL model. The amount of time dedicated to optimising each unique tensor operator o_i, depends on the rate of its execution latency improvement in relation to the other tensor operators within the DL model (o_j), as well as the overall improvement to the end-to-end DL model inference latency as a result of auto-tuning. More formally, for a DL model f and its inference latency f_t, Trimmer's goal is to optimise its individual

tensor operators at a low cost, such that a user-specified inference latency goal g is achieved, as follows:

$$\min \sum_{i=1}^{K} Cost_{auto-tune}(o_i) \quad \text{s.t. } f_t \leq g \tag{4.1}$$

where $Cost_{auto-tune}(o_i)$ represents the cost of auto-tuning tensor operator o_i. Trimmer's operation can be further constrained by additional objectives such as external cost functions specified by the user, as detailed in Section 4.1.3.1.

4.1.2 Cold Candidate Filtering

As outlined in Sections 2.6.2.3 and 2.6.2.2, auto-tuners utilise cost models and search algorithms to traverse candidate tensor program schedule space, in an attempt to discover parameters for schedule transformation that result in low execution latency of the resultant tensor program. During operation, the auto-tuner proposes candidate tensor program schedules to be compiled and executed on the target-device, to evaluate their execution latency. Within Chapter 3, it has been identified that a subset of these candidates are *cold* candidates that could be filtered to improve upon cost-efficiency of auto-tuning. Thus, Trimmer introduces a tensor operator-level, FC ANN-based *cold* candidate filter, that integrates with existing auto-tuning infrastructure (AutoTVM [40]) to remove *cold* candidates and replace them with new proposals, encouraging faster schedule space exploration and limiting impact of *cold* candidates on operational costs.

4.1.2.1 FC Network

The FC ANN network used to filter out *cold* candidates was designed with three layers, and leverages a ReLU activation function [217]. The network operates in two modes: (1) for training and regular inference, the network outputs predictions of execution latency for a given combination of schedule configuration and execution characteristics (outlined

Table 4.1: Details of the FC network architecture used by Trimmer to perform cold candidate filtering and re-sampling. Within the table, each model layer is represented as an x, y tuple where x is the layer's input size and y is the layer's output size.

Layer Type	**Layer Dimensions**
Input	$(\sum_{i=1}^{E} e_i, 32)$
*Embedding e_i	$(R_{space}, 10)$
Middle	(32, 32)
Output	(32, 1)

in the following section), and (2) during filtering, the network outputs middle-layer learned embeddings, which are then used to compare proposed candidate schedules to a historical database of embeddings. The network was implemented and trained using the Pytorch [1] DL framework. Listing G.1 found in Appendix G depicts the definition of Trimmer's FC ANN specified in the Pytorch specification format, while the high-level details of the network are provided within Table 4.1.

4.1.2.2 FC Network Inputs

As identified within Chapter 3, there are many characteristics of both the tensor operators, their schedule spaces, and the target-devices that may influence execution latency of a candidate tensor program, for a specific tensor operator and target-device combination during auto-tuning. These characteristics also have non-linear distributions which are challenging to leverage by more traditional methods such as Decision Trees [277]. As such, this non-linearity stands behind the choice of GBT-based cost models or other non-linear ML methods in SOTA DL auto-tuning, to traverse the candidate schedule space effectively. Taking into account the above assumptions, an input to Trimmer's filtering FC network consists of a numerical vector, composed of:

- *(1) Schedule Configuration:* Each entry represents a parameter used to configure a candidate tensor program schedule template, as described in Section 2.6.2.1. In

template-based auto-tuners such as AutoTVM [40], Chameleon [8] or AdaTune [183], template parameters consist of several unique values, such as: $tile\_x$, $tile\_y$, $auto\_unroll\_max\_step$ (See Table F.1 in Appendix F for details). During pre-processing, these were encoded as separate features.

- *(2) Hardware Features:* Characteristics of the target-device GPU. For example, the number of available cores, L1/L2 cache sizes, threads per warp.

- *(3) Tensor Operator CI:* This entry represents the complexity of the tensor operator expressed as total required FLOPs to be performed.

- *(4) Tensor Operator Features:* This entry represents a unique tensor operator identifier (for example, its class) and type, such as 1D, 2D, 3D Convolution.

- *(5) Encoded Tensor Operator Argument Features:* This set of entries encodes the representation of tensor operator parameters, for example, the *input/output shape*, *kernel size*, the amount of *input padding* or the *stride* during Convolution.

- *(6) Status of Measurement:* Given candidate tensor program measurements can be successful or failed due to instantiation errors or runtime errors, signifying an erroneous candidate schedule, these status features are taken into account within the model by encoding them as a unique integer value.

4.1.2.3 FC Network Training

The goal of the training pipeline for Trimmer's FC filtering ANN, was to learn the relationships between the tensor operator features, individual tensor program candidate configurations (schedule parameters) and the target-device characteristics, to then predict tensor program execution latency. During network training, the Mean Squared Error loss objective function was used, against candidate latency to determine loss, which was then leveraged to update layer weights during backpropagation. In terms of the

training dataset, it consisted of 20,000 individual candidate measurements, collected as part of the study documented within Chapter 3. The dataset was prepared by splitting it into three disjoint sets: *training* (70%), *validation* (10%) and *testing* (20%). The validation set was leveraged to adjust (tune) hyperparameters, following established methods and guidelines from research focused on predicting tensor program latency [376]. The training pipeline was configured to perform ten training epochs on the FC network, with the learning rate of $1e^{-3}$ (min $1e^{-9}$). To adjust the training process, Adam optimiser [157] with Plateau Patience hyperparameter of one was used, utilising batch size of 1024 simultaneous samples during forward-backward pass of the network.

Algorithm 1: Trimmer's Cold Candidate Filtering

Input: $Candidates$, ϵ, K, $Model$, $costModel$

begin
  // Batched inference using sample candidates proposed by the cost model
  $embeddings$ = NNPredict($Model, Candidates$, mode="filter")
  // For each candidate task's embedding
  $totalRemovedCount = 0$ **for** $e_i \in embeddings$ **do**
    $rand$ = random()
    // ϵ is an increasing parameter
    if $rand < (1 - \epsilon)$ **then**
      // Retrieve K similar candidates based on embeddings
      C_{sim} = similarInDatabase(K, e_i)
      // Check how many of the identified similar candidates are cold candidates
      $N_{sim.cold}$ = howManyAreCold(C_{sim})
      if $N_{sim.cold} == K$ **then**
        // Remove sample from the candidate set - removal is based on index $e_i == c_i$
        remove($Candidates$, c_i) $totalRemovedCount$ += 1
      end
    end
  end
  $Candidates^*$ = $Candidates$ + reSample($costModel$, $totalRemovedCount$)
  updateEpsilon(ϵ)
  return $Candidates^*$
end

4.1.2.4 Utilising the FC ANN for Filtering

Whilst trained to predict candidate tensor program execution latency, the network is utilised differently during *cold* candidate filtering within Trimmer. During training and regular inference, the network's forward pass outputs predictions of tensor program execution latency, based on inputs that specify candidate schedule parameters and the execution environment such as the target-device characteristics.

As depicted in Algorithm 1, during candidate filtering ("filter" mode), the ANN outputs the network's middle-layer embeddings - data structures commonly used in NLP tasks [309] to process input data with many features. Trimmer leverages embeddings by treating the input data as bag-of-words samples (indifferent of their order within the sample vector). These embeddings are then used to compare proposed candidates with an offline embeddings database extracted from the training dataset.

Comparison of embeddings is performed by measuring *cosine distance* - a frequently used method to identify samples that share characteristics within a vector space [300]. Top-K similar candidates are identified during this process based on their embeddings, for each candidate proposed by the auto-tuner. If embeddings identified to be similar, were previously associated with candidates that were identified as *cold* candidates, the currently examined candidate schedule is removed from the input candidate set.

To ensure an adequate number of candidates are measured on the target-device, the module re-samples $totalRemovedCount$ new candidates by re-querying the auto-tuner cost model. This updated candidate set is sent for measurement, and the process repeats for the subsequent batches of proposed candidate schedule configurations.

4.1.2.5 Exploration vs. Exploitation

From the cost-efficiency perspective, it is more desirable to reduce the time spent on measuring costly *cold* candidates. Inversely, from the perspective of the optimiser,

sampler and the cost model within an auto-tuner, it may be beneficial to explore many different points within the candidate schedule space to identify promising template parameter sets. As such, there exists a trade-off between reducing the cost of evaluating *cold* candidates, and allowing candidate discovery procedure to explore the schedule space in an unconstrained manner.

Trimmer balances this trade-off by leveraging an inverse ϵ-greedy policy, where the filtering of sampled schedules is encouraged within the early iterations of the auto-tuning session, whilst exploration is encouraged later on, as the auto-tuning progresses. To encourage exploration, the probability of candidate filtering decreases after each batch of candidates is measured and once the cost model has been updated - see line 5 of Algorithm 1 for details. This results in incentivisation of candidate exploration rather than filtering as soon as the optimiser and auto-tuner cost model become stable.

Comparing Trimmer's approach to prior work, Chameleon's sampling module [8] leverages the K-means [110] algorithm to filter out similar candidates, thus allowing the AutoTVM schedule space exploration algorithm (Simulated Annealing) to explore more candidate schedules within the schedule space quicker. Contrastingly, Trimmer heuristically prunes away candidates if and only if top-K similar configurations all exhibit high execution latency (for example, manifest *cold* candidate execution patterns). Since Trimmer prunes candidate tensor programs likely to exhibit high execution latency, the filtered candidates are replaced with new candidates by querying the auto-tuner cost model, further ensuring sufficient exploration of the candidate schedule space.

This also ensures coherence with the auto-tuner measurement infrastructure, which expects a given batch size of candidate tensor programs to be measured. By combining both the probabilistic and heuristic strategies, Trimmer accelerates exploration of the candidate schedule space, whilst limiting the number of *cold* candidates that need to be measured on the target-device, ultimately increasing the overall cost-efficiency of auto-tuning as lesser number of poor candidates are evaluated on the target-device.

4.1.3 Survey Tuning

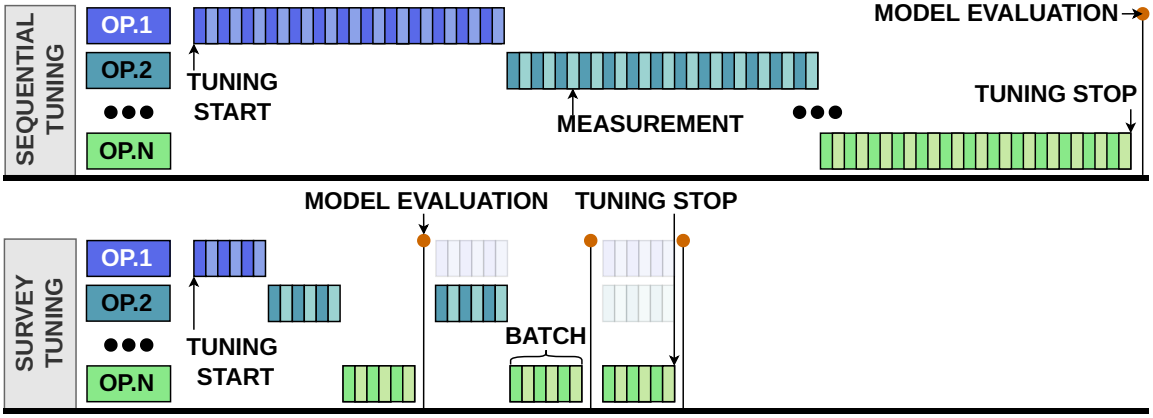


Figure 4.2: Differences between Sequential and Survey tuning. [OP = Tensor Operator]

Existing DL auto-tuners, optimise end-to-end DL models by extracting all tensor operators that are compatible with the auto-tuner, followed by sequential auto-tuning of each extracted tensor operator to completion. More specifically, each tensor operator auto-tuning session must be completed before auto-tuning of the next tensor operator can commence, as depicted in Figure 4.2. The duration of each tensor operator auto-tuning session depends primarily on the number of candidate tensor program measurements performed on the target-device - typically chosen by the user.

Considering the above constraints, within existing DL auto-tuning frameworks, end-to-end inference latency of a DL model being optimised can only be ascertained once the entirety of the individual auto-tuning sessions have been completed. During this often prolonged (tens of hours) process, the user has no knowledge of the impact of individual tensor operator auto-tuning on the global, end-to-end DL model inference latency. It is also unclear which tensor operators should be optimised to achieve the most significant end-to-end DL model inference latency improvement. Moreover, it is uncertain how this improvement compares to auto-tuning sessions of other tensor operators within the same model.

Algorithm 2: Trimmer's Survey Tuning

```
   Input: Model, B_size
1  Class SurveyManager is
2      Method update(b, lat) is
3          if b.lastBestLatency > lat then // reset plateu on this batcher
4              b.plateauCount = 0
5              b.lastBestLatency = lat
6          else
7              b.plateauCount += 1
8              if b.plateauCount > Op_plateau.thresh then
9                  b.shouldStop = true
10             end
11         end
12     Method step(modInfLat) is
13         infLatHistory.add(modInfLat)
14         if length(infLatHistory) < H_window then // window not large enough, continue
15             return false
16         else if length(infLatHistory) > H_window then // window too large, reduce
17             infLatHistory.popFromFront()
18         end
19         if mean(infLatHistory) > lastAvgHistory then
20             if Mod_plateau > Mod_plateau.thresh then
21                 return true
22             end
23             Mod_plateau += 1
24         else
25             Mod_plateau = 0
26             lastAvgHistory = mean(infLatHistory)
27         end
28         if Objective then // If user specified custom objective
29             return eval(Objective, mean(infLatHistory), modInfLat)          // Eq. 4.1
30         else if min(infLatHistory) < G_lat then
31             return true
32         end
33         return false
34 begin
35     for o ∈ Model.operators do
36         batchers.add(SurveyBatcher(CreateAutoTuner(o), o, B_size)
37     end
38     mgr = SurveyManager(batchers)
39     while not mgr.hasFinished() do // Begin epoch
40         for b ∈ mgr.batchers do // iterate through all tensor operator batchers
41             if not b.shouldStop then
42                 n = b.getNextBatchSize()
43                 bestFoundLat = autoTune(b.autoTuner, b.operator, bSize=n)
44                 mgr.update(b, bestFoundLat)        // update best-found candidate for the batcher
45             end
46         end
47         recs = mgr.getBestAutoTuningRecordsSoFar()
48         modInfLat = measure(compile(model, recs))              // evaluate model inference latency
49         if mgr.step(modInfLat) then // mgr.step() decides whether to continue auto-tuning
50             return
51         end
52     end
53 end
```

Furthermore, given the occurrence of *hot* candidates within early stages of auto-tuning (See insights derived within Chapter 3), it is unknown ahead of time whether auto-tuning each tensor operator to completion is cost-efficient, and whether it could have been more efficient to suspend select tensor operator auto-tuning sessions early[1]. Being able to leverage this information during auto-tuning automatically, would enable to focus more time on auto-tuning tensor operators that result in more significant end-to-end DL model inference latency improvement compared to others, ultimately leading to a more cost-efficient overall process. Trimmer leverages these insights to implement the concept of *Survey tuning*, as depicted in Figure 4.2. Survey-tuning is designed to operate both at the individual tensor operator and the DL model levels.

4.1.3.1 Tensor Operator-level Survey Tuning

Trimmer's Survey tuning approach is designed around the idea of a Survey Manager and a set of Survey Batchers managed by it, as depicted in Algorithm 2. At the tensor operator level (i.e. auto-tuning a single end-to-end DL model), Trimmer extracts all optimisable tensor operators and allocates a Survey Batcher per tensor operator to maintain auto-tuning state and manage batched optimisation of that particular tensor operator. Each Batcher maintains information about how many measurements have been performed in relation to its managed tensor operator and how many more are left (for example, due to hard measurement limits imposed by the user).

Partial Auto-tuning: Once initialised, Trimmer's Survey tuning module begins iterative optimisation of each tensor operator, splitting auto-tuning into *epochs*. During each auto-tuning epoch, Trimmer performs partial auto-tuning of each tensor operator, by allowing the auto-tuner to propose up to B_{size} candidate tensor programs for measurement. Within Trimmer, B_{size} is by default set equal to the *planSize*

[1]Existing auto-tuners provide means to stop auto-tuning early, however, the threshold of observed improvement is set arbitrarily by the user, and has to be decided manually for each individual tensor operator

hyperparameter of an auto-tuner (64 in AutoTVM and Chameleon), which specifies the number of candidate proposals and measurements, after which the cost model will be updated and a new set of candidates proposed. The B_{size} parameter is kept equal to $planSize$ in order to ensure Survey tuning does not interfere with DL auto-tuner cost model updates, however, it can be freely modified by the user if required.

Tensor operator Early-stopping: Once partial auto-tuning is performed for a given tensor operator, the best (lowest latency) candidate found within the current epoch is selected and used to update its Batcher's internal state (by calling its *update()* procedure - see. line 43 in Algorithm 2). During Batcher update for the current epoch, the current best found latency is compared with previous epoch's lowest latency candidate to establish whether the auto-tuner is making good progress at reducing the particular tensor operator's execution latency. In case the new best latency candidate is found to be worse than the one discovered within the last epoch, a $plateauCount$ is raised. Optimisation of this tensor operator will be early-stopped if $plateauCount$ exceeds $OP_{plateau.thresh}$ - set to 2 in Trimmer based on empirical findings. In other words, if the tensor operator does not exhibit latency improvement in three consecutive epochs, its optimisation will be halted to dedicate more time and resources towards other tensor operators. The epoch continues until all such partial auto-tuning sessions are performed for all the remaining tensor operators of the DL model.

End-to-end Model Early-stopping: At the end of each epoch, for each of the tensor operators, Trimmer extracts the best performing candidate tensor program found so far, and applies the schedule configurations to the end-to-end DL model. Configured model is then compiled and executed on the target-device to determine its inference latency ($modInfLat$). The $modInfLat$ is then used to perform an epoch step via the Survey Manager, which results in a decision whether to continue auto-tuning of the end-to-end DL model. During the *step()* procedure (see lines: 12 - 32 in Algorithm 2), the Survey Manager maintains a window of H_{window} epochs across which the end-

to-end DL model inference latency is compared. Initially, current epoch's end-to-end DL model inference latency is appended to a list of historical latency records. Then, an updated historical average latency $mean(infLatHistory)$, across H_{window} of prior latencies is compared with prior average latency, establishing whether the end-to-end DL model latency has improved as a result of the current epoch's partial auto-tuning sessions. If no such improvement is observed, the $Mod_{plateau}$ count is raised to reflect potentially plateauing optimisation. If the $Mod_{plateau}$ count exceeds $Mod_{plateau.thresh}$, the optimisation is halted. In Trimmer, H_{window} was set to 3 whilst $Mod_{plateau.thresh}$ to 2, based on empirical findings.

Objectives: Optimisation of the DL model can also be controlled by user-defined goals and cost functions. Trimmer utilises the achieved $min(infLatHistory)$ end-to-end inference latency measure to determine whether it has fallen below the goal threshold G_{lat}, in which case optimisation is halted for that DL model. Alternatively, a cost function (in the form of an external evaluating procedure) can be specified, in which case the Survey tuning module consults the cost function to determine whether optimisation should be halted - see line 29 of Algorithm 2. By default, Trimmer's Survey tuning module compares the rate of end-to-end DL model inference latency reduction across the current and prior Survey tuning epoch, as follows:

$$\begin{aligned} stop &= \omega_1 \frac{x_t + x_{t-1}}{x_{t-1}} \leq \phi \\ x_t &= \omega_2 \frac{\delta f_t}{\delta O_t} \end{aligned} \tag{4.2}$$

Where the difference in end-to-end DL model inference latency f between batch intervals t and $t-1$ is represented by x_t, as a ratio over the optimisation time cost O_t for the auto-tuning epoch, multiplied by configurable weight ω_2. Trimmer increases efficiency of auto-tuning by determining how significantly the DL model inference

latency is reducing on every epoch and at what rate, adjusting the optimisation process accordingly. ϕ is a configuration parameter that helps the policy to avoid unnecessary auto-tuning, by signalling when the ratio since the previous epoch becomes smaller than the parameter ϕ. This would occur when the change in model inference latency between epochs is insignificant. The ϕ option enables prioritisation of performance improvement or operational cost. Trimmer sets ϕ to -0.25 to avoid unnecessary time spent on optimising poorly improving operators. This setting was determined empirically.

To account for the cases where x_t and x_{t-1} are slower (+ive), and cases of x_t being quicker but x_{t-1} being slower, ω_1 is set to -1 and otherwise set to 1. Within Trimmer, ω_2 is set to -1 if both δf_t and δO_t are -ive, and set to 1 otherwise. These phenomena may occur when the space of candidate schedule is significantly non-linear, potentially resulting in rapid changes in measured candidate latency.

4.1.3.2 DL Model-level Survey Tuning

The aforementioned early suspension approach is also applied when optimising multiple models simultaneously, since Trimmer measures end-to-end DL model latency at each Survey tuning batch. This achieves *meta-tuning*, where a relative speedup (decrease in end-to-end inference latency) of DL models is compared to the speedup achieved by concurrently executing auto-tuning sessions across other models.

Taking inspiration from population-based DL model training [142], Trimmer ranks models using the measure of improvement to their inference latency at the end of a partial Survey tuning batch, accomplishing meta-tuning across multiple models. Models performing worse in comparison to others are suspended. The goal of this strategy is to focus optimisation efforts onto the most quickly improving DL model and auto-tuner combinations, when considering multiple of such pairs within a cluster, the user-set criteria for inference latency thresholds or cost objectives. Similar to single-model Survey tuning, the suspension criteria include configurable plateau iterations. Trimmer

synchronises auto-tuning sessions across multiple models and machines using RPC, and does so when a batch of measurements are completed for each model, allowing a short period of conventional auto-tuning before comparison, to initialise cost models.

4.2 Experiment Setup

To evaluate Trimmer's ability to increase cost-efficiency of auto-tuning, both at the level of a single DL model auto-tuning and across multiple models in a Cloud cluster scenario, experimentation was performed using several prominent DL models, comparing Trimmer to SOTA auto-tuners across different hardware platforms. Trimmer has been integrated partly as a module of the existing AutoTVM auto-tuner infrastructure and as a layer above the auto-tuner (Survey tuning) to manage multiple auto-tuning sessions.

4.2.1 Hardware, Software and Middleware

During experiments, Trimmer performed auto-tuning towards Nvidia GPUs situated within distinct hardware platforms as oulined within Table 3.2 found in Section 3.1.2 of Chapter 3. Each of the platforms was provisioned with identical Operating System and middleware, much like in the case of the experimentation performed as part of Chapter 3. For experiments involving multi-platform (distributed system), multi-model Survey tuning, Docker [134] and Docker Swarm [135] version 20.10.7 were used to deploy Trimmer and facilitate communications between instances.

4.2.2 Auto-tuners

Trimmer was evaluated against three SOTA auto-tuners: (1) *AutoTVM* (AT) [40], which utilises a GBT-based cost model to propose candidate schedule configurations and SA-based optimiser to traverse the schedule space using the feedback from the GBT cost model; (2) *RL* (RL) [8] - an auto-tuner which also leverages the GBT-based model,

however, replaces the SA-based optimiser with one based on PPO; and (3) Chameleon (CH) [8], which extends the RL auto-tuner with a K-means clustering sampler, to reduce the number of similar candidates being explored and improve optimisation quality.

4.2.3 Workloads

Experimentation performed as part of Trimmer's evaluation involved auto-tuning four distinct CNN DL models (AlexNet [168], MobileNet [121], VGG-16 [304] and ResNet-18 [113]), as previously shown in Table 3.1. Each model receives an input tensor for inference prediction of the shape of $1 \times 3 \times 224 \times 224$, with tensor layout of NCHW, and outputs a 1×1000 tensor containing predictions. Models range in the number of parameters and computational complexity, and were converted into TVM format from the Pytorch and Apache MXNet DL frameworks. Once converted into the TVM's Relay format, each model has been transformed using the graph-level optimisation infrastructure, applying level 3 optimisations, following prior work guidelines to avoid approximation of parameter values that occurs when applying level 4 graph optimisations. During experiments, only tensor operators compatible with the studied auto-tuners were extracted and auto-tuned - subject to limitations posed by TVM's TOPI template repository. For more information about graph-level optimisations and optimisation levels, see Section 2.5.3 as well as Table E.1 within Appendix E, whilst further details on the model architectures can be found within Appendices D.1, D.3, D.8 and D.7.

4.2.4 Collected Metrics

Several metrics were used to quantify Trimmer's effectiveness at increasing cost-efficiency of auto-tuning. End-to-end model latency was captured prior and post auto-tuning to determine optimisation quality, expressed as a raw measurement of execution latency (ms) or speedup compared to an un-optimised model. Total auto-tuning time and platform

energy consumption were also measured to determine operational costs of performing auto-tuning with Trimmer vs. other SOTA auto-tuners. Additionally, average CPU and GPU utilisation and memory usage were also collected during auto-tuning. Details about the specific metrics collection methods can be found in Section 3.1.6.

4.2.5 Experiment Scenarios

Experiments evaluating Trimmer consisted of several scenarios. Initially, Trimmer was evaluated in a *Single-platform* scenario where DL models were auto-tuned end-to-end, utilising isolated target-devices to perform candidate tensor program measurements. Within these experiments, Trimmer was compared to other DL auto-tuners that also utilised the same workload, target-device and platform. Further experiments followed the *Cloud-cluster* scenario where four auto-tuning instances were coordinated across four individual (platform A) machines, comparing Trimmer's Survey auto-tuning with performing Sequential auto-tuning concurrently at a model level. Trimmer assumes access to an offline repository of historical optimisation data, used to train its FC network. In cases where no such data is available, auto-tuning can be performed to populate such database with candidate measurement samples.

4.3 Evaluation Results

4.3.1 Single Platform, Sequential Auto-tuning

4.3.1.1 Achieved Inference Latency

As presented within Table 4.2 and Figure 4.4, Trimmer achieved on average greater reduction of inference latency when auto-tuning end-to-end DL models, compared to other auto-tuners. For AlexNet, VGG-16 and MobileNet, Trimmer achieved the lowest end-to-end DL model inference latency of 0.79ms, 4.68ms and 0.65ms respectively.

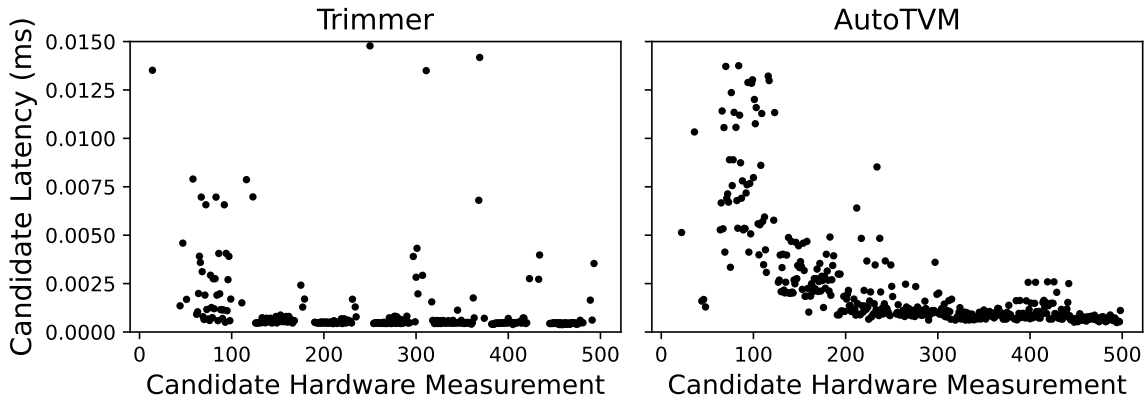


Figure 4.3: Comparison of candidate tensor program latency patterns as a result of auto-tuning tensor operator 5 of the VGG-16 model using Trimmer and AutoTVM.

Table 4.2: Achieved inference latency, auto-tuning time and energy costs incurred during auto-tuning (Single Platform)

	DL Model	**Trimmer**	**AutoTVM**	**RL**	**Chameleon**	**Base.**
Latency (ms)	Alexnet	**0.79**±0.02	**0.85**±0.05	**0.84**±0.09	**0.82**±0.08	4.42
	VGG-16	**4.68**±0.49	**4.78**±0.28	**5.85**±0.45	**5.83**±0.38	9.66
	Mobilenet	**0.65**±0.03	**0.67**±0.02	**0.76**±0.06	**0.74**±0.05	1.24
	ResNet-18	**1.39**±0.29	**0.86**±0.08	**1.11**±0.06	**1.03**±0.08	8.48
Tuning time (m)	Alexnet	**119**±0.29	**116**±0.25	**121**±0.42	**127**±0.40	
	VGG-16	**194**±0.49	**207**±0.22	**296**±0.98	**298**±0.30	
	Mobilenet	**213**±0.48	**286**±0.52	**216**±0.58	**214**±0.42	
	ResNet-18	**228**±0.24	**401**±0.59	**353**±0.74	**279**±0.44	
Tuning energy (MJ)	Alexnet	**1.6**±0.22	**1.9**±0.22	**2.3**±0.46	**2.4**±0.50	
	VGG-16	**3.4**±0.59	**3.4**±0.21	**5.5**±1.19	**5.6**±0.32	
	Mobilenet	**3.6**±0.81	**4.5**±0.48	**3.6**±0.58	**3.6**±0.41	
	ResNet-18	**21.2**±1.6	**26.6**±2.4	**29.5**±3.4	**31.8**±3.1	

Specifically for ResNet-18, Trimmer achieved results similar to Chameleon and RL. This is because the *cold* candidate filter employed by Trimmer, apart from filtering out poor candidates, enables rapid identification of more favourable schedules during the early phases of auto-tuning. This can be observed within Figure 4.3 where trimmer almost immediately explores high-quality candidate tensor programs, whereas approaches such

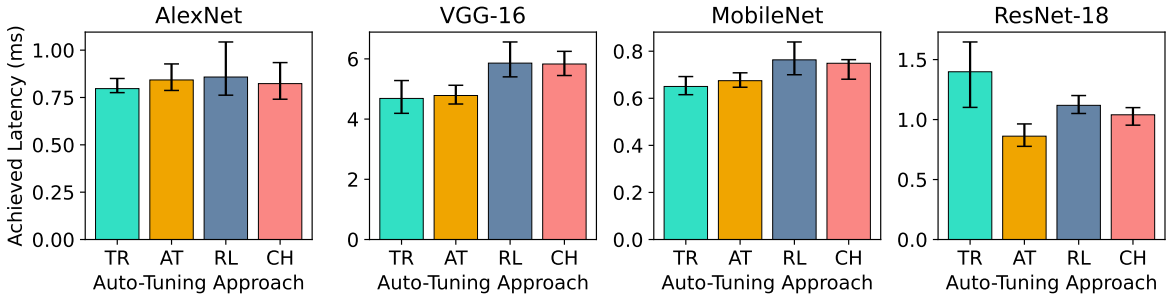


Figure 4.4: Achieved inference latency (ms)

as AutoTVM require some time to begin finding low latency schedules. Inadvertently for ResNet-18, Trimmer discovered good candidates early on during auto-tuning, potentially omitting slightly better ones that could have been found later on.

4.3.1.2 Auto-tuning Cost Efficiency

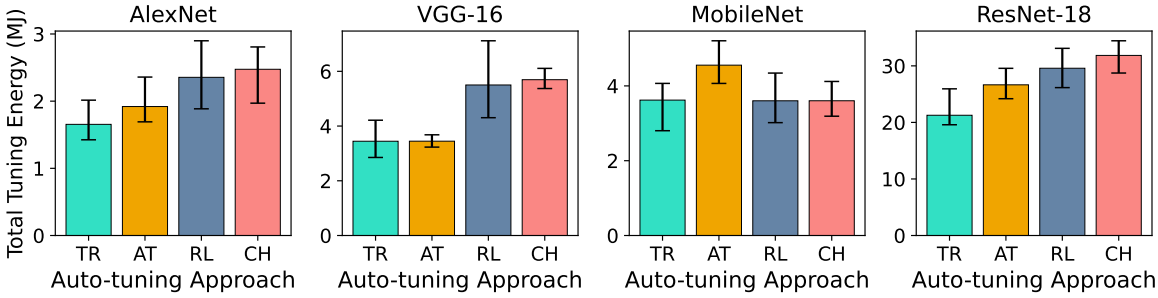


Figure 4.5: Total auto-tuning energy consumption (MJ)

As shown in Table 4.2, Trimmer incurred the lowest auto-tuning time cost compared to the other studied auto-tuners for the ResNet-18 and VGG-16 models, whilst performing comparably with the fastest auto-tuner for the remaining two models (within margin of error). In terms of energy costs of auto-tuning, Trimmer's energy consumption when auto-tuning DL models was lower, compared to the other auto-tuners for AlexNet and ResNet-18. In the cases of the remaining two DL models, Trimmer scored on par with the least energy-hungry framework (AutoTVM), as depicted in Figure 4.5.

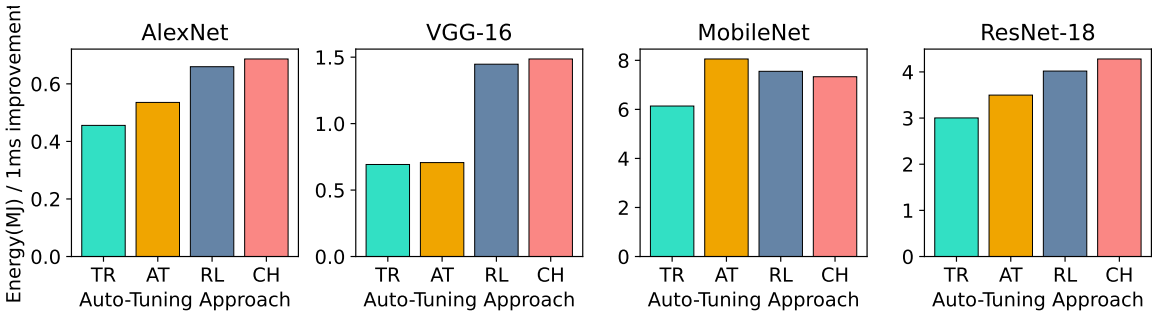
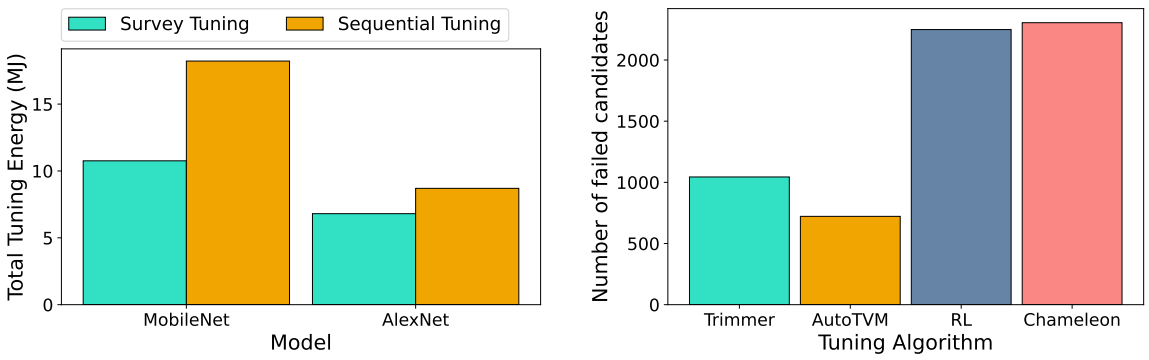


Figure 4.6: Average energy costs (measured in MJ) of auto-tuning that were incurred to achieve inference latency reduction of one millisecond across different DL models and auto-tuners, expressed as a ratio

However, crucially, both the time and energy costs should be understood in terms of cost-efficiency during auto-tuning. In other words, an auto-tuner would not be considered cost-efficient when it results in slightly lower inference latency, however, incurs significantly higher time or energy costs to achieve it. As depicted in Figure 4.6, within all experiments, Trimmer auto-tuning exhibited the best cost-efficiency, understood as: *"the energy cost of achieving a 1ms improvement to model execution latency via auto-tuning"*. Trimmer exhibited an average improvement in cost efficiency over other auto-tuners, between 14 and 29% for AlexNet, 2 and 54% for VGG-16, 16 and 24% for MobileNet, and 14 and 29% for ResNet-18, attributed to the candidate filtering and early finish when insubstantial inference latency improvement is detected.

4.3.1.3 Candidate Measurement Composition

Trimmer's evaluation suggests that when guided by Trimmer's *cold* candidate filtering, an auto-tuner's cost model proposes on average fewer globally poor candidates as well as a lesser number of erroneous candidates that fail during compilation or measurement, compared to the RL and Chameleon auto-tuners, as depicted in Figure 4.7b. Trimmer's re-sampling explores on average more total candidate schedules compared to AutoTVM, as during re-sampling, Trimmer filters out *cold* candidates from the batch currently



(a) Total tuning energy

(b) Total failed candidates

Figure 4.7: Total energy cost (MJ) & failed candidate measurements, when auto-tuning VGG-16 towards the GPU in Platform A – Trimmer's Survey tuning vs. Sequential tuning in parallel

proposed by the auto-tuner, whilst reintroducing a replacement batch of new candidate schedules by greedily querying the cost model. This in effect diversifies the measurement batch. With higher diversity of explored candidates compared to the filter-less AutoTVM auto-tuner, Trimmer inadvertently explores a wider variety of candidates, including erroneous ones. The additional processing of the Trimmer's ANN-based candidate filtering, results in a 3% increase of CPU utilisation compared to AutoTVM, which given the improvements to the overall cost-efficiency achieved by Trimmer is acceptable.

4.3.2 Cloud Clusters

Table 4.3: Comparison of Survey tuning and parallel model auto-tuning in a cluster scenario

	Model	**Survey**	**Parallel**	**Improvement**
Tuning energy (MJ)	MobileNet	10.76	18.21	40.9%
	AlexNet	6.80	8.70	21.8%
Model latency (ms)	MobileNet	0.629	0.684	8%
	AlexNet	0.797	0.805	1%

When performing model-level Survey tuning across multiple target-devices, Trimmer achieved between 21.8% and 40.9% improvement in total energy cost incurred during

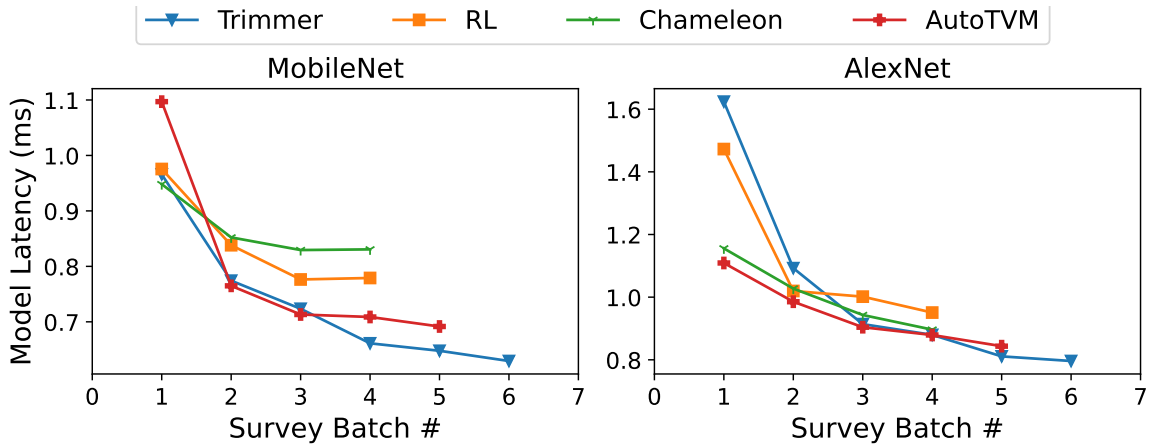


Figure 4.8: Achieved DL model inference latency after Survey auto-tuning. Results depict achieved end-to-end DL model inference latency at different batches (1 batch = 64 measurements)

optimisation, compared to performing parallel auto-tuning for AlexNet (8.70MJ to 6.80MJ) and MobileNet (18.21MJ to 10.76MJ) respectively, as shown in Figure 4.7a.

This reduction in energy consumption during auto-tuning stems from early-suspension of optimisation once insufficient end-to-end DL model inference latency reduction is observed during Trimmer's Survey tuning. This occurs both at the level of an individual model's tensor operators as well as at a multi-model level.

A major feature of Survey tuning implemented in Trimmer, is that it permits well-performing[2] combinations of DL models and auto-tuners to conduct additional partial Survey tuning batches compared to less-effective combinations, thus achieving better cost-efficiency. This can be observed in Table 4.3, which demonstrates that especially for MobileNet (0.684ms), Trimmer achieved inference latency 8% faster compared to AutoTVM permitted to continue auto-tuning until user-set threshold of candidate measurements is surpassed (i.e. completion). When Survey tuning is employed to evaluate the end-to-end DL model inference latency at Survey batch intervals, different auto-tuners can be observed to exhibit different patterns of achieved model latency, in scenarios of auto-tuning both MobileNet and AlexNet, as depicted in Figure 4.8.

[2]Exhibiting substantial improvement to end-to-end DL model latency

For MobileNet, all auto-tuners resulted in a sizeable end-to-end DL model latency improvement within the period of time between Survey batch 0 and 1 (between 10.1 and 30.2% reduction in latency). Chameleon and RL auto-tuners were stopped earlier than other scenarios as they have not exhibited sufficient improvements to the end-to-end DL model inference latency at Survey Batch 3. This suspension occurred both due to gradient convergence (at single model level) and in relation to other simultaneously executing auto-tuning sessions. AutoTVM was early-stopped at Survey Batch 5 as it achieved insignificant DL model inference latency improvement across two Survey batch intervals consecutively. This improvement was also smaller compared to Trimmer, and as such, the Survey tuning module decided to suspend it.

Overall, the evaluation results suggest that Trimmer is a more cost-efficient auto-tuner compared to standalone AutoTVM or Chameleon, as it allows the schedule space search algorithm (for example, SA) to greedily explore more candidate schedules, whilst incurring the same operational cost, and in many cases, achieving superior optimisation quality. This ultimately achieves a more cost-appropriate end-to-end DL model inference latency optimisation. This is further improved by the Survey tuning strategy at the tensor operator and single/multi-model levels. Leveraging this meta-strategy, Trimmer exhibits on average faster auto-tuning compared to conventional auto-tuning, which awaits completion of all tensor operators before testing end-to-end model latency, and faster auto-tuning when considering multiple models in a Cloud cluster setting.

4.4 Discussion and Limitations

4.4.1 Auto-tuner Compatibility

Trimmer's ANN-based filtering module is compatible with any AutoTVM-derived auto-tuners that leverage schedule templates and a cost model (see Section 2.6.3.1 for

examples). In cases where the templates utilise different schedule parameters to those supported by default in TVM's TOPI repository, the user can re-train Trimmer's FC ANN model with a dataset containing the alternative parameters as new features. To improve compatibility with autoschedulers or hybrid auto-tuners, Trimmer's filter and re-sampler could be adapted to support encoded schedule transformation rules / steps alongside template parameters during the FC model training and embeddings extraction.

Trimmer's Survey tuning is compatible with any auto-tuner that performs batched candidate proposals (for example, 64 at a time in TVM-based auto-tuners). This is a very common approach in all cost-model / search algorithm-based auto-tuners since the cost model must be periodically updated to improve search efficacy, and as such, requires iterative measurement result feedback.

4.4.2 Target-device Compatibility

Trimmer has been evaluated on Nvidia GPUs, and leverages their characteristics as input data features to the FC ANN. It is highly-likely that the same process could be adopted when auto-tuning towards non-Nvidia GPUs or CPUs, since their characteristics could also be used as input data features.

4.4.3 FC NN Model Training

One of Trimmer's limitations is the need for model training to prepare the filtering module for its operation. This design decision was made to account for the diverse environments in which Trimmer may operate (tensor operator classes, target-devices), enabling the user to adjust and fine-tune Trimmer's model to their specific needs. Prior candidate measurement data is typically easily accessible, especially given Trimmer's adopters are assumed to be prior practitioners of conventional auto-tuning, which produces measurement log files as a result of its operation. Furthermore, the TVM publishes an

online repository (TopHub [80]) of prior measurement data, containing measurement logs for the best found candidates for select combinations of tensor operators and target-devices. This could be used to supplement other available candidate measurement logs to enhance diversity of the training datasets. Due to its relatively low complexity (3× layers with small input/output dimensionality), Trimmer's FC ANN model training is a quick procedure (< 10m given 20,000 input samples obtained during exploratory study - see Chapter 3), and is performed as a one-off task ahead of auto-tuning multiple end-to-end DL models. Trimmer's model training can be completely avoided in cases of time constrained adoption by training the model online as the auto-tuning progresses, leveraging a limited number of *cold* candidate samples.

4.4.4 Workloads compatibility

During Trimmer's evaluation, CNN DL models have been used as the primary workloads to be optimised. Utilising CNNs to evaluate auto-tuners is a common practice [40, 8, 385, 183, 386] due to the occurrence of high-complexity Convolution tensor operators within these types of networks. Auto-tuning other workload types such as RNN or Transformer model tensor operators could also be achieved in Trimmer, by adopting the template parameters for these tensor operators as the ANN input features. When investigated, templates for such workloads could not be found within the online repositories.

4.4.5 Scalability

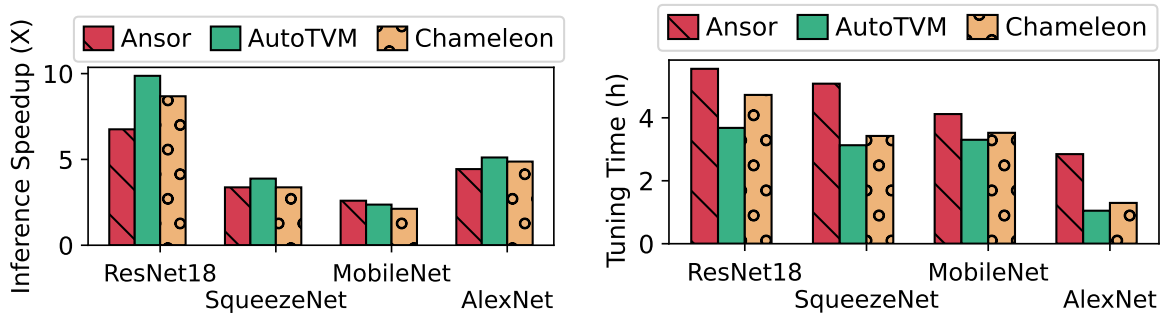
Trimmer's multi-model Survey tuning was evaluated on a cluster of four machines, each containing one GPU. It is unclear how utilising a larger cluster of machines (100s - 1000s) would impact Trimmer's efficacy of improving auto-tuning cost-efficiency. One approach to limit such potential impact would be to partition a larger cluster into smaller sub-clusters and manage multiple Trimmer instances separately.

This page is left intentionally blank

Chapter 5

A Naïvely-parallel Approach to Reduce DL Auto-tuning Costs

5.1 Overview



(a) Relative inference latency improvement from models compiled with default tensor operator schedules.

(b) Overall auto-tuning time cost (hrs) across different auto-tuners and models.

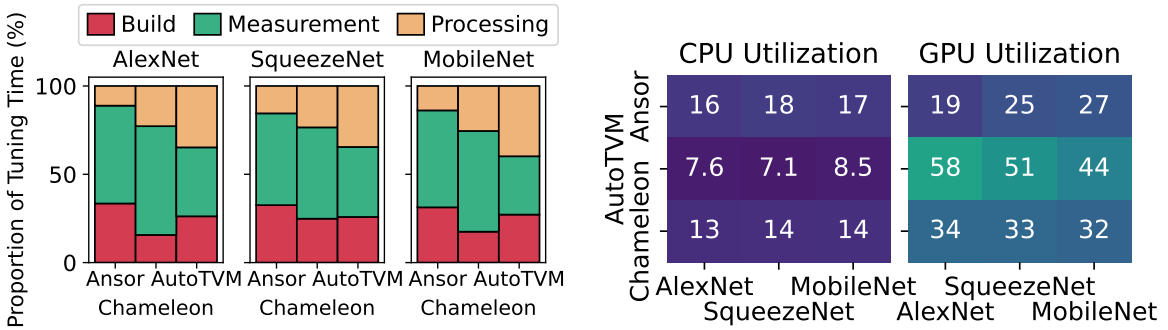
Figure 5.1: Auto-tuning four DL models with three DL auto-tuners towards Nvidia Volta V100 GPU target-device until 500 hardware measurements are performed.

As identified in Chapters 3 and 4, DL model auto-tuning via DL compilers can yield substantial reduction in model inference latency, however, it is a costly process often requiring hours of iterative search to obtain satisfactory performance improvements. Figure 5.1 depicts results (achieved inference latency speedup) and costs (auto-tuning

time) of performing auto-tuning for four prominent DL models towards Nvidia V100 GPU, utilising different auto-tuners (AutoTVM [40], Chameleon [8] and Ansor [385]).

It can be observed that different auto-tuners produce different performance improvement and incur varied time cost when paired with different models. At the algorithmic level, this stems from the varied abilities of auto-tuners to search through a large candidate schedule space, in quest to discover tensor program schedule candidates that exhibit low execution latency when applied, and the necessity to do so for every unique tensor operator within every optimised DL model.

Chapter 3 experimentally demonstrates that a substantial portion of candidates proposed for measurement on target-device are not meaningfully furthering the search process (identifying as them *cold* candidates), whilst Chapter 4 proposes several techniques to reduce their occurrence and impact, additionally improving the efficiency of the auto-tuning process at multi-operator and multi-model levels.



(a) Proportion of total auto-tuning time cost dedicated to generating candidates (*Processing*), compiling candidates (*Build*) and measuring their latency (*Measurement*)

(b) Average CPU and GPU utilisation during auto-tuning.

Figure 5.2: Auto-tuning three DL models with three DL auto-tuners towards Nvidia Volta V100 GPU target-device until 500 hardware measurements are performed.

Despite their increasingly more effective approaches to candidate space traversal, completing auto-tuning for a single DL model remains a lengthy process (hours to tens of hours), as demonstrated in Figure 5.1b and results presented in Section 3.3. To understand what causes this increased cost, several auto-tuners (AutoTVM [40],

Ansor [385] and Chameleon [8]) were instrumented to measure the proportion of total auto-tuning time spent on candidate schedule generation (*processing*), compilation (*build*) and candidate tensor program latency *measurements*, as shown in Figure 5.2a.

These results indicate that the processing phase is highly dependent on the type of auto-tuner, whilst the build phase depends mainly on the DL model, since more complex tensor operator classes commonly take similar time to compile. Importantly, it can be observed that performing measurements can be a time-consuming activity, costing on average 51.6% of total optimisation time during auto-tuning.

Upon closer inspection of the measurement infrastructure (see Section 2.6.2.4) shared by the analysed auto-tuners, it becomes apparent that the isolation of candidate execution during measurement (see Section 2.6.2.5) is the core bottleneck causing high time costs of performing measurements on target-devices during auto-tuning. The decision of isolating candidate executions during measurements has been widely adopted within the DL compiler optimisation community [40, 385, 8, 183, 184], and stems from the need to maintain measurement accuracy. It is an established assumption that simultaneous execution of more than one DL tensor program on the same device can result in kernel contention and interference [254, 367, 369], with no execution latency guarantees provided by the GPU scheduling infrastructure [239, 271, 44].

Within this thesis, this isolated execution approach is referred to as a *serial* measurement infrastructure, whereby latency of tensor programs is measured by executing them on the target-device sequentially, with no other concurrently executing workloads present within the same GPU. As depicted in Figure 5.2b, isolated execution of tensor programs does not utilise the GPU fully (as low as 19%), at the same time resulting in low host CPU utilisation (as low as 7%) due to frequent waiting for measurement results, ultimately reducing candidate throughput and platform availability.

All studied TVM-based auto-tuners rely upon the same, serial candidate measurement infrastructure. As such, there exists opportunity to address prolonged auto-tuning times

across multiple SOTA DL auto-tuners, by eliminating or reducing the bottlenecks present within their measurement infrastructures. The rest of this Chapter explores a naïve approach to solving the aforementioned bottlenecks within current auto-tuner candidate measurement infrastructure, in quest to discover opportunities for efficiency and performance gains during DL auto-tuning.

5.2 Leveraging Parallelism During Measurements

It is widely understood that parallelism can be used to improve compute resource efficiency and throughput across numerous areas of computing. Intuitively, performing candidate tensor program executions in parallel, within the same target-device GPU, could also manifest in similar improvements, accelerating DL auto-tuning. However, an established, held and practised view within the community is that leveraging measurements performed on simultaneously executing (within the same target-device) candidate tensor programs is not viable due to unpredictable resource contention effects occurring during parallel execution. In the case of GPUs, the proprietary kernel scheduling provides no workload latency guarantees. Moreover, existing auto-tuner measurement infrastructures are designed to strictly perform serial tensor program latency measurements in an isolated target-device. Within the existing SOTA DL auto-tuner measurement infrastructures, parallelism can only be achieved inter-device, where multiple local or remote devices perform serial measurements in parallel.

5.2.1 Existing Inter-device Parallel Measurements

Existing, serial measurement infrastructure underpinning SOTA DL auto-tuners is unable to parallelise candidate executions during measurement within a single GPU. Within the TVM-based auto-tuners, acceleration of measurements is achieved via a set of RPC-based components that enable leveraging multiple identical target-devices during

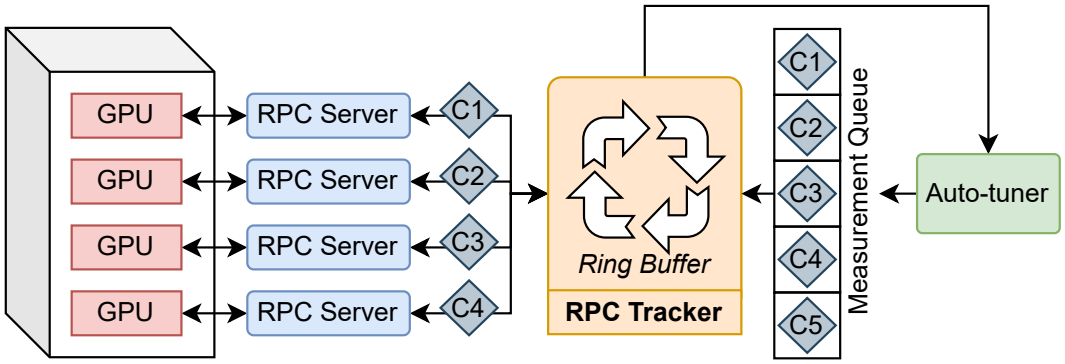


Figure 5.3: Existing, serial measurement infrastructure achieving inter-device parallelism during measurement via RPC calls and ring buffer tracker that isolate candidate executions.

the auto-tuning candidate measurement process. Any locally or remotely available GPUs can be leveraged for measurement by binding them to standalone RPC servers that facilitate kernel execution and latency measurement, as depicted in Figure 5.3. Multiple RPC servers are managed via an RPC tracker - a process that connects the auto-tuner to the servers and allocates measurement tasks to servers that are not yet occupied by existing measurement tasks, in a ring buffer fashion. The maximum parallelism achievable via this approach is bounded by the number of available target devices, and limited by the CPU's ability to execute multiple RPC servers and connections to the tracker simultaneously. To date, no existing DL auto-tuner is capable of performing intra-device parallel candidate measurements reliably - that is to execute multiple tensor programs on the same target-device whilst performing their latency measurements, and maintain satisfactory DL auto-tuning performance.

5.2.2 Naïve Intra-device Parallel Measurements

To introduce intra-device parallelism during candidate measurements, N local RPC servers could be spawned and bound to a single GPU, where each RPC server executes a process with an attached CUDA Stream [271], enabling simultaneous execution of up to N candidate tensor programs. This, however, introduces overhead of spawning

multiple long-running RPC servers per GPU target-device and further, makes it more challenging to control and analyse the pattern of candidate submissions for execution.

An initial, naïve approach to challenge the established view of the need for serial intra-device candidate measurements during DL auto-tuning, and avoid spawning multiple RPC servers, is to circumvent the existing RPC-based serial infrastructure and manage multiple candidate measurement processes separately. Within this approach, during each measurement round, d_p (Degree of Parallelism) processes are spawned and assigned a separate CUDA Stream that executes the respective GPU kernel for the tensor program, enabling d_p simultaneous tensor program executions. Hereafter, this approach is referred to as Naïve Parallel Measurement (NPM). The NPM approach considers a local DL auto-tuning deployment with a single GPU. Multi-GPU mechanism, amongst other improvements is presented in Chapter 6.

5.3 Experiment Setup

To analyse the impact of intra-device parallelism on tensor program latency measurement effectiveness and the cost of performing auto-tuning, a set of experiments was performed involving multiple hardware platforms, tensor programs and DL auto-tuners, comparing the conventional *serial* approach and the proposed naïve approach - NPM.

5.3.1 Hardware Platforms

Table 5.1: Details of hardware platforms used during experimentation

	Host		Target-device (GPU)		
Abrv.	**CPU**	**DRAM**	**Model**	**Arch.**	**DRAM**
A	64-cores 2x Intel Xeon 5218, 2.3Ghz	196GB	Nvidia V100	Volta	32GB
B	24-cores AMD 1920X, 3.5GHz	128GB	Nvidia GTX2080	Turing	8GB
C	12-cores Intel i7-6850K, 3.8GHz	32GB	Nvidia GTX1080	Pascal	8GB

All performed experiments involved auto-tuning DL tensor programs towards Nvidia GPUs contained within three distinct compute platforms (A, B and C), as shown in Table 5.1. Each platform differs in terms of the CPU model, available DRAM memory and the available GPUs. GPUs were selected across three distinct Nvidia architectures: Pascal [222], Turing [224] and Volta [223] to determine feasibility of parallel measurements and their impact across a wide range of popular DL auto-tuning target-devices.

5.3.2 Software, Middleware and Auto-tuners

Table 5.2: Details of middleware and software such as DL frameworks used during the study

Type	Specification	Version
Operating System	Ubuntu	20.04.02
GPU Driver / Compute Lib.	Nvidia CUDA [231] / Driver (Linux)	11.3.1 / 465.31
DL / Codegen Compiler	Apache TVM / LLVM [172]	0.8 / 11.0
DL Frameworks	Pytorch [1] / Apache MXNet [79]	1.11.0 / 1.8.0

Similar to the study performed within Chapter 3, each host platform contained identical deployments of an Operating System and middleware, as detailed within Table 5.2. Most recent versions of software were used at the time of experimentation.

To perform auto-tuning, three SOTA DL auto-tuners were used, selected based on general popularity within the TVM online community and availability of source code online, necessary for instrumentation and analysis. The following auto-tuners were used: (1) AutoTVM [40] - a template-based auto-tuner that leverages a GBT cost model and an SA-based search strategy (abbreviated as *AT* in figures); (2) Chameleon [8] - a template-based auto-tuner that leverages the same cost model as AutoTVM, replacing the search strategy with an RL-based approach (abbreviated as *CH* in figures); and (3) Ansor [385] - an autoscheduler that leverages a modified cost model adopted from AutoTVM and an Evolutionary search strategy to generate high-performance tensor program schedules (abbreviated as *AN* in figures).

Each auto-tuner was configured with its default parameters, obtained from the associated publications and codebases available online. Section 2.6 provides architectural and design details of the auto-tuners used for experimentation, while Tables F.1, F.5, F.6 and F.7 within Appendix F provide parameters used to configure the auto-tuners.

5.3.3 Workloads

Table 5.3: Tensor operator workloads used during experimentation. For more in-depth details, please see Table C.1 in Appendix C

Total Count	**Class**	**Tensor Op Type**	**Input Types**	**Input shapes**	**Variants**	**FLOPs range**
3	MatMul	Matrix multiply	fp32	NHW	Regular	180 - 6.71E+08
7	Conv1D	Convolution 1D	fp32	NCW, NWC	Regular, Transposed	36 - 221,448
16	Conv2D	Convolution 2D	fp32, int8	HWCN, NCHW, NHWC, NCHWc	Regular, Depthwise, Grouped	576 - 1.21E+09
6	Conv3D	Convolution 3D	fp32	NCDHW	Regular, Transposed	27,648 - 5.24E+08
1	Corr	Correlation operator	fp32	NCHW	Regular	70,308
3	Dense	Dense (FC) layer operator	fp32, int8	NHW	Regular	1,024,000 - 1.84E+08

Experimentation performed during evaluation of the NPM approach, involved auto-tuning 36 unique tensor operators, as outlined in 5.3. Tensor operators selected consisted of commonly occurring operators within CNN models, as well as custom operators with multiple variants (for full details on the tensor operator characteristics, see Table C.1 in Appendix C. Expressions for each tensor operator were specified using TVM's Tensor Expression (TE) language, ensuring that all operators are able to leverage existing schedule templates within TVM's TOPI, such that they can be auto-tuned using AutoTVM and Chameleon (Ansor does not require templates). Studied workloads range in shapes, sizes, variants and overall CI to represent many potential DNN architectures.

5.3.4 Collected Metrics

During experimentation, tensor program *execution latency* was captured via timestamps during DL auto-tuning measurements. Derived from the execution latency, metrics such as latency increase/decrease with respect to some latency setpoint were obtained. Furthermore, total DL auto-tuning time cost was captured via less granular start and end timestamps. In scenarios that involve NPM measurements, the measure of δ was also captured - difference between measured latency performed with some d_p (for example, two or four) vs. a measurement performed in serial for the same candidate.

5.3.5 Experiments

36 unique tensor operators were auto-tuned using three SOTA auto-tuners and the NPM infrastructure. Seven distinct d_p levels (1, 2, 4, 8, 16, 32, 64) were studied across all tensor operator and auto-tuner combinations. Experiments utilising the serial measurement infrastructure were also performed to obtain reference latency measurements of best found candidates per auto-tuner. Each experiment involved auto-tuning a tensor operator towards a given GPU until 500 measurements were performed.

5.3.5.1 Nvidia MPS

In addition to the NPM d_p level experimentation, a subset of tensor operators were selected for NPM auto-tuning with Nvidia Multi-process Service (MPS) [228] enabled.

Nvidia MPS is client-server implementation of the CUDA runtime API, that enables multiplexing of GPU kernels originating from different CPU processes, facilitating their concurrent execution on the GPU. Typically, kernels originating from separate threads within a single process may run concurrently as they are managed as part of the same CUDA context, however, kernels originating from separate processes cannot, and are by default allocated to separate execution queues that are managed by a black-box scheduler

within the driver. Nvidia MPS enables separate process kernels (for example, candidate tensor program executions) to share GPU compute resources concurrently, in turn increasing resource utilisation. The experiments involving Nvidia MPS were performed to determine whether it could be a viable strategy to increase overall GPU compute and memory utilisation, and speed up auto-tuning further, as well as to determine its impact on measurement accuracy.

5.4 Experiment Results

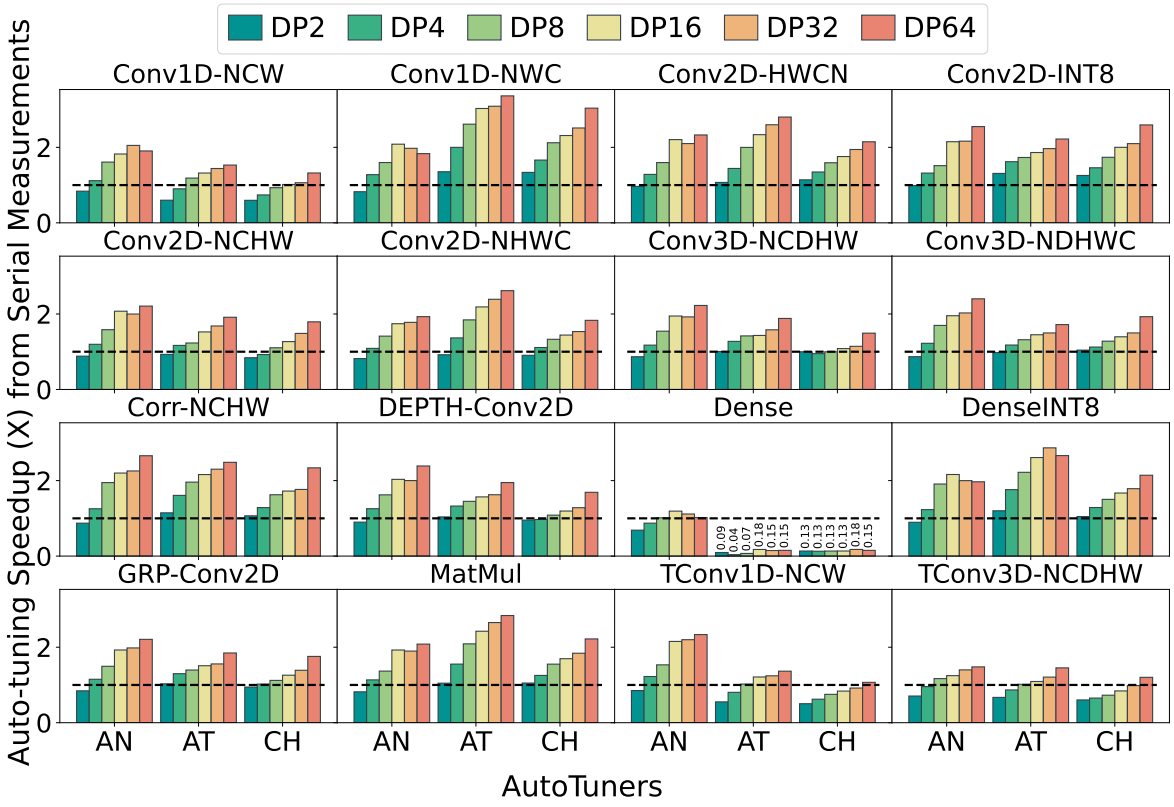


Figure 5.4: Impact of different degrees of parallelism during NPM: Reporting reduction in total auto-tuning time cost compared to serial measurement infrastructure across different tensor operators, degrees of parallelism and auto-tuners. Reporting averaged results across three platforms. It can be observed that the degree of speedup depends on the auto-tuner, tensor program and degree of parallelism, with times where parallelism produces counterintuitive slowdown compared to serial measurements.

5.4.1 Auto-tuning time cost

It can be observed that auto-tuning time is consistently reduced when performing measurements using the NPM infrastructure, as shown in Figure 5.4. At d_p levels d_p larger than eight (higher number of simultaneous candidate executions), auto-tuning with NPM infrastructure resulted in overall auto-tuning speedup of between 2 and 3.36× in relation to serial measurements. Lower number of simultaneous candidate executions ($d_p = 2$ or 4) resulted in less significant improvement of between 1.01 and 1.2×. This difference in speedup is intuitive since the more simultaneous candidate tensor program executions there are, the higher the measurement throughput. However, it can be observed that the achieved speedup is non-linear in relation to the applied d_p.

5.4.1.1 GPU Time-slice Scheduling

The lower than ideal speedup as a result of naïve parallelisation of tensor programs stems from the specificity of GPU scheduling with respect to kernels originating from separate host processes. Such kernels are managed by the Nvidia GPU scheduler, which disallows parallel execution of kernels originating from different processes (more specifically from different CUDA Streams). According to documentation [271], each such kernel receives a slice of the GPU execution time and will be pre-empted once its launch thread grid is completed, replacing it with another kernel to begin its thread grid computation.

Commonly, the majority of the time spent during tensor program execution consists of the CPU setting up the launch procedure, loading data onto the device, and waiting for the respective GPU kernel to complete its work and return back results. Whilst waiting for the kernel computation results might not be an issue during serial measurements, since there is no competition between tensor program kernels for the GPU time slices, it begins to become impactful during NPM. During NPM, the multiple CPU processes managing the awaiting tensor program kernels, become blocked at the CUDA Runtime

level, awaiting any straggler kernels. This prevents them from fully utilising the CPU capabilities for performing tasks such as data loading or setting up subsequent kernel launches, ultimately resulting in long idle CPU time and prolonged auto-tuning time.

5.4.1.2 Candidate Measurement Timeouts and Runtime Errors

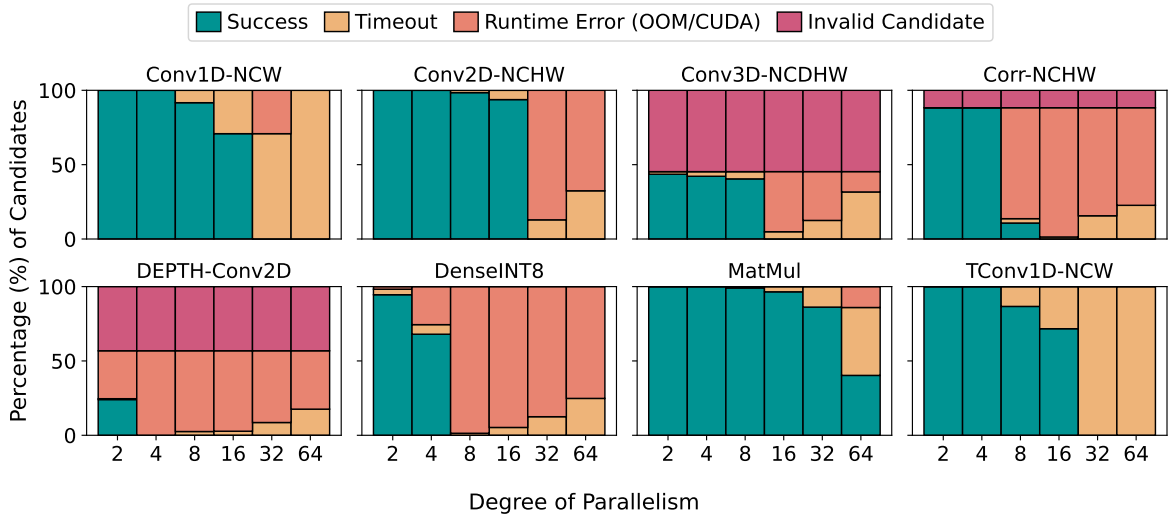


Figure 5.5: Proportion of successful and failed candidate measurements across different degrees of parallelism adopted during NPM measurements. All measurements performed with the *timeout* set to three seconds. Candidates were proposed using the Grid-index auto-tuner towards the GPU in platform A, to maintain candidate repeatability across experiments. See Figures H.3 - H.11, in Appendix H.2 for additional results across remaining platforms, other auto-tuners and additional tensor operators.

Conventionally, all candidate tensor programs submitted for measurement within the default candidate measurement infrastructure, utilise a timeout setting (set to three seconds). This guard ensures no tensor program can occupy the target-device for longer than three seconds during its measurement. Usually, candidates complete their computation far ahead of their three second timeout setpoint, however, in the worst case, the next candidate is prevented from being executed for more than three seconds.

The worst case occurs when a candidate is programmatically correct (and is thus directed to be measured), however, the loop transformations performed as part of its schedule, cause the program kernel to have a disproportionately long execution time

on the GPU. Other causes of timeouts are Out-of-memory (OOM) errors caused by the candidate attempting to utilise more GPU DRAM memory than is available, also commonly originating from inefficient schedules proposed by the auto-tuner.

In several experiments, it can be observed that auto-tuning with the NPM infrastructure at lower d_p levels of 2 or 4, results in auto-tuning time cost larger than that with serial measurement infrastructure ($d_p = 1$) - see series for Dense and TConv3D-NCDHW in Figure 5.4. This is because execution timeouts and OOM errors are more prevalent when multiple candidates are launched simultaneously via NPM, as depicted for several tensor operators and the Grid-index auto-tuner towards platform A in Figure 5.5. Results for other auto-tuners and platforms are depicted in Figures H.3, H.4, H.5, H.6, H.7, H.8, H.9, H.10 and H.11 found in Appendix H.2.

Given that multiple candidates have to wait for their turn to start executing kernels in NPM, there is a higher likelihood of their timeout and the programs being forcefully terminated. Furthermore, whilst tensor program kernels are ultimately serialised by default at the level of the CUDA time-slice scheduler, their inputs and intermediary data is pre-allocated when they are submitted for execution. As such, even a few simultaneously submitted, large-CI kernels, may result in more frequent OOM errors. Both in the cases of timeouts and runtime errors, the CPU is idle for prolonged period of time compared to the time spent on executing kernel on the GPU during measurement.

5.4.1.3 Amortisation of Slowdown

Higher d_p levels (16 - 64), in part amortise the inefficiencies associated with awaiting CPU processes that execute their tensor programs. This is due to the number of in-flight candidates being measured simultaneously, which saturates the available CPU cores, in part reducing idle time and enabling higher measurement throughput compared to a lower d_p level. Importantly, however, the CI and memory footprint of the tensor operator must be low enough to fit d_p tensor program candidates within the GPU simultaneously,

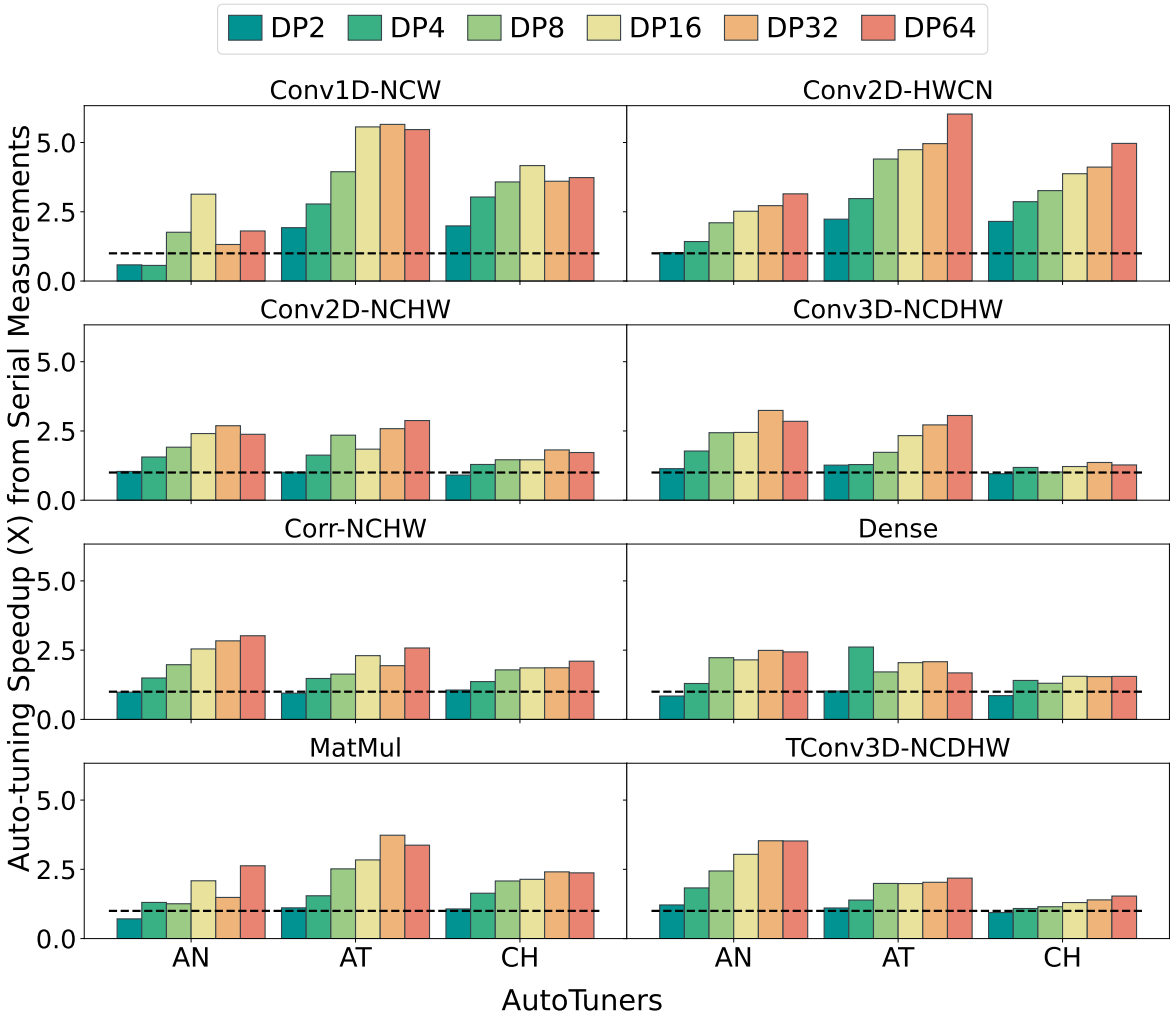


Figure 5.6: Impact of different degrees of parallelism during NPM with Nvidia MPS turned on: Reporting reduction in total auto-tuning time cost compared to serial measurement infrastructure across different tensor operators, degrees of parallelism and auto-tuners, averaged across three platforms.

without resulting in substantial increase of errors or timeouts. Candidate tensor programs for complex tensor operators with large memory footprint may result in increased errors and timeouts when auto-tuned with high d_p level. The overall throughput of both successful and failed candidate measurements is increased at higher d_p, however, linear auto-tuning speedup (for example, 64 times for $d_p = 64$) is rarely possible due to CPU and GPU capability limits, timeouts, runtime errors and the serialisation of tensor

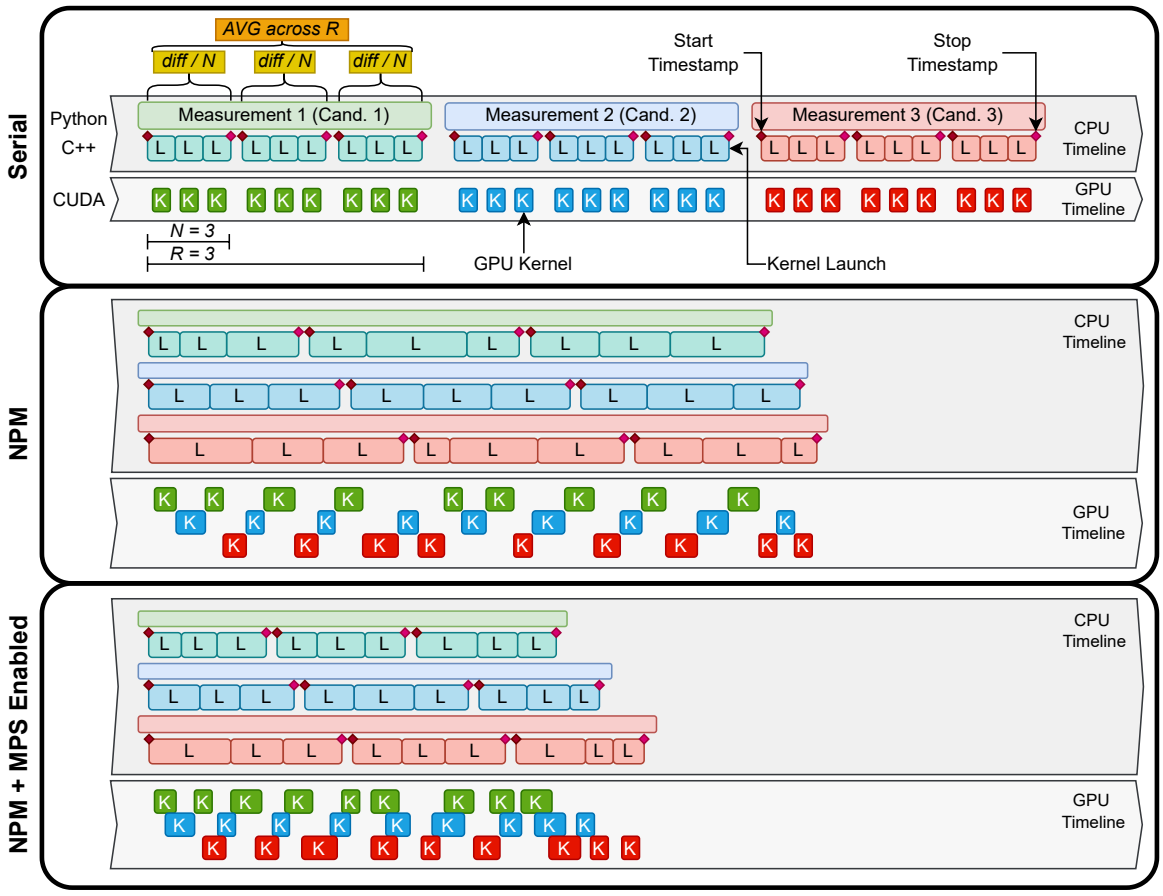


Figure 5.7: Differences in tensor program and kernel execution patterns when performing candidate measurements using serial, NPM and NPM + Nvidia MPS measurement infrastructures.

program kernels at the CUDA Runtime level. However, parallel execution of tensor programs helps to amortise inefficiencies at the CPU process level, leading to faster measurements compared to serial auto-tuning, as shown in Figure 5.7.

5.4.1.4 Impact of Nvidia MPS

As depicted in Figure 5.6, enabling Nvidia MPS during NPM auto-tuning results in moderate improvement in terms of auto-tuning time cost compared to auto-tuning with MPS disabled. With Nvidia MPS enabled, it was possible to further speed up auto-tuning by between 0.98% and 10.97% ($\delta = 25.6\%$) compared to NPM alone, across

the evaluated tensor programs and platforms This effect was especially observable at higher d_p levels. These results are intuitive as the larger number of simultaneous or overlapping kernels, can pack the GPU resources available better, thus increasing target-device utilisation and auto-tuning measurement throughput. Figure 5.7 depicts the differences in CPU process execution patterns and GPU kernel behaviour, between NPM auto-tuning alone and the combination of NPM and Nvidia MPS.

5.4.2 Quality of Candidate Measurements

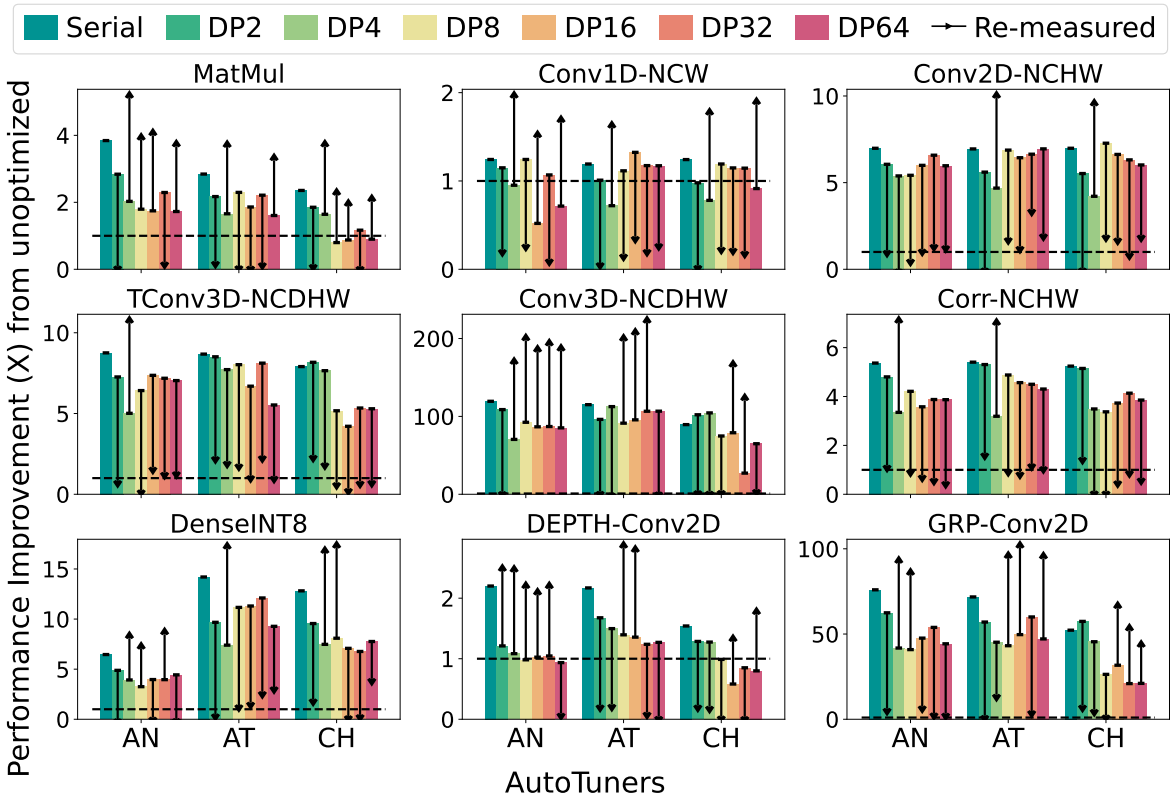


Figure 5.8: Improvement in achieved latency of the best found tensor program as a result of auto-tuning them with three different auto-tuners towards platform A's GPU. Results depict improvement in achieved latency compared to an un-optimised tensor program, across different degrees of parallelism (2 - 64) used during measurement of candidates. In each parallel scenario, the candidate was re-measured in isolation (represented by arrows) to obtain its un-affected execution latency measure, and determine whether and how parallel measurements affect the overall efficacy of DL auto-tuning.

The intuition behind serialising candidate measurements during auto-tuning is that any degree of parallelism will inadvertently reduce the quality of measurements performed (accuracy and consistency), rendering them useless for the auto-tuner when adjusting its cost model and traversing the tensor program schedule space. Attempting to challenge this assumption, requires understanding the source of the measurement quality reduction, and analysis of the magnitude of impact observed as a result of varied levels of d_p.

5.4.2.1 Candidate Measurement Accuracy

To understand how varying d_p affects quality of candidate measurements, and in turn the overall auto-tuning quality, it is necessary to establish the differences between auto-tuning performed using NPM infrastructure, with each distinct d_p level, in contrast to serial auto-tuning with existing measurement infrastructure. This difference is represented as δ, equal to $|\tilde{r_i} - r_i|$, where r_i represents the latency of candidate tensor program c_i as collected during a serial measurement, and the $\tilde{r_i}$ is the execution latency of the same candidate tensor program measured using the NPM infrastructure.

Figure 5.8 depicts tensor program latency improvement vs. un-optimised tensor operator implementation (for example, default DL compiler schedule), for auto-tuning performed across different tensor operators, auto-tuners and levels of d_p (including serial). It can be observed that the tensor program latency improvement achieved with the NPM infrastructure, is typically always reported as worse compared to improvement reported for auto-tuning with serial measurement infrastructure.

Furthermore, when measurement of the best-found candidate is repeated in isolation (serial), its reported latency differs substantially from the one reported by the NPM infrastructure - as depicted by black arrows in Figure 5.8. However, when individual candidate executions were examined using Nvidia Nsight Systems [236] and Nvidia Nsight Compute [235], their on-GPU kernel latencies, when executed using NPM, matched closely those obtained using the serial measurement infrastructure.

This is explained by the fact that by default, latency measurements are performed at the level of the CPU, within the process that manages the execution of the tensor program, as it can be observed in Figures 2.22 and 5.7. Within both serial and NPM candidate latency measurements, the measurement routine relies on *start* and *end* timestamps collected at the beginning and end of every measurement *repeat* of a single candidate. As outlined in Section 2.6.2.5, each candidate tensor program kernel is launched $R \times N$ times where R stands for a single measurement repeat, within which the kernel executes N times to obtain a reliable measure of execution latency. The NPM infrastructure introduces efficiency gains by enabling CPU-level parallelism. During NPM, multiple GPU kernels contained by independent CUDA Streams, driven by individual CPU processes ($d_p \geq 2$), are blocked and must await at the CUDA time-slice scheduler, however their CPU-level routines take advantage of parallel computation.

The above findings suggest that majority of the measurement inaccuracy that manifests during NPM auto-tuning, stems from the measurement method itself, which occurs at the level of kernel launch repeats (performed at the CPU) - see Figure 5.7. Since the time-slice scheduling of GPU kernels can be arbitrary (black-box), and cannot be externally influenced, the time between kernel being launched and the kernel executing on the GPU is factored into the measured latency. This poses no issues during serial measurements, however, results in unpredictably inflated tensor program candidate execution latency when measured using CPU-level timestamps, further amplified by differences in GPU SM utilisation and memory access contention between simultaneously executing tensor programs.

5.4.2.2 Candidate Measurement Consistency

DL auto-tuners rely upon consistently accurate candidate latency measurements to continually update their cost models that influence the schedule space traversal to discover high-performance tensor programs. The inaccuracy of measurements caused by

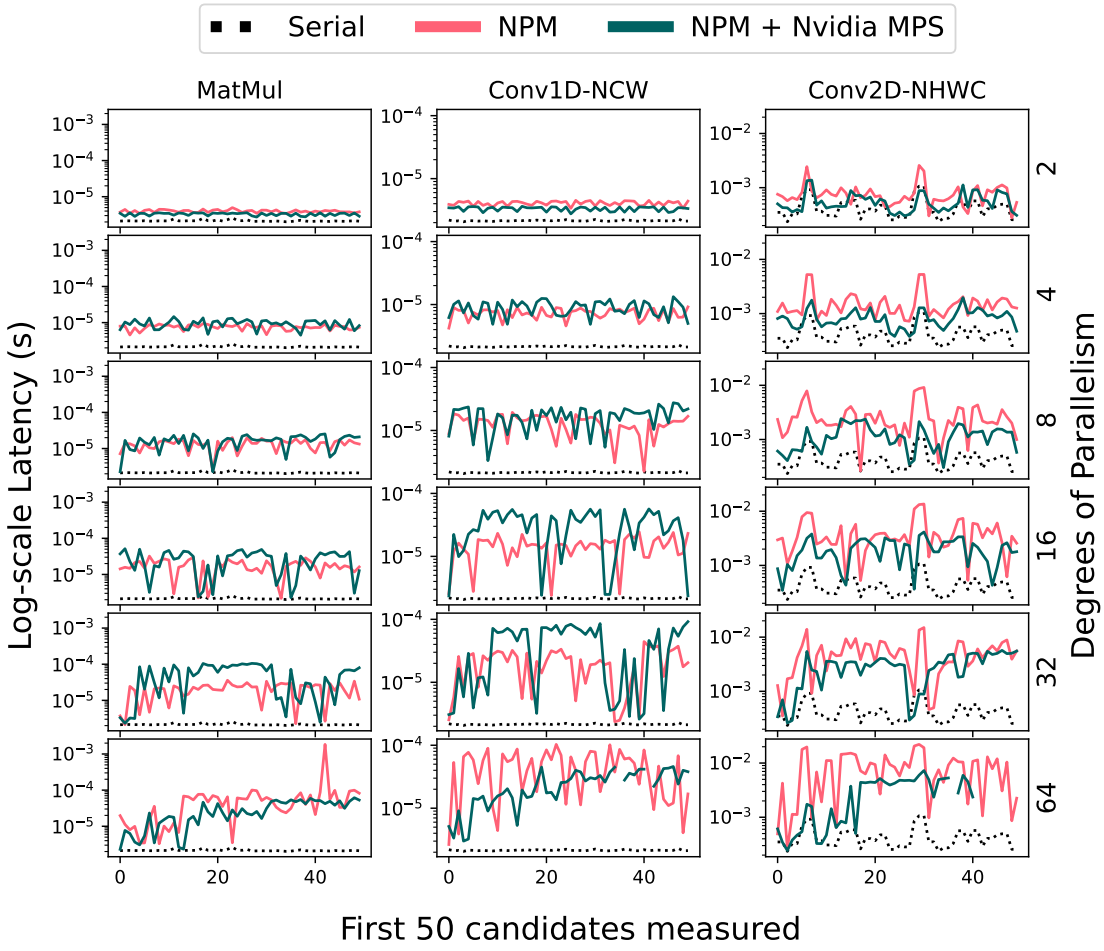


Figure 5.9: Measured latency of proposed candidate tensor programs when auto-tuning with a Grid-index auto-tuner (to ensure repeatability of candidate proposal pattern). Reported 50 first candidates proposed by the auto-tuner towards the GPU within platform A, across several tensor operator auto-tuning tasks. Compared latency measurements produced by serial, NPM and NPM + MPS measurement infrastructures. For extended results, see Figures H.1 and H.2 in Appendix H.1.

CPU-level timestamp collection at higher d_p also degrades the ability of the auto-tuner to discover high-quality candidates. Consistency of measurement accuracy is crucial such that auto-tuner cost models do not converge onto false minima and waste time and energy exploring high-latency schedule space areas.

Figure 5.9 depicts auto-tuning progress (latency of candidate executions across time) across different tensor operators, when auto-tuning with a Grid-index auto-tuner [39]

utilising the serial and NPM infrastructures, across different levels of d_p and with Nvidia MPS enabled or disabled. The Grid-index auto-tuner was selected for these experiments to ensure identical candidates are proposed for the same tensor operator across multiple auto-tuning sessions[1]. This enables one-to-one latency comparisons across different d_p levels, and across serial and NPM infrastructures. Observing results in Figure 5.9, it can be noted that for identical candidate measurements within auto-tuning sessions of identical tensor operators, higher levels of d_p produced increasingly more diverse distributions of the measured latency compared to the same candidate measurements performed in isolation. Different tensor operators (varied CI) had varying impact on such divergence, with further differing impact across measurements performed using NPM infrastructure with Nvidia MPS enabled/disabled.

This stems from the black-box scheduling decisions made at the CUDA Runtime level. At lower d_p levels, the fewer the number of simultaneous kernels originating from different CUDA streams can be more effectively packed within the available computational resources of the GPU (SMs), increasing the probability that the measured execution latency of the tensor program is closer to the same tensor program executing in isolation - in other words, reducing the impact of contention on execution characteristics. At higher d_p levels, more simultaneous kernels are awaiting their turn to execute, sometimes long-enough to violate the overall measurement timeout set point, and as such, the measurement processes that manage them, must wait longer for the kernel results to be returned and the end timestamps to be saved, causing measurements to be inaccurate. The accuracy divergence occurs because these phenomena are not consistent across time, as different candidates occupy different portions of the GPU compute resources for different lengths of time, further influenced by non-deterministic kernel scheduling decisions made by the black-box CUDA Runtime scheduler.

[1]Such fair comparison using auto-tuners such as Ansor or AutoTVM would be infeasible, given their schedule space search algorithms propose different schedules during new auto-tuning sessions, when optimising an identical tensor operator.

5.4.2.3 Impact of Nvidia MPS

Examining Figure 5.9, it appears that enabling Nvidia MPS during NPM, reduces overall measurement inaccuracy by 15% across all evaluated tensor operator classes. However, inconsistency of this reduction remains high (40% Median Absolute Deviation [181, 251]). Reduction in inaccuracy stems from kernel throughput as a result of enabling Nvidia MPS. As detailed in Section 5.3.5.1, Nvidia MPS combines kernels originating from different CUDA Streams (different CPU processes), into a single Stream, circumventing kernel serialisation at the GPU time-slice scheduler level. This results in more GPU compute resources being utilised simultaneously, thus affecting the inaccuracy measure, as kernel measurements complete earlier, resulting in δ being smaller across time.

Whilst marginally speeding up candidate measurements and reducing measurement inaccuracy, Nvidia MPS retains the effect of causing accuracy inconsistency, and in several cases (Conv1D-NCW at $d_p = 16, 32$, MatMul at $d_p = 16, 32$ in Figure 5.9), manifests further inconsistency artefacts (repeating periods of "flat", highly inflated latency, followed by latency "dips"). These stem from the aforementioned kernel funneling whereby many kernels execute on the device at the same time, preventing their measurements from being completed until all of them complete. While improving workload throughput, Nvidia MPS impacts the cache and/or memory transaction efficiency of concurrent CUDA Stream kernels, where kernel executions are interleaved and compete for memory access, particularly in cases of memory-bound programs [271].

The performed analysis suggests that enabling Nvidia MPS is counterproductive during parallel candidate measurements within auto-tuning, due to the negative effects on measurement accuracy and accuracy consistency. As it is important for auto-tuners to rely upon accurate and consistent measurements when exploring tensor operator schedule spaces, enabling Nvidia MPS, while utilising the inaccurate NPM infrastructure, can further impair auto-tuner's ability to propose optimal candidates.

5.5 Findings and Design Directions

The study explored NPM infrastructure for performing parallel measurements during tensor operator auto-tuning, identifying several important design decisions for a candidate measurement infrastructure used by auto-tuners, that if appropriately leveraged, could lead to significantly lower overall auto-tuning time costs, whilst maintaining auto-tuning quality across a range of different tensor operators, auto-tuners and target-devices.

Insight 1 - *Performing more accurate measurements*: The study indicates that tensor program latency measurements, when performed naïvely in parallel using CPU-level timestamps, inadvertently result in measurement inaccuracy. This occurs due to serialisation of kernels at the GPU-level. As the CPU-based kernel launch routines await kernel completion, the time spent between kernel launch and execution will be captured, producing inaccurate measurement. Furthermore, the inaccuracy pattern is affected both by candidate execution characteristics and the black-box GPU-level scheduling, causing accuracy inconsistency across candidates. This in turn hinders the auto-tuner's ability to navigate the candidate schedule space via cost models and search algorithms.

Thus, an important design direction is to *leverage more accurate measurement methods when parallelising candidate tensor program measurements, such that the auto-tuners can reliably leverage them during schedule space exploration.*

Insight 2 - *Managing trade-offs between concurrency and failures*: Another important observation is that naïvely increasing the number of concurrent tensor programs, leads to more frequent timeouts and runtime failures. This stems from the statically set, three second timeout used in conventional measurement infrastructure for all candidate tensor programs. A static timeout, combined with GPU time-slice scheduling of concurrent kernels, and measurement routines awaiting kernel completion, results in premature preemption of measurements that would have otherwise succeeded if performed serially.

In turn, when auto-tuning with the NPM infrastructure, this results in a

reduced number of successful measurements available for the auto-tuner cost model updates, ultimately reducing auto-tuning efficacy or in the worst case, rendering it counterproductive compared to un-optimised schedules. Additionally, the degree of computational resource utilisation varies across candidate tensor programs for a single tensor operator over time. As auto-tuning continues, candidates with increasingly better performance are proposed, that better utilise the available GPU compute resources. This results in different execution footprint across candidates as they reach lower latency, in turn affecting the GPU scheduling decisions.

This suggests that *the measurement timeout should adjust to d_p at which the measurements are performed. For example, the higher the number of simultaneous measurements, the larger the individual per-candidate timeout should be, minimising failures. Furthermore, the timeout adjustments must dynamically account for the varying execution patterns of candidates within a single auto-tuning session, such that progressively better candidates are not prematurely pre-empted causing them to fail.*

Insight 3 - *Adapting to different tensor operators*: The analysis suggests that certain tensor operators benefit more from the NPM infrastructure during their auto-tuning, with respect to reducing total auto-tuning time cost and less impactful measurement inaccuracy and inconsistency. This stems from the different overall characteristics of tensor operators, which include compute complexity, memory footprint and ability of the auto-tuner to generate schedules that parallelise tensor operator computation optimally across the available GPU cores. It is likely that for different combinations of tensor operators and target-device and host platform capabilities, there exists a different range of d_p levels that optimally balances the trade-off between concurrency (achieving high measurement throughput), measurement accuracy and measurement consistency.

As such, *to reduce auto-tuning operational costs, whilst delivering consistently accurate parallel measurements, the selection of an appropriate d_p level must take into account the varied performance profiles across tensor operators with different characteristics.*

This page is left intentionally blank

Chapter 6

DOPpler: Parallel Measurement Infrastructure for DL Auto-tuning

Guided by the insights and design directions established in Chapter 5, this Chapter describes the design, implementation and evaluation of DOPpler - a parallel DL auto-tuning measurement infrastructure that is capable of reducing total auto-tuning time cost by exploiting parallelism during candidate tensor program latency measurements, whilst maintaining optimisation quality equivalent to that of auto-tuning with serial measurement infrastructure. DOPpler actively adjusts the level of measurement concurrency (d_p) towards the given auto-tuning scenario, and monitors quality of performed candidate measurements to ensure results reported to the auto-tuner allow it to maintain high schedule space exploration efficacy. Simultaneously, DOPpler also works towards avoiding measurement timeouts and runtime errors (ones that are unrelated to erroneous candidate schedules), such that maximum number of valid candidate measurements complete successfully, and can be leveraged by the auto-tuner during cost model updates to then better navigate the schedule space.

6.1 System Design and Implementation

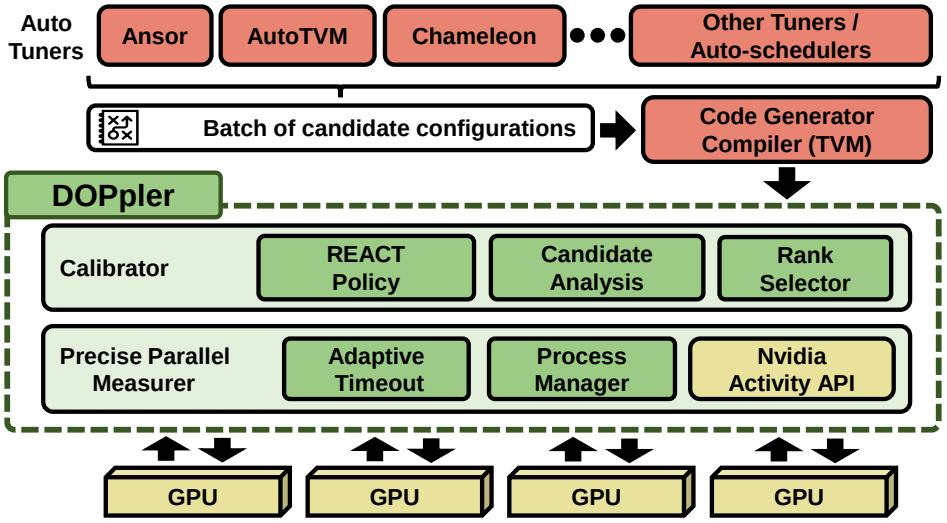


Figure 6.1: Design of the DOPpler measurement infrastructure

As shown in Figure 6.1, DOPpler acts as an intermediary layer between the auto-tuner and target-devices, replacing the conventional, serial measurement infrastructure to enable high-quality, intra and inter-device parallel candidate measurements. DOPpler comprises two major components that work together to achieve these goals:

(1) Precise Parallel Measurer (PPM) is a component within DOPpler, responsible for enabling parallelisation of candidate tensor program measurements within the same target-device GPU as well as across multiple GPUs. Moreover, PPM introduces a method for collecting kernel execution latency measurements directly from the GPU, as opposed to collecting timestamps at the CPU level - the method used by conventional candidate measurement infrastructure.

(2) Calibrator: The Calibrator module compliments the PPM to reactively adapt d_p towards an auto-tuning scenario, both in terms of the specific tensor operator characteristics, target-device capabilities and the changing execution characteristics of candidate tensor programs for a given tensor operator across time. Calibrator analyses

the accuracy and consistency of performed measurements in real-time, and continually adjusts the d_p to levels appropriate for the current batch of candidate measurements. Furthermore, the Calibrator ranks a small portion of best found candidates and re-measures their latency in isolation to ascertain selection of the globally best candidate out of all proposed during the auto-tuning session.

6.1.1 DOPpler's Objectives

Formally, DOPpler is designed towards three core objectives, as follows:

Objective 1: Maximise d_p during auto-tuning measurements, thus increasing measurement throughput and utilisation of the target-device GPU and platform CPU:

$$\arg\max \; d_p \tag{6.1}$$

Objective 2: Minimise measurement time[1] $\Upsilon = \{\upsilon_i \mid \; i = 1 \dots K\}$ for a set of candidates $C = \{c_i \mid i = 1 \dots K\}$ proposed by the auto-tuner given target-devices $H = \{h_j \mid j = 1 \dots N\}$ and operating with degree of parallelism $d_p = \{p = 1 \dots max_{dp}\}$, where E denotes tensor program execution during its latency measurement.

$$\arg\min_{i,j,p} \; \Upsilon = E(c_i, h_j, d_p) \tag{6.2}$$

Objective 3: Minimise measurement inaccuracy δ_{mean}, resultant from assigned d_p being too high for a given sub-batch of candidate tensor program latency measurements, that execute in parallel during a single DOPpler round[2], as follows:

$$\arg\min_{i,j,p} \; \delta_{mean} = E(c_i, h_j, d_p) \tag{6.3}$$

[1]Time taken to perform latency measurement of a single candidate. Note: In conventional (serial) and parallel measurement infrastructures, tensor program kernels are executed multiple times during their latency measurement to ensure measurement accuracy)

[2]DOPpler rounds are elaborated on within Section 6.1.3

6.1.2 Precise Parallel Measurer

Algorithm 3: Precise Parallel Measurer

Input: B_{curr}, d_p, G
Output: M_{curr}

init
  $W \leftarrow$ Instantiate max_{dp} workers across G devices
begin
  t_{out} = CalcTimeout(d_p) // Eq. 6.4
  while *candidates left to measure* **do**
    $B_{sel} \leftarrow$ take $d_p \times G$ candidates from B_{curr}
    // Allocate d_p candidates to each device $g_i \in G$ in a round-robin manner
    $BG_{alloc} \leftarrow$ allocate(B_{sel}, d_p, G)
    for $(b_i, g_j) \in BG_{alloc}$ **do**
      $subm \leftarrow$ measure(wrk=w_{ij},dev=g_j,cand=b_i,tout=t_{out})
    end
  end
  $M_{curr} \leftarrow$ Retrieve results from workers W
end

6.1.2.1 Parallel Candidate Execution

To ensure DOPpler (compared to the naïve approach proposed within Chapter 5) can manage parallel candidate tensor program execution and measurement more reliably, a process manager and candidate execution / measurement routines were designed across different levels of the DL compiler stack. Algorithm 3 describes the PPM, where B_{curr} is a set of compiled candidates and G is the number of available target-devices.

The process (worker) manager leverages the multiprocessing [81] Python library to construct W reusable CPU worker processes, where $W = \{w_i \mid i = 1 \dots max_{dp}\}$. max_{dp} specifies the maximum expected d_p possible for the auto-tuning session, and is a hyperparameter that can be set by the user. Within all performed experiments, max_{dp} has been set to 64 - the size of the batch of candidate tensor programs proposed for

latency measurement during auto-tuning (see Tables F.5, F.6 and F.7 within Appendix F for details on auto-tuner hyperparameters set during experimentation).

Each worker process $w_i \in W$ manages a separate CUDA Stream that encapsulates and executes the tensor program kernel on the GPU, and performs calls to the custom latency measurement subroutines added to the DL compiler tensor program runtime environment (C++). The workers are reusable[3], for the duration of auto-tuning of a single tensor operator. As such instantiating these workers is a one-off cost.

Tensor program latency measurement requests are sent to each worker $w_i \in W$, in the form of a compiled candidate tensor program and a set of execution parameters (timeout setpoint, number of measurement repeats). The requests are made by DOPpler's main routine via Inter-Process Communication. As detailed in Section 5.4.2, separate CUDA Stream kernels are serialised by the GPU time-slice scheduler. Thus, the core cost savings, in the form of increased measurement throughput achieved by the parallel worker manager, have their source at the platforms's CPU level. More specifically, caused by the amortisation of increased per-candidate measurement time Υ (stemming from workers waiting for kernel results) by the increased number of in-flight candidate measurements. DOPpler does not attempt to spatially multiplex candidate tensor program kernel executions at the GPU, for example, by enabling Nvidia MPS. As described in Sections 5.4.1.4 and 5.4.2.3, enabling Nvidia MPS imposes further measurement inaccuracy and inconsistency across time, whilst resulting in modest auto-tuning time cost reduction.

6.1.2.2 Adaptation of Timeout Setpoint

As described in Section 5.4.1.2, high d_p levels during candidate tensor program latency measurements, result in an increased likelihood of tensor programs violating their timeout setpoints, and causing runtime errors such as Out-of-memory (OOM) due to resource

[3]Unlike the ephemeral, serialised processes spawned within conventional serial infrastructure for each candidate tensor program measurement

availability violations. As such, to achieve timely and successful completion of parallel candidate measurements, DOPpler addresses this by introducing the concept of an adaptive timeout for individual candidate tensor program measurement processes. An appropriate timeout is determined dynamically for each candidate tensor program ahead of its measurement, using a modified Heaviside step function [23], as follows:

$$t_{out} = \lfloor \max\{\eta, \min\{\iota, 2\iota \times \ \tanh(\phi d_p \frac{G}{2})\}\}\rfloor \tag{6.4}$$

where η is the minimum t_{out}, ι is the maximum t_{out}, G is the set of target-devices available for measurement, and ϕ the 'steepness' parameter that controls how quickly t_{out} should increase given higher d_p and G. The t_{out} increases as d_p and G increase.

In terms of selected hyperparameters, η was set to conservative four seconds (+1 in relation to the default three seconds found within serial measurement infrastructure). ι was set to 20 seconds, which was guided empirically by auto-tuning tensor operators with progressively higher CI using the Grid-index auto-tuner. The maximum time taken to complete max_{dp} concurrent measurements successfully was noted and represented as ι, utilising highest-possible CI tensor operator that fits within the GPU resources, whilst avoiding timeout setpoint violations and runtime errors such as OOMs.

6.1.2.3 Maintaining Measurement Accuracy

As outlined in Section 5.4.2, utilising the conventional candidate latency measurement method (CPU-level timestamps) during parallel candidate tensor program executions, results in increased measurement inaccuracy. To alleviate measurement inaccuracy, DOPpler's PPM introduces a measurement routine that leverages direct on-device kernel execution latency information collection. To enable direct on-device latency measurements, PPM utilises calls to the Nvidia Activity API available within the Nvidia CUDA Profiling Tools Interface (CUPTI) library [226] - a set of low-level profiling tools

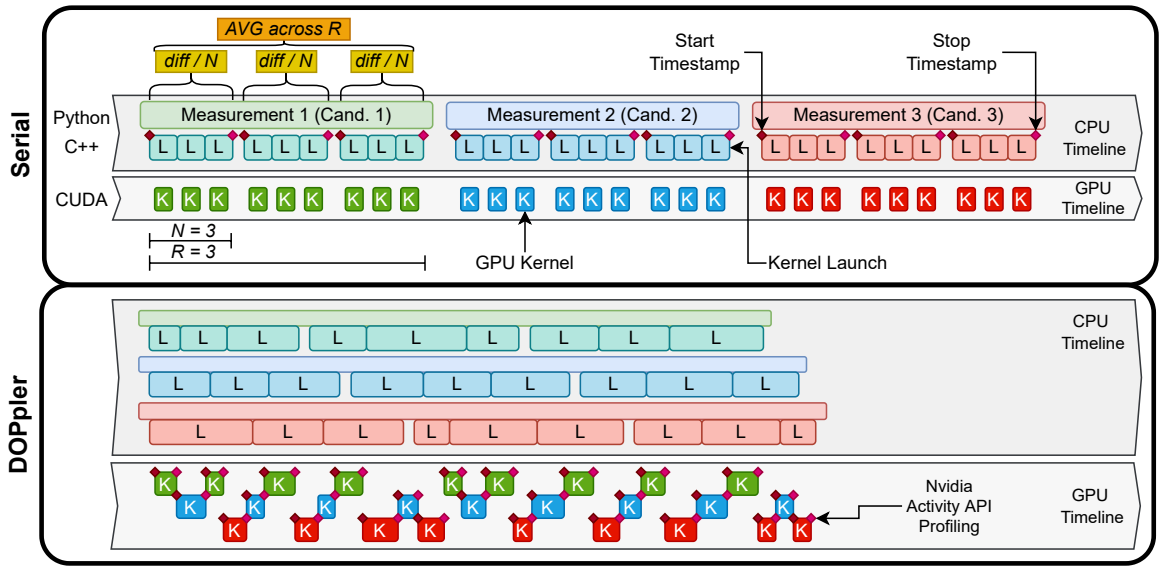


Figure 6.2: Differences in candidate tensor program and GPU kernel execution patterns when performing candidate measurements with serial measurement infrastructure vs. using DOPpler.

for Nvidia GPUs. More specifically, PPM utilises the *CUpti_ActivityKernel5* records made available by the Activity API, containing *start* and *end* timestamps of each kernel execution, collected by the GPU itself, as instructed by the CUDA driver.

DL systems engineers often utilise Nvidia Activity API to manually inspect and measure particular portions of GPU kernels. The Activity API also underpins Nvidia visual profiling suites as an internal library [236, 235]. DOPpler's PPM automates this instrumentation for each executed kernel, by modifying the low-level DL compiler runtime routine, responsible for launching tensor program kernels. Once the kernel records are collected, the PPM extracts necessary timing data and reconstructs a compatible candidate latency measurement record, returning it back to the auto-tuner. Since the kernel timing information originates directly from the target-device (via GPU driver), measurement accuracy is greatly enhanced compared to the CPU-level measurement approach, and further strengthened by repeated kernel executions during measurement, following conventional measurement infrastructure approach. The conceptual difference between serial measurement infrastructure and DOPpler's PPM is shown in Figure 6.2.

Whilst DOPpler does not leverage Nvidia MPS and the associated kernel multiplexing (see Section 5.4.2.3), and employs accurate kernel latency measurements via the PPM, execution of kernels in close succession manifests in additional GPU DRAM memory footprint, which inadvertently has an effect on the cache overhead and thus the ability of kernels to access their operational data efficiently. Any execution timing overheads resultant from memory and cache sharing are thus captured by the direct on-device measurements performed via Nvidia Activity API, reporting kernel execution latencies that are different from those obtained during serial candidate execution and measurement. Whilst executing candidate tensor programs serially on the GPU usually results in very low compute/memory resource utilisation, as the d_p grows, the available resources become better utilised and the aforementioned latent phenomena start to manifest. Thus, there is a specific d_p threshold where measurement inaccuracy and inconsistency become problematic for auto-tuning quality. This threshold is highly dependent on the unique combination of tensor operator and target-device characteristics. DOPpler manages the trade-off between achieving high measurement throughput (large d_p level) and maintaining measurement quality (d_p low-enough to avoid high degree of inaccuracy / inconsistency), using the Calibrator module - as detailed in Section 6.1.3.

6.1.3 Calibrator

Recalling the insights found in Chapter 5, different combinations of d_p levels and tensor operator classes and sizes, manifest different degrees of impact on measurement accuracy and consistency during parallel auto-tuning measurements. As described in the previous section, despite the significantly more accurate measurement approach enabled by DOPpler's PPM, kernel execution latency remains affected due to memory/cache access competition phenomena when multiple kernels are launched simultaneously. To address this, DOPpler's Calibrator module analyses and dynamically adjusts d_p based on changing operational characteristics.

Algorithm 4: Calibrator operation

Input: T - Tensor operator definition
C - Batch of candidate configurations for T
G - Available target devices
Output: M - Measurements of each B_i as configured by C_i

begin
  $B \leftarrow \text{Compile}(T, C)$ // B = Set of compiled candidates
  while $len(B) \geq 0$ **do**
    $M_{res} \leftarrow \text{Measure}(d_p \times G \text{ candidates from } B, \text{dp}=d_p, \text{devs}=G)$
    $M_{res}, N_{err.dp}, N_{succ} \leftarrow \text{RemeasureFailed}(M_{res})$
    $M_{smpl} \leftarrow \text{SampleCandidates}(M_{res}, param)$ // Eq. 6.6
    $M_{remeas} \leftarrow \text{Measure}(M_{smpl}, \text{dp}=1, \text{devs}=G)$
    $M_{res}, \delta_{mean} \leftarrow \text{Analyze}(M_{res}, M_{remeas})$ // Eq. 6.7
    $M_{updt} \leftarrow \text{Update}(M_{res}, \delta_{mean})$ // Eq. 6.8
    $M \leftarrow \text{Concat}(M, M_{updt})$
    // REACT Policy Update
    $A \leftarrow \alpha$ // Adaptive Max Degree of Parallelism, α is an initialiser, set to 128
    if $(\delta_{mean} > \tau)$ *or* $(N_{err.dp} > (d_p \times \tau))$ **then**
      $A, d_p \leftarrow MultiplicativeDecrease(A, d_p, \beta)$ // Eq. 6.9
    $\gamma_{cur} \leftarrow CalculateAdjustment(A, d_p, \gamma_{min}, \gamma_{max})$ // Eq. 6.10
    $d_p \leftarrow |d_p + \gamma_{cur}|$ // Binary Increase

The operation of the Calibrator is described within Algorithm 4, and involves performing the desired candidate tensor program measurements as a set of measurement iterations, each utilising a dynamically determined degree of parallelism d_p.

6.1.3.1 Initial Parallel Measurements

At the beggining of each auto-tuning measurement round (i.e. once an auto-tuner proposes a batch of candidate schedule configurations to be measured), the Calibrator compiles the tensor operator T, $|C|$ times using the TVM DL compiler, where C is a set of schedule configurations provided by the auto-tuner, producing set of compiled tensor programs B, each with different execution characteristics. The set of compiled tensor programs B is then passed onto DOPpler's PPM in order to perform $d_p \times |G|$ parallel

latency measurements with degree of parallelism d_p across $|G|$ available target-devices, where G is the set of handles for the devices. This process produces a set of $d_p \times |G|$ measurement results M_{res} containing tensor program execution latencies. Measurements M_{res} are inspected, and any failed measurements are repeated as a smaller sub-batch with $d_p = \frac{|M_{failed}|}{|G|}$ across the available target-devices G. This enables the Calibrator to establish two important measurement characteristics: (1) $N_{err.dp}$, which denotes how many measurements have failed (timed out, OOM) as a result of the d_p being set too high[4]; and (2) N_{succ} denoting the number of measurements that have succeeded. These two characteristics are later used in the process of reactive d_p adaptation.

6.1.3.2 DOPpler's Measurement Anatomy

As outlined within Sections 2.6.2.4 and 2.6.2.5, a latency measurement of a single candidate tensor program constitutes multiple executions of the tensor program kernel to increase measurement reliability. This process is further depicted in Figures 2.22 and 5.7 for serial and NPM measurement infrastructures, whilst Figure 6.2 compares the execution characteristics of DOPpler in relation to the serial infrastructure.

More specifically, let M denote a batch of K completed (successful or failed) measurements, where $M = \{m_i | i = 1 \dots K\}$. M is obtained by measuring $b_i \in B$ compiled tensor programs as configured by $c_i \in C$ candidate schedule configurations provided by the auto-tuner. In line with conventional auto-tuner measurement infrastructures, each execution of $b_i \in B$ is repeated $R \times N$ times within its measurement m_i to obtain mean execution latency $l_i \in L$. DOPpler extends each measurement record m_i by also capturing total measurement duration $\upsilon_i \in \Upsilon$. υ_i denotes the total time taken to perform latency measurement of a single candidate tensor program, which constitutes $R \times N$ executions of the tensor program.

[4]If initially failed measurements succeed once d_p is lowered, it indicates too high d_p has caused the initial failure

6.1.3.3 Latent Measurement Ratios

Examining candidate latency patterns in Figure 5.9, it can be observed that different candidate tensor programs are variably affected by their parallel operation across the auto-tuning duration, in terms of accuracy of reported execution latency. The Calibrator analyses measurement records to identify ones that were disproportionately affected by the latent phenomena identified in Chapter 5. To identify such measurements, the Calibrator initially calculates a population of ratios P between mean candidate execution latencies L and the durations of their associated measurements Υ, as follows:

$$P = \{\frac{l_i}{\upsilon_i} | l_i \in L, \upsilon_i \in \Upsilon\} \tag{6.5}$$

In an ideal scenario (for example, during serial execution), $N \times l_i \approx \upsilon_i$ for each m_i - that is the measurement time υ_i should have a duration close to N times l_i where l_i is the mean measured latency of the candidate tensor program, with small overhead related to kernel launch and data copy operations (where both are static across candidates). However, in cases of high tensor operator CI, constrained computational resources of the target-device, or the d_p being set too high, the ratios $p_i \in P$ for candidate tensor programs most affected by measurement inaccuracy (stemming from parallel execution), were found to be disproportionally inflated compared to other ratios within the population P. Detecting such disproportionally affected measurements can be advantageous to determine a population-wide measure of inaccuracy within a single DOPpler iteration, as well as enable to partially correct the initial population.

6.1.3.4 Outlier Detection and Isolated Re-measurements

Calibrator detects such outlier measurement records using a combination of Double Median Absolute Deviation [181] and modified Z-score [132]. A combination of these detectors was used due to the distribution of P being non-symmetric, where standalone

conventional Median Absolute Deviation outlier detectors fail to operate effectively within such distributions at lower population sizes [279]. Once discovered, the outlier measurements, coupled with a number of random samples from the rest of the result set, are selected for re-measurement in isolation. The Calibrator selectively calculates latency differences between parallel-measured candidates and their serially re-measured counterparts, without having to re-measure the entire population, and balances the tradeoff between measurement quality and auto-tuning cost reduction. Subsequently, a set of Q candidates are selected to be re-measured serially in isolation, constituting the aforementioned outliers and random samples, with Q constructed as follows:

$$Q = max(len(P_{outliers}),\ \lceil \zeta \times len(M_{success}) \rceil) \tag{6.6}$$

where $M_{success}$ denotes all successful measurements within DOPpler's current iteration and ζ denotes the re-measurement percentage factor - set to 0.2 based on initial empirical and sensitivity analyses. The re-measurement result records are then used to calculate δ_{mean}, a population-level measure of candidate measurement inaccuracy, as follows:

$$\delta_{mean} \leftarrow mean\left(\left\{\frac{|l_j - l_i|}{l_i} \mid l_j \in L_{remeas}, l_i \in L\right\}\right) \tag{6.7}$$

The obtained δ_{mean} is then leveraged two-fold in the Calibrator module: (1) to guide adjustment of d_p for the next Calibrator iteration, and (2) to scale current iteration's measurements reported to the auto-tuner in an attempt to correct them, as follows:

$$M_{updt} = \{p_i \times (\upsilon_i - (\upsilon_i \times \delta_{mean}))|p_i \in P, \upsilon_i \in \Upsilon\} \tag{6.8}$$

Since the latent ratios are unique to the specific candidate and its measurement, each such measurement can be adjusted with respect to the population-level δ_{mean}, whilst accounting for the impact of parallelism on the measurement's accuracy.

6.1.3.5 Reactive d_p Calibration

As outlined before, DOPpler continually adjusts d_p to ensure timely and reliable candidate latency measurements are performed. Ahead of the next measurement iteration, DOPpler's Calibrator adjusts next d_p by leveraging the REACT policy. The REACT policy is inspired by the operation of the Binary Increase Congestion control (BIC) [366] method, proposed initially as a network congestion control mechanism within the Transmission Control Protocol (TCP). REACT's goal is to rapidly respond to any changes in δ_{mean} and the number of measurement failures occurring due to current d_p being too high ($N_{err.dp}$), and whilst doing so, maximise d_p over time as outlined in Equation 6.1. The REACT policy is described at lines 11 - 18 of Algorithm 4.

Inspired by the operation of TCP BIC, REACT utilises an adaptive maximum d_p denoted A (and initialised by α) to determine d_p for the next measurement iteration. A is initialised by α to 128 - double of the typical auto-tuner candidate batch size of 64 and can be modified if necessary. To establish whether the d_p needs to be adjusted, REACT monitors current iteration's δ_{mean} and $N_{err.dp}$ for whether they have surpassed threshold τ, which would indicate that the current iteration has manifested high measurement inaccuracy or resulted in a large number of failed candidates as a result of the current d_p. In either case, to ensure reliable latency measurements can be obtained within the next measurement round, d_p must be reduced. To do so, inspired by the TCP BIC, REACT performs *Multiplicative Decrease* of A using hyperparameter β to adjust next iteration's d_p as follows:

$$
\begin{aligned}
A &\leftarrow \begin{cases} |d_p \times \frac{2-\beta}{2}|, & \text{if } d_p < A \\ |d_p| & \text{otherwise} \end{cases} \\
d_p &\leftarrow |d_p \times (1-\beta)|
\end{aligned}
\tag{6.9}
$$

Xu et al. (2004) [366] sets β to 0.125 motivated by higher network utilisation at the expense of convergence on the window size. In Calibrator, β is set to 0.2 to focus the policy on reducing measurement inaccuracy at a modest reduction of throughput. Once adjusted for inaccuracy and failures, the next iteration's d_p is established by performing *Binary Increase*, utilising the γ factor as follows:

$$\gamma \leftarrow \begin{cases} \frac{A-d_p}{2}, & \text{if } d_p < A \\ d_p - A, & \text{otherwise} \end{cases} \qquad (6.10)$$
$$\gamma \leftarrow max\{min\{\gamma, \gamma_{max}\}, \gamma_{min}\}$$

The γ_{min} and γ_{max} denote the lower and upper bounds for d_p adjustment during each DOPpler's iteration. This limits the degree of change of d_p between consecutive iterations and reduces sensitivity to transient candidate impact on auto-tuning progress. In REACT, γ_{min} and γ_{max} are set to two and twelve respectively, in line with [366].

The Calibrator continues to monitor d_p and measurement characteristics until auto-tuning completion, continually adjusting d_p in response to measurement inaccuracy, inconsistency and candidate measurement failure rate. Transitively, Calibrator also adjusts the d_p to both the tensor operator characteristics and target-device capabilities.

For example, when performing auto-tuning of a highly computationally complex and large tensor operator, which utilises a large portion of the target-device's computational capabilities and/or memory resources, the Calibrator will adjust d_p accordingly (for example, reduce d_p), working towards reliable measurements being produced. Another example involves a target-device with very low available resources, which may be capable of only accommodating a few simultaneous candidate tensor program executions without violating its resource limits. In extreme cases, d_p may be reduced to one, whereby DOPpler's operation becomes equivalent to the serial infrastructure, in which case candidate re-measurements are avoided entirely until d_p is increased. Importantly, these

decisions are made automatically, reactively and transparently, without the need to provide target-device, auto-tuner or tensor operator characteristics ahead of time.

6.1.3.6 Candidate Rank Selection

To increase reliability of latency measurement results returned back to the user, upon auto-tuning completion, the Calibrator module re-measures Top-K best-found candidate tensor programs in isolation to re-affirm the latency measurement of the globally-best candidate found during auto-tuning. During this process, DOPpler reverts back to the serial measurement infrastructure. The number of re-measured candidates, K, was empirically determined (via sensitivity study - see Section 6.3.10) and set to 1% of $|C|$.

6.2 Experiment Setup

To determine DOPpler's ability at reducing auto-tuning time cost and maintaining auto-tuner optimisation quality, an experimental evaluation was performed using several SOTA auto-tuners, a variety of workloads (including standalone tensor operators and end-to-end DL models), and three unique hardware platforms. It is important to stress that this evaluation focuses on comparing the effects of using DOPpler's candidate measurement infrastructure vs. the conventional serial measurement infrastructure during auto-tuning. The evaluation does not attempt to compare the achieved optimisation quality of individual auto-tuners towards different tensor operators, models or target-devices.

6.2.1 Hardware, Software, Middleware and Auto-tuners

During experimentation, DL auto-tuning was performed towards three distinct target-device GPUs, as listed in Table 5.1 found in Section 5.3 of Chapter 5. These GPUs span three distinct architectures: Pascal, Turing and Volta, all of which are supported by the Nvidia Activity API of the CUPTI library used by DOPpler's PPM. Software

packages deployed on the hardware platforms were identical to the ones listed within Table 5.2 found in Chapter 5, Section 5.3. Likewise, the auto-tuners used were identical to those used during the Chapter 5 study, that is the AutoTVM, Chameleon (+Ansor), configured with default parameters, as per Tables F.5, F.6 and F.7 in Appendix F.

6.2.2 Workloads

Table 6.1: Details of DL models used during DOPpler's evaluation

Model	Parameters	FLOPs	Input Size	Top-1 / 5 Acc.	Details
AlexNet [168]	62.30M	725.00M	1x3x224x224	57.20% / 80.30%	Appendix D.1
SqueezeNet [130]	1.25M	20.00B	1x3x224x224	57.50% / 80.30%	Appendix D.2
MobileNet-V1 [121]	4.20M	569.00M	1x3x224x224	66.60% / 90.40%	Appendix D.3
ConvNeXt [194]	350.00M	60.90B	1x3x224x224	87.00% / 98.04%	Appendix D.5
VGG-16 [304]	138.40M	19.60B	1x3x224x224	71.59% / 90.38%	Appendix D.8

DOPpler's evaluation consisted of auto-tuning 36 tensor operators, as previously outlined in Table 5.3 found in Section 5.3 of Chapter 5, each accepting an input with batch size = 1. Further information about the specific tensor operator characteristics are provided within Table C.1 of Appendix C. Additionally, DOPpler was evaluated by auto-tuning tensor operators of end-to-end DL models, including AlexNet[168], SqueezeNet [130], MobileNetV1 [121], VGG16 [304] and ConvNeXt [194], as outlined in Table 6.1. The models were configured to accept inputs shaped $1 \times 3 \times 224 \times 224$ in the NCHW layout, and output a prediction tensor of shape 1×1000. Further information about the individual model architectures used is provided within Appendices D.1, D.2, D.3, D.5 and D.8. The models originate from several DL frameworks, including Pytorch [1] and Apache MXNet [163, 79], as well as converted using the ONNX [82] interchange format whenever necessary. Once ingested into the DL compiler, DL model graphs were optimised using graph-level optimisation at level 3 (see Sections 3.1.4 and 3.2, as well as Table E.1 found in Appendix E. Level 3 is chosen in line with existing works [40, 8, 385, 39], as it is the highest level of graph optimisation that does not modify the DL

model's operational logic and does not introduce optimisations that modify input or weight data - for example, by enabling mathematical approximations.

6.2.3 Performed Experiments

Experiments performed as part of DOPpler's evaluation consisted of auto-tuning standalone tensor operators and tensor operators of end-to-end DL models, initially using the conventional serial candidate measurement infrastructure, followed by the same auto-tuning scenario with the measurement infrastructure replaced by DOPpler.

Unless otherwise stated, during all experiments, each auto-tuner is set to propose at most 500 candidate tensor programs to be measured on the target-device, or otherwise stop early due to the schedule space containing < 500 candidates. The choice of 500 measurements follows the experimental evaluations of existing projects [8, 183], that measure from 150 up to 800 candidates. Furthermore, within these works, it can be observed that auto-tuning commonly converges around the 500-candidates mark, across substantial majority of the experiments published. To ensure DOPpler's reliability during prolonged auto-tuning sessions, experiments involving a larger number of candidate tensor program measurements (2000 trials) were also performed.

For end-to-end DL model experiments, auto-tuners are allowed to extract all compatible tensor operators from the model graph. Once extracted, each auto-tuner performs either 500 candidate proposals (and measurements) for each of the extracted tensor operators, or self-allocates the number of candidate proposals for each tensor operator, up to $O \times 500$ of total measurements per model, where O denotes the number of unique tensor operators extracted. The latter approach is adopted by the Ansor autoscheduler. Within experiments utilising multiple target-devices G, the serial measurement infrastructure conventionally performs $500/G$ measurements in a round-robin manner across the available devices, whilst DOPpler allocates d_p measurements per available device, performing $d_p \times G$ simultaneous measurements.

6.2.4 DOPpler's Hyperparameters and Sensitivity Analysis

The majority of DOPpler's hyperparameters were selected using methods practiced by the TVM auto-tuning community [39, 80, 8] as well as mimic those described within published methodologies of works that DOPpler is inspired by [23, 132, 366]. Any hyperparameters that are unique to DOPpler's design were determined empirically by experimenting with different values on a subset of auto-tuning scenarios (six tensor operators, one auto-tuner, one hardware platform), which represents 3.7% of all scenarios performed as part of DOPpler's evaluation, with results of sensitivity analysis presented within Section 6.3.10. Unless otherwise stated, DOPpler's static hyperparameters do not change across experiments involving different tensor operators, platforms or auto-tuners.

6.2.4.1 Number of Measurement Repeats During Auto-tuning

A common approach within TVM-based auto-tuners is to determine the number of candidate measurement repeats dynamically. This is achieved by allowing each candidate tensor program to continue executing repeatedly until a minimum amount of time has elapsed. As such, the number of measurement repeats is variable depending on the composition of the auto-tuning scenario (for example, tensor operator complexity, device capabilities), and can often exceed hundreds or thousands, given the parameter deciding the minimum elapsed time is often set to 1000ms. Other practitioners select a static number of measurement repeats - for example, three ($N = 1, R = 3$) or 100 ($N = 10, R = 10$), with the choice being fairly arbitrary. During DOPpler's evaluation, all auto-tuning experiments perform 60 tensor program executions per measurement ($R = 3, N = 20$), disabling the dynamically allocated number of repeats, in order to maintain fairness when comparing the measurement infrastructures across different tensor operators, platforms and auto-tuners. This choice is further evaluated empirically by varying the number of repeats for a small subset of auto-tuning scenarios.

6.2.5 Collected Metrics

During all experiments, auto-tuning time cost has been observed and compared between the serial and DOPpler's candidate measurement infrastructures. Additionally, the latency of the best-found candidate per auto-tuning scenario was also captured, to then analyse the impact on auto-tuning quality as a result of performing candidate measurements with DOPpler's infrastructure vs. serial. Furthermore, platform-wide utilisation of computational resources (CPU and GPU) were also collected. CPU utilisation was collected via a custom profiler based on Python's *psutil* library [274], whilst GPU utilisation was queried from the GPU via the use of Nvidia's NVML library [234]. In both cases, granularity of profiler queries was restricted to 0.5 - 1 seconds due to architectural and design limitations of the platform and middleware / libraries.

Table 6.2: Comparison of total time cost and latency improvement as a result of auto-tuning 16 classes of tensor operators with three auto-tuners (AutoTVM, Chameleon, Ansor) towards three GPUs (platforms A, B and C). Table presents aggregate results across the tensor operator classes, auto-tuners and platforms.

Tensor Operator Class	**Auto-tuning time cost reduction with DOPpler rel. Serial**	**Tensor program latency reduction (Serial)**	**Tensor program latency reduction (DOPpler)**
Conv1D-NCW	56.24% ± 12.96%	31.62% ± 8.55%	33.18% ± 9.26%
Conv1D-NWC	57.89% ± 13.41%	30.61% ± 8.23%	32.55% ± 7.88%
Conv2D-HWCN	36.41% ± 26.21%	48.57% ± 32.95%	46.19% ± 30.84%
Conv2D-INT8	55.61% ± 13.59%	86.35% ± 3.32%	82.81% ± 9.30%
Conv2D-NCHW	49.28% ± 32.03%	77.74% ± 8.05%	74.81% ± 9.84%
Conv2D-NHWC	49.93% ± 18.31%	63.14% ± 23.04%	66.09% ± 18.99%
Conv3D-NCDHW	47.48% ± 20.48%	98.61% ± 1.54%	97.36% ± 5.12%
Conv3D-NDHWC	48.97% ± 13.08%	94.42% ± 2.36%	94.51% ± 2.30%
Corr-NCHW	55.17% ± 10.04%	80.51% ± 6.59%	81.13% ± 6.28%
DEPTH-Conv2D	45.61% ± 16.31%	30.91% ± 15.78%	25.38% ± 10.53%
Dense	51.47% ± 13.95%	9.56% ± 4.38%	11.49% ± 4.22%
DenseINT8	51.68% ± 15.36%	93.28% ± 3.85%	92.74% ± 4.17%
GRP-Conv2D	52.15% ± 13.91%	97.56% ± 1.31%	98.05% ± 1.01%
MatMul	53.84% ± 21.57%	45.71% ± 9.37%	45.47% ± 8.85%
TConv1D-NCW	57.41% ± 11.85%	54.13% ± 6.23%	56.47% ± 5.28%
TConv3D-NCDHW	46.15% ± 14.65%	88.79% ± 2.52%	88.79% ± 2.77%

6.3 Evaluation Results

6.3.1 Auto-tuning Time Cost - Single Target-device

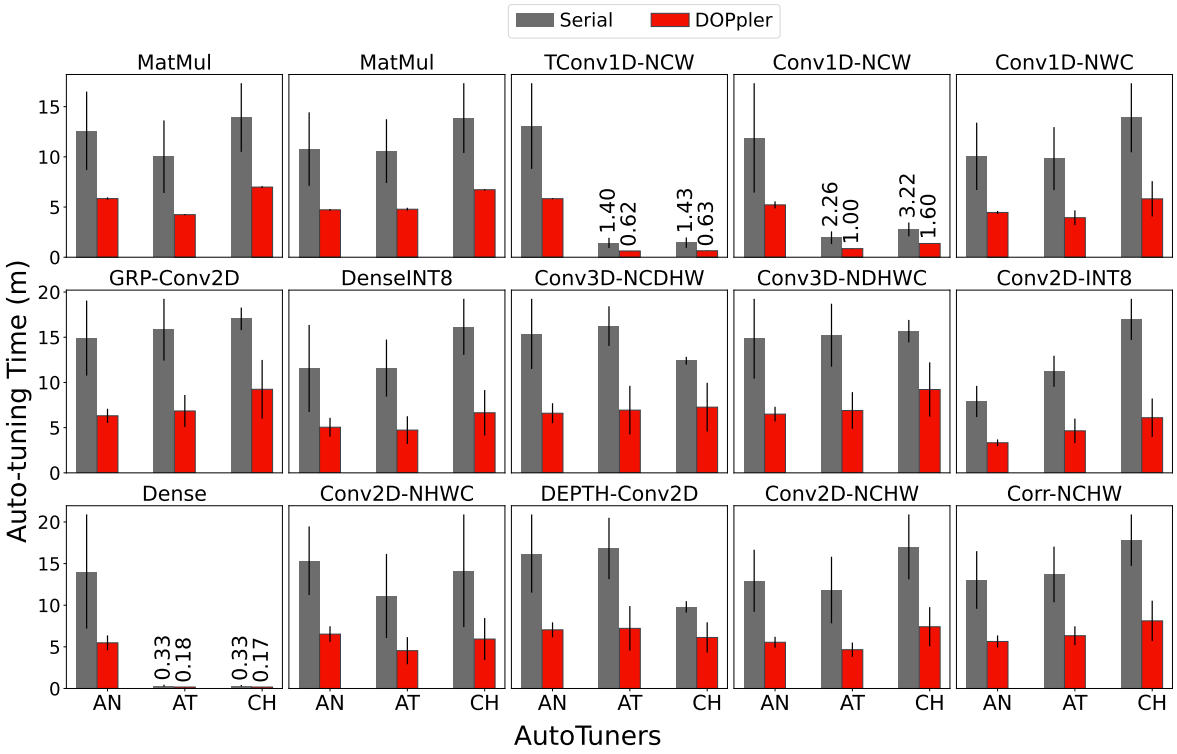


Figure 6.3: Auto-tuning time cost (minutes), when auto-tuning different classes of tensor operators with three different auto-tuners (AutoTVM, Chameleon, Ansor) across three platforms (A, B, C), comparing serial vs. DOPpler's measurement infrastructure.

As depicted in Figure 6.3 and Table 6.2, DOPpler was able to reduce auto-tuning time cost by 51.9%±18.6% compared to using the serial measurement infrastructure, across all studied platforms, auto-tuners and tensor operator classes. This stems from the increased measurement throughput when using DOPpler for candidate measurements, as a result of performing several simultaneous measurements on the target-device. Savings due to DOPpler can be particularly observed for less-complex or lower FLOP tensor operator classes such as Conv1D-NWC, Conv1D-NCW, TConv1D-NCW or Dense, where

auto-tuning scenarios that leveraged DOPpler, completed between 51.4 and 57.9% faster than those leveraging serial measurement infrastructure.

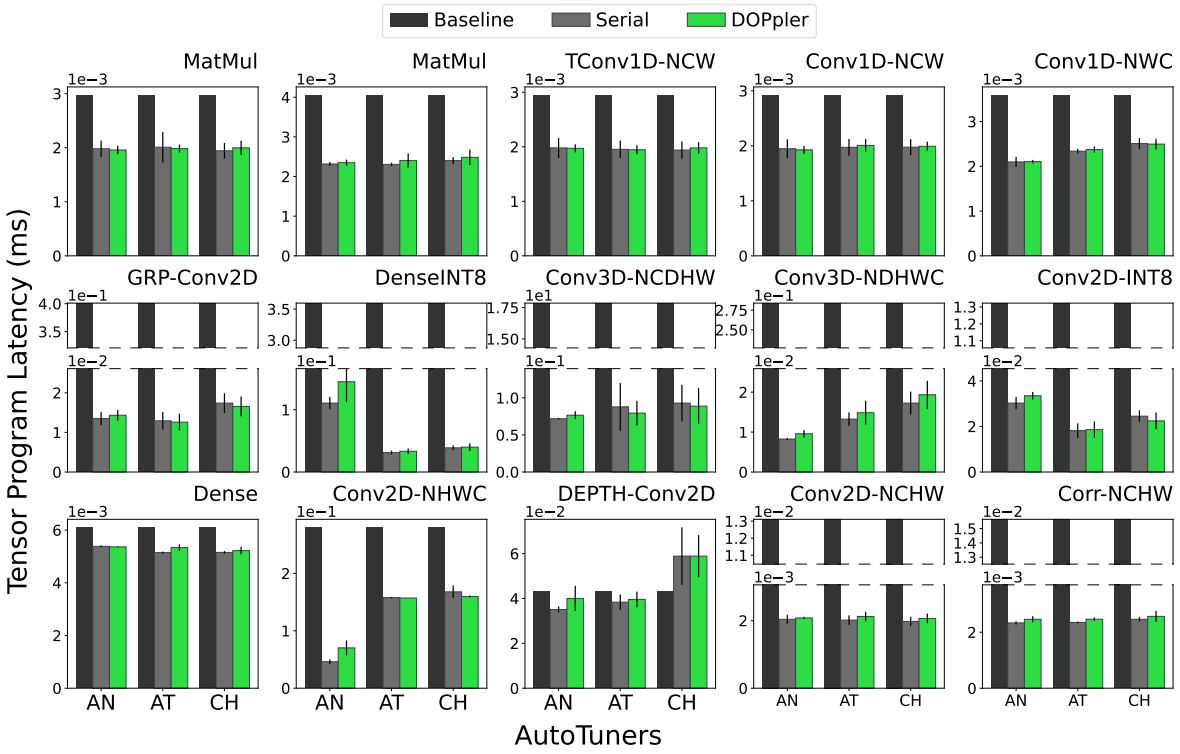


Figure 6.4: Achieved tensor program latency (ms) when auto-tuning different classes of tensor operators with three different auto-tuners (AutoTVM, Chameleon, Ansor) across three platforms (A, B, C), comparing serial vs. DOPpler's measurement infrastructure.

Within minority of experiment scenarios, DOPpler resulted in minimal auto-tuning time cost reduction, in specific combinations of tensor operators, auto-tuners and platforms - for example, DOPpler's was 2.53% faster compared to serial measurement infrastructure, when auto-tuning the MatMul tensor operator using Chameleon towards the GPU in platform B. When examined further, this minimal speedup was a result of a disproportionate number of erroneous scheduled being proposed by the auto-tuner, as well as timeouts, causing delays in measurement completion. Such erroneous candidate schedule proposals were observed primarily in auto-tuning with Chameleon and AutoTVM. Overall, the auto-tuning speedup when auto-tuning with DOPpler is in

part a result of better utilised platform's compute resources. Concurrently performed candidate measurements leverage more CPU cores compared to serial, confirmed by the overall increased CPU utilisation in DOPpler auto-tuning scenarios on all studied platforms: platform A (48.43%), platform B (54.23%) and platform C (51.46%).

6.3.2 Achieved Execution Latency - Single Target-device

As demonstrated within Figure 6.4 and Table 6.2, performing auto-tuning with DOPpler's measurement infrastructure, produced equivalent tensor operator optimisation quality compared to auto-tuning with the serial measurement infrastructure. On average, across all tensor operator classes, auto-tuners and platforms, there was ±1.37% difference in attained tensor program latency speedup between auto-tuning scenarios involving DOPpler vs. the serial measurement infrastructure.

Several experiment instances can be observed where one measurement approach outperforms the other with respect to attained tensor program execution latency, for example: auto-tuning Dense or MatMul tensor operators with the serial measurement infrastructure and auto-tuning Conv1D-NCW or DEPTH-Conv2D with DOPpler. This primarily stems from the non-deterministic exploration strategies of SOTA DL auto-tuners. This has been confirmed by performing an identical auto-tuning scenario with the serial measurement infrastructure ten times, which has yielded on average 2.16% deviation in achieved latency resultant from proposing different pattern of candidate schedules within each auto-tuning run. As such, the best-found candidate during auto-tuning with DOPpler, resides within an equivalent range to the best-found candidates proposed when auto-tuning with the serial measurement infrastructure.

For certain combinations of auto-tuners and tensor operators, auto-tuning with DOPpler discovered higher-latency candidates (Ansor + Dense-INT8). INT-8 (quantised) tensor operators are commonly considered esoteric and typically lack robust support within autoschedulers such as Ansor that rely upon rule-based schedule generation [385].

Despite that, auto-tuning with DOPpler maintains optimisation quality close to that of serial auto-tuning, in relation to un-optimised INT-8 schedules.

6.3.3 End-to-end DL Model Auto-tuning

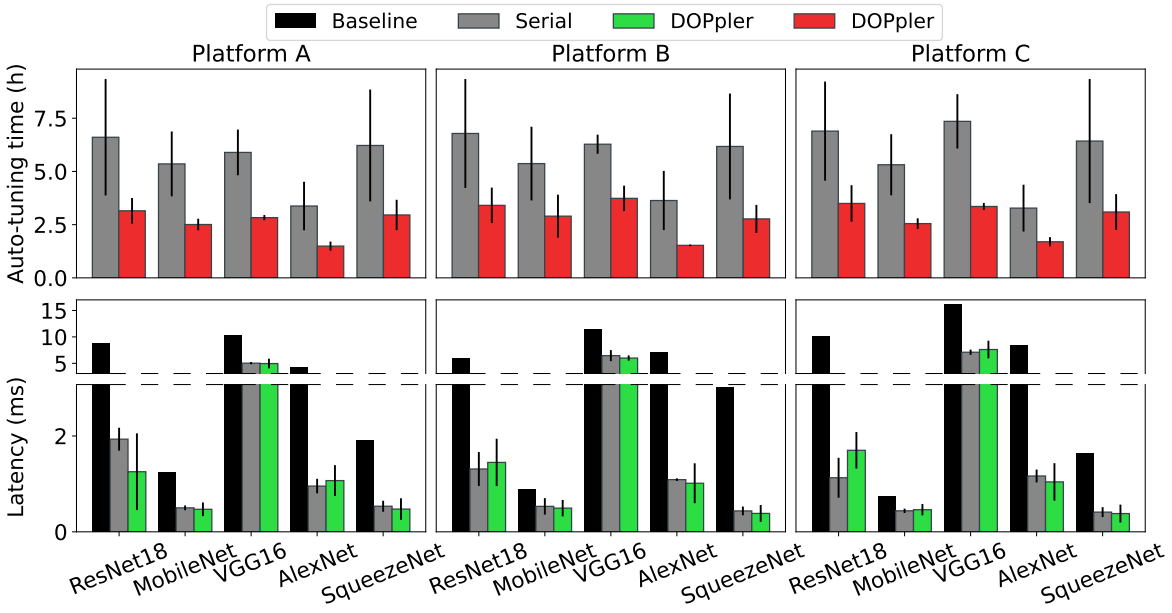


Figure 6.5: Comparison between DOPpler and serial measurement infrastructures, reporting achieved end-to-end DL model latency (ms) and total auto-tuning time cost (hrs) when auto-tuning five DL models using the Ansor autoscheduler. Each DL model layer tensor operator was auto-tuned until 500 measurements were performed in both serial and DOPpler infrastructure scenarios. Both **red** (auto-tuning time cost) and **green** (achieved inference latency) series in the Figure represent DOPpler.

Similar to standalone tensor operator auto-tuning, auto-tuning tensor operators extracted from end-to-end DL models with DOPpler, also benefits operational time cost and maintains model-level optimisation quality. As shown in Figure 6.5, auto-tuning end-to-end DL models with DOPpler, on average, reduced total auto-tuning time cost by 51.40% for all studied models and target-device combinations. In line with standalone tensor programs (see Figure 6.4), auto-tuning end-to-end DL models with DOPpler resulted in end-to-end model inference latency improvement equivalent to that produced when auto-tuning with the serial measurement infrastructure ($\sigma = 3.3\% \pm 7.8\%$).

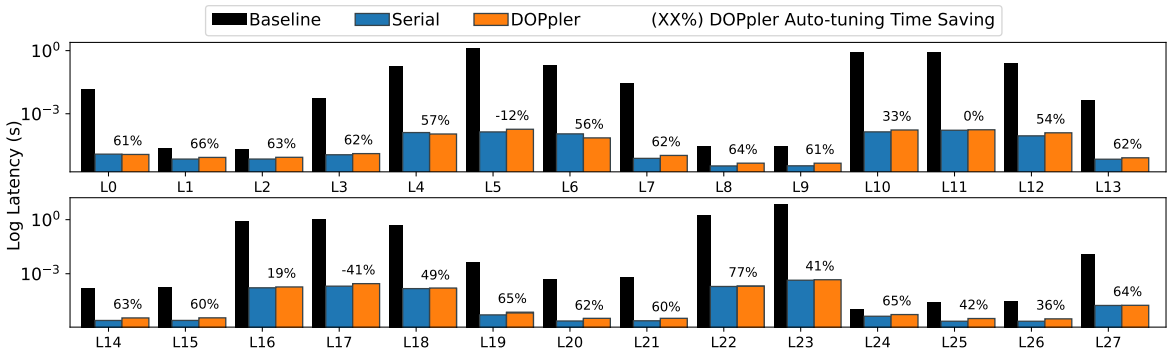


Figure 6.6: Achieved latency and total auto-tuning time cost improvement compared to serial measurement infrastructure, when auto-tuning 27 layer tensor operators of the ConvNeXt model using Ansor, leveraging DOPpler's measurement infrastructure. Each layer tensor operator was auto-tuned until 500 candidate measurements were performed.

On a DL model layer level, as shown in Figure 6.6 for the 27 unique tensor operators suitable for auto-tuning within the ConvNeXt DL model, these improvements are also observable. In these auto-tuning scenarios, DOPpler achieved sizeable auto-tuning time cost reduction (μ=48.67%, σ=27.0%), and achieved overall latency improvement equivalent to that of auto-tuning with serial measurement infrastructure. The auto-tuned layer tensor operators within the ConvNeXt model ranged in FLOP footprint between $4.7 \times e^{-6}$ and 0.92 GFLOPs. Within several scenarios (layers 5, 11 and 17) DOPpler auto-tuning is slower due to an increased number of invalid candidates proposed by the Ansor autoscheduler. This has caused DOPpler to perform more frequent re-measurements and thus reduced auto-tuning speedup.

6.3.4 Leveraging Multiple Target-devices

To evaluate DOPpler's ability at leveraging multiple target-devices during candidate measurements, six tensor operators were auto-tuned using Ansor, leveraging between one and four GPUs within platform A. Auto-tuning scenarios that utilised the serial measurement infrastructure, leveraged the RPC-based tracker / server round-robin measurements, with N RPC servers deployed locally for N used GPUs.

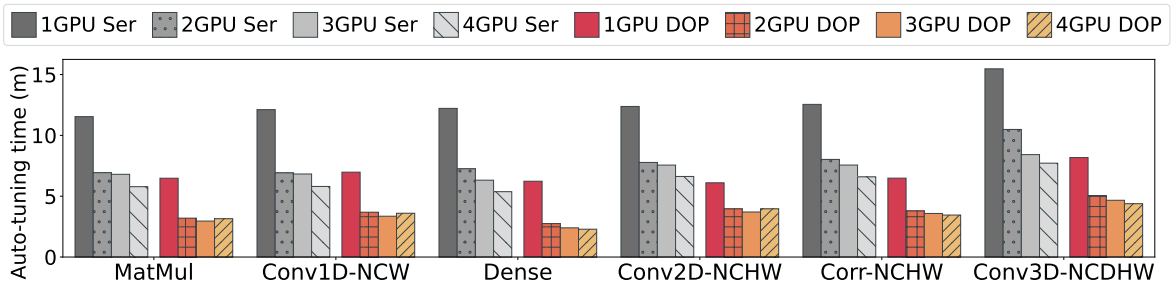


Figure 6.7: Depiction of the total auto-tuning time cost differences between the default TVM RPC-based multi-device measurement infrastructure vs. DOPpler's multi-device infrastructure, when auto-tuning six tensor programs using Ansor auto-tuner across one and more (up to four) devices used for measurement of candidate latencies.

As it can be observed within Figure 6.7, performing auto-tuning with DOPpler, whilst utilising a single target-device GPU for candidate latency measurements, yielded a time reduction equivalent to serial candidate measurement auto-tuning with 3.46 GPUs. This effect is further amplified when DOPpler can leverage multiple GPUs, providing further time reduction over the serial measurement infrastructure of 52.76%, 52.93% and 45.32% for two, three and four GPU scenarios.

Within specific auto-tuning scenarios such as MatMul, Conv1D-NCW or Conv2D-NCHW, it can be observed that leveraging three or more GPUs, yields a marginal 4.69% auto-tuning time cost improvement, in both cases of applying the serial measurement infrastructure and leveraging DOPpler. This stems from the platform's CPU cores becoming saturated by simultaneous candidate measurement procedure operations, limiting auto-tuning speedup.

Regarding optimisation quality, when leveraging multiple target-devices during auto-tuning with DOPpler, the best found tensor programs exhibited latencies that differed from those found during serial auto-tuning by between 0.98 and 1.35%, in line with expected auto-tuner deviation as previously presented within Table 6.2. DOPpler achieved on average 50.5% auto-tuning speedup across singular and multi-device auto-tuning scenarios, compared to the serial measurement infrastructure.

6.3.5 Auto-tuning Large Tensor Operators

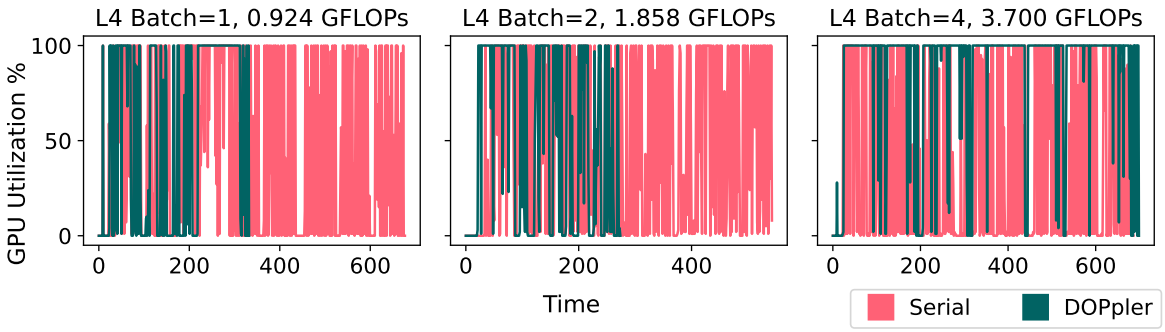


Figure 6.8: Comparison of achieved GPU utilisation during autoscheduling (with Ansor) Layer 4 of the ConvNeXt model with varied batch size (1, 2, 4) when relying upon serial vs. DOPpler measurement infrastructure. GPU utilisation was sampled using the Nvidia NVML library at one second intervals.

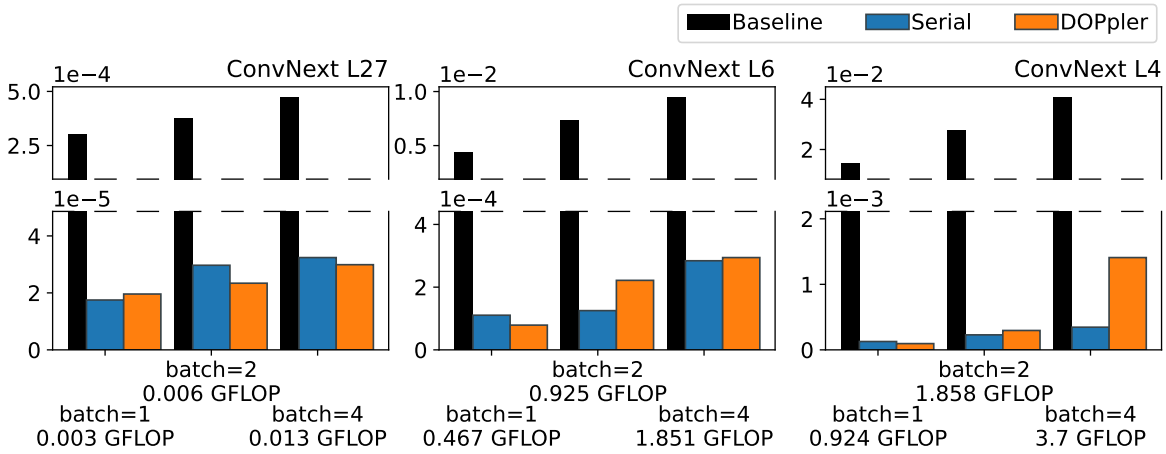


Figure 6.9: Achieved latency (s) when auto-tuning large layer tensor operators of the ConvNeXt model using Ansor and leveraging serial and DOPpler infrastructure for comparison. Batch sizes of the layers are varied between one and four to artificially increase CI (FLOPs).

To stress-test DOPpler's measurement infrastructure, several tensor programs that exhibit very high GPU utilisation were auto-tuned. Commonly, DL models that are considered "large", consist of a large number of low to medium complexity layer tensor operators, however, some models may contain tensor operator kernels that require a large number of floating point operations during computation, leading to high target-

device utilisation during their execution. This in turn may lead to increased likelihood of measurement inaccuracy and inconsistency when auto-tuning utilises concurrently performed candidate latency measurements.

Figure 6.8 depicts instantaneous GPU utilisation achieved during auto-tuning the tensor operator associated with layer 4 of the large ConvNeXt DL model. The tensor operators were then artificially enlarged in terms of their FLOPs requirement by increasing their input batch size up to two and four, resulting in required FLOPs of 0.924 GFLOPss to 3.70 GFLOPss. Modifications of batch size beyond four resulted in OOM errors stemming from violating the target-device resource limits.

Auto-tuning was successfully performed for these tensor operator auto-tuning scenarios using DOPpler's measurement infrastructure. As shown in Figure 6.8, these experiments resulted in high GPU instantaneous utilisation as well as increased frequency of on-device activity, compared to auto-tuning with the serial measurement infrastructure, also reducing auto-tuning time cost for the batch size one and two scenarios.

Despite the higher GPU utilisation and approaching resource limits, DOPpler was able to reduce auto-tuning time cost by on average 32.4% compared to serial auto-tuning, primarily by avoiding overheads associated with sequential tensor program launch procedures. In most cases, DOPpler auto-tuning maintained optimisation quality equivalent to serial auto-tuning, as depicted in Figure 6.9.

Figure 6.9 depicts achieved tensor program latency improvement when auto-tuning large tensor operators found in layers of the ConvNeXt model, with artificially inflated batch sizes, producing high-FLOP tensor operators. As the size and complexity of the tensor operator grew, optimisation quality decreased (see batch size four for ConvNeXT L4 example) due to increased number of OOM errors and timeouts resultant from violations of target-device resource limits. This behaviour is expected, and could occur within serial auto-tuning when considering isolated executions of tensor programs that require a large number of FLOPs to be performed during their execution.

In all cases, DOPpler reactively assigned lower d_p levels when auto-tuning these tensor operators (d_p between one and three). As such, whilst decreased d_p reduces auto-tuning time cost savings, it minimises likelihood of measurement inaccuracy. Within the ConvNeXt L4 (batch size four) example shown in Figure 6.9, DOPpler's achieved tensor program latency remains within the range of acceptable improvement, especially given the baseline (unoptimised, default schedule) exhibits an order of magnitude higher latency compared to both DOPpler's and serial measurement infrastructure result. This reaffirms that DOPpler is capable of providing good quality measurements when auto-tuning large tensor operators, including mitigating negative impact stemming from resource contention on target-device.

6.3.6 Number of Performed Measurements and Repeats

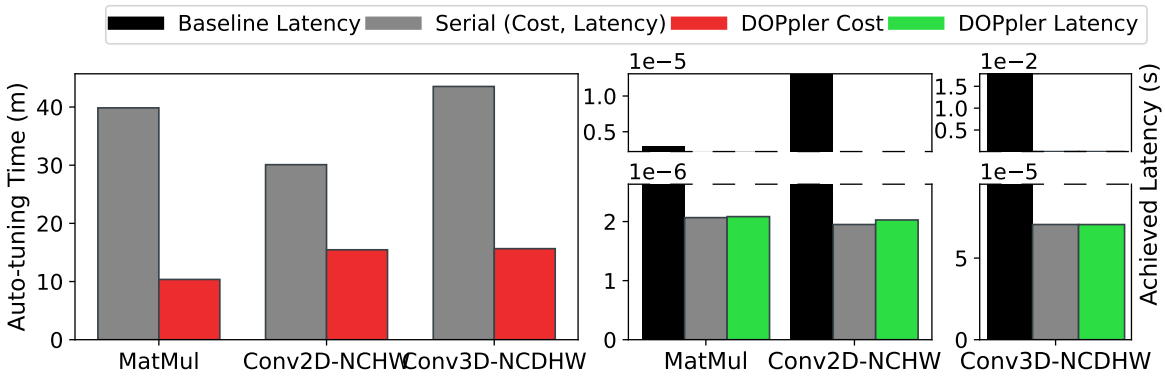


Figure 6.10: Impact on achieved tensor program latency and total auto-tuning time cost, as a result of applying DOPpler's measurement infrastructure during an increased number of candidate measurement trials (2000), when auto-tuning three tensor operators towards platform A with Ansor. Comparison with default serial measurement infrastructure.

To evaluate DOPpler's effectiveness during prolonged auto-tuning sessions (signified by more candidate latency measurements being performed), several tensor operator classes were auto-tuned using the Ansor autoscheduler towards platform A's GPU, continuing auto-tuning until 2000 candidate measurements were performed. As it can

be observed in Figure 6.10, auto-tuning tensor operators with DOPpler, resulted in equivalent optimisation quality attained by the auto-tuner, whilst reducing auto-tuning time cost by between 46.5 and 64.0% across the three considered tensor operators.

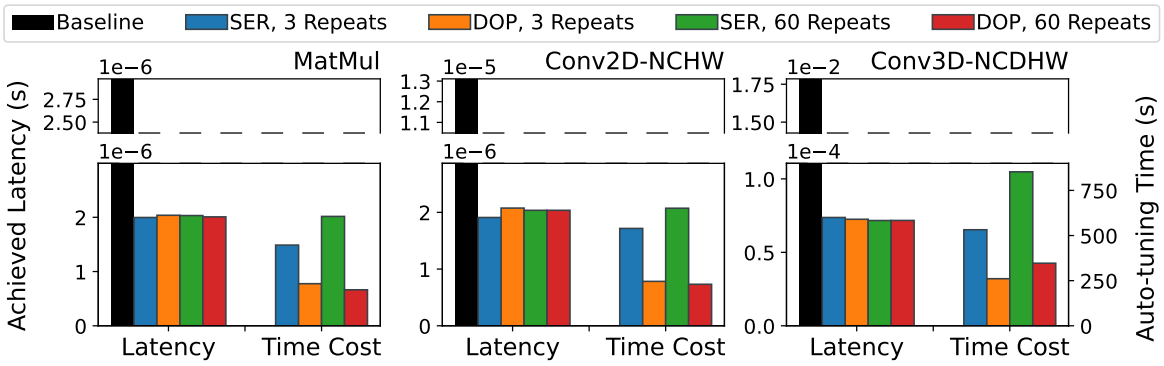


Figure 6.11: Impact of the difference in the number of repeated executions of the same candidate (3 vs 60 times) during its latency measurement on the achieved tensor program latency and the overall time cost of auto-tuning. Comparing between serial and DOPpler measurement infrastructures.

Furthermore, to evaluate DOPpler's ability to maintain optimisation quality and auto-tuning time speedup when performing a lower number of measurement repeats for each candidate tensor program, several tensor operators were optimised using the Ansor autoscheduler towards platform A's GPU using a reduced number (three from the default 60) of measurement repeats. In such scenarios, the auto-tuning time cost was marginally reduced, as depicted in Figure 6.11. Compared to serial auto-tuning with three measurement repeats per candidate, auto-tuning with DOPpler (with 60 measurement repeats) reduced auto-tuning by between 47.8 and 54.4%. The marginal difference in auto-tuning time between low and high number of measurement repeats stems from the fact that the majority of time spent during candidate measurements, involves process and context management, both in the case of serial and DOPpler.

In some cases (MatMul), utilising three measurement repeats as opposed to 60 (DOPpler's default) was disadvantageous for DOPpler as it resulted in increased measurement inconsistency, which required DOPpler to re-measure candidates more

often, ultimately reducing the time cost savings potential.

As such, by default, DOPpler utilises 60 candidate tensor program executions per candidate measurement. Figures I.7, I.8 and Table I.1 found in Appendix I provide additional information about the timings of individual operations happening during a candidate measurement and across an entire auto-tuning session, comparing DOPpler's approach vs. the serial measurement infrastructure.

6.3.7 d_p and Dynamic Timeout

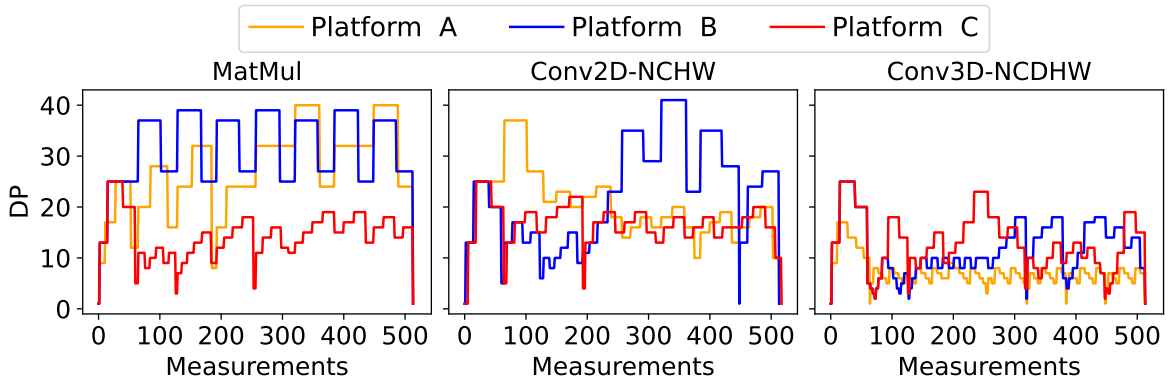


Figure 6.12: Pattern of change of DOPpler's dynamic degree-of-parallelism observed when auto-tuning three tensor operators towards GPUs of platforms A, B and C with Ansor.

As depicted in Figure 6.12, DOPpler continually adjusts d_p for different tensor operator and platform combinations, including during individual auto-tuning sessions as the characteristics of proposed candidate tensor programs change. Notably, DOPpler reduces d_p when auto-tuning increasingly complex tensor operators (for example, Conv3D-NCDHW), due to the increased measurement inaccuracy. Measurement inaccuracy becomes impactful more quickly for more complex tensor operators as the d_p increases. This is because lesser number of complex tensor programs can share the target-device GPU resources simultaneously than it would have been the case for a less complex tensor operator. DOPpler dynamically adjusts d_p levels in accordance with these characteristics

(for example, lowers d_p in scenarios involving the less capable Nvidia GTX 1080 GPU - platform C).

Figures I.1, I.2 and I.3 found in Appendix I depict how d_p varied across time when auto-tuning different tensor operators towards different platforms with the Ansor autoscheduler, while Figures I.4, I.5 and I.6 depict how the dynamically determined timeout setpoint has varied within DOPpler within the same auto-tuning scenarios.

6.3.8 Platform Utilisation

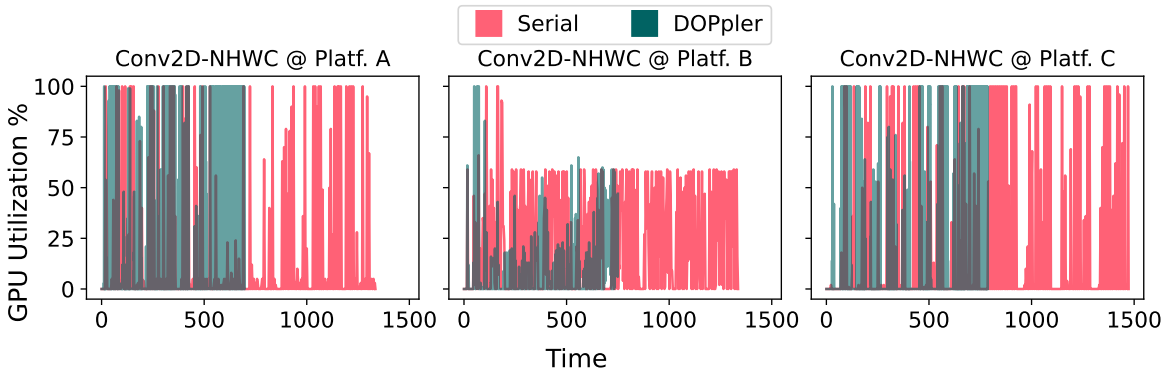


Figure 6.13: Comparison of achieved GPU utilisation during auto-tuning (with Ansor) a single Conv2D tensor operator, comparing serial vs. DOPpler's measurement infrastructure impact across three different platforms (A, B, C) containing different GPUs. Utilisation sampled using Nvidia NVML library API [234], which limits sampling granularity to once every 0.5 - 1.0s depending on GPU capabilities.

Performing auto-tuning using DOPpler's measurement infrastructure, increased platform's CPU utilisation across all considered platforms by 8% (A), 14% (B) and 24% (C), compared to the serial infrastructure. This stems from the increased number of candidate measurement processes being allocated to the available CPU cores, which also increased cost-efficiency by amortising the process waiting time. Figure 6.13 depicts GPU utilisation attained when auto-tuning several tensor operators using Ansor towards the three platforms (A, B and C). It can be observed that relying on DOPpler's candidate measurement infrastructure, causes the GPU target-device to be utilised more effectively.

This stems from more frequent kernel launch events when measuring using DOPpler, since kernels from candidate "B" no longer have to wait for candidate "A" to finish being measured - unlike when measuring serially.

6.3.9 Alternative Calibrator Policies

During DOPpler's evaluation, Calibrator policies other than REACT were considered for dynamically adapting d_p during auto-tuning measurements. The LEARN policy utilised a Sequential Model-based Bayesian Optimisation [30] and a Gradient Boosted Random Forest [38], and successfully provided measurements that maintained optimisation quality comparable to serial measurement infrastructure, however, resulted in 4.8% higher auto-tuning time cost compared to REACT across different tensor operators, auto-tuners and platforms. The LEARN policy had a tendency to *exploit* rather than *explore* different d_p ranges, and thus conservatively didn't react to changes in measurement characteristics as quickly as REACT does. Another explored policy called RLEARN, was based on RL and PPO [290], where the reward function was specified as follows:

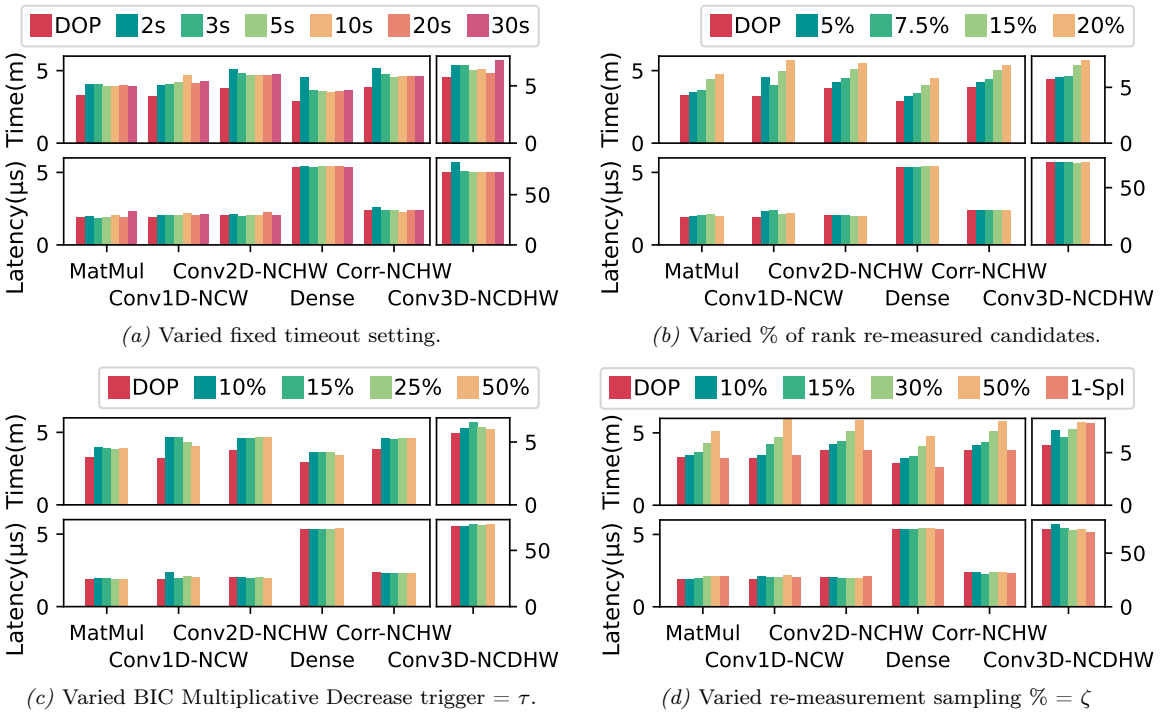
$$r = \alpha \times (1 - \frac{(\upsilon_{current} - \upsilon_{min}}{\upsilon_{min}}) + (1 - \alpha) \times (1 - \delta_{mean}) \tag{6.11}$$

with α controlling the tradeoff between auto-tuning time cost and quality of performed measurements. The reward function assigned high reward to a certain d_p range when both the auto-tuning time cost and measurement inaccuracy were small.

The RLEARN approach was unable to converge to an optimal d_p level and adapt to the changing operational circumstances appropriately, producing inaccurate measurements with d_p often "stuck" within a range discovered as appropriate in the beginning of operation. This is due to the need for large number of training samples in RL, to reach an optimal policy. The specific application area of quickly adapting d_p within DOPpler, rendered the RLEARN policy unsuitable for this particular purpose.

6.3.10 Hyperparameter Sensitivity Analysis

To determine how varying DOPpler's hyperparameters affects auto-tuning time cost and optimisation quality, six tensor operators were auto-tuned using the Ansor autoscheduler towards platform A's GPU, varying DOPpler's hyperparameter values.



(a) Varied fixed timeout setting.

(b) Varied % of rank re-measured candidates.

(c) Varied BIC Multiplicative Decrease trigger = τ.

(d) Varied re-measurement sampling % = ζ

Figure 6.14: DOPpler hyperparameter sensitivity study results. Reported achieved latency (μs) and total auto-tuning time cost (minutes) when auto-tuning six tensor programs until 500 measurements are completed using Ansor autoscheduler. "DOP" series represents DOPpler's default parameters obtained empirically or set dynamically during operation (timeout).

6.3.10.1 Dynamic Timeout

Figure 6.14a depicts differences in auto-tuning time cost and achieved tensor program execution latency of the best-found candidate, when varying the static candidate timeout setpoint parameter (2s, 3s, 5s, 10s, 20s, 30s, DOP {DOPpler}). DOPpler's dynamic timeout setpoint resulted in on average 18.84% reduction in auto-tuning time

cost compared to the static timeout setpoint of three seconds (default value used in AutoTVM, Chameleon and Ansor). The dynamically determined timeout setpoint is able to avoid both the excessive candidate pre-emptions, and the idle time spent waiting for the candidate tensor program kernel to return its results and allow measurement to be completed. At the same time, the dynamic approach can quickly adapt to the differences between tensor operator complexities and target-device capabilities. The results presented in Figure 6.14a suggest that there does not exist a one-size-fits-all timeout setpoint that is applicable to all tensor operator and platform auto-tuning scenarios, and that this can be alleviated using a dynamic approach.

6.3.10.2 Re-measurement Sampling Threshold

Figure 6.14d demonstrates differences in auto-tuning time cost and achieved tensor program execution latency of the best-found candidate, when varying the number of samples to be re-measured during Calibrator's operation (ahead of analysis)(ζ). DOPpler adopts 20% as the default number of samples to be re-measured during each iteration of the Calibrator's loop. Other sampling percentages were considered as part of the sensitivity study: 10%, 15%, 30%, 50% and one sample.

Whilst increasing ζ from one sample to 50% of measured candidates within the iteration, negatively impacted auto-tuning time by 30.1%, a low choice for ζ (for example, one sample) resulted in insufficient number of samples being re-measured, in turn causing δ_{mean} to become unreliable. This often led to infrequent Multiplicative Decrease events within the REACT policy, causing DOPpler to utilise d_p that is higher than optimal for that particular scenario, ultimately manifesting in moderately increased number of timeouts / OOM, reducing measurement reliability. DOPpler's 20% ζ, whilst re-measuring more samples than some of the other choices, facilitated the REACT policy in responding to quickly changing measurement accuracy and appropriate scaling of d_p such that auto-tuning speedup and measurement quality were maintained.

6.3.10.3 Multiplicative Decrease Threshold

To understand how the DOPpler's τ threshold, responsible for determining when to perform *Multiplicative Decrease* during the REACT update, affects auto-tuning time and achieved tensor program execution latency, several tensor operators were auto-tuned with varied τ threshold. The τ threshold triggers *Multiplicative Decrease* when δ_{mean} exceeds τ (expressed as a number between 0 and 1) across the iteration's population. For example, with τ set to 5%, when δ_{mean} exceeds it, *Multiplicative Decrease* will be triggered to reduce d_p in the next iteration. DOPpler by default utilises $\tau = 5\%$.

As demonstrated in Figure 6.14c, the lower the τ threshold, the more conservative the policy becomes, reacting to small variances in measurement accuracy and frequently triggering *Multiplicative Decrease*. On the other hand, permissive τ of 25 - 50%, results in quickly growing d_p (with *Binary Increase*), resulting in higher likelihood of intermittent timeout violations during measurement, as there are more simultaneous candidates in-flight. Additionally, the likelihood of OOM error occurrences increases as the target-device computational and memory limits are approached at high d_p levels.

The REACT policy allows d_p to continually increase until the τ threshold is reached. As such, low DOPpler's τ of 5% is appropriate when one of the primary goals is to maintain measurement quality, whilst attempting to continually increase d_p until high δ_{mean} begins to manifest. Setting τ lower than 5% sensitises the policy to the naturally occurring differences in measured execution latency, that occur both across DOPpler and serial measurement infrastructures during auto-tuning, as outlined in Section 6.3.2.

6.3.10.4 Rank Re-measurement

At the end of each auto-tuning session, DOPpler performs isolated rank selection re-measurement of the candidates measured during auto-tuning. This has been put in place to ascertain that the globally best candidate found as part of auto-tuning is returned

back to the user as the optimised schedule configuration. As demonstrated in Figure 6.14b, selecting a large percentage of candidates to be rank ordered and re-measured in isolation, drastically reduces auto-tuning time cost savings obtained via the use of DOPpler. For example, there is a reduction of 31.34% in total auto-tuning time cost savings when the rank selection percentage is varied from 1% to 20%; however, no major differences in achieved latency were observed as the rank selection percentage varied.

This suggests that small percentage of candidates can be selected for re-measurement in isolation, with little to no adverse effect on overall auto-tuning optimisation quality. As such, the default selection for the rank re-measurement percentage used in DOPpler is 1%. As the rank re-measurement percentage is a hyperparameter, it can be configured to an even lower value by the user if permissible given specific optimisation scenario (for example, ahead of time knowledge of perfect-accuracy measurements).

6.4 Discussion and Limitations

6.4.1 Auto-tuner Compatibility

DOPpler is capable of accelerating many prominent DL auto-tuners [40, 8, 385, 386, 183, 105, 377], as it replaces their default measurement infrastructure (isolated serial measurements) without the need for major modifications to their codebases. This is a significant advantage for projects such as NeuralMagic [136], Amazon SageMaker Neo [296], Ampere Computing [44] or OctoML [241] that rely upon DL compiler-based auto-tuning to optimise DL models at datacenter scale.

6.4.2 Target-device Compatibility

DOPpler has been evaluated with three distinct Nvidia GPU architectures (Pascal, Turing, Volta), due to their prevalence in DL research, and being commonly used in

DL auto-tuner research [385, 8, 40]. With high probability of success, DOPpler could also leverage other Nvidia GPU architectures such as the Ampere [229] A100 Graphics Processing Unit (GPU). This is because the Ampere architecture continues to utilise Nvidia Activity API, submission of kernels for execution using CUDA Streams, and has recently been used to evaluate other DL auto-tuning projects [321].

Furthermore, the Ampere architecture is one of the first GPUs within the Multi Instance GPU (MIG) class of GPU that enable virtual GPU instances to be partitioned and separately utilised by the user. Importantly, the separate MIG instances entail separate cache and DRAM access paths. If used for auto-tuning, the auto-tuner could target each of the individual virtual GPU instances as a target-device, further increasing DOPpler's utility as existing workloads would have to be re-optimised towards each unique instance (if differently partitioned). With minimal modification to the PPM, DOPpler could also support non-Nvidia (AMD or Intel) GPUs via OpenCL Cmd-Queues for multi-"Stream" workload submission and the *clWaitForEvents(...)* callback that enables kernel latency information collection directly from device [137].

6.4.3 Workloads compatibility

Whilst evaluated in part on CNN architectures, and a wide range of standalone tensor operator classes, DOPpler can support any DL workloads already supported by the auto-tuners compatible with DOPpler. It is important to stress that DOPpler does not actively analyse the tensor operator or DL model characteristics during its operation, and instead relies on a reactive approach that leverages real-time tensor program execution characteristics. As such, DOPpler has no dependencies on any particular workload type or target-device characteristics. Any limitations on supported workloads stem from the capabilities of auto-tuners themselves, for example, missing templates in template-based auto-tuners such as AutoTVM or Chameleon or incompatible schedule generation rules within autoschedulers such as Ansor.

6.4.4 Cost vs. Quality

DOPpler is capable of significantly speeding up auto-tuning time, without compromising optimisation quality attained by the auto-tuner. However, given the set of hyperparameters and design decisions adopted within DOPpler's Calibrator module, it can also be used to balance optimisation quality and measurement throughput (i.e. speedup). Controlling such balance may be advantageous, for example, in applications such as large-scale Neural Architecture Search (NAS) [19] that generate and evaluate thousands of candidate DL model architectures, where rapidly discovering ones characterised by high inference accuracy and low inference latency is paramount. Alternatively, DL optimisation MLaaS provider, servicing many customers, may desire to prioritise throughput of auto-tuning jobs over optimisation quality, for example, to offer 'fast but good' auto-tuning immediately, vs offering 'slow but excellent' auto-tuning that is enqueued for hours or days, occupying compute resources. Such scenario would require minimal modifications to the DOPpler's REACT policy and/or default hyperparameters.

6.4.5 Scalability

DOPpler has been evaluated in single-machine, single-target-device and single-machine, multi-target-device scenarios. It has not been evaluated in scenarios involving multiple machines. When considering deploying DOPpler in multi-machine scenarios, the default auto-tuner candidate batch size of 64 may not saturate multiple machines containing multiple target-devices. To remediate this, modifications must be made to the PPM, whereby multiple candidates are sent to different machines simultaneously. Increasing batch size, may affect the operation of the auto-tuner's cost model, given its re-training occurs once every batch of candidates. Understanding how best to split a batch of candidates across multiple machines containing variable number of target-devices is beyond the scope of this work, however, remains a foremost direction for future work.

Chapter 7

Conclusion

This chapter concludes the thesis by summarising the explored research area and the identified problems within the space of DL inference performance optimisation, followed by how the major research questions and contributions have been addressed. The costs incurred as a result of performing experiments associated with this work are also discussed, including how a change in geographical location or electricity generation fuel mix could affect the monetary and environmental costs when reproducing this research. The chapter concludes with a discussion on future research directions that could be explored by building upon the research contributions presented within this thesis.

7.1 Research Problem Summary

DL models are becoming increasingly integrated within industrial applications such as stock market analysis, autonomous driving, genomics or automated protein synthesis. In part, this has been enabled by the emergence of high-performance, massively-parallel processors such as GPUs that greatly accelerate complex DL computation, facilitating research progress. However, given the DL model operators each exhibit different computational footprint, without carefully designed implementations that take advantage

of these powerful processors, the computational resources become under-utilised. This leads to sub-optimal inference performance and large DL operational costs.

In recent years, several solutions such as graph-level optimisations and auto-tuning (both facilitated by DL compilers) have been proposed for optimising model inference performance, aligning model implementations towards target-device characteristics. DL auto-tuning demonstrates impressive inference latency improvements, however, it is a lengthy and energy-intensive process, requiring a large number of candidate schedules to be proposed and evaluated on target-devices in isolation (serially), to ascertain their performance profiles and discover optimal implementations.

A substantial portion of such candidate proposals exhibits high execution latency, thus not contributing positively towards optimisation, and at the same time, isolating the already under-utilised target-devices for prolonged periods of time. Moreover, SOTA DL auto-tuners optimise tensor operators sequentially, one at a time until completion, making it difficult to predict the effects of single tensor operator optimisation on the end-to-end DL model performance and to introduce operational cost controls.

Sub-optimal candidate proposals coupled with inefficient, serial candidate measurements and sequential operator optimisation, constrain auto-tuners from being used reliably at scale, where cost-efficiency and control over cost objectives are highly desirable system characteristics for DL deployment pipelines. Tackling these challenges has a potential to not only reduce costs of optimising DL inference performance, but also to reduce the barrier to entry to compiler-based optimisations.

7.2 Summary of Contributions

The key insight derived within this thesis is that within the space of DL performance optimisation, and more specifically auto-tuning, there exist a number of inefficiencies, stemming from strongly-held assumptions and design decisions, which result in cost-

ineffective and prolonged optimisation. Addressing the research hypothesis and questions posited within Chapter 1, the work presented within this thesis demonstrates that these inefficiencies can be measurably alleviated via targeted improvements to multiple components of existing DL auto-tuners, leveraging probabilistic, heuristic and reactive techniques. More specifically the following research contributions are proposed:

7.2.1 Analysis of DL Optimisation Costs and Inefficiencies

Chapter 3 presents an in-depth analysis of time and energy costs associated with performing DL inference optimisations. This analysis experimentally evaluated existing DL compiler-based optimisation methods (model graph level and low-level auto-tuning), across different optimisation scenarios, varying DL models and hardware platforms. Such experimental analysis discovered insights into inefficiencies plaguing existing DL inference optimisation. As described in Section 3.2.3, the costs of DL model graph transformations are negligible, given their substantial end-to-end latency improvements when applied. This is not the case during low-level DL auto-tuning that targets individual tensor operators within a DL model. DL auto-tuning, while offering substantial performance improvements is notoriously slow, partly due to the enormous candidate schedule space it has to navigate, and partly because some of the discovered candidate schedules must be measured on the target-device in isolation.

Additionally, during the analysis, *cold* candidates were discovered - proposed schedules that exhibit poor performance despite being proposed farther in the auto-tuning process than otherwise expected, with their impact further propagated to the measurement process resulting in additional operational costs. Quantifiably, as a result of their proposals and measurements, *cold* candidates result in between 28 - 43% of additional energy costs during auto-tuning. Chapter 3 concludes with recommendations on design directions for future DL auto-tuners that would help to mitigate impact of *cold* candidates during DL auto-tuning and improve auto-tuning cost-efficiency.

7.2.2 Cost-efficient DL Auto-tuning Filtering and Meta-tuning

Chapter 4 builds upon the design directions established within Chapter 3, proposing Trimmer - a multi-component, cost-efficient DL auto-tuning filter and meta-tuner, capable of reducing auto-tuning time and energy costs. Trimmer's design enables it to introduce new cost objectives when auto-tuning singular and multiple end-to-end models across standalone target-devices or clusters of machines. To alleviate the impact of *cold* candidates, Trimmer introduces an ANN-based filter that predicts whether a candidate proposed by the auto-tuner is a *cold* candidate, and replaces such candidates with previously unexplored ones, leading to more cost-effective auto-tuning.

At the single and multi-model levels, Trimmer introduces Survey tuning - a meta-tuning approach for scheduling tensor operator auto-tuning sessions. For the first time, Survey tuning enables end-to-end DL model inference latency to influence auto-tuning progress and control operational costs. While conventional auto-tuners await completion of all tensor operator auto-tuning sessions before measuring end-to-end DL model latency, Trimmer's Survey tuning performs partial auto-tuning of all tensor operators, measures the end-to-end model latency, and determines which tensor operators should continue to be optimised. Trimmer's policy takes into account model-level latency improvements as well as relative improvements across tensor operators when making these decisions.

The same decision making process is also applied at multi-model level, whereby models that exhibit substantial performance improvement during their optimisation, are permitted to continue being optimised. Such an arrangement enables Trimmer to utilise cost objectives such as a ratio of inference latency improvement over time, or custom objectives to be specified by the user, including time or energy budgets. Based on empirical evaluation, compared to SOTA DL auto-tuners, Trimmer delivers the most cost-efficient auto-tuning and reduces optimisation energy costs by between 21.8 and 40.9% when used in optimisation scenarios utilising a cluster of machines.

7.2.3 Analysis of Parallel Candidate Measurements

During experimentation it became apparent that one of the major bottlenecks causing prolonged DL auto-tuning is the serial candidate measurement infrastructure, utilised by majority of SOTA DL auto-tuners. Performing candidate measurements serially ensures measurement accuracy, however, it is time consuming and isolates the target-device.

In Chapter 5, a naïve solution involving parallel candidate measurements is proposed (NPM), acting as an exploratory analysis experiment towards breaking the aforementioned assumption. This involved comparing the effects of concurrent candidate measurements at different degrees of parallelism, across diverse operator classes, auto-tuners and hardware platforms, determining reasons behind concurrent measurement inaccuracy, and design directions for a more comprehensive solution.

Crucially, it has been identified that most of the inaccuracy during parallel candidate measurements stems from CPU-level measurements, inherently subject to latent phenomena of process blocking and waiting for multiple simultaneously executing GPU workloads. Furthermore, the analysis suggests that when no controls are put in place within the naïve solution, a high degree of simultaneous measurements increases measurement timeouts or exhausts device resources (for example, DRAM).

This suggests that to achieve fast yet accurate auto-tuning measurements, a trade-off must be made between high degree of concurrency, measurement accuracy and minimisation of measurement failures. A more comprehensive solution for reliable intra-device parallel measurements, must account for inherent differences in tensor program execution patterns, reacting to sudden changes in such patterns during auto-tuning.

7.2.4 Parallel Intra-device Measurement Infrastructure

Given the design directions established in Chapter 5, we propose DOPpler - a reliable, intra and inter-device parallel candidate measurement infrastructure, described within

Chapter 6. DOPpler builds upon the naïve solution presented within Chapter 5, by introducing an adaptive candidate measurement component that is compatible with existing DL auto-tuners.

To ensure appropriate degree of parallelism is selected during each auto-tuning session and across time, DOPpler continually analyses optimisation quality based on measurement accuracy and consistency metrics, and reactively modifies the number of in-flight candidates using a method adapted from TCP BIC. At the same time, to maintain measurement reliability, DOPpler performs on-device measurements, leveraging GPU's internal kernel timing reports as opposed to CPU-level timestamps.

To facilitate the trade-off between maximising parallelism during measurements and minimising the timeouts and runtime errors, DOPpler adapts the timeout setpoint for each measured candidate in real-time, adjusting it in accordance with the current degree of parallelism, and the number of available target-devices. DOPpler's experimental evaluation demonstrates that on average, DOPpler reduces auto-tuning time costs by 50.5%, when evaluated on a large variety of standalone tensor operator classes and DL models, acting as the measurement infrastructure layer for several SOTA DL auto-tuners. The experiments include auto-tuning scenarios that utilise singular and multiple target-devices simultaneously, where in all cases DOPpler outperforms the conventional, serial measurement infrastructure, whilst maintaining equivalent optimisation quality.

7.3 Review of Research Questions

The insights derived from the research presented within this thesis, and the artefacts developed to tackle the identified problems, collectively answer the research questions posited within Section 1.3 of Chapter 1. More specifically, this section identifies the connections between discussions provided in each Chapter of this thesis and the aforementioned research questions.

[RQ1] *How significant are the time and energy costs of applying high and low-level DL inference performance optimisations and where do they originate from, when analysed across individual tensor operators, end-to-end models and DL auto-tuner levels? Could such cost analysis identify key architectural reasons for prolonged and cost-inefficient DL auto-tuning, considering different optimisation scenarios?*

This question is answered in particular in Chapter 3, where cost implications of high and low-level DL model optimisations are discussed, with Chapters 4, 5 and 6 also experimentally quantifying these optimisations. More specifically, Section 3.2.3 demonstrates that high-level optimisations cost less compared to low-level optimisations, and intuitively, in majority of the cases, their costs are proportional to their optimisation complexity and achieved performance improvement.

DL auto-tuning exhibits more complex relationships between optimisation quality and operational costs. As shown in Section 3.3, operational cost patterns vary substantially across auto-tuners, with less complex methods (for example, Random) exhibiting large operational costs. This can be attributed to lower quality candidate proposals that are measured on target-device in isolation, occupying compute resources while achieving sub-optimal performance improvements.

Within Chapters 3 and 5, it has been identified that the occurrence of *cold* candidates, coupled with serial measurements performed in isolation on the target-device, are significant reasons for inefficient and prolonged DL auto-tuning. The entirety of Chapter 5 focuses on iteratively exploring the inefficiencies of serial candidate measurements and describes an evaluation of a naïve approach to parallelising candidate measurements intra-device. Section 3.4 of Chapter 3, identifies and quantifies the issue of *cold* candidates and demonstrates that they can severely impair auto-tuning efficacy, preventing more favourable *hot* candidates from being measured more frequently.

[RQ2] *Is it possible to train a DL model to identify low-quality (high latency) tensor program candidates ahead of their latency measurements, during DL auto-tuning? If so, could such a model be leveraged to reduce the negative impact of poor candidates on auto-tuning operational costs?*

In Chapter 4, Trimmer was proposed. Trimmer incorporates an ANN-based candidate filter capable of identifying and filtering out *cold* candidates during auto-tuning. Within experimental evaluation of Trimmer, it has been demonstrated that it is indeed possible to leverage such a model for *cold* candidate filtering, and that utilising such filtering improves cost-efficiency of auto-tuning substantially.

Trimmer combines this ANN-based approach with an ϵ-greedy exploration policy that adapts the balance between exploitation and exploration of tensor program candidates when utilising them to advance auto-tuning progress. The efficacy of Trimmer's filtering model has been experimentally shown in Section 4.3.

[RQ3] *Could the end-to-end DL model inference latency be measured and subsequently leveraged to control the auto-tuning process for cost-effectiveness trade-offs, whilst auto-tuning of the model operators is underway?*

This question has been answered by developing Trimmer's Survey tuning module, as presented in Chapter 4, with its evaluation presented within Section 4.3.

As Trimmer's Survey tuning facilitates performing end-to-end DL model inference latency measurements during auto-tuning, Trimmer can leverage this information to preemptively suspend auto-tuning of tensor operators that do not exhibit significant performance improvement over time. Trimmer re-uses this strategy for multi-model auto-tuning scenarios where it redistributes auto-tuning time and computational resources towards more promising auto-tuning sessions and models.

This is in part enabled by performing partial (batched) auto-tuning of all tensor operators within the model, permitting limited candidate measurements to be

performed, and frequently checking how the auto-tuning performed so far, has impacted the overall DL model inference latency. Given more control is now possible over the progress of end-to-end model auto-tuning, Trimmer's Survey tuning can facilitate adding new cost objectives when auto-tuning singular and multiple end-to-end DL models.

[RQ4] *Could the long-standing requirement for serial, isolated candidate measurements during DL auto-tuning be avoided, instead enabling parallel candidate tensor program execution intra-target-device reliably? If so, what would be the optimisation quality and operational cost implications of such an approach, when applied across different classes of tensor operators, DL models, auto-tuners and target-devices?*

To answer this question, an experimental analysis has been performed, involving existing SOTA DL auto-tuners and their serial measurement infrastructures, as well as a naïve parallel measurement infrastructure, as presented in Chapter 5.

This approach enabled to identify that naïvely parallelising candidate measurements intra-device, exhibits both measurement inaccuracy and inaccuracy inconsistency, reaffirming the reasoning behind the requirement for isolated candidate measurements in SOTA DL auto-tuners. These phenomena occur largely due to the widely-adopted method of performing measurements (CPU-level timestamps), which is inherently affected by latent phenomena caused by process blocking when executing multiple GPU workloads simultaneously.

Furthermore, Chapter 5 describes and quantifies phenomena such as timeouts and runtime errors, frequently occurring during naïvely-parallel measurements at high degrees of parallelism, suggesting that high degrees of measurement parallelism do not always translate into substantial time cost reduction during auto-tuning.

All together, these insights helped to form design directions for a more reliable parallel measurement solution - DOPpler, presented in Chapter 6. To

alleviate measurement inaccuracy, DOPpler introduces a direct, on-target-device measurement technique, leveraging GPU vendor libraries and APIs. At the same time, to ensure the degree of parallelism is appropriate for the given optimisation scenario, and to mitigate inaccuracy inconsistency, DOPpler proposes the Calibrator module, which continually analyses candidate measurement quality and adjusts the operational conditions using a reactive policy.

DOPpler's experimental evaluation and hyperparameter sensitivity analyses, demonstrate that parallelism can be leveraged to substantially reduce DL auto-tuning operational costs, while maintaining high optimisation quality.

Experimental analyses of DL optimisations, and in particular DL auto-tuning, helped to identify design directions for two distinct yet complimentary solutions (Trimmer and DOPpler). Their empirical evaluations demonstrate that it is possible to reduce operational costs associated with DL auto-tuning and retain optimisation efficacy by addressing multiple inefficiencies at different levels of the DL auto-tuner design architecture. Importantly, answering the above research questions, validated the hypothesis posited within Section 1.3 of Chapter 1.

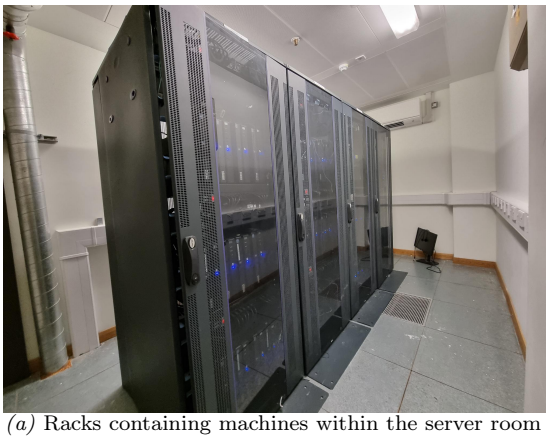
7.4 Self-analysis of Research Costs

Globally, DL computation is doubling every three to four months, which has resulted in an overall $300,000\times$ increase of required DL compute capability during the period between 2012 and 2018 [14]. These increases in demand, translate into increases in energy consumption attributed to DL models, both in terms of their development and use for inference [319], which in turn translate to increased Greenhouse Gas (GHG) emissions. Strubell et al. [318] report that training a large NLP Transformer model to convergence, including a priori NAS, results in GHG emissions equivalent to those produced by a globally average human during 60 years of their life, or nearly equivalent to the emissions

produced by five mid-sized vehicles since their production until decommissioning. Given the substantial environmental costs of performing DL research, several recent works [291, 115] advocate for at least minimal reporting of the energetic footprint required to replicate published DL research results, in the hope of creating more responsible and accountable research moving forwards.

The following analysis presents estimates for the time, energy, monetary and environmental costs incurred as part of the experimental studies and evaluations of the work presented within this thesis. Furthermore, the implications of geography and energy source composition (fuel mix) are discussed, projecting potential variance in these costs across different geographical regions within and outside of the UK borders.

7.4.1 Assumptions and Context



(a) Racks containing machines within the server room

(b) Mitsubishi Electric PKA-M100KAL CRAC

Figure 7.1: Compute and cooling equipment within the server room

7.4.1.1 Location and Temperature Control

All experiments performed as part of the research presented within this thesis, utilised machines located within Lancaster University UK, and more specifically a single room dedicated to housing the compute equipment utilised by the Experimental Distributed

Systems Lab, as shown in Figure 7.1. The room contains several server racks, that in the period of 2019 - 2022 contained between 50 and 150 machines. The room's temperature is controlled via a wall-mounted Mitsubishi Electric PKA-M100KAL Computer Room Air Conditioning (CRAC), which is rated at 10kW of cooling capacity, drawing a maximum of 2.81kW during operation [70].

The computational resources located within this room are shared across multiple research projects within the areas of DL, distributed systems, computer networking, resource management and security, all with different workload execution patterns and computational, energy and heat dissipation footprints. Since only a portion of machines were leveraged for the experiments, at intermittent time intervals across the span of three to four years, it is beyond the scope of this work to account for the proportion of energy consumed by the networking equipment, the CRAC unit, or the transmission and transformation losses originating from the power delivery to the server room.

7.4.1.2 Location and Energy Sourcing

To understand how energy consumption can differently affect financial and environmental costs, it is important to understand the context in which the electricity is sourced and used. According to reports on environmental sustainability, Lancaster University is supplied by several electricity sources: around 14% of total electricity is sourced from on-premises wind turbine, between 25 and 40% from a natural gas powered Combined Heat and Power (CHP) engine and the rest from the national power grid [341].

GHG Emissions: UK's national power grid is composed of several distinct downstream electricity generation sources, each with different proportion of contribution to the overall energetic grid composition, including: natural gas (65.8%), nuclear (14.5%), biomass (7.3%), coal (3.6%), hydro (2.0%), wind (4.7%), offshore wind (0.9%) and other (1.2%) respectively [340, 119]. Each of these downstream sources results in different environmental footprint, measured in grams of carbon dioxide equivalent per

kilowatt-hour of electricity generated (g CO2 eq/kWh).

On average, in the years 2019, 2020, 2021 and 2022, UK-wide national grid carbon intensity factor of electricity generation was reported as 264.39g CO2 eq/kWh, 242.15g CO2 eq/kWh, 264.51g CO2 eq/kWh and 261.55g CO2 eq/kWh respecively (avg. 258.15g CO2 eq/kWh) [71]. In terms of the other electricity sources at Lancaster University, on average, UK wind power generators result in low levellised GHG emissions of 11.8g CO2 eq/kWh [119, 333]. As reported by Rehva [285], a natural gas powered CHP installed in the UK in 2014, has an average carbon intensity factor of 217g CO2 eq/kWh.

As such, it is estimated that on average, dissipating 1kWh of electricity at Lancaster University during the period between 2019 - 2022, resulted in GHG emissions of 207.20g CO2 eq/kWh in the best case scenario (40% sourced from CHP) and 213.37g CO2 eq/kWh in the worst case scenario (25% sourced from CHP), non-inclusive of transmission losses or otherwise incurred power delivery losses.

Financial Costs: UK national grid prices for non-domestic electricity use were on average £0.1224/kWh, £0.1303/kWh, £0.1431/kWh and £0.1839/kWh in the years 2019, 2020, 2021 and 2022 respectively, with an average price across the duration of this research work of around £0.14/kWh [339]. Likewise, in the case of natural gas prices, the UK's Department for Business Energy and Industrial Strategy reports that 1kWh worth of natural gas costed £0.0232, £0.0228, £0.0290 and £0.0443 (avg. £0.0298) in the years 2019, 2020, 2021 and 2022 respectively [339].

Furthermore, relating natural gas prices to Lancaster University's CHP engine, it is estimated that for every 1kWh of electricity generated, 1.1kWh of heat is also released and recaptured for residential heating [338]. As such, to generate 1kWh of electrical energy, the CHP must consume up to 2.1kWh worth of natural gas. Estimating financial costs of operating the wind turbine, as of 2015, Thomson and Harrison [333] report that during a lifetime of a typical UK wind turbine, an average levellised (deployment and operation) cost is £0.083 for every 1kWh supplied.

As such, it is estimated that dissipating 1kWh of electricity at Lancaster University during the period between 2019 and 2022, costed on average £0.101 in the best case and £0.112 in the worst case, non-inclusive of any transmission and power delivery losses.

7.4.1.3 Machine Power Model

Experimentation performed as part of Chapters 3 and 4 made use of energy and instantaneous power measurements to quantify costs of performing DL optimisations. As previously outlined in Section 3.1.6, energy metrics were collected using Nvidia NVML library for GPUs and the RAPL MSR interface for CPUs. During experiments described within Chapters 5 and 6, energy sampling for CPUs was not performed, instead collecting machine-wide CPU utilisation during every experiment, while GPU instantaneous power samples were collected using Nvidia NVML.

Accounting for Other Costs: Since machine-wide energy data was not collected in any of the experiments performed, other costs such as the energy consumed by the machine's motherboard, DRAM memory, internal cooling fans or the energy lost during transformations at the Power Supply Unit (PSU) were instead estimated using available information about their power draw, found in data sheets published by their respective manufacturers. For experiments performed as part of Chapters 5 and 6, CPU instantaneous power was modelled by leveraging a widely used power curve model proposed by Fan, Weber, and Barroso [75], which relies on CPU utilisation, minimum idle power draw and maximum safe TDP to estimate average power drawn.

Energy lost as part of PSU power transformations was accounted for by observing PSU efficiency designations. For example, a PSU with 85% energy efficiency, delivers up to 85% of energy drawn from the socket to the machine's components, with the total out-of-socket power equal to component power draw $\times 1.15$. This has been adjusted for every unique machine. Energy costs associated with the server room's CRAC cooling system, the networking equipment or any other auxiliary costs were not accounted for.

7.4.2 Experimental Research Costs

Table 7.1: Estimated time, energy, environmental and financial costs incurred as a result of performing experimentation presented within this thesis. GHG emissions and financial costs calculated for the Lancaster University setting. Results grouped by Chapter (CH)

CH	Total Experiments	Auto-tuning Sessions	Candidate Measurements	Compute Time (h)	Energy Consumption (kWh)	GHG Emissions (kg CO2 eq)		Electricity Costs (£)	
						Best	Worst	Best	Worst
3	1760	2160	1,080,000	2003.12	751.64	155.74	160.38	75.91	84.18
4	293	3871	1,935,500	3122.76	1246.23	258.22	265.91	125.87	139.57
5	30,180	30,180	15,090,000	9944.03	3187.26	660.40	680.07	321.91	356.97
6	10,805	14,345	7,172,500	8684.59	2682.96	555.91	572.47	270.97	300.49
Total	**43,038**	**50,556**	**25,278,000**	**23,754.51**	**7868.10**	**1630.28**	**1678.84**	**794.67**	**881.22**

Table 7.1 presents estimates for time, energy, environmental and financial costs incurred as a result of experimentation related to this thesis. As it can be observed, both the GHG emissions and monetary costs are higher when Lancaster University's CHP constitutes 25% of total electricity generation (vs. 40% in the best case). In comparative terms, given the most favourable conditions, the 7868.10kWh consumed as part of the experimentation, would be enough to drive 48,269 miles in a standard Tesla Model 3 electric vehicle, boil the kettle enough times to brew 251,779 cups of tea or heat an average UK home for almost two years using an electric heat pump.

Different world regions exhibit different carbon intensity factors and costs of electricity per kWh, and as such, affect the potential footprint of performing computationally expensive experiments, as shown in Table 7.2. For example, to replicate the experimentation in the UK, selecting a location with the average UK-wide non-domestic electricity costs, it would have cost on average 28% more in terms of electricity and released 20% more GHG emissions than it did at Lancaster University. The differences in monetary costs are even more stark between domestic and non-domestic electricity tarrifs in the UK. For example, if a researcher from the UK was to replicate experiments associated with this thesis in their own home, using the exact equipment, it would

Table 7.2: Estimated environmental and financial costs incurred as a result of performing experimentation presented within this thesis in different locations. Carbon intensity data obtained from [96] while electricity prices (as of December 2022) obtained from [122]. Financial conversions performed at rate 1 USD = 0.81 GBP. Best case scenario selected for Lancaster University

Location	**Carbon intensity (g CO2 eq/kWh)**	**Total GHG emissions (kg CO2 eq)**	**Avg Electricity Price/kWh(£)**	**Total Electricity Cost (£)**
Sweden	12.00	94.417	0.114	898.61
France	58.00	456.350	0.140	1102.55
Lancaster University*	210.12	1630.280	0.101	794.67
United Kingdom (non-dom)	264.50	2031.150	0.140	1101.53
United Kingdom (price-cap)	264.50	2031.150	0.340	2675.15
United States	357.00	2808.915	0.088	694.67
China	541.00	4256.646	0.068	535.03
South Africa	665.00	5232.292	0.062	487.82
Poland	728.00	5727.983	0.056	440.61

have cost them 2.42 times more than a business buying electricity on a non-domestic tarriff due to economies of scale, and 3.4× more than Lancaster University. When analysed at a multinational level, the experimentation could have resulted in 17.2× less electricity-related GHG emissions if it was to be performed in Sweden (12% more expensive), or 3.5× more if it was to be performed in Poland (45% less expensive).

7.4.2.1 Operational Costs

Table 7.3: Estimated costs of access to on-demand Cloud instances equivalent to Platform A used within experimentation. Total cost provided for the total compute hours spent during experimentation across Chapters 3, 4, 5 and 6. Prices adjusted to GBP using ratio of 1 USD = 0.81 GBP.

Cloud Provider	**Instance Type**	**On-demand (£/h)**	**Total Cost (£)**	**Location**
Google Cloud	a2-highgpu-4g	12.1800	289,330	US (Iowa)
Amazon AWS	p3.8xlarge	9.9144	235,511	US (N. Virginia)
Huawei Cloud	g5.8xlarge.4	4.7628	113,138	Hong Kong
Huawei Cloud	g5.8xlarge.4	10.2400	243,400	Ireland (Dublin)
Microsoft Azure	Standard_NC24rs_v3	13.2192	314,015	US (Central)

Typically, electricity costs are only a portion of the total costs of operating high-performance compute infrastructures, with other costs involving levellised initial

investment, maintenance, internet access, location rental and cooling equipment upkeep. As such, on-demand Cloud access costs can greatly exceed the costs of electricity dissipated by computation[1234]. Table 7.3 depicts estimates for total costs of experiments associated with this thesis using different on-demand Cloud infrastructures. It can be observed that the overall costs of performing the aforementioned experiments can be vastly different depending on the selected Cloud provider as well as region in which the rented Cloud instances are located.

7.4.3 Observations

As detailed above, the energy mix composition and geographical location can have a substantial impact on the monetary and environmental costs of performing DL research. Performing identical amount of computation can result in differences in the realm of tonnes of CO2 eq GHG emissions, when performed at different locations with different energy mix compositions. At the same time, certain locations, while exhibiting substantially higher GHG emissions per every kWh of electricity produced, provide the electricity at much lower price points - see Poland in Table 7.2.

Furthermore, it is unclear whether utilising on-demand Cloud infrastructures for computation-heavy research experimentation is a cost-effective strategy. For example, purchase, installation, maintenance and energy consumption costs associated with the *Platform A* machines used during experimentation amounts to roughly £50,000, inclusive of estimated margins to account for power transmission losses. Comparing this with Table 7.3, it can be observed that for equivalent compute capability, it may be more than 60% more cost efficient to deploy and maintain bespoke compute infrastructures in research settings, especially if they can be shared across multiple research projects.

[1]Google Cloud: `https://cloud.google.com/products/calculator`
[2]Amazon AWS: `https://aws.amazon.com/ec2/instance-types/p3/`
[3]Huawei Cloud: `https://www.huaweicloud.com/intl/en-us/pricing/index.html#/ecs`
[4]Microsoft Azure: `https://learn.microsoft.com/azure/virtual-machines/ncv3-series`

7.4.4 Broader Research Context

In the course of this research, surprising and, at times, alarming costs associated with DL computation have been encountered, both in terms of monetary and environmental impact. These findings highlight the need for further investigation of the intricate balance between the value provided by DL systems and the costs of resources required to build and operate them. Additionally, there is a growing awareness within the research community about the necessity to measure and report the material, economic and environmental costs associated with DL computation. This shift in mindset is crucial, as it fosters responsible and sustainable research and development practices.

Although this work does not directly address the broader questions of assessing the value of DL against its costs, it contributes to the pursuit of more energy-proportional and energy-efficient DL systems. Additionally, by raising awareness of the environmental and operational costs associated with this research and promoting transparency in reporting, this thesis takes an active part in the collective work towards a more sustainable and responsible future for DL and its applications. While the immediate future research stemming from this work is unlikely to focus on addressing the broader cost vs. value questions, it is important to acknowledge that these issues are of paramount importance and require higher-level attention, such as government policy formulation, driving the research and development of more environmentally-responsible DL systems.

7.5 Future Work

Given the research area of DL auto-tuning is continually growing and is becoming increasingly adopted within large MLaaS platforms, there is an opportunity to extend Trimmer and DOPpler to improve quality and efficiency of DL auto-tuning at scale.

7.5.1 Alternative Candidate Filters

While the FC ANN-based candidate filtering approach used in Trimmer results in cost-effective and high-quality auto-tuning, exploring alternative filtering methods could yield interesting discoveries in terms of efficacy of different algorithms at optimising different classes of tensor operators and DL models. Future work could involve experimenting with different ANN architectures, manually or utilising existing NAS [392] approaches for rapid network discovery. The incremental search for high-performance schedules performed during auto-tuning could also be thought of as an index search problem. Recently proposed work [166], explores the idea of learned index structures - testing a hypothesis that all existing index structures (for example, B-Trees, Hashmaps, Bitmaps), could be replaced by learned models, demonstrating promising improvements to search efficiency, and as such, a potential direction to explore when improving Trimmer.

7.5.2 Polymorphic Auto-tuning

As described in Section 2.6, SOTA DL auto-tuners are frameworks composed of several components such as cost-models, search algorithms, the schedule space and the measurement infrastructure. Trimmer demonstrates how augmenting an existing auto-tuner (AutoTVM) by adding a filtering component can improve cost-efficiency of optimising end-to-end DL models, while DOPpler provides a unified, parallel alternative to the serial measurement infrastructure component within DL auto-tuners.

While auto-tuners typically exist within monolithic code bases, recent innovation in the area of auto-tuning primarily consists of replacing existing components with alternative ones, that are better suited for a subset of optimisation scenarios [8, 183, 105, 377, 386, 385]. This suggests that there exist a set of auto-tuner component combinations that are better suited for specific workload classes, or that certain optimisation scenarios may require different exploration strategies during different phases of auto-tuning.

Building an experimental framework capable of dynamic construction and adaptation of auto-tuners at a component level, guided by different objectives such as cost reduction, quality improvements or arbitrary priority designations, would be an interesting immediate area to explore. In such a system, Trimmer's Survey tuning could be leveraged to take advantage of its ability to balance cost vs. optimisation quality, while DOPpler's parallel measurement infrastructure could accelerate the process of measurements that most auto-tuners rely upon.

7.5.3 Alternative Calibration Policies

As briefly outlined in Section 6.3.9, this research work considered two other calibration policies for DOPpler: LEARN (Bayesian Optimisation), and RLEARN (Reinforcement Learning), however, it has been observed that they result in sub-optimal performance compared to the TCP BIC-based REACT policy. Further work on DOPpler could involve evaluating different TCP congestion control algorithm variants such as TCP Highspeed [77], TCP CUBIC [272], TCP Reno [28], TCP Vegas [196] amongst other implementations. This could lead to further cross-area discoveries of adapting well-established methods to more recent problems such as DL compiler auto-tuning.

7.5.4 Auto-tuning as a Service

Currently, the majority of SOTA DL auto-tuners utilise a single-machine environment, where a single CPU process executes the candidate search procedure, several processes are responsible for candidate compilation and a single process is responsible for isolated candidate measurements performed on each locally or remotely available target-device. Analyses in Chapters 3 and 5 suggest that this can lead to resource under-utilisation, both at the CPU and GPU levels and crucially, result in prolonged optimisation. This is because the measurement procedure must wait for candidate compilation, while the

compilation procedure awaits for the schedule space search and cost model to propose new candidates for measurement, which can only do so once a batch of measurements is generated and fed back to it by the measurement infrastructure.

Furthermore, different auto-tuning phases have distinct computational resource requirements. For instance, candidate compilation benefits from multi-core CPUs to perform as many simultaneous compilations as possible, since these processes are non-dependent and disjoint from each other. The measurement phase would benefit from access to machines equipped with multiple powerful GPUs or a diverse set of other target-devices that could be rapidly used for candidate latency measurements. Lastly, the search procedure is best deployed on machines with high-frequency clocked CPUs to accelerate the often single-core schedule space traversal.

To efficiently interleave these different auto-tuning phases across multiple simultaneous auto-tuning sessions, auto-tuning could be split across multiple disjoint micro-services, leveraging distributed computing paradigms such as the Service Oriented Architecture, allocating computation to resources best suited for it. Exploring cost-effective approaches for performing large-scale DL optimisation is becoming an important area of research, given large Cloud providers such as Amazon [296], Huawei [128] or Alibaba [102] are beginning to make DL optimisation available as one of their MLaaS service offerings, alongside more conventional training and inference Cloud-based DL services.

Trimmer's multi-model Survey tuning provides foundations for achieving cost-efficient, multi-model and multi-machine DL auto-tuning, while DOPpler improves utilisation of target-devices during candidate measurements. Combining these advancements with the micro-service-based approach, could be an interesting research area to explore. Transforming DL auto-tuning into a massively-distributed process could lead to new research frontiers, one of which could involve developing new Cloud paradigms such as Auto-tuning as a Service (AaaS), which could in turn spark exploration of novel resource orchestration methods dedicated towards DL auto-tuning at scale.

7.6 Recommendations for Stakeholders

Building upon the research in this thesis, this section provides a set of recommendations tailored for different stakeholders involved in DL workflows and pipelines, such as: DL Engineers, MLaaS Cloud Providers and DL Compiler Engineers.

7.6.1 DL Engineers

DL Engineers should aim to leverage DL compilers and DL compiler auto-tuners to optimise their models ahead of deployment, as this would lead to faster model inference, and over time, lead to reduced financial and environmental footprint. More specifically, to incentivise adoption of model optimisation and reduce operational costs, DL Engineers could adopt Trimmer (as described in Chapter 4), which can reduce the time spent on evaluating sub-optimal candidate tensor programs. Engineers could also integrate DOPpler (see Chapter 6) in their existing auto-tuning deployments to further speed up optimisation by parallelising candidate evaluations inter and intra-device.

Similar to other metrics commonly used in DL workflows such as: accuracy, validation loss or inference throughput, DL engineers should also actively control the optimisation cost during the auto-tuning process. This can be achieved by monitoring end-to-end model auto-tuning using cost objectives, ensuring that the auto-tuning process is always within acceptable cost limits or budgets. Trimmer's Survey tuning could prove beneficial in this task by dynamically monitoring and controlling optimisation progress.

7.6.2 MLaaS Cloud Providers

Prominent MLaaS Providers with optimisation service offerings such as Amazon SageMaker Neo [296] or OctoML Octomizer [241], should consider offering Trimmer and DOPpler as part of their existing service suite. The operational cost reductions enabled

by both Trimmer and DOPpler could also incentivise other MLaaS Providers that currently do not have a model optimisation Cloud offering. Accelerated optimisation enabled by Trimmer and DOPpler could help attract more customers who are looking for efficient and cost-effective DL solutions.

MLaaS providers should also consider implementing DOPpler's approach to parallelise candidate measurements intra-device. This approach allows multiple candidate measurements to be performed simultaneously, thereby significantly reducing the time and computational resources required for auto-tuning. This can lead to faster model deployment and lower operational costs at a datacenter scale.

7.6.3 DL Compiler Engineers

One of the factors that limit adoption of DL auto-tuning within DL pipelines are the large search spaces associated with the optimisation process. DL Compiler Engineers are often tasked with developing novel methods of exploring these candidate spaces.

Trimmer's approach to filtering out poorly performing candidates can be a good starting point. By focusing on promising candidates, DL Compiler Engineers can reduce the time it takes to evaluate their methods. DL Compiler Engineers should also consider integrating DOPpler as the default measurement infrastructure in the auto-tuning methods they develop. By parallelising measurements, engineers can significantly reduce the time required for auto-tuning, thereby speeding up evaluation of their methods.

Trimmer's ANN-based filtering module is compatible with any AutoTVM-derived auto-tuners that leverage schedule templates and a cost model. At the same time, DOPpler is a flexible replacement measurement infrastructure that can be integrated with both template-based and generative auto-tuners such as Ansor [385]. As such, both systems could be leveraged by DL Compiler Engineers with minimum amount of development effort.

This page is left intentionally blank

References

[1] Facebook's AI Research lab (FAIR). *PyTorch*. Meta Platforms, Inc. 2022. URL: https://pytorch.org/ (visited on 07/07/2023).

[2] Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, et al. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems". In: *CoRR* abs/1603.04467 (2016). arXiv: 1603.04467. URL: http://arxiv.org/abs/1603.04467.

[3] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, et al. "Learning to Optimize Halide with Tree Search and Random Programs". In: *ACM Transactions on Graphics* 38.4 (July 2019). ISSN: 0730-0301. DOI: 10.1145/3306346.3322967. URL: https://doi.org/10.1145/3306346.3322967.

[4] Advanced Micro Devices, Inc. *AMD EPYC 7763 | AMD*. Advanced Micro Devices, Inc. 2022. URL: https://www.amd.com/en/products/cpu/amd-epyc-7763 (visited on 07/07/2023).

[5] Advanced Micro Devices, Inc. *AMD ROCm Open Software Platform | AMD*. Advanced Micro Devices, Inc. 2022. URL: https://www.amd.com/en/graphics/servers-solutions-rocm (visited on 07/07/2023).

[6] Advanced Micro Devices, Inc. *rocBLAS*. Advanced Micro Devices, Inc. 2022. URL: https://github.com/ROCmSoftwarePlatform/rocBLAS (visited on 07/07/2023).

[7] Charu C Aggarwal, Lagerstrom-Fife Aggarwal, and Lagerstrom-Fife. *Linear Algebra and Optimization for Machine Learning: A Textbook*. Vol. 156. Springer International Publishing, 2020. ISBN: 9783030403447. URL: https://books.google.co.uk/books?id=EdTkDwAAQBAJ.

[8] Byung Hoon Ahn, Prannoy Pilligundla, Amir Yazdanbakhsh, and Hadi Esmaeilzadeh. "Chameleon: Adaptive Code Optimization for Expedited Deep Neural Network Compilation". In: *CoRR* abs/2001.08743 (2020). arXiv: 2001.08743. URL: https://arxiv.org/abs/2001.08743.

[9] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermüller, Dzmitry Bahdanau, et al. "Theano: A Python framework for fast computation of mathematical expressions". In: *CoRR* abs/1605.02688 (2016). arXiv: 1605.02688. URL: http://arxiv.org/abs/1605.02688.

[10] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. "Understanding of a convolutional neural network". In: *2017 International Conference on Engineering and Technology (ICET)*. 2017, pp. 1–6. DOI: 10.1109/ICEngTechnol.2017.8308186.

[11] Naomi S Altman. "An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression". In: *The American Statistician* 46.3 (1992), pp. 175–185. ISSN: 00031305. URL: http://www.jstor.org/stable/2685209 (visited on 07/15/2023).

[12] 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). Atlantic City, New Jersey: ACM, 1967, pp. 483–485. ISBN: 9781450378956. DOI: 10.1145/1465482.1465560. URL: https://doi.org/10.1145/1465482.1465560.

[13] Hossein Amiri and Asadollah Shahbahrami. "SIMD programming using Intel vector extensions". In: *Journal of Parallel and Distributed Computing* 135 (2020), pp. 83–100. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2019.09.012. URL: https://www.sciencedirect.com/science/article/pii/S074373151830813X.

[14] Dario Amodei, Danny Hernandez, Girish Sastry, Jack Clark, Greg Brockman, et al. *AI and Compute*. 2018. URL: https://openai.com/research/ai-and-compute (visited on 07/07/2023).

[15] Luke Anderson, Andrew Adams, Karima Ma, Tzu-Mao Li, Tian Jin, et al. "Efficient Automatic Scheduling of Imaging and Vision Pipelines for the GPU". In: *Proceedings of the ACM Programming Languages* 5.OOPSLA (Oct. 2021). DOI: 10.1145/3485486. URL: https://doi.org/10.1145/3485486.

[16] Stuart Andrews, Ioannis Tsochantaridis, and Thomas Hofmann. "Support Vector Machines for Multiple-Instance Learning". In: *Proceedings of the 15th International Conference on Neural Information Processing Systems*. NIPS'02. Cambridge, MA, USA: MIT Press, 2002, pp. 577–584.

[17] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. "SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39.12 (2017), pp. 2481–2495. DOI: 10.1109/TPAMI.2016.2644615.

[18] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, et al. "Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code". In: *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2019, pp. 193–205.

[19] Bowen Baker*, Otkrist Gupta*, Ramesh Raskar, and Nikhil Naik. "Accelerating Neural Architecture Search using Performance Prediction". In: *6th International*

Conference on Learning Representations (ICLR). 2018. URL: https://openreview.net/forum?id=BJypUGZ0Z.

[20] Debrath Banerjee. "A microarchitectural study on Apple's A11 bionic processor". In: *Arkansas State University: Jonesboro, AR, USA* (2018).

[21] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, et al. "Cython: The Best of Both Worlds". In: *Computing in Science & Engineering* 13.2 (2011), pp. 31–39. DOI: 10.1109/MCSE.2010.118.

[22] Tal Ben-Nun and Torsten Hoefler. "Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis". In: *ACM Computing Surveys (CSUR)* 52.4 (Aug. 2019). ISSN: 0360-0300. DOI: 10.1145/3320060. URL: https://doi.org/10.1145/3320060.

[23] Ernst Julius Berg. *Heaviside's Operational Calculus as Applied to Engineering and Physics*. McGraw-Hill, 1936.

[24] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. "Algorithms for Hyper-Parameter Optimization". In: *Proceedings of the 24th International Conference on Neural Information Processing Systems*. NIPS'11. Granada, Spain: Curran Associates Inc., 2011, pp. 2546–2554. ISBN: 9781618395993.

[25] Daniel Berlin, David Edelsohn, and Sebastian Pop. "High-level loop optimizations for GCC". In: *Proceedings of the 2004 GCC Developers Summit*. Citeseer. 2004, pp. 37–54.

[26] Gérard Biau and Erwan Scornet. "A random forest guided tour". In: *TEST* 25.2 (June 2016), pp. 197–227. ISSN: 1863-8260. DOI: 10.1007/s11749-016-0481-7. URL: https://doi.org/10.1007/s11749-016-0481-7.

[27] Ekaba Bisong. "Batch vs. online learning". In: *Building Machine Learning and Deep Learning Models on Google Cloud Platform*. Springer, 2019, pp. 199–201.

[28] Ethan Blanton, Dr. Vern Paxson, and Mark Allman. *TCP Congestion Control*. RFC 5681. Sept. 2009. DOI: 10.17487/RFC5681. URL: https://www.rfc-editor.org/info/rfc5681.

[29] Léon Bottou. "Large-Scale Machine Learning with Stochastic Gradient Descent". In: *Proceedings of COMPSTAT'2010*. Ed. by Yves Lechevallier and Gilbert Saporta. Heidelberg: Physica-Verlag HD, 2010, pp. 177–186. ISBN: 978-3-7908-2604-3.

[30] Eric Brochu, Vlad M. Cora, and Nando de Freitas. "A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning". In: *CoRR* abs/1012.2599 (2010). arXiv: 1012.2599. URL: http://arxiv.org/abs/1012.2599.

[31] Andy Brock, Soham De, Samuel L Smith, and Karen Simonyan. "High-performance large-scale image recognition without normalization". In: *International Conference on Machine Learning*. PMLR. 2021, pp. 1059–1071.

[32] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, et al. "A Survey of Monte Carlo Tree Search Methods". In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.1 (2012), pp. 1–43. DOI: 10.1109/TCIAIG.2012.2186810.

[33] Jacques Bughin, Eric Hazan, Sree Ramaswamy, Michael Chui, Tera Allas, et al. "Artificial intelligence: the next digital frontier?" In: *McKinsey Global Institute* 47.3.6 (2017).

[34] David Callahan and Ken Kennedy. "Compiling programs for distributed-memory multiprocessors". In: *The Journal of Supercomputing* 2.2 (Oct. 1988), pp. 151–169. ISSN: 1573-0484. DOI: 10.1007/BF00128175. URL: https://doi.org/10.1007/BF00128175.

[35] M Emre Celebi and Kemal Aydin. *Unsupervised learning algorithms*. Springer, 2016.

[36] Karanbir Singh Chahal, Manraj Singh Grover, Kuntal Dey, and Rajiv Ratn Shah. "A Hitchhiker's Guide On Distributed Training Of Deep Neural Networks". In: *Journal of Parallel and Distributed Computing* 137 (2020), pp. 65–76. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2019.10.004. URL: https://www.sciencedirect.com/science/article/pii/S0743731518308712.

[37] Tianfeng Chai and Roland R Draxler. "Root mean square error (RMSE) or mean absolute error (MAE)? – Arguments against avoiding RMSE in the literature". In: *Geoscientific Model Development Discussions* 7.3 (2014), pp. 1247–1250. DOI: 10.5194/gmd-7-1247-2014. URL: https://gmd.copernicus.org/articles/7/1247/2014/.

[38] 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. San Francisco, California, USA: ACM, 2016, pp. 785–794. ISBN: 9781450342322. DOI: 10.1145/2939672.2939785. URL: https://doi.org/10.1145/2939672.2939785.

[39] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, 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. Carlsbad, CA, USA: USENIX Association, 2018, pp. 579–594. ISBN: 9781931971478.

[40] Tianqi Chen, Lianmin Zheng, Eddie Q. Yan, Ziheng Jiang, Thierry Moreau, et al. "Learning to Optimize Tensor Programs". In: *CoRR* abs/1805.08166 (2018). arXiv: 1805.08166. URL: http://arxiv.org/abs/1805.08166.

[41] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, et al. "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1724–1734. DOI: 10.3115/v1/D14-1179. URL: https://aclanthology.org/D14-1179.

[42] Ronald Christensen. *Log-linear models and logistic regression*. Springer Science & Business Media, 2006.

[43] William S. Cleveland and Susan J. Devlin. "Locally Weighted Regression: An Approach to Regression Analysis by Local Fitting". In: *Journal of the American Statistical Association* 83.403 (1988), pp. 596–610. DOI: 10.1080/01621459.1988.10478639. eprint: https://www.tandfonline.com/doi/pdf/10.1080/01621459.1988.10478639. URL: https://www.tandfonline.com/doi/abs/10.1080/01621459.1988.10478639.

[44] Ampere Computing. *Ampere AI*. 2022. URL: https://solutions.amperecomputing.com/solutions/ampere-ai (visited on 07/07/2023).

[45] Keith D. Cooper, L. Taylor Simpson, and Christopher A. Vick. "Operator Strength Reduction". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23.5 (Sept. 2001), pp. 603–625. ISSN: 0164-0925. DOI: 10.1145/504709.504710. URL: https://doi.org/10.1145/504709.504710.

[46] Marius Cornea and Intel Corporation. *Intel AVX-512 instructions and their use in the implementation of math functions*. 2015. URL: http://www.fit.vutbr.cz/~iklubal/IPA/AVX-512_Cornea.pdf.

[47] Intel Corporation. *Accelerate Fast Math with Intel oneAPI Math Kernel Library*. Intel Corporation. 2022. URL: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html (visited on 07/07/2023).

[48] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 2: Instruction Set Reference*. Intel Corporation. 2022. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.html (visited on 07/07/2023).

[49] Intel Corporation. *Intel Distribution of OpenVINO Toolkit*. Intel Corporation. 2022. URL: https://docs.openvino.ai/latest/index.html (visited on 07/07/2023).

[50] Intel Corporation. *Intel Movidius Myriad X Vision Processing Unit 0GB*. Intel Corporation. 2022. URL: https://www.intel.com/content/www/us/en/products/sku/204770/intel-movidius-myriad-x-vision-processing-unit-0gb/specifications.html (visited on 07/07/2023).

[51] Intel Corporation. *Intel oneAPI Deep Neural Network Library (oneDNN)*. Intel Corporation. 2022. URL: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onednn.html (visited on 07/07/2023).

[52] Intel Corporation. *Intel Xeon Platinum 8380 Processor*. Intel Corporation. 2022. URL: https://www.intel.co.uk/content/www/uk/en/products/sku/212287/intel-xeon-platinum-8380-processor-60m-cache-2-30-ghz/specifications.html (visited on 07/07/2023).

[53] Intel Corporation. *Intrinsics for Intel(R) Advanced Matrix Extensions (Intel(R) AMX) Instructions*. Intel Corporation. 2022. URL: https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compilerreference/intrinsics/intrinsics-for-intel-advanced-matrix-extensions-intelamx-instructions.html (visited on 07/07/2023).

[54] Microsoft Corporation. *Azure Machine Learning - ML as a Service | Microsoft Azure*. Microsoft Corporation. 2022. URL: https://azure.microsoft.com/en-us/products/machine-learning/#product-overview (visited on 07/07/2023).

[55] Microsoft Corporation. *Azure VM sizes - GPU - Azure Virtual Machines*. Microsoft Corporation. 2022. URL: https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-gpu (visited on 07/07/2023).

[56] Microsoft Corporation. *Deploy ML models to FPGAs - Azure Machine Learning | Microsoft Learn*. Microsoft Corporation. 2022. URL: https://learn.microsoft.com/en-us/azure/machine-learning/v1/how-to-deploy-fpga-web-service (visited on 07/07/2023).

[57] Microsoft Corporation. *ONNX Runtime | Home*. Microsoft Corporation. 2022. URL: https://onnxruntime.ai/ (visited on 07/07/2023).

[58] Scott Cyphers, Arjun K. Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, et al. "Intel nGraph: An Intermediate Representation, Compiler, and Executor for Deep Learning". In: *CoRR* abs/1801.08058 (2018). arXiv: 1801.08058. URL: http://arxiv.org/abs/1801.08058.

[59] Jacek Czaja, Michal Gallus, Joanna Wozna, Adam Grygielski, and Luo Tao. "Applying the Roofline model for Deep Learning performance optimizations". In: *CoRR* abs/2009.11224 (2020). arXiv: 2009.11224. URL: https://arxiv.org/abs/2009.11224.

[60] Pawel Czarnul, Jerzy Proficz, and Adam Krzywaniak. "Energy-Aware High-Performance Computing: Survey of State-of-the-Art Tools, Techniques, and Environments". In: *Scientific Programming* 2019 (Apr. 2019). ISSN: 1058-9244. DOI: 10.1155/2019/8348791. URL: https://doi.org/10.1155/2019/8348791.

[61] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. "RAPL: Memory power estimation and capping". In: *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*. IEEE. 2010, pp. 189–194. DOI: 10.1145/1840845.1840883.

[62] Jack W. Davidson and Sanjay Jinturkar. *An Aggressive Approach to Loop Unrolling*. Tech. rep. USA: University of Virginia, 2001.

[63] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, et al. "Large Scale Distributed Deep Networks". In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper_files/paper/2012/file/6aca97005c68f1206823815f66102863-Paper.pdf.

[64] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, et al. "ImageNet: A large-scale hierarchical image database". In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.

[65] Radosvet Desislavov, Fernando Martínez-Plumed, and José Hernández-Orallo. "Compute and Energy Consumption Trends in Deep Learning Inference". In: *CoRR* abs/2109.05472 (2021). arXiv: 2109.05472. URL: https://arxiv.org/abs/2109.05472.

[66] Bin Ding, Huimin Qian, and Jun Zhou. "Activation functions and their characteristics in deep neural networks". In: *2018 Chinese Control And Decision Conference (CCDC)*. 2018, pp. 1836–1841. DOI: 10.1109/CCDC.2018.8407425.

[67] Jesse Dodge, Suchin Gururangan, Dallas Card, Roy Schwartz, and Noah A. Smith. "Show Your Work: Improved Reporting of Experimental Results". In: *CoRR* abs/1909.03004 (2019). arXiv: 1909.03004. URL: http://arxiv.org/abs/1909.03004.

[68] Stephan Dreiseitl and Lucila Ohno-Machado. "Logistic Regression and Artificial Neural Network Classification Models: A Methodology Review". In: *Journal of Biomedical Informatics* 35.5/6 (Oct. 2002), pp. 352–359. ISSN: 1532-0464. DOI: 10.1016/S1532-0464(03)00034-0. URL: https://doi.org/10.1016/S1532-0464(03)00034-0.

[69] Nan Du, Yanping Huang, Andrew M Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, et al. "Glam: Efficient scaling of language models with mixture-of-experts". In: *International Conference on Machine Learning*. PMLR. 2022, pp. 5547–5569.

[70] Mitsubishi Electric. *Light Commercial PKA100 High Wall Heat Pump*. 2022. URL: https://www.mitsubishi-electric.co.nz/heatpump/i/69228B/light-commercial-pka100-high-wall-heat-pump (visited on 07/07/2023).

[71] electricity info. *Fuel Mix Archive - electricity info*. 2022. URL: https://electricityinfo.org/fuel-mix-archive/#data (visited on 07/07/2023).

[72] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. "Neural Architecture Search: A Survey". In: *The Journal of Machine Learning Research* 20.1 (Jan. 2019), pp. 1997–2017. ISSN: 1532-4435.

[73] Lee D. Erman, Frederick Hayes-Roth, Victor R. Lesser, and D. Raj Reddy. "The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty". In: *ACM Computing Surveys (CSUR)* 12.2 (June 1980), pp. 213–253. ISSN: 0360-0300. DOI: 10.1145/356810.356816. URL: https://doi.org/10.1145/356810.356816.

[74] Heiko Falk and Peter Marwedel. "Loop Nest Splitting". In: *Source Code Optimization Techniques for Data Flow Dominated Embedded Software*. Boston, MA: Springer US, 2004, pp. 53–117. ISBN: 978-1-4020-2829-8. DOI: 10.1007/978-1-4020-2829-8_5. URL: https://doi.org/10.1007/978-1-4020-2829-8_5.

[75] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. "Power Provisioning for a Warehouse-Sized Computer". In: *ACM SIGARCH Computer Architecture News* 35.2 (June 2007), pp. 13–23. ISSN: 0163-5964. DOI: 10.1145/1273440.1250665. URL: https://doi.org/10.1145/1273440.1250665.

[76] Paul Feautrier. "Automatic parallelization in the polytope model". In: *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*. Ed. by Guy-René Perrin and Alain Darte. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 79–103. ISBN: 978-3-540-70646-5. DOI: 10.1007/3-540-61736-1_44. URL: https://doi.org/10.1007/3-540-61736-1_44.

[77] Sally Floyd. *HighSpeed TCP for Large Congestion Windows*. RFC 3649. Dec. 2003. DOI: 10.17487/RFC3649. URL: https://www.rfc-editor.org/info/rfc3649.

[78] Michael J. Flynn. "Some Computer Organizations and Their Effectiveness". In: *IEEE Transactions on Computers* C-21.9 (1972), pp. 948–960. DOI: 10.1109/TC.1972.5009071.

[79] Apache Software Foundation. *Apache MXNet | A flexible and efficient library for deep learning*. 2022. URL: https://mxnet.apache.org (visited on 07/07/2023).

[80] Apache Software Foundation. *github - apache/tvm: open deep learning compiler stack for cpu, gpu and specialized accelerators*. 2022. URL: https://github.com/apache/tvm/ (visited on 07/07/2023).

[81] Python Software Foundation. *Python multiprocessing — Process-based parallelism*. Python Software Foundation. 2022. URL: https://docs.python.org/3/library/multiprocessing.html (visited on 07/07/2023).

[82] The Linux Foundation. *ONNX: Open Neural Network Exchange*. 2022. URL: https://onnx.ai/ (visited on 07/07/2023).

[83] The Linux Foundation. *PyTorch Hub | Pytorch*. 2022. URL: https://pytorch.org/hub (visited on 07/07/2023).

[84] The Linux Foundation. *TorchScript - PyTorch 1.12 documentation*. 2022. URL: https://pytorch.org/docs/stable/jit.html (visited on 07/07/2023).

[85] The Linux Foundation. *TorchServe - PyTorch/Serve master documentation*. 2022. URL: https://pytorch.org/serve/ (visited on 07/07/2023).

[86] 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.

[87] Jerome H. Friedman. "Greedy Function Approximation: A Gradient Boosting Machine". In: *The Annals of Statistics* 29.5 (2001), pp. 1189–1232. ISSN: 00905364. URL: http://www.jstor.org/stable/2699986 (visited on 07/15/2023).

[88] Rolando Garcia, Vikram Sreekanti, Neeraja Yadwadkar, Daniel Crankshaw, Joseph E Gonzalez, and Joseph M Hellerstein. "Context: The Missing Piece in the Machine Learning Lifecycle". In: *KDD CMI Workshop*. Vol. 114. 2018.

[89] Vikas Garg and Adam T Kalai. "Supervising Unsupervised Learning". In: *Advances in Neural Information Processing Systems*. Ed. by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett. Vol. 31. Curran Associates, Inc., 2018. URL: https://proceedings.neurips.cc/paper_files/paper/2018/file/72e6d3238361fe70f22fb0ac624a7072-Paper.pdf.

[90] Ioannis Gatopoulos, Romain Lepert, Auke Wiggers, Giovanni Mariani, and Jakub Tomczak. "Evolutionary Algorithm with Non-parametric Surrogate Model for Tensor Program optimization". In: *2020 IEEE Congress on Evolutionary Computation (CEC)*. 2020, pp. 1–8. DOI: 10.1109/CEC48606.2020.9185646.

[91] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. "Vision meets Robotics: The KITTI Dataset". In: *The International Journal of Robotics Research* 32.11 (2013), pp. 1231–1237. DOI: 10.1177/0278364913491297. eprint: https://doi.org/10.1177/0278364913491297. URL: https://doi.org/10.1177/0278364913491297.

[92] Isaac Gelado and Michael Garland. "Throughput-Oriented GPU Memory Allocation". In: *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. PPoPP '19. Washington, District of Columbia: ACM, 2019, pp. 27–37. ISBN: 9781450362252. DOI: 10.1145/3293883.3295727. URL: https://doi.org/10.1145/3293883.3295727.

[93] 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. Chicago, Illinois: ACM, 2023, pp. 28–39. ISBN: 9781450398688. DOI: 10.1145/3559009.3569682. URL: https://doi.org/10.1145/3559009.3569682.

[94] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. "Region-Based Convolutional Networks for Accurate Object Detection and Segmentation". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 38.1 (2016), pp. 142–158. DOI: 10.1109/TPAMI.2015.2437384.

[95] George Gkotsis, Anika Oellrich, Sumithra Velupillai, Maria Liakata, Tim J. P. Hubbard, et al. "Characterisation of mental health conditions in social media using Informed Deep Learning". In: *Scientific Reports* 7.1 (Mar. 2017), p. 45141. ISSN: 2045-2322. DOI: 10.1038/srep45141. URL: https://doi.org/10.1038/srep45141.

[96] Global Change Data Lab. *Carbon intensity of electricity - Our World in Data*. 2022. URL: https://ourworldindata.org/grapher/carbon-

intensity-electricity?tab=chart&region=Europe&country=AUT~FRA~EU-27~SWE~POL~CHN~ZAF~USA~GBR (visited on 07/07/2023).

[97] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[98] Google. *AI & Machine Learning Products | Google Cloud*. Google. 2022. URL: https://cloud.google.com/products/ai (visited on 07/07/2023).

[99] Google. *Cloud GPUs | Google Cloud*. Google. 2022. URL: https://cloud.google.com/gpu (visited on 07/07/2023).

[100] Google. *Train and run machine learning models faster | Cloud TPU | Google Cloud*. Google. 2022. URL: https://cloud.google.com/tpu (visited on 07/07/2023).

[101] Andreas Griewank. "On automatic differentiation". In: *Mathematical Programming: Eecent Developments and Applications* 6.6 (1989), pp. 83–107.

[102] Alibaba Group. *AI Compiler at Alibaba*. Alibaba Group. 2019. URL: https://sampl.cs.washington.edu/tvmconf/slides/2019/Xiaoyong-Liu-Alibaba.pdf (visited on 07/07/2023).

[103] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. "Improving the Cache Locality of Memory Allocation". In: *SIGPLAN Notices* 28.6 (June 1993), pp. 177–186. ISSN: 0362-1340. DOI: 10.1145/173262.155107. URL: https://doi.org/10.1145/173262.155107.

[104] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, et al. "The Architectural Implications of Facebook's DNN-Based Personalized Recommendation". In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2020, pp. 488–501. DOI: 10.1109/HPCA47549.2020.00047.

[105] Ameer Haj-Ali, Hasan Genc, Qijing Huang, William S. Moses, John Wawrzynek, et al. "ProTuner: Tuning Programs with Monte Carlo Tree Search". In: *CoRR* abs/2005.13685 (2020). arXiv: 2005.13685. URL: https://arxiv.org/abs/2005.13685.

[106] Jawad Haj-Yahya, Avi Mendelson, Yosi Ben Asher, and Anupam Chattopadhyay. *Energy Efficient High Performance Processors: Recent Approaches for Designing Green High Performance Computing*. Springer, 2018.

[107] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, et al. "EIE: Efficient Inference Engine on Compressed Deep Neural Network". In: *ACM SIGARCH Computer Architecture News* 44.3 (June 2016), pp. 243–254. ISSN: 0163-5964. DOI: 10.1145/3007787.3001163. URL: https://doi.org/10.1145/3007787.3001163.

[108] Song Han, Huizi Mao, and William J Dally. "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding". In: *arXiv preprint arXiv:1510.00149* abs/1510.00149 (2015). arXiv: 1510.00149. URL: https://arxiv.org/abs/1510.00149.

[109] Leon D. Harmon. "Studies with artificial neurons, I: properties and functions of an artificial neuron". In: *Kybernetik* 1.3 (Dec. 1961), pp. 89–101. ISSN: 1432-0770. DOI: 10.1007/BF00290179. URL: https://doi.org/10.1007/BF00290179.

[110] J. A. Hartigan and M. A. Wong. "Algorithm AS 136: A K-Means Clustering Algorithm". In: *Applied Statistics* 28.1 (1979), pp. 100–108. ISSN: 00359254. DOI: 10.2307/2346830. URL: http://dx.doi.org/10.2307/2346830.

[111] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, et al. "Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective". In: *2018 IEEE International Symposium on High Performance*

Computer Architecture (HPCA). 2018, pp. 620–629. DOI: 10.1109/HPCA.2018.00059.

[112] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 37.9 (2015), pp. 1904–1916. DOI: 10.1109/TPAMI.2015.2389824.

[113] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[114] Robert Hecht-Nielsen. "Theory of the backpropagation neural network". In: *International 1989 Joint Conference on Neural Networks*. 1989, 593–605 vol.1. DOI: 10.1109/IJCNN.1989.118638.

[115] Peter Henderson, Jieru Hu, Joshua Romoff, Emma Brunskill, Dan Jurafsky, et al. "Towards the Systematic Reporting of the Energy and Carbon Footprints of Machine Learning". In: *Journal of Machine Learning Research* 21.1 (Jan. 2020). ISSN: 1532-4435.

[116] Sepp Hochreiter. "The Vanishing Gradient Problem during Learning Recurrent Neural Nets and Problem Solutions". In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6.2 (Apr. 1998), pp. 107–116. ISSN: 0218-4885. DOI: 10.1142/S0218488598000094. URL: https://doi.org/10.1142/S0218488598000094.

[117] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. eprint: https://direct.mit.edu/neco/

article-pdf/9/8/1735/813796/neco.1997.9.8.1735.pdf. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[118] Mark Horowitz. "1.1 Computing's energy problem (and what we can do about it)". In: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 2014, pp. 10–14. DOI: 10.1109/ISSCC.2014.6757323.

[119] Houses of Parliament - Parliamentary Office of Science & Technology. *Carbon Footprint of Electricity Generation - Post Note Update*. 2022. URL: https://www.parliament.uk/globalassets/documents/post/postpn_383-carbon-footprint-electricity-generation.pdf (visited on 07/07/2023).

[120] A. Howard, M. Sandler, B. Chen, W. Wang, L. Chen, et al. "Searching for MobileNetV3". In: *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. Los Alamitos, CA, USA: IEEE Computer Society, Nov. 2019, pp. 1314–1324. DOI: 10.1109/ICCV.2019.00140. URL: https://doi.ieeecomputersociety.org/10.1109/ICCV.2019.00140.

[121] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, et al. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". In: *CoRR* abs/1704.04861 (2017). arXiv: 1704.04861. URL: http://arxiv.org/abs/1704.04861.

[122] Dan Howdle. *Worldwide Electricity Pricing | Energy Cost Per KWh in 230 Countries*. Existent Ltd. 2022. URL: https://www.cable.co.uk/energy/worldwide-pricing/ (visited on 07/07/2023).

[123] Yitao Hu, Swati Rallapalli, Bongjun Ko, and Ramesh Govindan. "Olympian: Scheduling GPU Usage in a Deep Neural Network Model Serving System". In: *Proceedings of the 19th International Middleware Conference*. Middleware '18. Rennes, France: ACM, 2018, pp. 53–65. ISBN: 9781450357029. DOI: 10.1145/3274808.3274813. URL: https://doi.org/10.1145/3274808.3274813.

[124] Yuming Hua, Junhai Guo, and Hua Zhao. "Deep Belief Networks and deep learning". In: *Proceedings of 2015 International Conference on Intelligent Computing and Internet of Things*. 2015, pp. 1–4. DOI: 10.1109/ICAIOT.2015.7111524.

[125] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger. "Densely Connected Convolutional Networks". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, July 2017, pp. 2261–2269. DOI: 10.1109/CVPR.2017.243. URL: https://doi.ieeecomputersociety.org/10.1109/CVPR.2017.243.

[126] Xun Huang, Yixuan Li, Omid Poursaeed, John Hopcroft, and Serge Belongie. "Stacked Generative Adversarial Networks". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 1866–1875. DOI: 10.1109/CVPR.2017.202.

[127] Huawei. *MindSpore AI Computing Framework Research | Huawei Global*. 2022. URL: https://e.huawei.com/uk/products/cloud-computing-dc/atlas/mindspore (visited on 07/07/2023).

[128] Huawei. *TBE Operator Development Modes | Ascend CANN (20.0, training) | TBE Custom Operator Development Guide | TBE Introduction | Huawei Cloud*. 2022. URL: https://support.huaweicloud.com/intl/en-us/odevg-A800_9000_9010/atlaste_10_0015.html (visited on 07/07/2023).

[129] Brody Huval, Tao Wang, Sameep Tandon, Jeff Kiske, Will Song, et al. "An Empirical Evaluation of Deep Learning on Highway Driving". In: *CoRR* abs/1504.01716 (2015). arXiv: 1504.01716. URL: http://arxiv.org/abs/1504.01716.

[130] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, et al. "SqueezeNet: AlexNet-level accuracy with 50x fewer

parameters and <0.5MB model size". In: *5th International Conference on Learning Representations (ICLR)*. 2017. URL: https://openreview.net/forum?id=S1xh5sYgx.

[131] IBM. *GPUs for Cloud Servers | IBM*. 2022. URL: https://www.ibm.com/cloud/gpu (visited on 07/07/2023).

[132] Boris Iglewicz and David Caster Hoaglin. *How to detect and handle outliers*. Vol. 16. Asq Press, 1993.

[133] Apple Inc. *Core ML | Apple Developer Documentation*. 2022. URL: https://developer.apple.com/documentation/coreml (visited on 07/07/2023).

[134] Docker Inc. *Docker: Accelerated, Containerized Application Development*. 2022. URL: https://www.docker.com/ (visited on 07/07/2023).

[135] Docker Inc. *Swarm mode overview | Docker Documentation*. 2022. URL: https://docs.docker.com/engine/swarm/ (visited on 07/07/2023).

[136] Neuralmagic Inc. *Software-Delivered AI - Neural Magic*. 2022. URL: https://neuralmagic.com/ (visited on 07/07/2023).

[137] The Khronos Group Inc. *clWaitForEvents*. 2022. URL: https://man.opencl.org/clWaitForEvents.html (visited on 07/07/2023).

[138] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*. ICML'15. Lille, France: JMLR.org, 2015, pp. 448–456.

[139] Alekseí Grigorevich Ivakhnenko and Valentin Grigorévich Lapa. *Cybernetic Predicting Devices*. CCM Information Corporation, 1965.

[140] H Jabbar and Rafiqul Zaman Khan. "Methods to avoid over-fitting and under-fitting in supervised machine learning (comparative study)". In: *Computer Science, Communication and Instrumentation Devices* 70 (2015).

[141] Benoit Jacob, Gael Guennebaud, et al. *Eigen*. 2022. URL: https://eigen.tuxfamily.org/ (visited on 07/07/2023).

[142] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, et al. "Population Based Training of Neural Networks". In: *CoRR* abs/1711.09846 (2017). arXiv: 1711.09846. URL: http://arxiv.org/abs/1711.09846.

[143] Suresh Jagannathan and Andrew Wright. "Flow-Directed Inlining". In: *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*. PLDI '96. Philadelphia, Pennsylvania, USA: ACM, 1996, pp. 193–205. ISBN: 0897917952. DOI: 10.1145/231379.231417. URL: https://doi.org/10.1145/231379.231417.

[144] Paras Jain, Xiangxi Mo, Ajay Jain, Harikaran Subbaraj, Rehan Sohail Durrani, et al. "Dynamic Space-Time Scheduling for GPU Inference". In: *CoRR* abs/1901.00041 (2019). arXiv: 1901.00041. URL: http://arxiv.org/abs/1901.00041.

[145] Anoop Jeerige, Doina Bein, and Abhishek Verma. "Comparison of Deep Reinforcement Learning Approaches for Intelligent Game Playing". In: *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*. 2019, pp. 0366–0371. DOI: 10.1109/CCWC.2019.8666545.

[146] EunJin Jeong, Jangryul Kim, Samnieng Tan, Jaeseong Lee, and Soonhoi Ha. "Deep Learning Inference Parallelization on Heterogeneous Processors With TensorRT". In: *IEEE Embedded Systems Letters* 14.1 (2022), pp. 15–18. DOI: 10.1109/LES.2021.3087707.

[147] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, et al. "Caffe: Convolutional Architecture for Fast Feature Embedding". In: *Proceedings of the 22nd ACM International Conference on Multimedia*. MM '14. Orlando, Florida, USA: ACM, 2014, pp. 675–678. ISBN: 9781450330633. DOI: 10.1145/2647868.2654889. URL: https://doi.org/10.1145/2647868.2654889.

[148] Xiaotang Jiang, Huan Wang, Yiliu Chen, Ziqi Wu, Lichuan Wang, et al. "MNN: A Universal and Efficient Inference Engine". In: *Proceedings of Machine Learning and Systems*. Ed. by I. Dhillon, D. Papailiopoulos, and V. Sze. Vol. 2. 2020, pp. 1–13. URL: https://proceedings.mlsys.org/paper_files/paper/2020/file/bc19061f88f16e9ed4a18f0bbd47048a-Paper.pdf.

[149] Stephen C Johnson. "Hierarchical clustering schemes". In: *Psychometrika* 32.3 (1967), pp. 241–254.

[150] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, 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: ACM, 2017, pp. 1–12. ISBN: 9781450348928. DOI: 10.1145/3079856.3080246. URL: https://doi.org/10.1145/3079856.3080246.

[151] Wookeun Jung, Thanh Tuan Dao, and Jaejin Lee. "DeepCuts: A Deep Learning Optimization Framework for Versatile GPU Workloads". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: ACM, 2021, pp. 190–205. ISBN: 9781450383912. DOI: 10.1145/3453483.3454038. URL: https://doi.org/10.1145/3453483.3454038.

[152] Daniel Justus, John Brennan, Stephen Bonner, and Andrew Stephen McGough. "Predicting the Computational Cost of Deep Learning Models". In: *2018 IEEE International Conference on Big Data (Big Data)*. IEEE. 2018, pp. 3873–3882. DOI: 10.1109/BigData.2018.8622396.

[153] James Kennedy and Russell Eberhart. "Particle swarm optimization". In: *Proceedings of ICNN'95-international conference on neural networks*. Vol. 4. IEEE. 1995, pp. 1942–1948.

[154] Ken Kennedy and John R Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2001.

[155] Shauharda Khadka, Estelle Aflalo, Mattias Marder, Avrech Ben-David, Santiago Miret, et al. "Optimizing Memory Placement using Evolutionary Graph Reinforcement Learning". In: *International Conference on Learning Representations*. 2021. URL: https://openreview.net/forum?id=-6vS_4Kfz0.

[156] Jehandad Khan, Paul Fultz, Artem Tamazov, Daniel Lowell, Chao Liu, et al. "MIOpen: An Open Source Library For Deep Learning Primitives". In: *CoRR* abs/1910.00078 (2019). arXiv: 1910.00078. URL: http://arxiv.org/abs/1910.00078.

[157] Diederik Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *International Conference on Learning Representations (ICLR)*. San Diega, CA, USA, 2015.

[158] Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. "Optimization by Simulated Annealing". In: *Science* 220.4598 (1983), pp. 671–680. DOI: 10.1126/science.220.4598.671. eprint: https://www.science.org/doi/pdf/10.1126/science.220.4598.671. URL: https://www.science.org/doi/abs/10.1126/science.220.4598.671.

[159] Fredrik Kjolstad, Stephen Chou, David Lugato, Shoaib Kamil, and Saman Amarasinghe. "Taco: A Tool to Generate Tensor Algebra Kernels". In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2017, pp. 943–948. DOI: 10.1109/ASE.2017.8115709.

[160] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. "Partial Dead Code Elimination". In: *SIGPLAN Notices* 29.6 (June 1994), pp. 147–158. ISSN: 0362-1340. DOI: 10.1145/773473.178256. URL: https://doi.org/10.1145/773473.178256.

[161] Jens Kober, J. Andrew Bagnell, and Jan Peters. "Reinforcement learning in robotics: A survey". In: *The International Journal of Robotics Research* 32.11 (2013), pp. 1238–1274. DOI: 10.1177/0278364913495721. eprint: https://doi.org/10.1177/0278364913495721. URL: https://doi.org/10.1177/0278364913495721.

[162] Will Koehrsen. *Overfitting vs. underfitting: A complete example*. 2018. URL: https://towardsdatascience.com/overfitting-vs-underfitting-a-complete-example-d05dd7e19765 (visited on 07/07/2023).

[163] Jing Yu Koh. *Model Zoo - Deep learning code and pre-trained models for transfer learning, educational purposes, and more*. 2022. URL: https://modelzoo.co/ (visited on 07/07/2023).

[164] John F. Kolen and Stefan C. Kremer. "Gradient Flow in Recurrent Nets: The Difficulty of Learning LongTerm Dependencies". In: *A Field Guide to Dynamical Recurrent Networks*. IEEE, 2001, pp. 237–243. DOI: 10.1109/9780470544037.ch14.

[165] Mark A. Kramer. "Nonlinear principal component analysis using autoassociative neural networks". In: *AIChE Journal* 37.2 (1991), pp. 233–243. DOI: https://doi.org/10.1002/aic.690370209. eprint: https://aiche.onlinelibrary.wiley.

com/doi/pdf/10.1002/aic.690370209. URL: https://aiche.onlinelibrary.wiley.com/doi/abs/10.1002/aic.690370209.

[166] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. "The Case for Learned Index Structures". In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: ACM, 2018, pp. 489–504. ISBN: 9781450347037. DOI: 10.1145/3183713.3196909. URL: https://doi.org/10.1145/3183713.3196909.

[167] Alex Krizhevsky. *Learning Multiple Layers of Features from Tiny Images*. Tech. rep. 2009, pp. 32–33. URL: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

[168] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Communications of the ACM* 60.6 (May 2017), pp. 84–90. ISSN: 0001-0782. DOI: 10.1145/3065386. URL: https://doi.org/10.1145/3065386.

[169] Thorsten Kurth, Jian Zhang, Nadathur Satish, Evan Racah, Ioannis Mitliagkas, et al. "Deep Learning at 15PF: Supervised and Semi-Supervised Classification for Scientific Data". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '17. Denver, Colorado: ACM, 2017. ISBN: 9781450351140. DOI: 10.1145/3126908.3126916. URL: https://doi.org/10.1145/3126908.3126916.

[170] Rasmus Munk Larsen and Tatiana Shpeisman. *TensorFlow Graph Optimizations*. 2019. URL: https://research.google/pubs/pub48051/ (visited on 07/07/2023).

[171] Samuel Larsen and Saman Amarasinghe. "Exploiting Superword Level Parallelism with Multimedia Instruction Sets". In: *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. PLDI '00.

Vancouver, British Columbia, Canada: ACM, 2000, pp. 145–156. ISBN: 1581131992. DOI: 10.1145/349299.349320. URL: https://doi.org/10.1145/349299.349320.

[172] Chris Lattner. "LLVM and Clang: Next generation compiler technology". In: *The BSD conference*. Vol. 5. 2008.

[173] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, et al. "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation". In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2021, pp. 2–14. DOI: 10.1109/CGO51591.2021.9370308.

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

[175] Quoc V. Le, Navdeep Jaitly, and Geoffrey E. Hinton. "A Simple Way to Initialize Recurrent Networks of Rectified Linear Units". In: *CoRR* abs/1504.00941 (2015). arXiv: 1504.00941. URL: http://arxiv.org/abs/1504.00941.

[176] Chris Leary and Todd Wang. *XLA: TensorFlow, compiled*. TensorFlow Dev Summit. 2017. URL: https://www.youtube.com/watch?v=kAOanJczHA0 (visited on 07/07/2023).

[177] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791.

[178] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. *The MNIST database of handwritten digits*. 1998. URL: http://yann.lecun.com/exdb/mnist/ (visited on 07/07/2023).

[179] Honglak Lee, Roger Grosse, Rajesh Ranganath, and Andrew Y. Ng. "Convolutional Deep Belief Networks for Scalable Unsupervised Learning of Hierarchical Representations". In: *Proceedings of the 26th Annual International Conference on Machine Learning*. ICML '09. Montreal, Quebec, Canada: ACM, 2009, pp. 609–616. ISBN: 9781605585161. DOI: 10.1145/1553374.1553453. URL: https://doi.org/10.1145/1553374.1553453.

[180] Sergey Levine and Vladlen Koltun. "Guided Policy Search". In: *Proceedings of the 30th International Conference on Machine Learning*. Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. Proceedings of Machine Learning Research 3. Atlanta, Georgia, USA: PMLR, June 2013, pp. 1–9. URL: https://proceedings.mlr.press/v28/levine13.html.

[181] Christophe Leys, Christophe Ley, Olivier Klein, Philippe Bernard, and Laurent Licata. "Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median". In: *Journal of Experimental Social Psychology* 49.4 (2013), pp. 764–766. ISSN: 0022-1031. DOI: https://doi.org/10.1016/j.jesp.2013.03.013. URL: https://www.sciencedirect.com/science/article/pii/S0022103113000668.

[182] Chao Li, Yi Yang, Min Feng, Srimat Chakradhar, and Huiyang Zhou. "Optimizing Memory Efficiency for Deep Convolutional Neural Networks on GPUs". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '16. Salt Lake City, Utah: IEEE Press, 2016. ISBN: 9781467388153.

[183] Menghao Li, Minjia Zhang, Chi Wang, and Mingqin Li. "AdaTune: Adaptive Tensor Program Compilation Made Efficient". In: *Proceedings of the 34th International Conference on Neural Information Processing Systems*. NIPS'20. Vancouver, BC, Canada: Curran Associates Inc., 2020. ISBN: 9781713829546.

[184] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, et al. "The Deep Learning Compiler: A Comprehensive Survey". In: *IEEE Transactions on Parallel and Distributed Systems* 32.3 (2021), pp. 708–727. DOI: 10.1109/TPDS.2020.3030548.

[185] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. "Differentiable Programming for Image Processing and Deep Learning in Halide". In: *ACM Transactions on Graphics (ToG)* 37.4 (July 2018). ISSN: 0730-0301. DOI: 10.1145/3197517.3201383. URL: https://doi.org/10.1145/3197517.3201383.

[186] Leo Liberti, Carlile Lavor, Nelson Maculan, and Antonio Mucherino. "Euclidean Distance Geometry and Applications". In: *SIAM Review* 56.1 (2014), pp. 3–69. DOI: 10.1137/120875909. eprint: https://doi.org/10.1137/120875909. URL: https://doi.org/10.1137/120875909.

[187] Aristidis Likas, Nikos Vlassis, and Jakob J. Verbeek. "The global k-means clustering algorithm". In: *Pattern Recognition* 36.2 (2003). Biometrics, pp. 451–461. ISSN: 0031-3203. DOI: https://doi.org/10.1016/S0031-3203(02)00060-2. URL: https://www.sciencedirect.com/science/article/pii/S0031320302000602.

[188] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, et al. "Microsoft COCO: Common Objects in Context". In: *European Conference on Computer Vision – ECCV 2014*. Ed. by David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars. Cham: Springer International Publishing, 2014, pp. 740–755. ISBN: 978-3-319-10602-1.

[189] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. "Isolation forest". In: *2008 Eighth International Conference on Data Mining (ICDM)*. IEEE. 2008, pp. 413–422.

[190] Li Liu, Wanli Ouyang, Xiaogang Wang, Paul Fieguth, Jie Chen, et al. "Deep Learning for Generic Object Detection: A Survey". In: *International Journal of Computer Vision* 128.2 (Feb. 2020), pp. 261–318. ISSN: 1573-1405. DOI: 10.1007/s11263-019-01247-4. URL: https://doi.org/10.1007/s11263-019-01247-4.

[191] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, et al. "SSD: Single Shot MultiBox Detector". In: *European Conference on Computer Vision – ECCV 2016*. Ed. by Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling. Cham: Springer International Publishing, 2016, pp. 21–37. ISBN: 978-3-319-46448-0.

[192] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, et al. "Optimizing CNN Model Inference on CPUs". In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019, pp. 1025–1040. ISBN: 978-1-939133-03-8. URL: https://www.usenix.org/conference/atc19/presentation/liu-yizhi.

[193] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, et al. "Swin Transformer: Hierarchical Vision Transformer using Shifted Windows". In: *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*. Los Alamitos, CA, USA: IEEE Computer Society, Oct. 2021, pp. 9992–10002. DOI: 10.1109/ICCV48922.2021.00986. URL: https://doi.ieeecomputersociety.org/10.1109/ICCV48922.2021.00986.

[194] Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, et al. "A ConvNet for the 2020s". In: *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2022, pp. 11966–11976. DOI: 10.1109/CVPR52688.2022.01167.

[195] Chris Lomont. *Introduction to Intel® Advanced Vector Extensions*. 2011. URL: https://hpc.llnl.gov/sites/default/files/intelAVXintro.pdf (visited on 07/07/2023).

[196] Steven H. Low, Larry L. Peterson, and Limin Wang. "Understanding TCP Vegas: A Duality Model". In: *Journal of ACM* 49.2 (Mar. 2002), pp. 207–235. ISSN: 0004-5411. DOI: 10.1145/506147.506152. URL: https://doi.org/10.1145/506147.506152.

[197] D. Lu and Q. Weng. "A survey of image classification methods and techniques for improving classification performance". In: *International Journal of Remote Sensing* 28.5 (2007), pp. 823–870. DOI: 10.1080/01431160600746456. eprint: https://doi.org/10.1080/01431160600746456. URL: https://doi.org/10.1080/01431160600746456.

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

[199] Andrew L Maas, Awni Y Hannun, Andrew Y Ng, et al. "Rectifier nonlinearities improve neural network acoustic models". In: *Proceedings of the 30th International Conference on Machine Learning*. Vol. 30. 1. Citeseer. 2013, p. 3.

[200] Wolfgang Maass. "Networks of spiking neurons: The third generation of neural network models". In: *Neural Networks* 10.9 (1997), pp. 1659–1671. ISSN: 0893-6080. DOI: https://doi.org/10.1016/S0893-6080(97)00011-7. URL: https://www.sciencedirect.com/science/article/pii/S0893608097000117.

[201] Nishant Malpani. *A Brief Review of TVM: An Automated End-to-End Optimizing Compiler for Deep Learning*. URL: https://layman-n-ish.github.io/pdfs/TVM_Review_Report.pdf (visited on 07/07/2023).

[202] Christopher Manning and Hinrich Schutze. *Foundations of statistical natural language processing*. MIT press, 1999.

[203] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. "NVIDIA Tensor Core Programmability, Performance & Precision". In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2018, pp. 522–531. DOI: 10.1109/IPDPSW.2018.00091. URL: https://doi.ieeecomputersociety.org/10.1109/IPDPSW.2018.00091.

[204] Vladimir M. Matyushok, Vera A. Krasavina, and Sergey V. Matyushok. "Global artificial intelligence systems and technology market: formation and development trends". In: *RUDN Journal of Economics* 28.3 (2020), pp. 505–521. ISSN: 2313-2329. URL: https://journals.rudn.ru/economics/article/view/24669.

[205] Rowan McAllister, Yarin Gal, Alex Kendall, Mark van der Wilk, Amar Shah, et al. "Concrete Problems for Autonomous Vehicle Safety: Advantages of Bayesian Deep Learning". In: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*. International Joint Conferences on Artificial Intelligence Organization, Aug. 2017. DOI: 10.24963/ijcai.2017/661. URL: https://doi.org/10.24963%2Fijcai.2017%2F661.

[206] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. "Improving Data Locality with Loop Transformations". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 18.4 (July 1996), pp. 424–453. ISSN: 0164-0925. DOI: 10.1145/233561.233564. URL: https://doi.org/10.1145/233561.233564.

[207] Xinxin Mei and Xiaowen Chu. "Dissecting GPU Memory Hierarchy Through Microbenchmarking". In: *IEEE Transactions on Parallel and Distributed Systems* 28.1 (2017), pp. 72–86. DOI: 10.1109/TPDS.2016.2549523.

[208] Nicholas Metropolis and Stanislaw Ulam. "The monte carlo method". In: *Journal of the American statistical association* 44.247 (1949), pp. 335–341.

[209] Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernocky, and Sanjeev Khudanpur. "Recurrent neural network based language model". In: *Proceedings of Interspeech 2010*. 2010, pp. 1045–1048. DOI: 10.21437/Interspeech.2010-343.

[210] Sparsh Mittal and Shraiysh Vaishay. "A Survey of Techniques for Optimizing Deep Learning on GPUs". In: *J. Syst. Archit.* 99.C (Oct. 2019). ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2019.101635. URL: https://doi.org/10.1016/j.sysarc.2019.101635.

[211] Takeru Miyato, Andrew M. Dai, and Ian Goodfellow. "Adversarial Training Methods for Semi-Supervised Text Classification". In: (2017). URL: https://openreview.net/forum?id=r1X3g2_xl.

[212] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2018.

[213] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Yan, et al. "A Hardware–Software Blueprint for Flexible Deep Learning Specialization". In: *IEEE Micro* 39.5 (2019), pp. 8–16. DOI: 10.1109/MM.2019.2928962.

[214] Steven Muchnick et al. *Advanced compiler design implementation*. Morgan kaufmann, 1997.

[215] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. "Automatically Scheduling Halide Image Processing Pipelines". In: *ACM Transactions on Graphics (TOG)* 35.4 (July 2016). ISSN:

0730-0301. DOI: 10.1145/2897824.2925952. URL: https://doi.org/10.1145/2897824.2925952.

[216] Berndt Müller, Joachim Reinhardt, and Michael T Strickland. *Neural networks: an introduction*. Springer Science & Business Media, 1995.

[217] 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. Haifa, Israel: Omnipress, 2010, pp. 807–814. ISBN: 9781605589077.

[218] Supun Nakandala, Karla Saur, Gyeong-In Yu, Konstantinos Karanasos, Carlo Curino, et al. "A Tensor Compiler for Unified Machine Learning Prediction Serving". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 899–917. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/nakandala.

[219] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, et al. "Reading Digits in Natural Images with Unsupervised Feature Learning". In: (2011). URL: http://ufldl.stanford.edu/housenumbers/nips2011_housenumbers.pdf.

[220] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. "DNNFusion: Accelerating Deep Neural Networks Execution with Advanced Operator Fusion". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: ACM, 2021, pp. 883–898. ISBN: 9781450383912. DOI: 10.1145/3453483.3454083. URL: https://doi.org/10.1145/3453483.3454083.

[221] Zhaoyang Niu, Guoqiang Zhong, and Hui Yu. "A review on the attention mechanism of deep learning". English. In: *Neurocomputing* 452 (Sept. 2021).

Funding Information: This work was supported by the National Key Research and Development Program of China under Grant No. 2018AAA0100400, the Joint Fund of the Equipments Pre-Research and Ministry of Education of China under Grant No. 6141A020337, the Natural Science Foundation of Shandong Province under Grant No. ZR2020MF131, and the Science and Technology Program of Qingdao under Grant No. 21-1-4-ny-19-nsh. Publisher Copyright: © 2021 Elsevier B.V., pp. 48–62. ISSN: 0925-2312. DOI: 10.1016/j.neucom.2021.03.091.

[222] NVIDIA. *Nvidia Tesla P100 GPU Architecture*. NVIDIA. 2016. URL: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf (visited on 07/07/2023).

[223] NVIDIA. *Nvidia Tesla V100 GPU Architecture*. NVIDIA. 2017. URL: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf (visited on 07/07/2023).

[224] NVIDIA. *Nvidia Turing GPU Architecture*. NVIDIA. 2018. URL: https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf (visited on 07/07/2023).

[225] NVIDIA. *CUDA LLVM Compiler | NVIDIA Developer*. NVIDIA. 2022. URL: https://developer.nvidia.com/cuda-llvm-compiler#cudacompilersdk (visited on 07/07/2023).

[226] NVIDIA. *CUPTI :: CUPTI Documentation*. NVIDIA. 2022. URL: https://docs.nvidia.com/cupti/Cupti/index.html (visited on 07/07/2023).

[227] NVIDIA. *CUTLASS: CUDA templates for linear algebra subroutines*. NVIDIA. 2022. URL: https://github.com/NVIDIA/cutlass (visited on 07/07/2023).

[228] NVIDIA. *Multi-Process Service - MPS*. NVIDIA. 2022. URL: https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf (visited on 07/07/2023).

[229] NVIDIA. *NVIDIA A100 Tensor Core GPU Architecture*. NVIDIA. 2022. URL: https://images.nvidia.com/aem-dam/en-zz/Solutions/datacenter/nvidia-ampere-architecture-whitepaper.pdf (visited on 07/07/2023).

[230] NVIDIA. *NVIDIA cuBLAS*. NVIDIA. 2022. URL: https://developer.nvidia.com/cublas (visited on 07/07/2023).

[231] NVIDIA. *NVIDIA CUDA*. NVIDIA. 2022. URL: https://developer.nvidia.com/cuda-toolkit (visited on 07/07/2023).

[232] NVIDIA. *NVIDIA cuDNN*. NVIDIA. 2022. URL: https://developer.nvidia.com/cudnn (visited on 07/07/2023).

[233] NVIDIA. *NVIDIA Deep Learning Accelerator*. NVIDIA. 2022. URL: https://nvdla.org (visited on 07/07/2023).

[234] NVIDIA. *NVIDIA Management Library (NVML)*. NVIDIA. 2022. URL: https://developer.nvidia.com/nvidia-management-library-nvml (visited on 07/07/2023).

[235] NVIDIA. *NVIDIA Nsight Compute | NVIDIA Developer*. NVIDIA. 2022. URL: https://developer.nvidia.com/nsight-compute (visited on 07/07/2023).

[236] NVIDIA. *NVIDIA Nsight Systems*. NVIDIA. 2022. URL: https://developer.nvidia.com/nsight-systems (visited on 07/07/2023).

[237] NVIDIA. *NVIDIA Tensor Cores: Versatility for HPC & AI*. NVIDIA. 2022. URL: https://www.nvidia.com/en-gb/data-center/tensor-cores/ (visited on 07/07/2023).

[238] NVIDIA. *NVIDIA Tools Extension (NVTX) Documentation*. NVIDIA. 2022. URL: https://docs.nvidia.com/nvtx/index.html (visited on 07/07/2023).

[239] NVIDIA. *Programming Guide :: CUDA Toolkit Documentation*. NVIDIA. 2022. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (visited on 07/07/2023).

[240] NVIDIA. *TensorRT*. NVIDIA. 2022. URL: https://developer.nvidia.com/tensorrt (visited on 07/07/2023).

[241] OctoML. *Accelerated Machine Learning Deployment | OctoML*. 2022. URL: https://octoml.ai/ (visited on 07/07/2023).

[242] Jian Ouyang, Mijung Noh, Yong Wang, Wei Qi, Yin Ma, et al. "Baidu Kunlun An AI processor for diversified workloads". In: *2020 IEEE Hot Chips 32 Symposium (HCS)*. 2020, pp. 1–18. DOI: 10.1109/HCS49909.2020.9220641.

[243] PaddlePaddle. *PaddlePaddle-Parallel Distributed Deep Learning, efficient and extensible deep learning framework.* 2022. URL: https://www.paddlepaddle.org.cn/en (visited on 07/07/2023).

[244] Sinno Jialin Pan and Qiang Yang. "A Survey on Transfer Learning". In: *IEEE Transactions on Knowledge and Data Engineering* 22.10 (2010), pp. 1345–1359. DOI: 10.1109/TKDE.2009.191.

[245] Dhabaleswar K Panda, Xiaoyi Lu, and Dipti Shankar. *High-Performance Big Data Computing*. MIT Press, 2022.

[246] P.R. Panda, H. Nakamura, N.D. Dutt, and A. Nicolau. "Augmenting loop tiling with data alignment for improved cache performance". In: *IEEE Transactions on Computers* 48.2 (1999), pp. 142–149. DOI: 10.1109/12.752655.

[247] Hyunbin Park and Shiho Kim. "Chapter Three - Hardware accelerator systems for artificial intelligence and machine learning". In: *Hardware Accelerator Systems for Artificial Intelligence and Machine Learning*. Ed. by Shiho Kim and Ganesh Chandra Deka. Vol. 122. Advances in Computers. Elsevier, 2021, pp. 51–95. DOI: https://doi.org/10.1016/bs.adcom.2020.11.005. URL: https://www.sciencedirect.com/science/article/pii/S0065245820300929.

[248] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, et al. "Deep Learning Inference in Facebook Data Centers: Characterization, Performance Optimizations and Hardware Implications". In: *CoRR* abs/1811.09886 (2018). arXiv: 1811.09886. URL: http://arxiv.org/abs/1811.09886.

[249] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, et al. "Automatic Differentiation in PyTorch". In: (2017). URL: https://openreview.net/forum?id=BJJsrmfCZ.

[250] D. Petkov, R. Harr, and S. Amarasinghe. "Efficient pipelining of nested loops: unroll-and-squash". In: *Proceedings 16th International Parallel and Distributed Processing Symposium*. 2002, p. 6. DOI: 10.1109/IPDPS.2002.1015491.

[251] T. Pham-Gia and T.L. Hung. "The mean and median absolute deviations". In: *Mathematical and Computer Modelling* 34.7 (2001), pp. 921–936. ISSN: 0895-7177. DOI: https://doi.org/10.1016/S0895-7177(01)00109-1. URL: https://www.sciencedirect.com/science/article/pii/S0895717701001091.

[252] George Philipp, Dawn Song, and Jaime G. Carbonell. "Gradients explode - Deep Networks are shallow - ResNet explained". In: *CoRR* abs/1712.05577 (2017). arXiv: 1712.05577. URL: http://arxiv.org/abs/1712.05577.

[253] Robert Philipp, Andreas Mladenow, Christine Strauss, and Alexander Völz. "Machine Learning as a Service: Challenges in Research and Applications". In:

Proceedings of the 22nd International Conference on Information Integration and Web-Based Applications & Services. iiWAS '20. Chiang Mai, Thailand: ACM, 2021, pp. 396–406. ISBN: 9781450389228. DOI: 10.1145/3428757.3429152. URL: https://doi.org/10.1145/3428757.3429152.

[254] Rajat Phull, Cheng-Hong Li, Kunal Rao, Hari Cadambi, and Srimat Chakradhar. "Interference-Driven Resource Management for GPU-Based Heterogeneous Clusters". In: *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '12. Delft, The Netherlands: ACM, 2012, pp. 109–120. ISBN: 9781450308052. DOI: 10.1145/2287076.2287091. URL: https://doi.org/10.1145/2287076.2287091.

[255] PlaidML. *PlaidML is a framework for making deep learning work everywhere*. 2022. URL: https://github.com/plaidml/plaidml (visited on 07/07/2023).

[256] Matthias Pohl, Sascha Bosse, and Klaus Turowski. "A Data-Science-as-a-Service Model". In: *Proceedings of the 8th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER*. INSTICC. SciTePress, 2018, pp. 432–439. ISBN: 978-989-758-295-0. DOI: 10.5220/0006703104320439.

[257] M. Potkonjak, M.B. Srivastava, and A.P. Chandrakasan. "Multiple constant multiplications: efficient and versatile framework and algorithms for exploring common subexpression elimination". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 15.2 (1996), pp. 151–165. DOI: 10.1109/43.486662.

[258] Julia M Puaschunder. "Artificial Intelligence market disruption". In: *Proceedings of the International RAIS Conference on Social Sciences and Humanities organized by Research Association for Interdisciplinary Studies (RAIS) at Johns Hopkins University, Montgomery County Campus, Rockville, MD, United States*. 2019, pp. 1–8.

[259] Qualcomm. *Hexagon SDK - DSP Processor - Qualcomm Developer Network*. 2022. URL: https://developer.qualcomm.com/software/hexagon-dsp-sdk/dsp-processor (visited on 07/07/2023).

[260] Alec Radford, Luke Metz, and Soumith Chintala. "Unsupervised representation learning with deep convolutional generative adversarial networks". In: *arXiv preprint arXiv:1511.06434* abs/1511.06434 (2015). arXiv: 1511.06434. URL: http://arxiv.org/abs/1511.06434.

[261] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, et al. "Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines". In: *ACM Transactions on Graphics (TOG)* 31.4 (July 2012). ISSN: 0730-0301. DOI: 10.1145/2185520.2185528. URL: https://doi.org/10.1145/2185520.2185528.

[262] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, et al. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA: ACM, 2013, pp. 519–530. ISBN: 9781450320146. DOI: 10.1145/2491956.2462176. URL: https://doi.org/10.1145/2491956.2462176.

[263] Rajat Raina, Anand Madhavan, and Andrew Y. Ng. "Large-Scale Deep Unsupervised Learning Using Graphics Processors". In: *Proceedings of the 26th Annual International Conference on Machine Learning*. ICML '09. Montreal, Quebec, Canada: ACM, 2009, pp. 873–880. ISBN: 9781605585161. DOI: 10.1145/1553374.1553486. URL: https://doi.org/10.1145/1553374.1553486.

[264] Hassan Ramchoun, Youssef Ghanou, Mohamed Ettaouil, and Mohammed Amine Janati Idrissi. "Multilayer Perceptron: Architecture Optimization and Training

with Mixed Activation Functions". In: BDCA'17 (2017). DOI: 10.1145/3090354.3090427. URL: https://doi.org/10.1145/3090354.3090427.

[265] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, et al. "Zero-Shot Text-to-Image Generation". In: *Proceedings of the 38th International Conference on Machine Learning*. Ed. by Marina Meila and Tong Zhang. Vol. 139. Proceedings of Machine Learning Research. PMLR, July 2021, pp. 8821–8831. URL: https://proceedings.mlr.press/v139/ramesh21a.html.

[266] Ari Rasch, Michael Haidl, and Sergei Gorlatch. "ATF: A Generic Auto-Tuning Framework". In: *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 2017, pp. 64–71. DOI: 10.1109/HPCC-SmartCity-DSS.2017.9.

[267] Joseph Redmon. *Darknet: Open Source Neural Networks in C*. 2013. URL: http://pjreddie.com/darknet/ (visited on 07/07/2023).

[268] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. "You Only Look Once: Unified, Real-Time Object Detection". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, June 2016, pp. 779–788. DOI: 10.1109/CVPR.2016.91. URL: https://doi.ieeecomputersociety.org/10.1109/CVPR.2016.91.

[269] Payam Refaeilzadeh, Lei Tang, and Huan Liu. "Cross-Validation". In: *Encyclopedia of Database Systems*. Ed. by LING LIU and M. TAMER ÖZSU. Boston, MA: Springer US, 2009, pp. 532–538. ISBN: 978-0-387-39940-9. DOI: 10.1007/978-0-387-39940-9_565. URL: https://doi.org/10.1007/978-0-387-39940-9_565.

[270] Oliver Reiche, Christof Kobylko, Frank Hannig, and Jürgen Teich. "Auto-Vectorization for Image Processing DSLs". In: *Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES 2017. Barcelona, Spain: ACM, 2017, pp. 21–30. ISBN: 9781450350303. DOI: 10.1145/3078633.3081039. URL: https://doi.org/10.1145/3078633.3081039.

[271] Steven Rennich. *CUDA Streams and Concurrency*. NVIDIA. 2022. URL: https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf (visited on 07/07/2023).

[272] Injong Rhee, Lisong Xu, Sangtae Ha, Alexander Zimmermann, Lars Eggert, et al. *CUBIC for Fast Long-Distance Networks*. RFC 8312. Feb. 2018. DOI: 10.17487/RFC8312. URL: https://www.rfc-editor.org/info/rfc8312.

[273] Mauro Ribeiro, Katarina Grolinger, and Miriam A.M. Capretz. "MLaaS: Machine Learning as a Service". In: *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*. 2015, pp. 896–902. DOI: 10.1109/ICMLA.2015.152.

[274] Giampaolo Rodola. *giampaolo/psutil: Cross-platform lib for process and system monitoring in Python*. 2022. URL: https://github.com/giampaolo/psutil (visited on 07/07/2023).

[275] Andres Rodriguez. *Deep Learning Systems: Algorithms, Compilers, and Processors for Large-Scale Production*. Springer Nature, 2022.

[276] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, et al. "Relay: A New IR for Machine Learning Frameworks". In: *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. MAPL 2018. Philadelphia, PA, USA: ACM, 2018,

pp. 58–68. ISBN: 9781450358347. DOI: 10.1145/3211346.3211348. URL: https://doi.org/10.1145/3211346.3211348.

[277] Lior Rokach and Oded Maimon. "Decision trees". In: *Data mining and knowledge discovery handbook*. Springer, 2005, pp. 165–192.

[278] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, et al. "Glow: Graph Lowering Compiler Techniques for Neural Networks". In: *CoRR* abs/1805.00907 (2018). arXiv: 1805.00907. URL: http://arxiv.org/abs/1805.00907.

[279] Peter J. Rousseeuw and Christophe Croux. "Alternatives to the Median Absolute Deviation". In: *Journal of the American Statistical Association* 88.424 (1993), pp. 1273–1283. ISSN: 01621459. URL: http://www.jstor.org/stable/2291267 (visited on 07/15/2023).

[280] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: *CoRR* abs/1609.04747 (2016). arXiv: 1609.04747. URL: http://arxiv.org/abs/1609.04747.

[281] Kanokwan Rungsuptaweekoon, Vasaka Visoottiviseth, and Ryousei Takano. "Evaluating the power efficiency of deep learning inference on embedded GPU systems". In: *2017 2nd International Conference on Information Technology (INCIT)*. 2017, pp. 1–5. DOI: 10.1109/INCIT.2017.8257866.

[282] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, et al. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision* 115.3 (Dec. 2015), pp. 211–252. ISSN: 1573-1405. DOI: 10.1007/s11263-015-0816-y. URL: https://doi.org/10.1007/s11263-015-0816-y.

[283] Jaehun Ryu, Eunhyeok Park, and Hyojin Sung. "One-Shot Tuner for Deep Learning Compilers". In: *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*. CC 2022. Seoul, South Korea: ACM,

2022, pp. 89–103. ISBN: 9781450391832. DOI: 10.1145/3497776.3517774. URL: https://doi.org/10.1145/3497776.3517774.

[284] Jaehun Ryu and Hyojin Sung. "MetaTune: Meta-Learning Based Cost Model for Fast and Efficient Auto-tuning Frameworks". In: *CoRR* abs/2102.04199 (2021). arXiv: 2102.04199. URL: https://arxiv.org/abs/2102.04199.

[285] Faramarz Sadr. *Heat Pump or CHP – which one is greener?* Rehva. 2014. URL: https://www.rehva.eu/fileadmin/REHVA_Journal/REHVA_Journal_2014/RJ_issue_5/P.26/26-29_RJ1405_WEB.pdf (visited on 07/07/2023).

[286] S Rasoul Safavian and David Landgrebe. "A survey of decision tree classifier methodology". In: *IEEE Transactions on Systems, Man, and Cybernetics* 21.3 (1991), pp. 660–674. DOI: 10.1109/21.97458.

[287] Augusto Sampaio. *An Algebraic Approach to Compiler Design*. Vol. 4. World Scientific, 1997.

[288] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L. Chen. "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. URL: https://doi.ieeecomputersociety.org/10.1109/CVPR.2018.00474.

[289] Vivek Sarkar. "Optimized Unrolling of Nested Loops". In: *Proceedings of the 14th International Conference on Supercomputing*. ICS '00. Santa Fe, New Mexico, USA: ACM, 2000, pp. 153–166. ISBN: 1581132700. DOI: 10.1145/335231.335246. URL: https://doi.org/10.1145/335231.335246.

[290] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. "Proximal Policy Optimization Algorithms". In: *CoRR* abs/1707.06347 (2017). arXiv: 1707.06347. URL: http://arxiv.org/abs/1707.06347.

[291] Roy Schwartz, Jesse Dodge, Noah A. Smith, and Oren Etzioni. "Green AI". In: *Communications of the ACM* 63.12 (Nov. 2020), pp. 54–63. ISSN: 0001-0782. DOI: 10.1145/3381831. URL: https://doi.org/10.1145/3381831.

[292] Frank Seide and Amit Agarwal. "CNTK: Microsoft's Open-Source Deep-Learning Toolkit". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: ACM, 2016, p. 2135. ISBN: 9781450342322. DOI: 10.1145/2939672.2945397. URL: https://doi.org/10.1145/2939672.2945397.

[293] Amazon Web Services. *Amazon EC2 F1 Instances*. Amazon Inc. 2022. URL: https://aws.amazon.com/ec2/instance-types/f1/ (visited on 07/07/2023).

[294] Amazon Web Services. *Amazon EC2 Inf1 Instances*. Amazon Inc. 2022. URL: https://aws.amazon.com/ec2/instance-types/inf1/ (visited on 07/07/2023).

[295] Amazon Web Services. *Amazon EC2 P3 - Ideal for Machine Learning and HPC - AWS*. Amazon Inc. 2022. URL: https://aws.amazon.com/ec2/instance-types/p3/ (visited on 07/07/2023).

[296] Amazon Web Services. *Amazon SageMaker Neo*. Amazon Inc. 2022. URL: https://aws.amazon.com/sagemaker/ (visited on 07/07/2023).

[297] Amazon Web Services. *AWS Neuron - Amazon Web Services*. Amazon Inc. 2022. URL: https://aws.amazon.com/machine-learning/neuron/ (visited on 07/07/2023).

[298] Amazon Web Services. *Machine Learning and Artificial Intelligence - Amazon Web Services*. Amazon Inc. 2022. URL: https://aws.amazon.com/machine-learning/ (visited on 07/07/2023).

[299] Jing-Cheng Shi, Yang Yu, Qing Da, Shi-Yong Chen, and An-Xiang Zeng. "Virtual-Taobao: Virtualizing Real-World Online Retail Environment for Reinforcement Learning". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 01. July 2019, pp. 4902–4909. DOI: 10.1609/aaai.v33i01.33014902. URL: https://ojs.aaai.org/index.php/AAAI/article/view/4419.

[300] Stephen H. Shum, Najim Dehak, Réda Dehak, and James R. Glass. "Unsupervised Methods for Speaker Diarization: An Integrated and Iterative Approach". In: *IEEE Transactions on Audio, Speech, and Language Processing* 21.10 (2013), pp. 2015–2028. DOI: 10.1109/TASL.2013.2264673.

[301] Laurent Sifre and Prof Stéphane Mallat. "Rigid-motion scattering for image classification author". In: *Ecole Polytechnique, CMAP PhD thesis* (2014).

[302] Andreas Simbürger, Sven Apel, Armin Größlinger, and Christian Lengauer. "The potential of polyhedral optimization: An empirical study". In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2013, pp. 508–518. DOI: 10.1109/ASE.2013.6693108.

[303] Jean Paul Simon. "Artificial intelligence: scope, players, markets and geography". In: *Digital Policy, Regulation and Governance* 21.3 (Jan. 2019), pp. 208–237. ISSN: 2398-5038. DOI: 10.1108/DPRG-08-2018-0039. URL: https://doi.org/10.1108/DPRG-08-2018-0039.

[304] K Simonyan and A Zisserman. "Very deep convolutional networks for large-scale image recognition". In: Computational and Biological Learning Society, 2015, pp. 1–14.

[305] Shashi Pal Singh, Ajai Kumar, Hemant Darbari, Lenali Singh, Anshika Rastogi, et al. "Machine translation using deep learning: An overview". In: *2017 International Conference on Computer, Communications and Electronics (Comptelix)*. 2017, pp. 162–167. DOI: 10.1109/COMPTELIX.2017.8003957.

[306] Shikhar Singh, James Hegarty, Hugh Leather, and Benoit Steiner. "A Graph Neural Network-Based Performance Model for Deep Learning Applications". In: *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. MAPS 2022. San Diego, CA, USA: ACM, 2022, pp. 11–20. ISBN: 9781450392730. DOI: 10.1145/3520312.3534863. URL: https://doi.org/10.1145/3520312.3534863.

[307] Muthian Sivathanu, Tapan Chugh, Sanjay S. Singapuram, and Lidong Zhou. "Astra: Exploiting Predictability to Optimize Deep Learning". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: ACM, 2019, pp. 909–923. ISBN: 9781450362405. DOI: 10.1145/3297858.3304072. URL: https://doi.org/10.1145/3297858.3304072.

[308] EN Smirnov and SA Lukyanov. "Development of the global market of artificial intelligence systems". In: *Economy of Region* 1 (15 2019), pp. 57–69. URL: http://hdl.handle.net/10995/90350.

[309] David Snyder, Daniel Garcia-Romero, Daniel Povey, and Sanjeev Khudanpur. "Deep Neural Network Embeddings for Text-Independent Speaker Verification". In: *Proc. Interspeech 2017*. 2017, pp. 999–1003. DOI: 10.21437/Interspeech.2017-620.

[310] Petr Spelda and Vit Stritecky. "The future of human-artificial intelligence nexus and its environmental costs". In: *Futures* 117 (2020), p. 102531. ISSN: 0016-3287. DOI: https://doi.org/10.1016/j.futures.2020.102531. URL: https://www.sciencedirect.com/science/article/pii/S0016328720300215.

[311] Richard M Stallman. *the GCC Developer Community (2008) Using the GNU compiler collection*.

[312] Markus Steinberger, Michael Kenzel, Bernhard Kainz, and Dieter Schmalstieg. "ScatterAlloc: Massively parallel dynamic memory allocation for the GPU". In: *2012 Innovative Parallel Computing (InPar)*. 2012, pp. 1–10. DOI: 10.1109/InPar.2012.6339604.

[313] Benoit Steiner, Chris Cummins, Horace He, and Hugh Leather. "Value Learning for Throughput Optimization of Deep Learning Workloads". In: 3 (2021). Ed. by A. Smola, A. Dimakis, and I. Stoica, pp. 323–334. URL: https://proceedings.mlsys.org/paper_files/paper/2021/file/a7e5da037a0afc90fa84386586929a26-Paper.pdf.

[314] Dave Steinkraus, Ian Buck, and PY Simard. "Using GPUs for machine learning algorithms". In: *Eighth International Conference on Document Analysis and Recognition (ICDAR'05)*. Vol. 2. 2005, pp. 1115–1120. DOI: 10.1109/ICDAR.2005.251.

[315] Rainer Storn and Kenneth Price. "Differential Evolution – A Simple and Efficient Heuristic for global Optimization over Continuous Spaces". In: *Journal of Global Optimization* 11.4 (Dec. 1997), pp. 341–359. ISSN: 1573-2916. DOI: 10.1023/A:1008202821328. URL: https://doi.org/10.1023/A:1008202821328.

[316] Gilbert Strang. "A Proposal for Toeplitz Matrix Calculations". In: *Stud. Appl. Math.* 74.2 (Apr. 1986), pp. 171–176. ISSN: 0022-2526. DOI: 10.1002/sapm1986742171. URL: https://doi.org/10.1002/sapm1986742171.

[317] Volker Strassen. "Gaussian elimination is not optimal". In: *Numerische Mathematik* 13.4 (Aug. 1969), pp. 354–356. ISSN: 0945-3245. DOI: 10.1007/BF02165411. URL: https://doi.org/10.1007/BF02165411.

[318] Emma Strubell, Ananya Ganesh, and Andrew McCallum. "Energy and Policy Considerations for Deep Learning in NLP". In: *CoRR* abs/1906.02243 (2019). arXiv: 1906.02243. URL: http://arxiv.org/abs/1906.02243.

[319] Emma Strubell, Ananya Ganesh, and Andrew McCallum. "Energy and Policy Considerations for Modern Deep Learning Research". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 09. Apr. 2020, pp. 13693–13696. DOI: 10.1609/aaai.v34i09.7123. URL: https://ojs.aaai.org/index.php/AAAI/article/view/7123.

[320] Jiacheng Sun, Xiangyong Cao, Hanwen Liang, Weiran Huang, Zewei Chen, et al. "New Interpretations of Normalization Methods in Deep Learning". In: vol. 34. 04. Apr. 2020, pp. 5875–5882. DOI: 10.1609/aaai.v34i04.6046. URL: https://ojs.aaai.org/index.php/AAAI/article/view/6046.

[321] Wei Sun, Savvas Sioutas, Sander Stuijk, Andrew Nelson, and Henk Corporaal. "Efficient Tensor Cores support in TVM for Low-Latency Deep learning". In: *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2021, pp. 120–123. DOI: 10.23919/DATE51398.2021.9473984.

[322] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[323] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. *Efficient processing of deep neural networks*. Springer, 2020.

[324] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. "Efficient Processing of Deep Neural Networks: A Tutorial and Survey". In: *Proceedings of the IEEE* 105.12 (2017), pp. 2295–2329. DOI: 10.1109/JPROC.2017.2761740.

[325] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. "Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks". In: (July 2015), pp. 1556–1566. DOI: 10.3115/v1/P15-1150. URL: https://aclanthology.org/P15-1150.

[326] Shigeyuki Takano. *Thinking machines: machine learning and its hardware implementation*. Academic Press, 2021.

[327] Google Brain Team. *Architecture | TFX | TensorFlow*. Google. 2022. URL: https://www.tensorflow.org/tfx/serving/architecture (visited on 07/07/2023).

[328] Google Brain Team. *TensorFlow*. Google. 2022. URL: https://www.tensorflow.org/ (visited on 07/07/2023).

[329] Google Brain Team. *TensorFlow Lite | ML for Mobile and Edge Devices | TensorFlow*. Google. 2022. URL: https://www.tensorflow.org/lite (visited on 07/07/2023).

[330] Google Brain Team. *TensorFlow.js | Machine Learning for JavaScript Developers | TensorFlow*. Google. 2022. URL: https://www.tensorflow.org/js (visited on 07/07/2023).

[331] Keras Team. *Keras: the Python deep learning API*. 2022. URL: https://keras.io/ (visited on 07/07/2023).

[332] Neil C. Thompson, Kristjan H. Greenewald, Keeheon Lee, and Gabriel F. Manso. "The Computational Limits of Deep Learning". In: *CoRR* abs/2007.05558 (2020). arXiv: 2007.05558. URL: https://arxiv.org/abs/2007.05558.

[333] Camilla Thomson and Gareth Harrison. *Life cycle costs and carbon emissions of wind power: Executive Summary*. English. ClimateXChange, June 2015.

[334] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. "Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations". In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. MAPL 2019. Phoenix, AZ, USA: ACM, 2019, pp. 10–19. ISBN: 9781450367196. DOI: 10.1145/3315508.3329973. URL: https://doi.org/10.1145/3315508.3329973.

[335] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. "Chainer: a next-generation open source framework for deep learning". In: *Proceedings of Workshop on Machine Learning Systems (LearningSys) in the Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*. Vol. 5. 2015, pp. 1–6.

[336] Aditya Singh Tomar, Animesh Sharma, Aditya Shrivastava, Anurag Singh Rana, and Pradeep Yadav. "A Comparative Analysis of Activation Function, Evaluating their Accuracy and Efficiency when Applied to Miscellaneous Datasets". In: (2023), pp. 1035–1042. DOI: 10.1109/ICAAIC56838.2023.10140823.

[337] Giovanni Tripepi, KJ Jager, FW Dekker, and Carmine Zoccali. "Linear and logistic regression analysis". In: *Kidney Disease and Population Health* 73.7 (2008), pp. 806–810.

[338] UK Deptartment for Business, Energy & Industrial Strategy. *Combined heat and power*. 2022. URL: https://www.gov.uk/guidance/combined-heat-and-power (visited on 07/07/2023).

[339] UK Deptartment for Business, Energy & Industrial Strategy. *Gas and electricity prices in the non-domestic sector*. 2022. URL: https://www.gov.uk/government/statistical-data-sets/gas-and-electricity-prices-in-the-non-domestic-sector (visited on 07/07/2023).

[340] UK Deptartment for Business, Energy & Industrial Strategy. *UK Energy in Brief 2022*. 2022. URL: https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/1094025/UK_Energy_in_Brief_2022.pdf (visited on 07/07/2023).

[341] Lancaster University. *Energy and Carbon | Lancaster University*. Lancaster University. 2022. URL: https://www.lancaster.ac.uk/sustainability/action/energy-and-carbon/ (visited on 07/07/2023).

[342] Peter JM Van Laarhoven, Emile HL Aarts, Peter JM van Laarhoven, and Emile HL Aarts. *Simulated annealing*. Springer, 1987.

[343] Ben van Werkhoven. "Kernel Tuner: A search-optimizing GPU code auto-tuner". In: *Future Generation Computer Systems* 90 (2019), pp. 347–358. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2018.08.004. URL: https://www.sciencedirect.com/science/article/pii/S0167739X18313359.

[344] Nicolas Vasilache, Jeff Johnson, Michael Mathieu, Soumith Chintala, Serkan Piantino, et al. "Fast convolutional nets with fbfft: A GPU performance evaluation". In: *CoRR* abs/1412.7580 (2014). arXiv: 1412.7580. URL: http://arxiv.org/abs/1412.7580.

[345] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, et al. "Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions". In: *CoRR* abs/1802.04730 (2018). arXiv: 1802.04730. URL: http://arxiv.org/abs/1802.04730.

[346] Aravind Vasudevan, Andrew Anderson, and David Gregg. "Parallel Multi Channel convolution using General Matrix Multiplication". In: *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. Los Alamitos, CA, USA: IEEE Computer Society, July 2017, pp. 19–24. DOI: 10.1109/ASAP.2017.7995254. URL: https://doi.ieeecomputersociety.org/10.1109/ASAP.2017.7995254.

[347] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, et al. "Attention is All you Need". In: 30 (2017). Ed. by I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[348] David J. Wales and Jonathan P. K. Doye. "Global Optimization by Basin-Hopping and the Lowest Energy Structures of Lennard-Jones Clusters Containing up to 110 Atoms". In: *The Journal of Physical Chemistry A* 101.28 (1997), pp. 5111–5116. DOI: 10.1021/jp970984n. eprint: https://doi.org/10.1021/jp970984n. URL: https://doi.org/10.1021/jp970984n.

[349] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, et al. "PET: Optimizing Tensor Programs with Partially Equivalent Transformations and Automated Corrections". In: *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, July 2021, pp. 37–54. ISBN: 978-1-939133-22-9. URL: https://www.usenix.org/conference/osdi21/presentation/wang.

[350] Leyuan Wang, Zhi Chen, Yizhi Liu, Yao Wang, Lianmin Zheng, et al. "A Unified Optimization Approach for CNN Model Inference on Integrated GPUs". In: *Proceedings of the 48th International Conference on Parallel Processing*. ICPP '19. Kyoto, Japan: ACM, 2019. ISBN: 9781450362955. DOI: 10.1145/3337821.3337839. URL: https://doi.org/10.1145/3337821.3337839.

[351] Qi Wang, Yue Ma, Kun Zhao, and Yingjie Tian. "A Comprehensive Survey of Loss Functions in Machine Learning". In: *Annals of Data Science* 9.2 (Apr. 2022), pp. 187–212. ISSN: 2198-5812. DOI: 10.1007/s40745-020-00253-5. URL: https://doi.org/10.1007/s40745-020-00253-5.

[352] Christopher J.C.H. Watkins and Peter Dayan. "Technical Note: Q-Learning". In: *Machine Learning* 8.3 (May 1992), pp. 279–292. ISSN: 1573-0565. DOI: 10.1023/A:1022676722315. URL: https://doi.org/10.1023/A:1022676722315.

[353] Tsung-Hsien Wen, Milica Gašić, Nikola Mrkšić, Pei-Hao Su, David Vandyke, et al. "Semantically Conditioned LSTM-based Natural Language Generation for Spoken

Dialogue Systems". In: (Sept. 2015), pp. 1711–1721. DOI: 10.18653/v1/D15-1199. URL: https://aclanthology.org/D15-1199.

[354] Ben van Werkhoven, Jason Maassen, Henri E. Bal, and Frank J. Seinstra. "Optimizing Convolution Operations on GPUs Using Adaptive Tiling". In: *Future Gener. Comput. Syst.* 30.C (Jan. 2014), pp. 14–26. ISSN: 0167-739X.

[355] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *Commun. ACM* 52.4 (Apr. 2009), pp. 65–76. ISSN: 0001-0782. DOI: 10.1145/1498765.1498785. URL: https://doi.org/10.1145/1498765.1498785.

[356] Cort J. Willmott and Kenji Matsuura. "Advantages of the mean absolute error (MAE) over the root mean square error (RMSE) in assessing average model performance". In: *Climate Research* 30.1 (2005), pp. 79–82. DOI: 10.3354/cr030079. URL: https://doi.org/10.3354/cr030079.

[357] Shmuel Winograd. *Arithmetic complexity of computations*. Vol. 33. Siam, 1980.

[358] Svante Wold, Kim Esbensen, and Paul Geladi. "Principal component analysis". In: *Chemometrics and Intelligent Laboratory Systems* 2.1 (1987). Proceedings of the Multivariate Statistical Workshop for Geologists and Geochemists, pp. 37–52. ISSN: 0169-7439. DOI: https://doi.org/10.1016/0169-7439(87)80084-9. URL: https://www.sciencedirect.com/science/article/pii/0169743987800849.

[359] M. Wolfe. "More Iteration Space Tiling". In: *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*. Supercomputing '89. Reno, Nevada, USA: ACM, 1989, pp. 655–664. ISBN: 0897913418. DOI: 10.1145/76263.76337. URL: https://doi.org/10.1145/76263.76337.

[360] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, et al. "Machine Learning at Facebook: Understanding Inference at the Edge". In: *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2019, pp. 331–344. DOI: 10.1109/HPCA.2019.00048.

[361] Haicheng Wu, Gregory Diamos, Jin Wang, Srihari Cadambi, Sudhakar Yalamanchili, et al. "Optimizing Data Warehousing Applications for GPUs Using Kernel Fusion/Fission". In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. 2012, pp. 2433–2442. DOI: 10.1109/IPDPSW.2012.300.

[362] Jiajun Wu, Chengkai Zhang, Tianfan Xue, Bill Freeman, and Josh Tenenbaum. "Learning a Probabilistic Latent Space of Object Shapes via 3D Generative-Adversarial Modeling". In: 29 (2016). Ed. by D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett. URL: https://proceedings.neurips.cc/paper_files/paper/2016/file/44f683a84163b3523afe57c2e008bc8c-Paper.pdf.

[363] Zhang Xianyi. *OpenBLAS : An optimized BLAS library*. 2022. URL: https://www.openblas.net/ (visited on 07/07/2023).

[364] Jiarong Xing, Leyuan Wang, Shang Zhang, Jack Chen, Ang Chen, et al. "Bolt: Bridging the Gap between Auto-tuners and Hardware-native Performance". In: *Proceedings of Machine Learning and Systems*. Vol. 4. 2022, pp. 204–216.

[365] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. "Empirical Evaluation of Rectified Activations in Convolutional Network". In: *CoRR* abs/1505.00853 (2015). arXiv: 1505.00853. URL: http://arxiv.org/abs/1505.00853.

[366] Lisong Xu, K. Harfoush, and Injong Rhee. "Binary increase congestion control (BIC) for fast long-distance networks". In: *IEEE INFOCOM 2004*. Vol. 4. 2004, 2514–2524 vol.4. DOI: 10.1109/INFCOM.2004.1354672.

[367] Tyler Yandrofski and Jingyuan Chen. "Towards Demystifying Cache Interference on NVIDIA GPUs". In: *The 14th Junior Researcher Workshop on Real-Time Computing (JRWRTC)*. 2021, p. 13.

[368] Gingfung Yeung, Damian Borowiec, Adrian Friday, Richard Harper, and Peter Garraghan. "Towards GPU Utilization Prediction for Cloud Deep Learning". In: *Proceedings of the 12th USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'20. USA: USENIX Association, 2020.

[369] Gingfung Yeung, Damian Borowiec, Renyu Yang, Adrian Friday, Richard Harper, et al. "Horus: An Interference-Aware Resource Manager for Deep Learning Systems". In: *Algorithms and Architectures for Parallel Processing*. Ed. by Meikang Qiu. Cham: Springer International Publishing, 2020, pp. 492–508. ISBN: 978-3-030-60239-0. DOI: 10.1007/978-3-030-60239-0_33.

[370] Asano YM., Rupprecht C., and Vedaldi A. "Self-labelling via simultaneous clustering and representation learning". In: *International Conference on Learning Representations*. 2020. URL: https://openreview.net/forum?id=Hyx-jyBFPr.

[371] Dong Yu, Balakrishnan Varadarajan, Li Deng, and Alex Acero. "Active Learning and Semi-Supervised Learning for Speech Recognition: A Unified Framework Using the Global Entropy Reduction Maximization Criterion". In: *Computer Speech & Language* 24.3 (July 2010), pp. 433–444. ISSN: 0885-2308. DOI: 10.1016/j.csl.2009.03.004. URL: https://doi.org/10.1016/j.csl.2009.03.004.

[372] Jinhui Yuan, Xinqi Li, Cheng Cheng, Juncheng Liu, Ran Guo, et al. "OneFlow: Redesign the Distributed Deep Learning Framework from Scratch". In: *CoRR* abs/2110.15032 (2021). arXiv: 2110.15032. URL: https://arxiv.org/abs/2110.15032.

[373] Xi Zeng, Tian Zhi, Zidong Du, Qi Guo, Ninghui Sun, et al. "ALT: Optimizing Tensor Compilation in Deep Learning Compilers with Active Learning". In: *2020*

IEEE 38th International Conference on Computer Design (ICCD). IEEE. 2020, pp. 623–630. DOI: 10.1109/ICCD50377.2020.00108.

[374] Tim Zerrell and Jeremy Bruestle. "Stripe: Tensor Compilation via the Nested Polyhedral Model". In: *CoRR* abs/1903.06498 (2019). arXiv: 1903.06498. URL: http://arxiv.org/abs/1903.06498.

[375] Jingzhao Zhang, Tianxing He, Suvrit Sra, and Ali Jadbabaie. "Why Gradient Clipping Accelerates Training: A Theoretical Justification for Adaptivity". In: *International Conference on Learning Representations*. 2020. URL: https://openreview.net/forum?id=BJgnXpVYwS.

[376] Li Lyna Zhang, Shihao Han, Jianyu Wei, Ningxin Zheng, Ting Cao, et al. "Nn-Meter: Towards Accurate Latency Prediction of Deep-Learning Model Inference on Diverse Edge Devices". In: *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*. MobiSys '21. Virtual Event, Wisconsin: ACM, 2021, pp. 81–93. ISBN: 9781450384438. DOI: 10.1145/3458864.3467882. URL: https://doi.org/10.1145/3458864.3467882.

[377] Minjia Zhang, Menghao Li, Chi Wang, and Mingqin Li. "DynaTune: Dynamic Tensor Program Optimization in Deep Neural Network Compilation". In: *International Conference on Learning Representations*. 2021. URL: https://openreview.net/forum?id=GTGb3M_KcUl.

[378] Shanjun Zhang, Mingzhen Li, Hailong Yang, Yi Liu, Zhongzhi Luan, et al. "FamilySeer: Towards Optimized Tensor Codes by Exploiting Computation Subgraph Similarity". In: *CoRR* abs/2201.00194 (2022). arXiv: 2201.00194. URL: https://arxiv.org/abs/2201.00194.

[379] Yao Zhang and John D Owens. "A quantitative performance analysis model for GPU architectures". In: *2011 IEEE 17th international symposium on high*

performance computer architecture. IEEE. 2011, pp. 382–393. DOI: 10.1109/HPCA.2011.5749745.

[380] Zhenhua Zhang, Qing He, Jing Gao, and Ming Ni. "A deep learning approach for detecting traffic accidents from social media data". In: *Transportation Research Part C: Emerging Technologies* 86 (2018), pp. 580–596. ISSN: 0968-090X. DOI: https://doi.org/10.1016/j.trc.2017.11.027. URL: https://www.sciencedirect.com/science/article/pii/S0968090X1730356X.

[381] Zhilu Zhang and Mert R. Sabuncu. "Generalized Cross Entropy Loss for Training Deep Neural Networks with Noisy Labels". In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. NIPS'18. Montréal, Canada: Curran Associates Inc., 2018, pp. 8792–8802.

[382] Jie Zhao, Xiong Gao, Ruijie Xia, Zhaochuang Zhang, Deshi Chen, et al. "Apollo: Automatic Partition-based Operator Fusion through Layer by Layer Optimization". In: *Proceedings of Machine Learning and Systems* 4 (2022), pp. 1–19.

[383] Zhihe Zhao, Xian Shuai, Neiwen Ling, Nan Guan, Zhenyu Yan, et al. "Moses: Exploiting Cross-Device Transferable Features for on-Device Tensor Program Optimization". In: *Proceedings of the 24th International Workshop on Mobile Computing Systems and Applications*. HotMobile '23. Newport Beach, California: ACM, 2023, pp. 22–28. DOI: 10.1145/3572864.3580330. URL: https://doi.org/10.1145/3572864.3580330.

[384] Bojian Zheng, Ziheng Jiang, Cody Hao Yu, Haichen Shen, Josh Fromm, et al. "DietCode: Automatic optimization for dynamic tensor program". In: *Proceedings of Machine Learning and Systems (MLSys) 2022*. Vol. 4. 2022, pp. 848–863. URL: https://www.amazon.science/publications/dietcode-automatic-optimization-for-dynamic-tensor-program.

[385] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, et al. "Ansor: Generating High-Performance Tensor Programs for Deep Learning". In: *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. OSDI'20. USA: USENIX Association, 2020. ISBN: 978-1-939133-19-9.

[386] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. "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: ACM, 2020, pp. 859–873. ISBN: 9781450371025. DOI: 10.1145/3373376.3378508. URL: https://doi.org/10.1145/3373376.3378508.

[387] Zhen Zheng, Xuanda Yang, Pengzhan Zhao, Guoping Long, Kai Zhu, et al. "AStitch: Enabling a New Multi-Dimensional Optimization Space for Memory-Intensive ML Training and Inference on Modern SIMT Architectures". In: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '22. Lausanne, Switzerland: ACM, 2022, pp. 359–373. ISBN: 9781450392051. DOI: 10.1145/3503222.3507723. URL: https://doi.org/10.1145/3503222.3507723.

[388] Zhen Zheng, Pengzhan Zhao, Guoping Long, Feiwen Zhu, Kai Zhu, et al. "FusionStitching: Boosting Memory Intensive Computations for Deep Learning Workloads". In: *CoRR* abs/2009.10924 (2020). arXiv: 2009.10924. URL: https://arxiv.org/abs/2009.10924.

[389] Shusen Zhou, Qingcai Chen, and Xiaolong Wang. "Discriminative Deep Belief Networks for image classification". In: *2010 IEEE International Conference on Image Processing*. 2010, pp. 1561–1564. DOI: 10.1109/ICIP.2010.5649922.

[390] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, et al. "ROLLER: Fast and Efficient Tensor Compilation for Deep Learning". In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, July 2022, pp. 233–248. ISBN: 978-1-939133-28-1. URL: https://www.usenix.org/conference/osdi22/presentation/zhu.

[391] Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, et al. "A Comprehensive Survey on Transfer Learning". In: *Proceedings of the IEEE* 109.1 (2021), pp. 43–76. DOI: 10.1109/JPROC.2020.3004555.

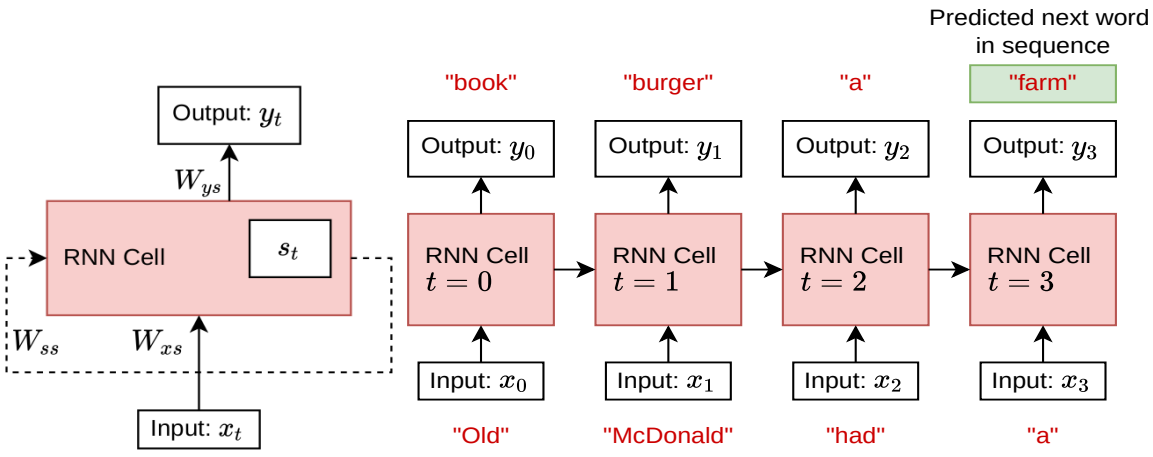
[392] Barret Zoph and Quoc Le. "Neural Architecture Search with Reinforcement Learning". In: *International Conference on Learning Representations*. 2017. URL: https://openreview.net/forum?id=r1Ue8Hcxg.

This page is left intentionally blank

Appendix A

Recurrent Cell Layers and LSTMs

A.1 Recurrent Cell Layers



(a) RNN cell *(b)* RNN cell across time

Figure A.1: RNN cell depicted as a recurrent block and unrolled across time steps

RNNs are built using *cells*, where each cell is recurrent (feeds back on itself). Figure A.1a depicts operation of a single RNN cell. Each cell accepts an input vector x_t where t designates the current time step or element in the sequence. Before producing output y_t, the RNN cell updates its internal state s by performing the following:

$$s_t = \text{nonlinear}_{func}(W_{ss}^T s_{t-1} + W_{xs}^T x_t) \tag{A.1}$$

where nonlinear_{func} is a non-linear function of choice such as Sigmoid, Hyperbolic Tangent or ReLU, W_{ss}^T is a transposed state weight matrix, s_{t-1} is the previous state of the cell, W_{xs}^T is a transposed input weight matrix and x_t is the current input vector element. Once updated, the state s is used to produce output y_t as follows:

$$y_t = W_{ys}^T s_t \tag{A.2}$$

where W_{ys}^T is a transposed output weight matrix and s_t is the current cell state.

During inference, RNNs process elements in the input sequence as described above, until the last element is reached. Figure A.1b depicts how an RNN cell (shown unrolled in time) would process a set of words in a sequence to predict the next word. For clarity, the diagram does not depict separate RNN cells but rather a single RNN cell at different points in time, sharing the same set of weights and periodically updating its internal state s. As with FFNNs, during training RNN utilise a loss function (Section 2.1.1) to determine how well the network is performing and to use the loss values to progressively update their weights. Since the update of weights involves computation of derivatives with respect to parameters (described in the following section) tracing back to the initial state, it involves many factors of weights and gradient computations, causing a phenomena such as exploding gradients [252] or vanishing gradients [164]. This occurs when values involved in the computation of gradient are greater than one or very small, causing calculated gradients to become very large or very small due to repeated calculations over the same values, rendering the training sub-optimal or in worst-case scenarios useless. These issues can be prevented using techniques such as gradient clipping [375], initialising weights to identity matrices [175] or an appropriate choice of an activation function, however have been solved altogether using a modified RNN architecture - the Long-short Term Memory (LSTM) networks [117].

A.2 Long Short Term Memory Layers

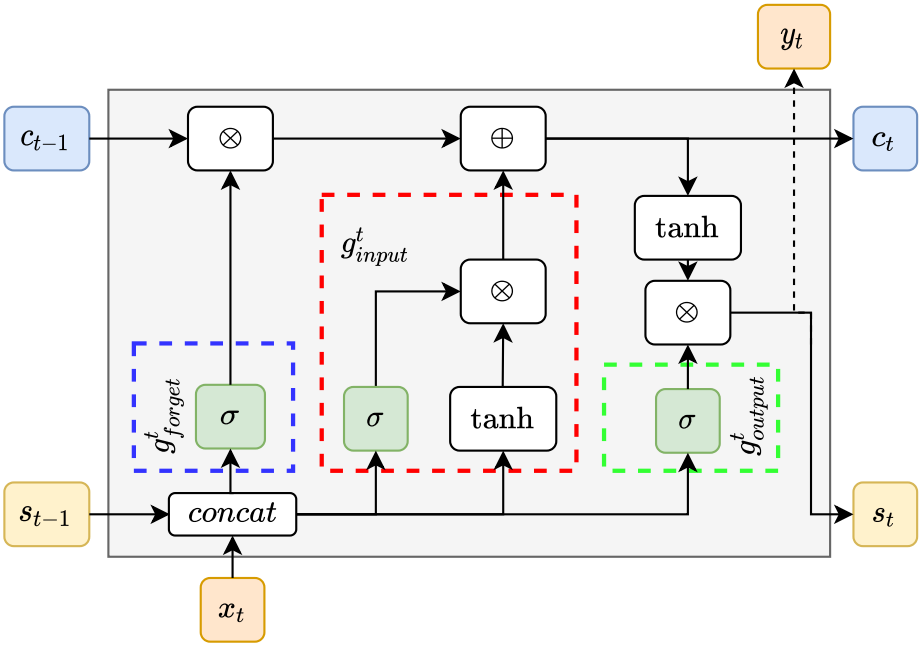


Figure A.2: LSTM gated cell

As depicted in Figure A.2, LSTM recurrent cells are composed of the internal cell state s, an *input gate* g^t_{input}, *output gate* g^t_{output} and a *forget gate* g^t_{forget}, where t is the current time step. Gated information flow enables LSTM networks to remember information over arbitrarily large spans of time, depending on information importance. On top of the internal state akin to classic RNNs, LSTM include a *cell state* c for storing information. Whilst in theory, generic RNN networks are also capable of remembering information over arbitrarily large sequences of input, the longer those spans are, the higher is the chance of exploding or vanishing gradients occurring.

LSTMs improve upon this via their architecture that enables gradient calculation during training to occur along the outputs of the gates. On every time step, the LSTM cell updates the state of each of the three gates. First, the state of the forget gate is processed for time step t to determine what information from prior internal state is kept

and what is forgotten, as follows:

$$g^t_{forget} = \sigma(W^g_{forget} x_t + S^s_{forget} s_{t-1} + \beta_{forget}) \tag{A.3}$$

where σ is the Sigmoid function, W^g_{forget} is the weight matrix of the forget gate, S^s_{forget} is the weight matrix of the cell's internal state w.r.t. the forget gate and β_{forget} is the bias of the gate. Next the input and output gates' states are processed, as follows:

$$g^t_{input} = \sigma(W^g_{input} x_t + S^s_{input} s_{t-1} + \beta_{input}) \tag{A.4}$$

$$g^t_{output} = \sigma(W^g_{output} x_t + S^s_{output} s_{t-1} + \beta_{output}) \tag{A.5}$$

New cell state C_t is then determined as follows:

$$C_t = \tanh(W^l_{hyp} x_t + S^l_{hyp} s_{t-1} + \beta_{hyp}) \odot g^t_{input} + g^t_{forget} \odot C_{t-1} \tag{A.6}$$

where C_{t-1} is the previous cell state and $W^l_{hyp}\&S^l_{hyp}\&\beta_{hyp}$ represent the weight matrices and bias of the hyperbolic tangent layer resultant from the input gate.

Finally, the new internal state (also the cell's output) is produced by:

$$s_t = g^t_{output} \odot \tanh(C_t) \tag{A.7}$$

Appendix B

Spatial Dataflow Processors (SDP)

In contrast with Temporal (Instruction) Processors (TIPs), Spatial (Dataflow) Processors (SDPs) enable communication between their Arithmetic Logic Units to provide means for forming computation chaining [247, 245]. The Arithmetic Logic Units found in SDPs are often less versatile compared to TIP Arithmetic Logic Units and focus on a specific task such as a the Multiply Accumulate (MAC) operation often used in tensor computation and DL. SDP Arithmetic Logic Units often include a limited amount of scratchpad memory and a simple control unit, together forming a Processing Element (PE). As PEs in SDPs are chained and do not rely on instructions to guide computation, they are suitable for high-throughput, embarrassingly-parallel, however, simpler computations such as MACs, found in DL workloads. This results in higher energy-efficiency as a higher portion of silicon is dedicated for the task at hand compared to more complex and feature-redundant TIPs. However, SDPs provide a decreased programming flexibility and increased engineering effort required to leverage them effectively. Prime examples of SDPs are Application Specific Integrated Circuits (ASICs) and Field-Programmable Gate Arrays (FPGAs).

B.1 Application-specific Integrated Circuits (ASIC)

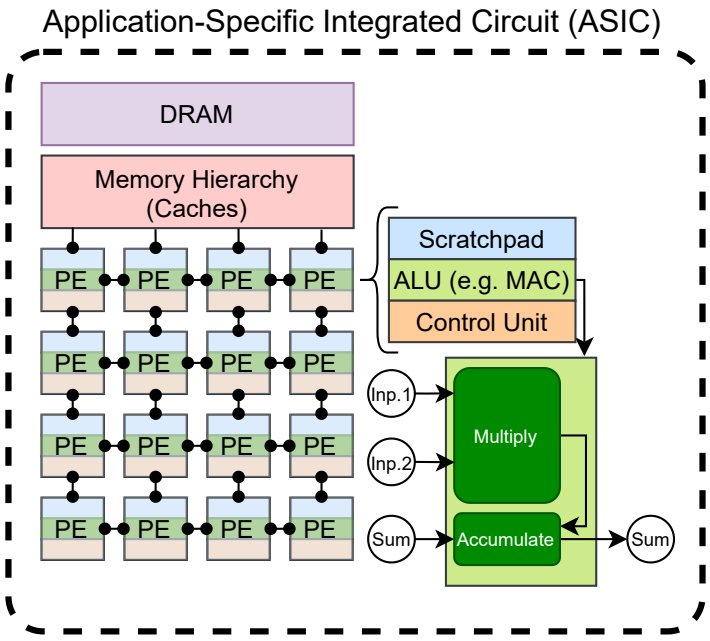


Figure B.1: An example of an Application Specific Integrated Circuit (ASIC) architecture based around systolic array configuration - a common design for DNN accelerator chips such as the Google Tensor Processing Unit (TPU). In the Figure, PE stands for Processing Element, MAC stands for Multiply-Accumulate and ALU stands for Arithmetic Logic Unit.

In the domain of DL, ASICs are SDPs used to perform compute-intensive MAC computations. ASICs are inflexible to program for and rely on predictable code that must be mapped from software implementation to hardware by experienced engineers, to execute efficiently and achieve high device utilisation. Whilst each ASIC design is different, they often employ arrays of PEs, each linked to each other and capable of performing simple arithmetic such as MACs. The core advantage of ASICs is their ability to perform massively parallel computation in far fewer clock cycles compared to a complex CPU core, however, can only do so for simple DL tensor operators or parts of operators (matrix multiplication) and require other processors to handle more complex tasks and data movement. One of the first commercially viable DL

ASICs was Google's Tensor Processing Unit (TPU) designed for inference and training [150, 100]. Other Cloud-based ASICs include Baidu Kunlun [242] or Amazon's AWS Inferentia chips [294]. Oftentimes marketed as Neural Processing Units (NPUs), Vision Processing Units (VPUs) or Image/Intelligence Processing Units (IPUs), ASICs have also started to become prevalent at the Edge with Intel Movidius Myriad-X architecture [50], Apple's Neural Engine [20] or mobile Digital Signal Processor (DSP) ASICs such as the Qualcomm Hexagon [259]. Figure B.1 depicts an example of an ASIC architecture based on the systolic array configuration. Systolic array ASICs are monolithic networks of fixed, identical processing elements, where each such processing element is independent in terms of computation. Typically, simple operations such as addition or multiplication are performed by these processing elements, which store and then pass computation results downstream. This makes systolic array ASICs especially suitable for massively parallel DNN computation.

B.2 Field-programmable Gate Arrays (FPGA)

FPGAs contain a large number of re-programmable execution elements, also known as Configurable Logic Blocks (CLBs), where each contains a Lookup Table (LUT) and a flip-flop gate and where each CLB can be adapted to perform a desired Boolean logic function or set of functions. Alongside the re-configurable portion of silicon, modern FPGAs often include Digital Signal Processor (DSP) tiles that provide higher-level functionality such as adders or multipliers without having to dedicate a portion of the re-configurable silicon to implement them. Moreover, FPGAs are sometimes coupled with DRAM memory installed on the same accelerator board to reduce memory access latency. FPGAs enable engineers to prototype other processors such as ASICs and design hardware-based implementations for specific DL operations such as Convolution or matrix multiplication, achieving much higher computation throughput compared to

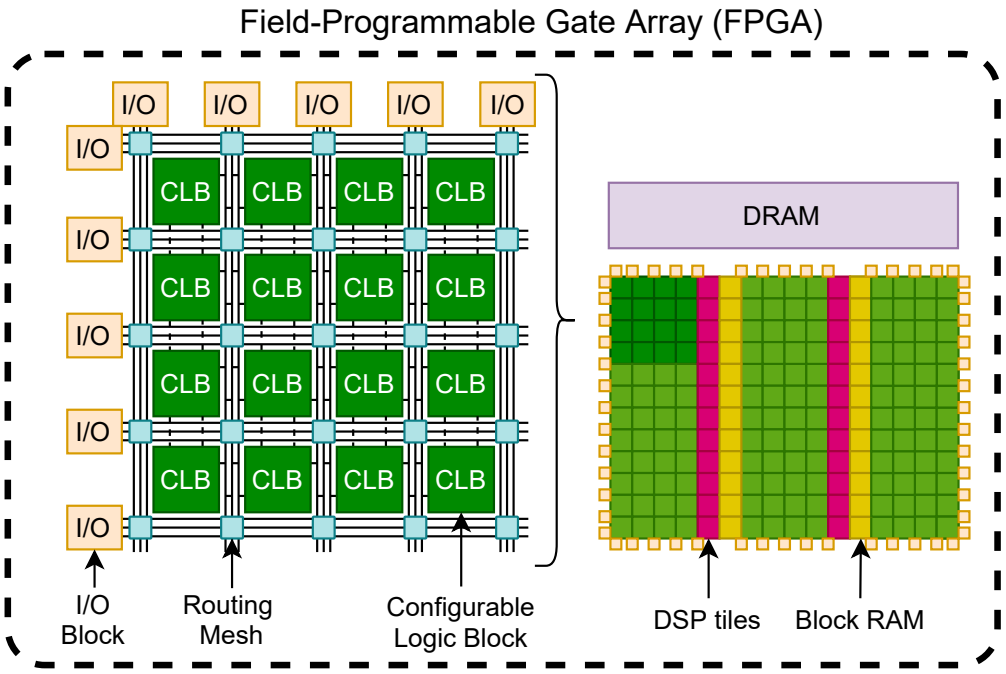


Figure B.2: An example of an Field-Programmable Gate Array (FPGA) architecture. In the Figure, CLB stands for Configurable Logic Block and DSP stands for Digital Signal Processor.

temporal processors. However, developing FPGA-based solutions for DL is challenging as it requires expertise in low-level hardware design and in-depth understanding of the specifics of DL computation. Despite that, some FPGA adoption can be observed for large-scale DL processing with Amazon's AWS EC2 F1 FPGA instances [293] containing Xillinx (now AMD) FPGAs or Microsoft Azure ML FPGA instances containing Altera (now Intel) FPGAs [56]. Furthermore, Nvidia released an open-source set of designs for a DL accelerator - Nvidia NVDLA [233], compatible with FPGAs, enabling research and development of novel DL accelerators.

Figure B.2 depicts an example of an FPGA architecture with the commonly occurring components such as CLBs, DSPs, the routing mesh and input/output blocks for communications. The Figure does not depict any specific FPGA device but rather a conceptual representation of an FPGA architecture that could be observed in platforms such as AWS EC2 F1.

Appendix C

Details of standalone tensor operators used in experimentation

Table C.1: Tensor Operator Characteristics

ID	Type	FLOPs	Input Type	Batch size	In Channels	Filters	Padding	Strides	Input Size	Filter Size	Dilation	Weight Inp Dim	Weight Out Dim	Input depth	Filter depth	Out-pad	Groups
0	MatMul	180	fp32	1					3,5			3,6					
1	MatMul	245760	fp32	5					32,16			32,48					
2	MatMul	6.71E+08	fp32	5					32,1024			32,2048					
3	TConv1D-NCW	221448	fp32	1	3	32	0	1	224	5						0	
4	TConv1D-NCW	134	fp32	1	1	1	2,3	1	10	5						0	
5	Conv1D-NCW	36	fp32	1	1	1	VALID	1	8	3	1						
6	Conv1D-NCW	16384	fp32	1	16	16	SAME	1	32	1	1						
7	Conv1D-NCW	13500	fp32	1	5	18	VALID	1	27	3	1						
8	Conv1D-NWC	36	fp32	1	1	1	VALID	1	8	3	1						
9	Conv1D-NCW	16384	fp32	1	16	16	SAME	1	32	1	1						
10	Conv2D-HWCN	1.21E+09	fp32	1	256	256	SAME	1	32	3	1						
11	Conv2D-HWCN	2.36E+08	fp32	4	128	256	VALID	2	16	5	1						
12	Conv2D-HWCN	1.21E+09	fp32	1	256	256	SAME	1	32	3	2						
13	Conv2D-NCHW	2.15E+08	fp32	1	64	64	1	1	56	3	2						
14	Conv2D-NCHW	576	fp32	2	2	2	2	2	2	2	1						
15	Conv2D-NCHW	4.64E+08	fp32	1	2048	126	1	1	10	3	1						
16	Conv2D-NCHW	23658496	fp32	1	512	64	SAME	1	19	1	1						
17	Conv2D-NHWC	2.1E+08	fp32	4	128	128	SAME	2	16	5	1						
18	Conv2D-NHWC	1.13E+09	fp32	1	256	256	1,1,2,2	1	32	3	2						
19	TConv3D-NCDHW	27648	fp32	1	3	1	0,0,0,0,0,0	1,1,1	24,24	1,1				24	1	0,0,0	
20	Conv3D-NCDHW	4.53E+08	fp32	1	16	16	SAME	1	32	3	1			32	3		
21	Conv3D-NCDHW	2.06E+08	fp32	1	1	32	SAME	1,2,2	256,256	3,3	1			20	1		
22	Conv3D-NCDHW	5.24E+08	fp32	1	4	8	0,2,2	1,2,2	256,256	5,5	1			20	1		
23	Conv3D-NDHWC	52428800	fp32	1	32	5	0	1	32	1	1			32	1		
24	Conv3D-NDHWC	57499200	fp32	1	32	5	0,0,0,1,1,1	1	32	1	1			32	1		
25	Corr-NCHW	70308	fp32				4	1,1	1,3,10,10			1					
26	Dense	2048000	fp32	1					1024			1024	1000				
27	Dense-INT8	1024000	int8	2					256			256	1000				
28	Dense-INT8	1.84E+08	int8	9					2048			2048	5000				
29	DEPTH-Conv2D-NCHW	10136448	fp32	1	32		SAME	1	112	3	1						
30	DEPTH-Conv2D-NCHW	77237888	fp32	1	728		SAME	1	64	3	2						
31	GRP-Conv2D-NCHW	28901376	fp32	1	1024	1024	1	1	7	3	1						32
32	Conv2D-NCHWc-INT8	2.31E+08	int8	1	64	64	1	1	56	3	1						
33	Conv2D-NCHWc-INT8	18432	int8	4	4	4	4	4	4	4	1						
34	Conv2D-NCHWc-INT8	11829248	int8	1	512	32	SAME	1	19	1	1						
35	Conv2D-NCHWc-INT8	1492992	int8	1	32	32	1,2,2,1	1	8	3	1						

Appendix D

DL model architecture details

D.1 AlexNet

Table D.1: AlexNet [168] architecture details. Input / Output shape in NHWC layout

#	Input	Output	Type	Pad	Stride	Kernel*CI*CO
L1	1x227x227x3	1x55x55x96	Conv	0	4	11x11x3x96
	1x55x55x96	1x27x27x96	MaxPool	0	2	3x3
L2	1x27x27x96	1x27x27x256	Conv	2	1	5x5x96x256
	1x27x27x256	1x13x13x256	MaxPool	0	2	3x3
L4	1x13x13x256	1x13x13x384	Conv	1	1	3x3x256x384
L5	1x13x13x384	1x13x13x384	Conv	1	1	3x3x384x384
L6	1x13x13x384	1x13x13x256	Conv	1	1	3x3x384x256
	1x13x13x256	1x6x6x256	MaxPool	0	2	3x3
	1x6x6x256	1x9216	Flatten			
L7	1x9216	1x4096	Dense			
L8	1x4096	1x4096	Dense			
L9	1x4096	1x1000	Dense			

D.2 SqueezeNet

To decrease the number of parameters, whilst maintaining acceptable accuracy, SqueezeNet [130] includes Fire blocks. Each Fire block consists of two phases (sub-layers): Squeeze and Expand. The Squeeze phase consists of a Convolution layer with some number of 1x1 kernels (specified by "(S) 1x1" in the table), followed by ReLU activation function. The Squeeze phase directly feeds into the Expand phase, which consists of a Convolution layer with some number of 1x1 kernels (specified by "(E) 1x1" in the table) and some number of 3x3 kernels (specified by "(E) 3x3" in the table), followed by ReLU activation function.

Table D.2: SqueezeNet [130] architecture details. Input / Output shape in NHWC layout. "(S) 1x1" is the number of 1x1 kernels within the Squeeze phase of each Fire block. "(E) 1x1" and "(E) 3x3" correspond to the number of 1x1 and 3x3 kernels within the Expand phase of each Fire block.

#	Input	Output	Type	Stride	Kernel*CI*CO	(S) 1x1	(E) 1x1	(E) 3x3
L1	1x224x224x3	1x111x111x96	Conv	2	7x7x3x96			
	1x111x111x96	1x55x55x96	MaxPool	2	3x3			
L2	1x55x55x96	1x55x55x128	FireBlock			16	64	64
L3	1x55x55x128	1x55x55x128	FireBlock			16	64	64
L4	1x55x55x128	1x55x55x256	FireBlock			32	128	128
	1x55x55x256	1x27x27x256	MaxPool	2	3x3			
L5	1x27x27x256	1x27x27x256	FireBlock			32	128	128
L6	1x27x27x256	1x27x27x384	FireBlock			48	192	192
L7	1x27x27x384	1x27x27x384	FireBlock			48	192	192
L8	1x27x27x384	1x27x27x512	FireBlock			64	256	256
	1x27x27x512	1x13x13x512	MaxPool	2	3x3			
L9	1x13x13x512	1x13x13x512	FireBlock			64	256	256
L10	1x13x13x512	1x13x13x1000	Conv	1	1x1x512x1000			
	1x13x13x1000	1x1x1x1000	AvgPool	1	13x13			
	1x1x1x1000	1x1000	Flatten					

D.3 MobileNetV1

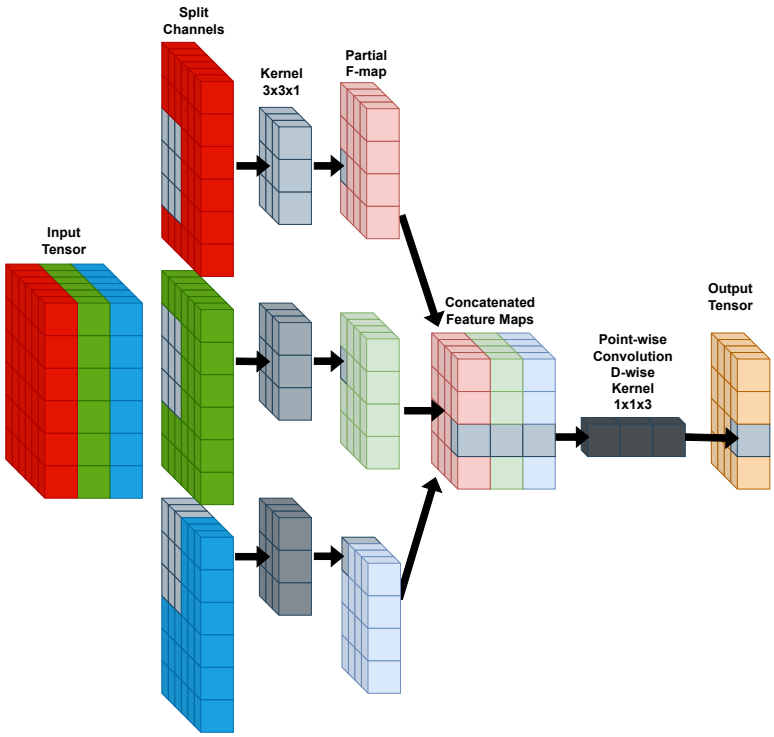


Figure D.1: The effect of applying 3x3x1 (x3) Depth-wise Separable Convolution to a 6x6x3 input tensor (stride = 1, padding = 0)

To maintain accuracy whilst reducing the number of required parameters within the model, MobileNetV1 [121] substitutes Convolution with Depth-wise Separable Convolution blocks [301]. Each Depth-wise Separable Convolution block consists of a single Depth-wise Convolution layer (apply single kernel channel to single input channel - as opposed to convolving directly in depth dimension) followed by an application of a 1x1 filter to the depth dimension of the resultant output, with BN and ReLU activation function applied after the Depth-wise and Point-wise (1x1) convolution. Depth-wise Separable Convolution block is shown in Figure D.1

Table D.3: MobileNetV1 [121] architecture details. Input / Output shape in NHWC layout

#	**Input**	**Output**	**Type**	**Stride**	**Kern*CI*CO**
L1	1x224x224x3	1x112x112x32	Conv	2	3x3x3x32
L2	1x112x112x32	1x112x112x32	Conv D-wise	1	3x3x32x1
L3	1x112x112x32	1x112x112x64	Conv P-wise	1	1x1x32x64
L4	1x112x112x64	1x56x56x64	Conv D-wise	2	3x3x64x1
L5	1x56x56x64	1x56x56x128	Conv P-wise	1	1x1x64x128
L6	1x56x56x128	1x56x56x128	Conv D-wise	1	3x3x128x1
L7	1x56x56x128	1x56x56x128	Conv P-wise	1	1x1x128x128
L8	1x56x56x128	1x28x28x128	Conv D-wise	2	3x3x128x1
L9	1x28x28x128	1x28x28x256	Conv P-wise	1	1x1x128x256
L10	1x28x28x256	1x28x28x256	Conv D-wise	1	3x3x256x1
L11	1x28x28x256	1x28x28x256	Conv P-wise	1	1x1x256x256
L12	1x28x28x256	1x14x14x256	Conv D-wise	2	3x3x256x1
L13	1x14x14x256	1x14x14x512	Conv P-wise	1	1x1x256x512
L14,16,18,20,22	1x14x14x512	1x14x14x512	Conv D-wise	1	3x3x512x1
L15,17,19,21,23	1x14x14x512	1x14x14x512	Conv P-wise	1	1x1x512x512
L24	1x14x14x512	1x7x7x512	Conv D-wise	2	3x3x512x1
L25	1x7x7x512	1x7x7x1024	Conv P-wise	1	1x1x512x1024
L26	1x7x7x1024	1x7x7x1024	Conv D-wise	2	3x3x1024x1024
L27	1x7x7x1024	1x7x7x1024	Conv P-wise	1	1x1x1024x1024
	1x7x7x1024	1x1024	AvgPool	1	7x7
L28	1x1024	1x1000	Dense		1x1000

D.4 MobileNetV2

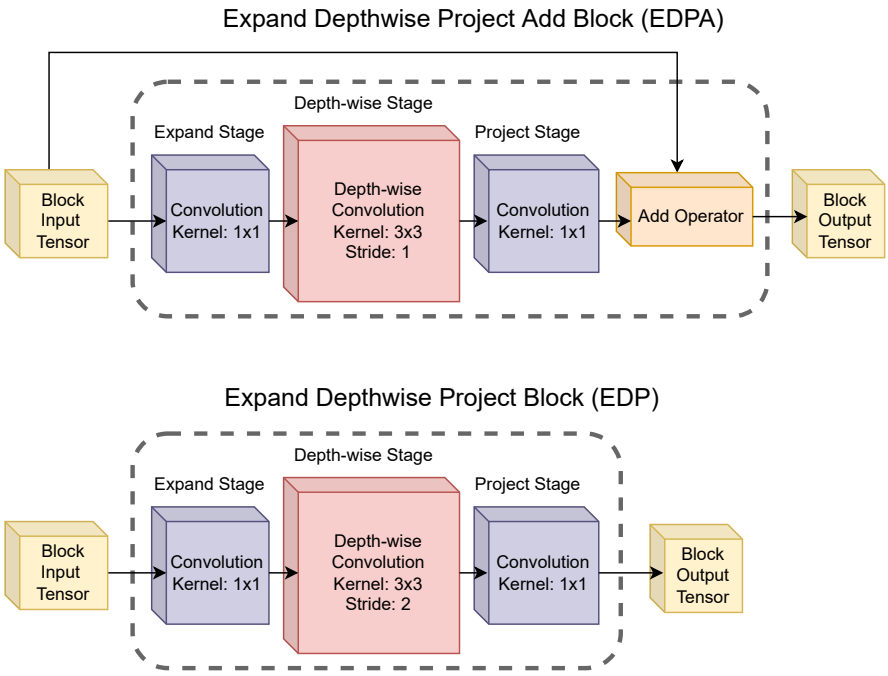


Figure D.2: The Expand Depth-wise Project (Add) Blocks in MobileNetV2.

The goal of the MobileNetV2 [288] architecture is to maintain the small computational footprint of MobileNetV1 [121], whilst improving upon the achieved inference performance (accuracy) and provide a backbone for other CV tasks such as object detection. The V2 architecture partially re-uses the Depth-wise and Point-wise Convolution layers found in V1, whilst expanding the architecture using inverted residual structures - Expand-Depth-wise-Project (EDP) and Expand-Depth-wise-Project-Add (EDPA) blocks, as depicted in Figure D.2.

Table D.4: MobileNetV2 [288] architecture details. Input / Output shape in NHWC layout.

#	Input	Output	Type	Stride	Kern*CI*CO
L1	1x224x224x3	1x112x112x32	Conv	2	3x3x3x32
L2	1x112x112x32	1x112x112x32	Conv D-wise	1	3x3x32x1
L3	1x112x112x32	1x112x112x16	Conv P-wise	1	1x1x32x16
Blk: 1 **L4-6**	1x112x112x16	1x56x56x24	EDP Blk	2	1x1x16x96 3x3x96x1 1x1x96x24
Blk: 2 **L7-9**	1x56x56x24	1x56x56x24	EDPA Blk	1	1x1x24x144 3x3x144x1 1x1x144x24
Blk: 3 **L10-12**	1x56x56x24	1x28x28x32	EDP Blk	2	1x1x24x144 3x3x144x1 1x1x144x32
Blk: 4-5 **L13-15** **L16-18**	1x28x28x32	1x28x28x32	EDPA Blk	1	1x1x32x192 3x3x192x1 1x1x192x32
Blk: 6 **L19-21**	1x28x28x32	1x14x14x64	EDP Blk	2	1x1x32x192 3x3x192x1 1x1x192x64
Blk: 7-9 **L22-24** **L25-27** **L28-30**	1x14x14x64	1x14x14x64	EDPA Blk	1	1x1x64x384 3x3x384x1 1x1x384x64
Blk: 10 **L31-33**	1x14x14x64	1x14x14x96	EDP Blk	2	1x1x64x384 3x3x384x1 1x1x384x96
Blk: 11-12 **L34-36** **L37-39**	1x14x14x96	1x14x14x96	EDPA Blk	1	1x1x96x576 3x3x576x1 1x1x576x96
Blk: 13 **L40-42**	1x14x14x96	1x7x7x160	EDP Blk	2	1x1x96x576 3x3x576x1 1x1x576x160
Blk: 14-15 **L43-45** **L46-48**	1x7x7x160	1x7x7x160	EDPA Blk	1	1x1x160x960 3x3x960x1 1x1x960x160
Blk: 16 **L49-51**	1x7x7x160	1x7x7x320	EDP Blk	2	1x1x160x960 3x3x960x1 1x1x960x320
L52	1x7x7x320	1x7x7x1280	Conv	1	1x1x320x1280
	1x7x7x1280	1x1x1x1280	AvgPool		
L53	1x1x1x1280	1x1000	Dense		

D.5 ConvNeXt

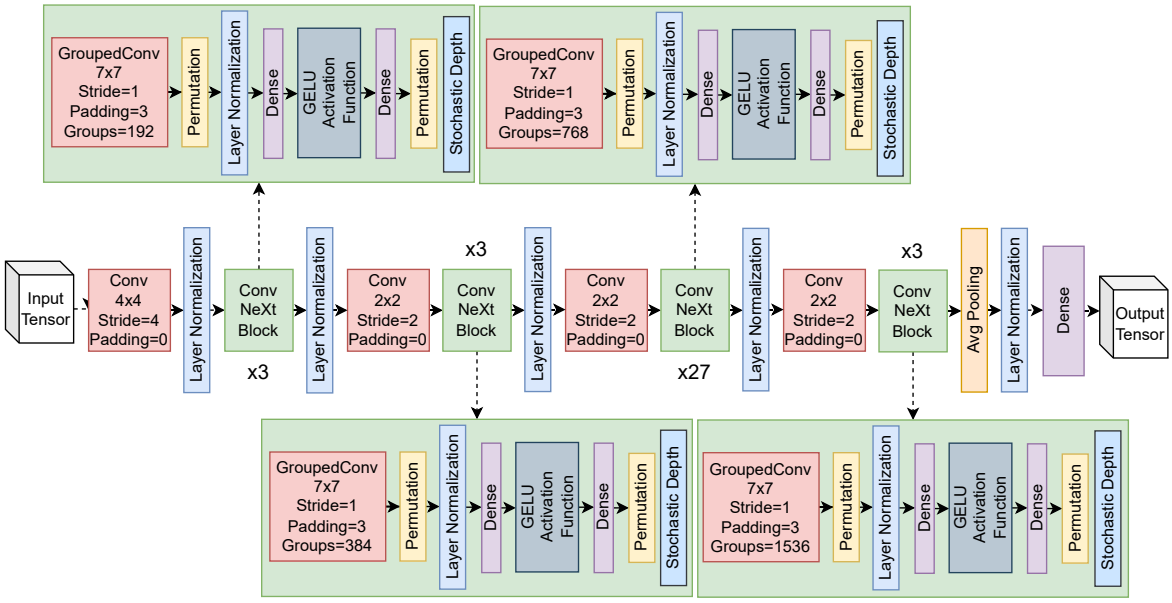


Figure D.3: The ConvNeXt Blocks, Layers and overall architecture.

The ConvNeXt [194] architecture is an attempt to test the limits of conventional CNN architecture in terms of accuracy and efficiency when performing CV tasks. Via a series of modernisations of the ResNet [113] architecture and adoption of training methods from Vision Transformer models [193], which enable it to outperform SOTA in image classification and other CV tasks, whilst remaining a pure CNN. Figure D.3 depicts the structure of the ConvNeXt-L model.

Table D.5: ConvNeXt [194] architecture details. Input / Output shape in NHWC layout. "N" denotes number of repeats of the block

#	N	Input	Output	Type	Kern*CI*CO	Stride	Pad	Groups
L1	1	1x3x224x224	1x192x56x56	Conv	4x4x3x192	4	0	
B1-3 **L2-10**	3	1x192x56x56 1x192x56x56 1x56x56x192 1x56x56x768 1x56x56x192	1x192x56x56 1x56x56x192 1x56x56x768 1x56x56x192 1x192x56x56	GrpConv Permute Dense Dense Permute	7x7x192x192	1	4	192
L11	1	1x192x56x56	1x384x28x28	Conv	2x2x192x384	2	0	
B4-6 **L12-20**	3	1x384x28x28 1x384x28x28 1x28x28x384 1x28x28x1536 1x28x28x384	1x384x28x28 1x28x28x384 1x28x28x1536 1x28x28x384 1x384x28x28	GrpConv Permute Dense Dense Permute	7x7x384x384	1	3	384
L21	1	1x384x28x28	1x768x14x14	Conv	2x2x384x768	2	0	
B7-33 **L22-102**	27	1x768x14x14 1x768x14x14 1x14x14x768 1x14x14x3072 1x14x14x768	1x768x14x14 1x14x14x768 1x14x14x3072 1x14x14x768 1x768x14x14	GrpConv Permute Dense Dense Permute	7x7x768x768	1	3	768
L103	1	1x768x14x14	1x1536x7x7	Conv	2x2x768x1536	2	0	
B34-36 **L104-112**	3	1x1536x7x7 1x1536x7x7 1x7x7x1536 1x7x7x6144 1x7x7x1536	1x1536x7x7 1x7x7x1536 1x7x7x6144 1x7x7x1536 1x1536x7x7	GrpConv Permute Dense Dense Permute	7x7x1536x1536	1	3	1536
L113	1 1 1	1x1536x7x7 1x1536x1x1 1x1536	1x1536x1x1 1x1536 1x1000	AvgPool Flatten Dense	7x7	1		

D.6 DenseNet-121

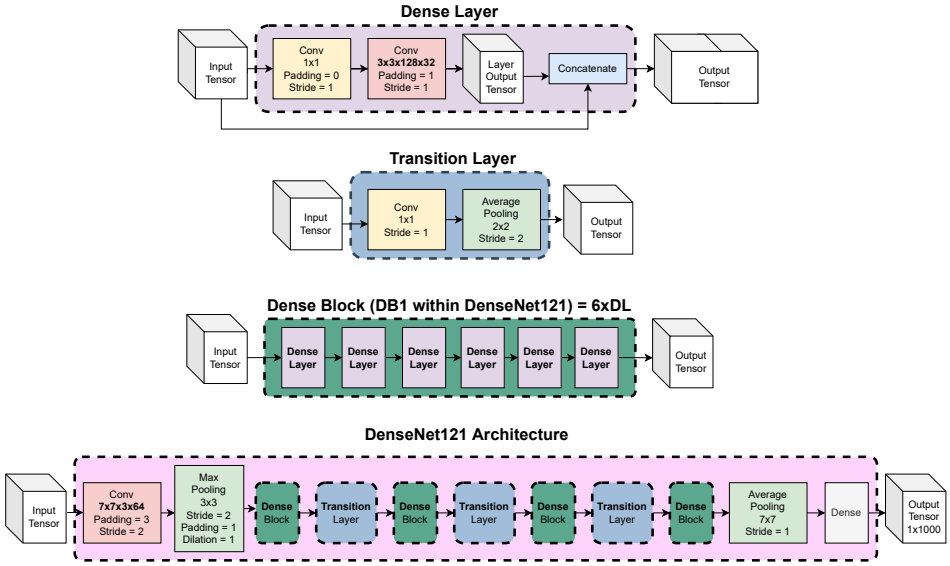


Figure D.4: Dense / Transition layers, Blocks and DNN architecture of DenseNet121

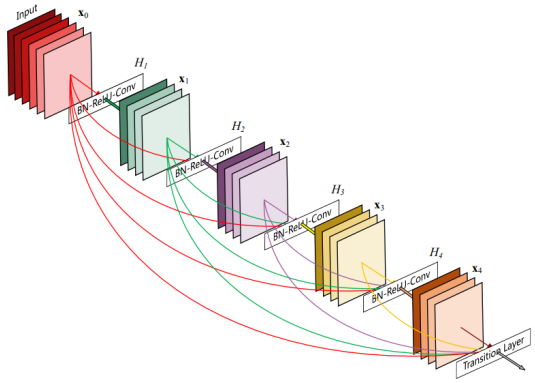


Figure D.5: Connectivity between Dense layers within DenseNet121. Figure obtained from [125].

The DenseNet [125] architecture has been designed to exploit an assumption where very deep DNNs can be effectively trained and avoid the diminishing gradient problem as long as the connections between layers close to the DNN input and output are

Table D.6: DenseNet121 [125] architecture details. Input / Output shape in NHWC layout. "*" denotes 3x3 Convolution, "L" denotes Layer numbers, "DB" denotes Dense Block, "dl" denotes Dense Layer

#	**Input**	**Output**	**Type**	**Kern*CI*CO dl1, dl2, dl3, dl4, dl5, ...**	**Padding = 1 for 3x3 Conv Unless otherwise stated**		**Stride**
L1	1x224x224x3	1x112x112x64	Conv	7x7x3x64 - Padding = 3			2
	1x114x114x64	1x56x56x64	MaxPool	3x3 (Dilation = 1)			2
DB1 L2-13	1x56x56x64	1x56x56x256	Conv+* Conv+*	1x1x64x128 1x1x160x128	1x1x96x128 1x1x192x128	1x1x128x128 1x1x224x128	1 1
TL1 L14	1x56x56x256	1x28x28x128	Conv MaxPool		1x1x256x128 2x2		1 2
DB2 L15-38	1x28x28x128	1x28x28x512	Conv+* Conv+* Conv+* Conv+*	1x1x128x128 1x1x224x128 1x1x320x128 1x1x416x128	1x1x160x128 1x1x256x128 1x1x352x128 1x1x448x128	1x1x192x128 1x1x288x128 1x1x384x128 1x1x480x128	1 1 1 1
TL1 L39	1x28x28x512	1x14x14x256	Conv MaxPool		1x1x512x256 2x2		1 2
DB3 L40-87	1x14x14x256	1x14x14x1024	Conv+* Conv+* Conv+* Conv+* Conv+* Conv+* Conv+* Conv+*	1x1x256x128 1x1x352x128 1x1x448x128 1x1x544x128 1x1x640x128 1x1x736x128 1x1x832x128 1x1x928x128	1x1x288x128 1x1x384x128 1x1x480x128 1x1x576x128 1x1x672x128 1x1x768x128 1x1x864x128 1x1x960x128	1x1x320x128 1x1x416x128 1x1x512x128 1x1x608x128 1x1x704x128 1x1x800x128 1x1x896x128 1x1x992x128	1 1 1 1 1 1 1 1
TL1 L88	1x14x14x1024	1x7x7x512	Conv MaxPool		1x1x1024x512 2x2		1 2
DB4 L89-120	1x7x7x512	1x7x7x1024	Conv* Conv* Conv* Conv* Conv* Conv*	1x1x512x128 1x1x608x128 1x1x704x128 1x1x800x128 1x1x896x128 1x1x992x128	1x1x544x128 1x1x640x128 1x1x736x128 1x1x832x128 1x1x928x128	1x1x576x128 1x1x672x128 1x1x768x128 1x1x864x128 1x1x960x128	1 1 1 1 1 1
L121	1x7x7x1024 1x1024	1x1024 1x1000	AvgPool Dense		7x7		1

shorter. In DenseNets, connections are made between every other layer as opposed to between neighbouring layers, reducing the required number of parameters. This effectively enables very deep networks to be trained, whilst reducing their computational burden.

The architecture leverages blocked-layered design as shown in Figure D.4. Consecutiveness of Dense layers within each block, enables concatenation of early feature maps (outputs of each layer) together as the block grows, effectively sharing parameters the deeper the network becomes, as shown in Figure D.5.

D.7 ResNet-18

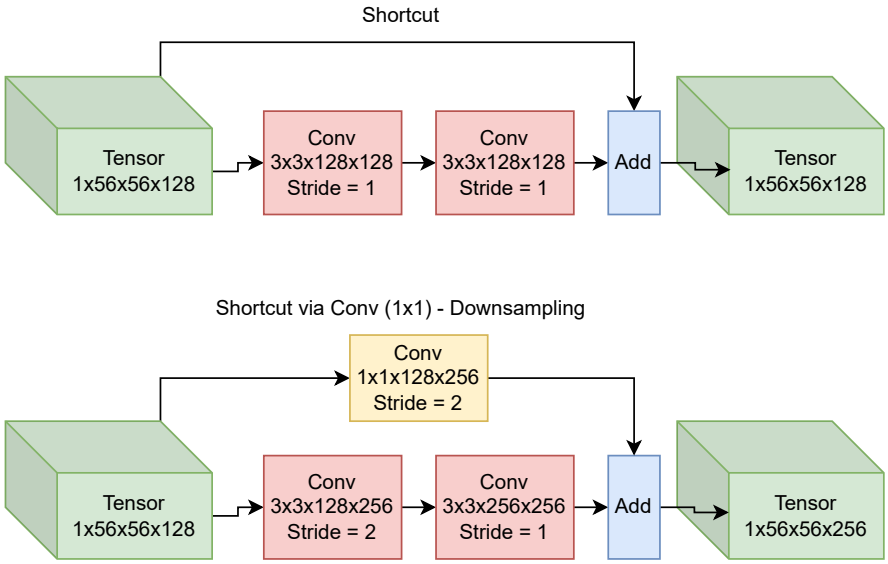


Figure D.6: Depiction of the Residual Blocks in ResNet architectures.

The ResNet [113] (Residual Nets) architecture was designed in response to the vanishing gradient problem which can occur in deep convolutional networks trained using backpropagation (see Section 2.2.1 for more details). Within the ResNet architecture, Residual Blocks of convolutions with shortcut residual mappings are used (see Figure D.6), such that during training, layers can be skipped when updating weights.

Table D.7: ResNet-18 [113] architecture details. Input / Output shape in NHWC layout. Each ResBlk is a Residual Block as depicted in Figure D.6

#	**Input**	**Output**	**Type**	**Stride**	**Kernel*CI*CO**
L1	1x224x224x3	1x112x112x64	Conv	2	7x7x3x64
	1x112x112x64	1x56x56x64	MaxPool	2	3x3
L2-3	1x56x56x64	1x56x56x64	ResBlk	1	3x3x64x64
				1	3x3x64x64
L4-5	1x56x56x64	1x56x56x64	ResBlk	1	3x3x64x64
				1	3x3x64x64
L6-7	1x56x56x64	1x56x56x128	ResBlk	2	3x3x64x128
				1	3x3x128x128
Conv Shortcut				2	**1x1x64x128**
L8-9	1x56x56x128	1x56x56x128	ResBlk	1	3x3x128x128
				1	3x3x128x128
L10-11	1x56x56x128	1x56x56x256	ResBlk	2	3x3x128x256
				1	3x3x256x256
Conv Shortcut				2	**1x1x128x256**
L12-13	1x56x56x256	1x56x56x256	ResBlk	1	3x3x256x256
				1	3x3x256x256
L14-15	1x56x56x256	1x56x56x512	ResBlk	2	3x3x256x512
				1	3x3x512x512
Conv Shortcut				2	**1x1x256x512**
L16-17	1x56x56x512	1x56x56x512	ResBlk	1	3x3x512x512
				1	3x3x512x512
	1x56x56x512	1x1x1x512	AvgPool	1	7x7
L18	1x1x1x512	1x1x1x1000	Dense		

D.8 VGG-16

Table D.8: VGG-16 [304] architecture details. Input / Output shape in NHWC layout

#	**Input**	**Output**	**Type**	**Stride**	**Kernel*CI*CO**
L1	1x224x224x3	1x224x224x64	Conv	1	3x3x3x64
L2	1x224x224x64	1x224x224x64	Conv	1	3x3x64x64
	1x224x224x64	1x112x112x64	MaxPool	2	2x2
L3	1x112x112x64	1x112x112x128	Conv	1	3x3x64x128
L4	1x112x112x128	1x112x112x128	Conv	1	3x3x128x128
	1x56x56x128	1x56x56x128	MaxPool	2	2x2
L5	1x56x56x128	1x56x56x256	Conv	1	3x3x128x256
L6-7	1x56x56x256	1x56x56x256	Conv	1	3x3x256x256
	1x56x56x256	1x28x28x256	MaxPool	2	2x2
L8	1x28x28x256	1x28x28x512	Conv	1	3x3x256x512
L9-10	1x28x28x512	1x28x28x512	Conv	1	3x3x512x512
	1x28x28x512	1x14x14x512	MaxPool	2	2x2
L11-13	1x14x14x512	1x14x14x512	Conv	1	3x3x512x512
	1x14x14x512	1x7x7x512	MaxPool	2	2x2
	1x7x7x512	1x25088	Flatten		
L14	1x25088	1x4096	Dense		
L15	1x4096	1x4096	Dense		
L16	1x4096	1x1000	Dense		

D.9 VGG-19

Table D.9: VGG-19 [304] architecture details. Input / Output shape in NHWC layout

#	**Input**	**Output**	**Type**	**Stride**	**Kernel*CI*CO**
L1	1x224x224x3	1x224x224x64	Conv	1	3x3x3x64
L2	1x224x224x64	1x224x224x64	Conv	1	3x3x64x64
	1x224x224x64	1x112x112x64	MaxPool	2	2x2
L3	1x112x112x64	1x112x112x128	Conv	1	3x3x64x128
L4	1x112x112x128	1x112x112x128	Conv	1	3x3x128x128
	1x56x56x128	1x56x56x128	MaxPool	2	2x2
L5	1x56x56x128	1x56x56x256	Conv	1	3x3x128x256
L6-8	1x56x56x256	1x56x56x256	Conv	1	3x3x256x256
	1x56x56x256	1x28x28x256	MaxPool	2	2x2
L9	1x28x28x256	1x28x28x512	Conv	1	3x3x256x512
L10-12	1x28x28x512	1x28x28x512	Conv	1	3x3x512x512
	1x28x28x512	1x14x14x512	MaxPool	2	2x2
L13-16	1x14x14x512	1x14x14x512	Conv	1	3x3x512x512
	1x14x14x512	1x7x7x512	MaxPool	2	2x2
	1x7x7x512	1x25088	Flatten		
L17	1x25088	1x4096	Dense		
L18	1x4096	1x4096	Dense		
L19	1x4096	1x1000	Dense		

Appendix E

TVM DLC high-level graph optimisation details

Table E.1: TVM's Relay IR transformation passes (high-level optimisations) and their corresponding designator levels

Lvl	Pass Name	Description
0	SimplifyExpr	Algebraically simplify expressions
0	SimpligyInference	Simplify DAG specifically for inference pass
0	InferType	Given a Relay expression, infer its resultant type
0	PartitionGraph	Partition model DAG into subgraphs
1	EtaExpand	Add abstraction over global variable or constructor
1	Inline	Inline appropriate functions within Relay module
1	RemoveUnusedFunctions	Remove unused global functions within Relay module
1	FuseOps	Perform operator fusion within Relay module
1	ToBBNF/ToANF/ToGNF	Convert Relay Expression to BBNF (Basic Blocks), ANF (Let-binding) or GNF (DAG)
1	PartialEvaluate	Evaluate static expressions at compile time
1	DeadCodeElimination	Eliminate unreachable expressions
2	FoldConstant(Expr)	Fold constant Relay expressions (replace with values)
2	SplitArgs	Split function with many arguments into multiple
2	DynamicToStatic	If possible, convert dynamic operators to static
3	EliminateCommonSubexpr	Common Relay subexpression elimination
3	For/Back-wardFoldScaleAxis	Forward / Backwards folding of axis into weights of Conv-2D or Dense operators
3	CanonicalizeOps	Standardise format of special operators
3	AlterOpLayout	Alternate layouts of consecutive operators
3	CanonicalizeCast	Standardise format of cast expressions
4	FastMath	Convert expensive non-linear activations to approximation functions, decreasing computation
4	CombineParallelConv2D	Join parallel Conv-2D ops into large Conv-2D
4	CombineParallelDense	Join parallel Dense ops into large Dense
4	CombineParallelBatchMatmul	Join multiple Batched GEMM ops into large GEMM

Appendix F

Auto-tuner hyperparameter details

Table F.1: Schedule parameters used to parameterise templates during auto-tuning.

Parameters	Description
tile_f, tile_y, tile_x	Tiling factors for the feature map and filter loop nests
tile_rc, tile_ry, tile_rx	Tiling factors for the reduction axis
auto_unroll_max_step	Unroll factor for automatic loop unrolling
unroll_explicit	Whether to explicitly unroll loop (annotate)

Table F.2: Configuration of the Grid-index search auto-tuner used during experimentation [40].

Configuration Option	Value	Description
NumAvgRuns	20	Number of candidate runs during 1 measurement repeat
NumMeasureRepeat	3	Number of measurement repeats per candidate
MinRepeatMs	0	Minimum duration of a single measurement Set to 0 to perform 20*3 runs

Table F.3: Configuration of the Random search auto-tuner used during experimentation [40].

Configuration Option	Value	Description
NumAvgRuns	20	Number of candidate runs during 1 measurement repeat
NumMeasureRepeat	3	Number of measurement repeats per candidate
MinRepeatMs	0	Minimum duration of a single measurement Set to 0 to perform 20*3 runs

Table F.4: Configuration of the Genetic search auto-tuner used during experimentation [40].

Configuration Option	Value	Description
PopSize	100	Population size during genetic search
NumEliteKeep	3	Number of elite candidates to keep in every iteration
MutationProb	0.1	Probability of mutation during each iteration
NumAvgRuns	20	Number of candidate runs during 1 measurement repeat
NumMeasureRepeat	3	Number of measurement repeats per candidate
MinRepeatMs	0	Minimum duration of a single measurement Set to 0 to perform 20*3 runs

Table F.5: Configuration of the AutoTVM auto-tuner used during experimentation [40].

Configuration Option	Value	Description
FeatureType	*itervar*	Features considered in cost model
LossType	*rank*	Cost model loss type
BatchSize	64	Planning batch size
NumMarkovChains	128	Number of Markov chains in parallel SA
MaxSteps	500	Maximum umber of steps in a single SA run
FilteringEps	0.05	Epsilon parameter used in candidate filtering
NumAvgRuns	20	Number of candidate runs during 1 measurement repeat
NumMeasureRepeat	3	Number of measurement repeats per candidate
MinRepeatMs	0	Minimum duration of a single measurement Set to 0 to perform 20*3 runs

Table F.6: Configuration of the Chameleon auto-tuner used during experimentation [8].

Configuration Option	Value	Description
FeatureType	itervar	Features considered in cost model
LossType	reg	Cost model loss type
BatchSize	64	Planning batch size
NumSteps	500	Number of steps in one reinforcement learning episode
NumIterations	16	Number of iterations of the optimiser process
NumEpisodes	128	Number of episodes performed during RL
PPO-AdamStepSize	1×10^{-3}	Step size of the Adam Optimiser used in the PPO strategy (search algorithm)
PPO-DiscountFactor	0.9	Discount factor parameter of the PPO strategy
PPO-GAE	0.99	GAE parameter of the PPO strategy
PPO-Epochs	3	Number of epochs used in the PPO strategy
PPO-Clip	0.3	Clipping parameter for the PPO strategy
PPO-Coeff	1.0	Value coefficient for the PPO strategy
PPO-Entropy	0.1	Entropy parameter for the PPO strategy
FilteringEps	0.05	Epsilon parameter used in the candidate filtering process ahead of proposal for measurement
NumAvgRuns	20	Number of candidate runs during 1 measurement repeat
NumMeasureRepeat	3	Number of measurement repeats per candidate
MinRepeatMs	0	Minimum duration of a single measurement Set to 0 to perform 20*3 runs

Table F.7: Configuration of the Ansor autoscheduler used during experimentation [385].

Configuration Option	**Value**	**Description**
CostModelType	XGBoost	Type of cost model used
EpsGreedy	0.05	Epsilon-greedy parameter for the cost model
RetSOROE	1	Search strategy parameter: Retry Search One Round On Empty
SmplInitMinPop	50	Sample size of the initial minimum population
SmplInitMeasRatio	0.2	Ratio of samples to be measured
EvoPop	2048	Population size for evolutionary search strategy
EvoIters	4	Number of iterations in evolutionary search strategy
EvoMutProb	0.85	Probability of mutation in evolutionary search strategy
CPUTiling	SSRSRS	Pattern specification for CPU-targeted multi-level tiling transformation
GPUTiling	SSSRRSRS	Pattern specification for GPU-targeted multi-level tiling transformation
MaxInmSplit	64	Maximum innermost split factor for the split schedule transformation
MaxVec	16	Maximum vectorisation factor for the vectorise schedule transformation
DisChgCompLoc	false	Whether to disable change in compute location
ModelAlpha	0.2	Alpha parameter when autoscheduling entire DNN
ModelBeta	2	Beta parameter when autoscheduling entire DNN
BckwrdWin	3	Backward window size parameter when autoscheduling entire DNN
NumWarmupSmpl	5	Number of warmup samples to initialise cost model
ObjectiveFunc	sum	Type of objective function when autoscheduling entire DNN
NumAvgRuns	20	Number of candidate runs during 1 measurement repeat
NumMeasureRepeat	3	Number of measurement repeats per candidate
MinRepeatMs	0	Minimum duration of a single measurement Set to 0 to perform 20*3 runs

Appendix G

Trimmer FC ANN Model Definition

```
class TrimmerFCNNModel(torch.nn.Module):
    def __init__(self, cat_feats, padding_indexes, embedding_dims=10, hidden_dim=32):
        super(TrimmerFCNNModel, self).__init__()
        self.cat_feats = cat_feats
        self.embeddings = {}
        total = 0
        for i, (feat, size) in enumerate(self.cat_feats.items()):
            embed = torch.nn.Embedding(size, embedding_dims, padding_idx=padding_indexes[feat])
            self.embeddings[feat] = embed
            total+=1
        self.linear = torch.nn.Linear((total * embedding_dims) + 1, hidden_dim)
        self.linear2 = torch.nn.Linear(1, 1)
        self.relu = torch.nn.ReLU6()
        self.final = torch.nn.Linear(hidden_dim, 1)

    def get_features(self, x):
        feats = []
        for k,v in x.items():
            embed = self.embeddings.get(k, None)
            if embed is None:
                continue
            feats.append(embed(v.long().detach()))
        feats = torch.cat(feats, axis=1)
        batch = x["flop"].size(0)
        flops = x["flop"].reshape(batch,1).float().detach()
        x2 = self.linear2(flops)
        x2 = self.relu(x2)
        return self.linear(torch.cat((feats, x2), axis=1))

    def forward(self, x, feats_only=False):
        x = self.get_features(x)
        if feats_only:
            return x
        return self.final(self.relu(x))
```

Listing G.1: Pytorch implementation of Trimmer's FC ANN filtering model.

Appendix H

Naïve Parallel Auto-tuning - Additional Results

H.1 Measurement Inaccuracy during NPM/MPS

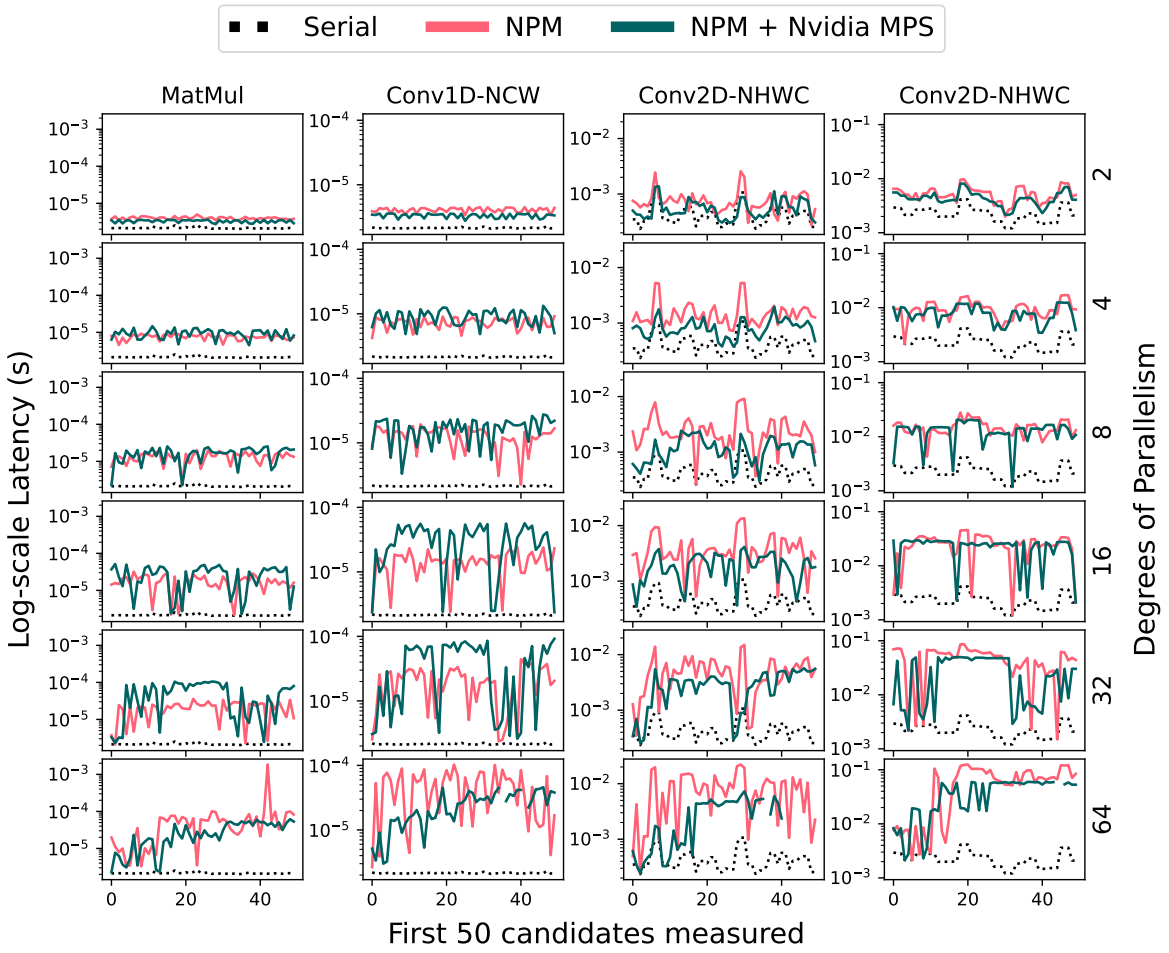


Figure H.1: {PART 1:} Measured latency of proposed candidate tensor programs when auto-tuning with a Grid-index auto-tuner. Reported 50 first candidates proposed by the auto-tuner towards the GPU within platform A, across several tensor operator auto-tuning tasks. Compared latency measurements as reported via serial, NPM and NPM + MPS measurement infrastructures.

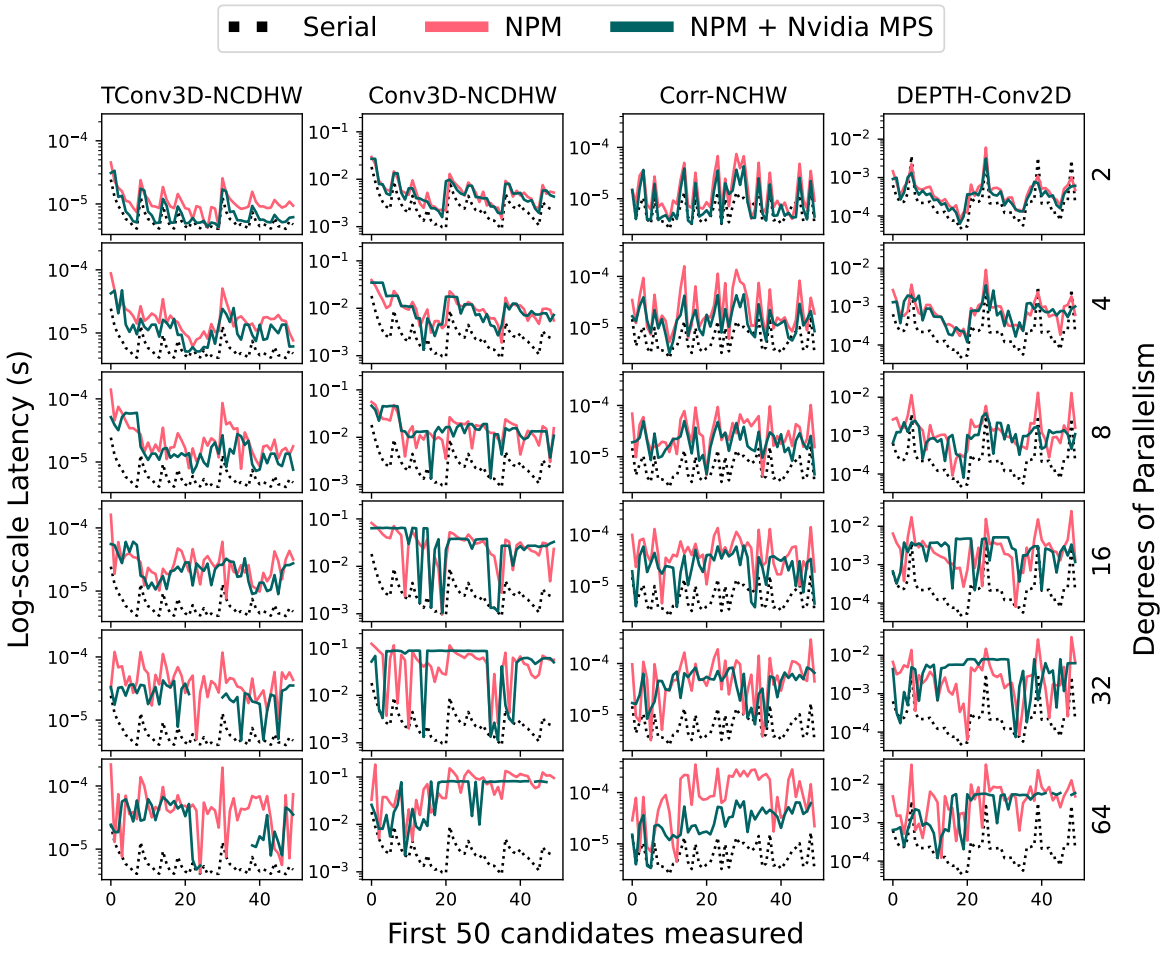


Figure H.2: {PART 2:} Measured latency of proposed candidate tensor programs when auto-tuning with a Grid-index auto-tuner. Reported 50 first candidates proposed by the auto-tuner towards the GPU within platform A, across several tensor operator auto-tuning tasks. Compared latency measurements as reported via serial, NPM and NPM + MPS measurement infrastructures.

H.2 Measurement Outcomes During NPM Auto-tuning

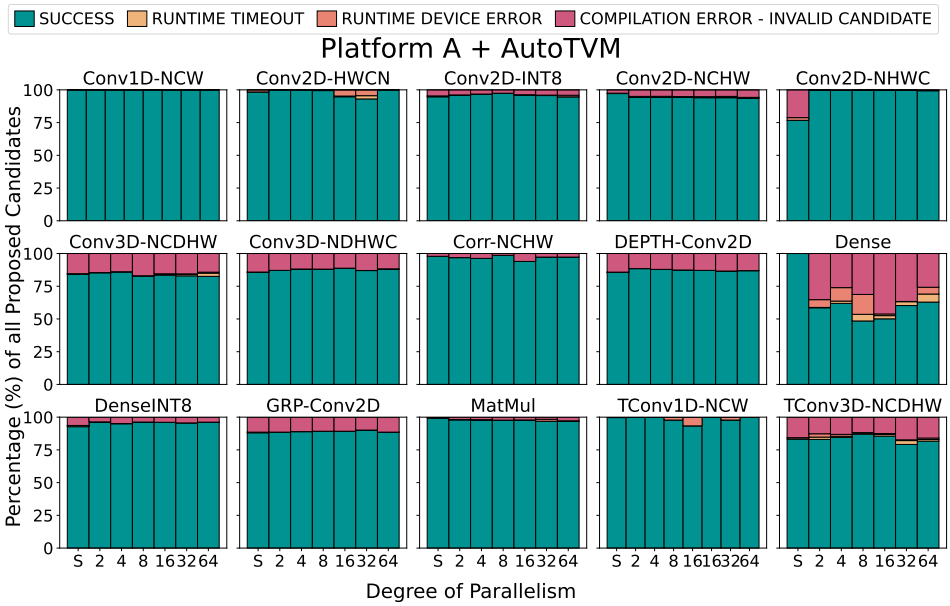


Figure H.3: Auto-tuning 15 groups of tensor operators using AutoTVM auto-tuner on Platform A (Intel CPU + Nvidia V100 GPU) until 500 hardware measurements are performed.

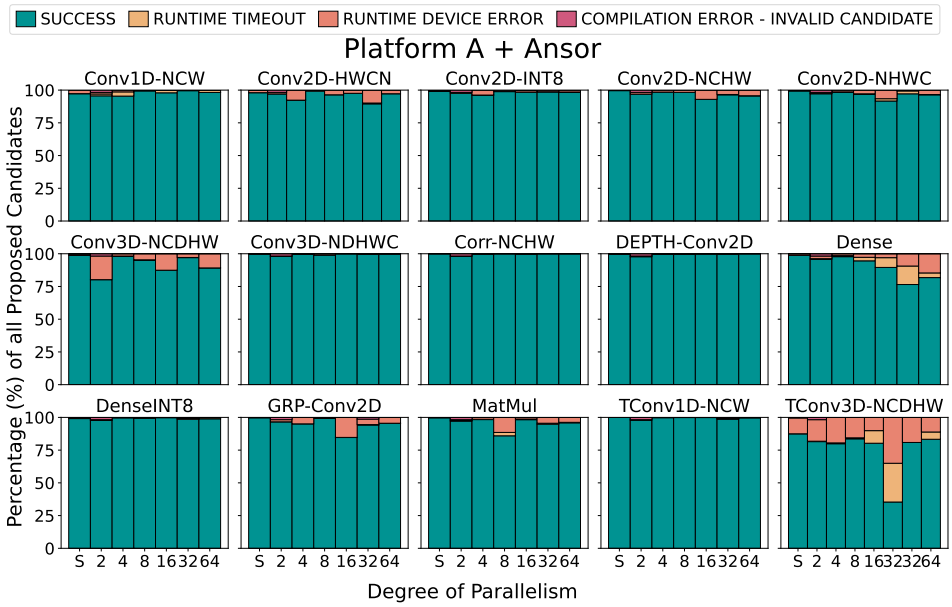


Figure H.4: Auto-tuning 15 groups of tensor operators using Ansor autoscheduler, Platform A (Intel CPU + Nvidia V100 GPU) until 500 hardware measurements are performed.

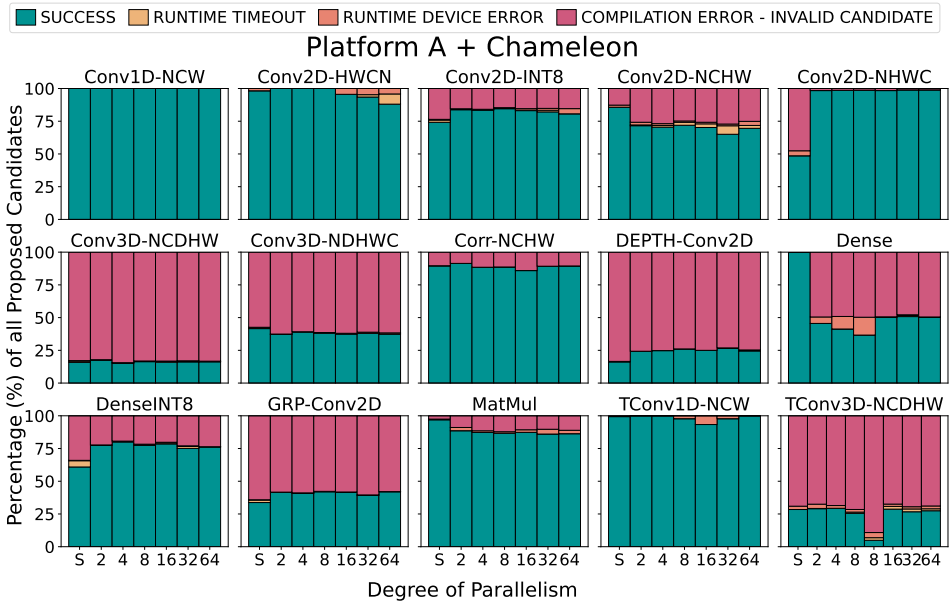


Figure H.5: Auto-tuning 15 groups of tensor operators using Chameleon auto-tuner on Platform A (Intel CPU + Nvidia V100 GPU) until 500 hardware measurements are performed.

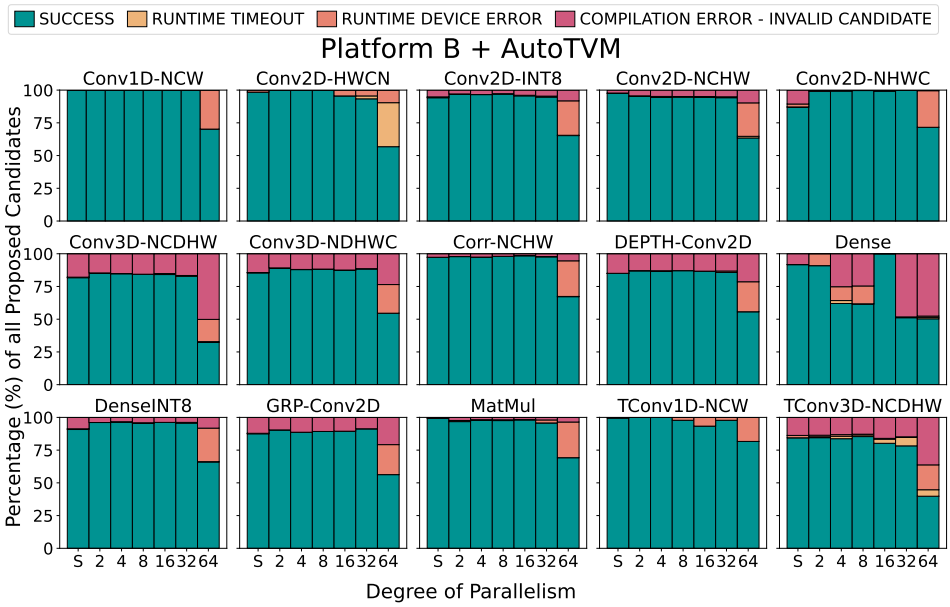


Figure H.6: Auto-tuning 15 groups of tensor operators using AutoTVM auto-tuner on Platform B (AMD CPU + Nvidia GTX 2080 GPU) until 500 hardware measurements are performed.

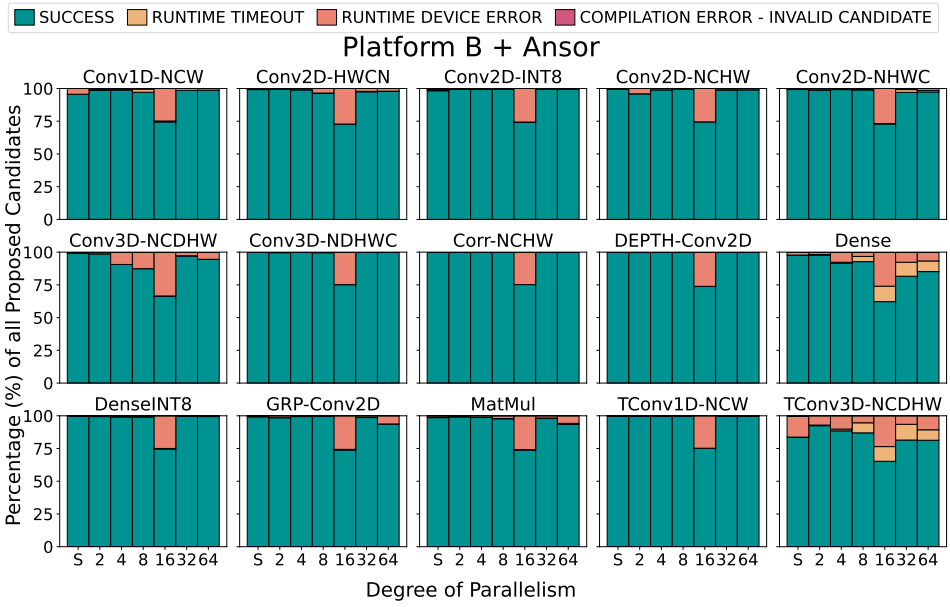


Figure H.7: Auto-tuning 15 groups of tensor operators using Ansor autoscheduler, Platform B (AMD CPU + Nvidia GTX 2080 GPU) until 500 hardware measurements are performed.

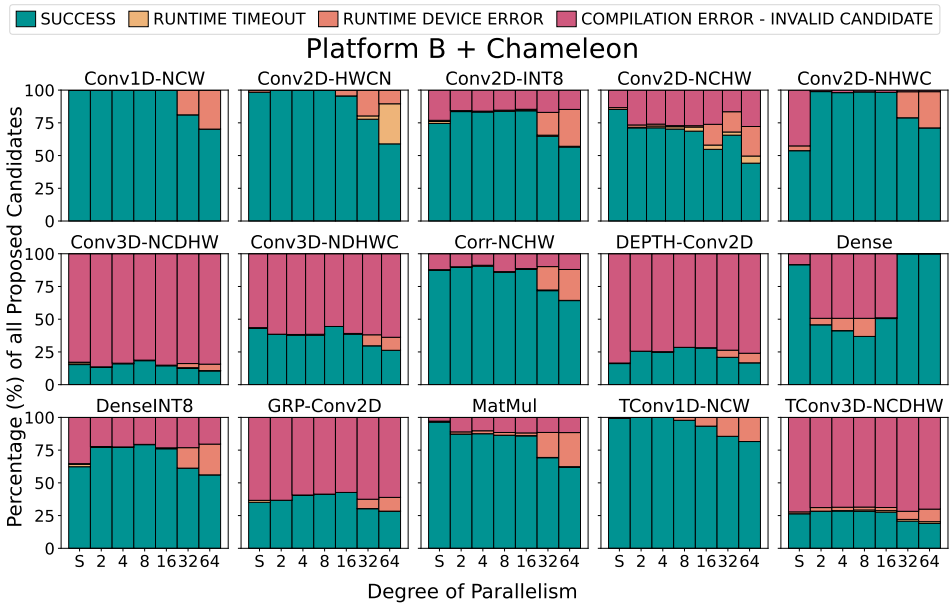


Figure H.8: Auto-tuning 15 groups of tensor operators using Chameleon auto-tuner on Platform B (AMD CPU + Nvidia GTX 2080 GPU) until 500 hardware measurements are performed.

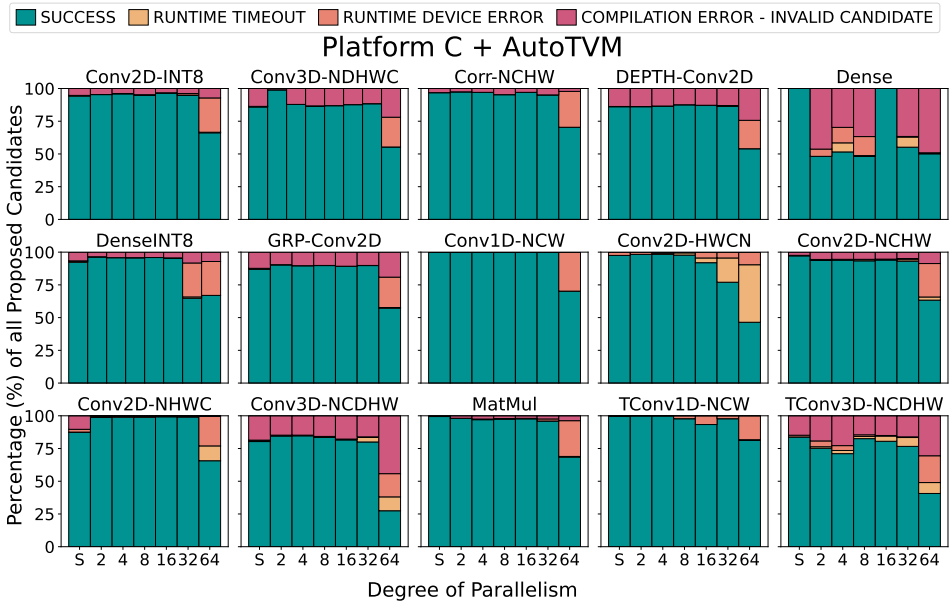


Figure H.9: Auto-tuning 15 groups of tensor operators using AutoTVM auto-tuner on Platform C (Intel CPU + Nvidia GTX 1080 GPU) until 500 hardware measurements are performed.

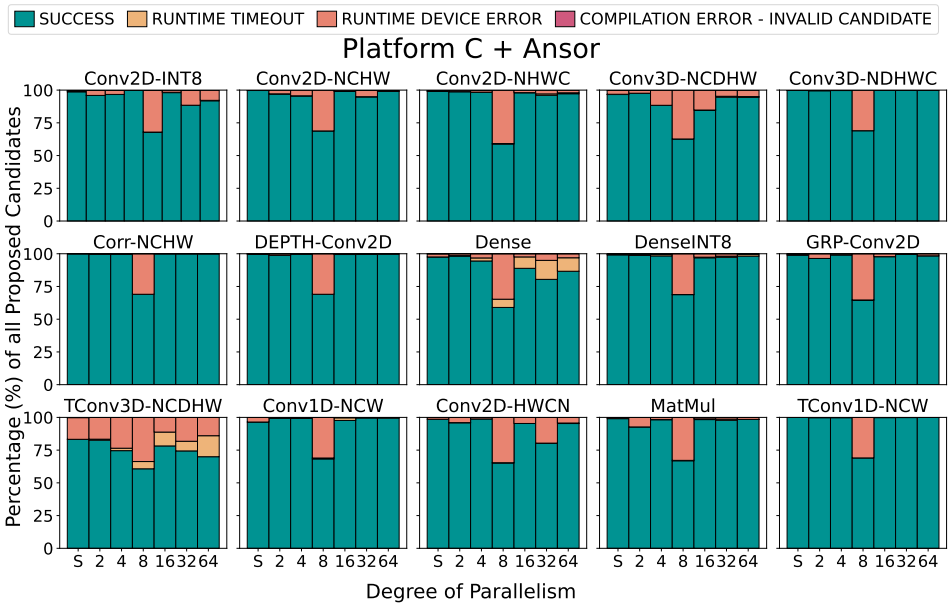


Figure H.10: Auto-tuning 15 groups of tensor operators using Ansor autoscheduler, Platform C (Intel CPU + Nvidia GTX 1080 GPU) until 500 hardware measurements are performed.

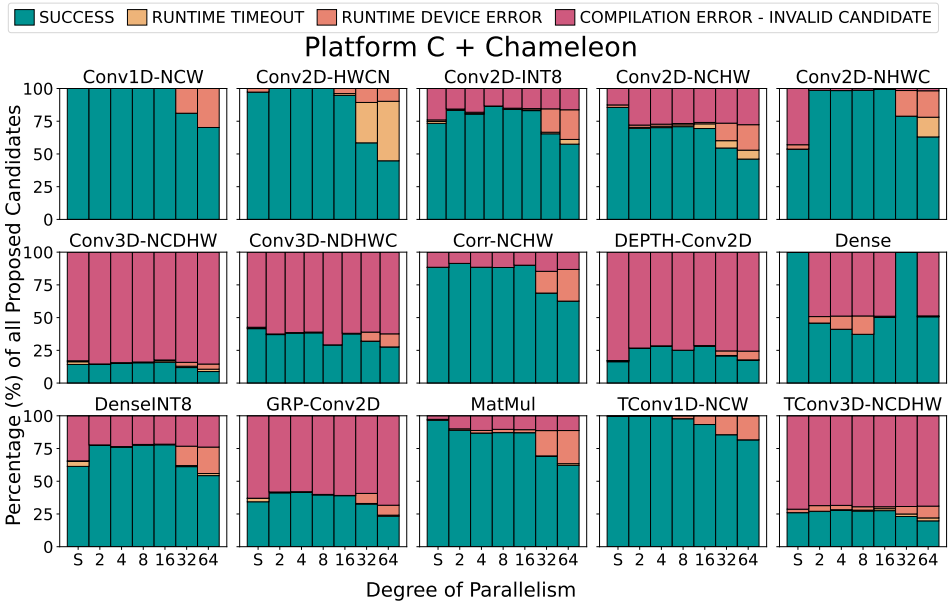
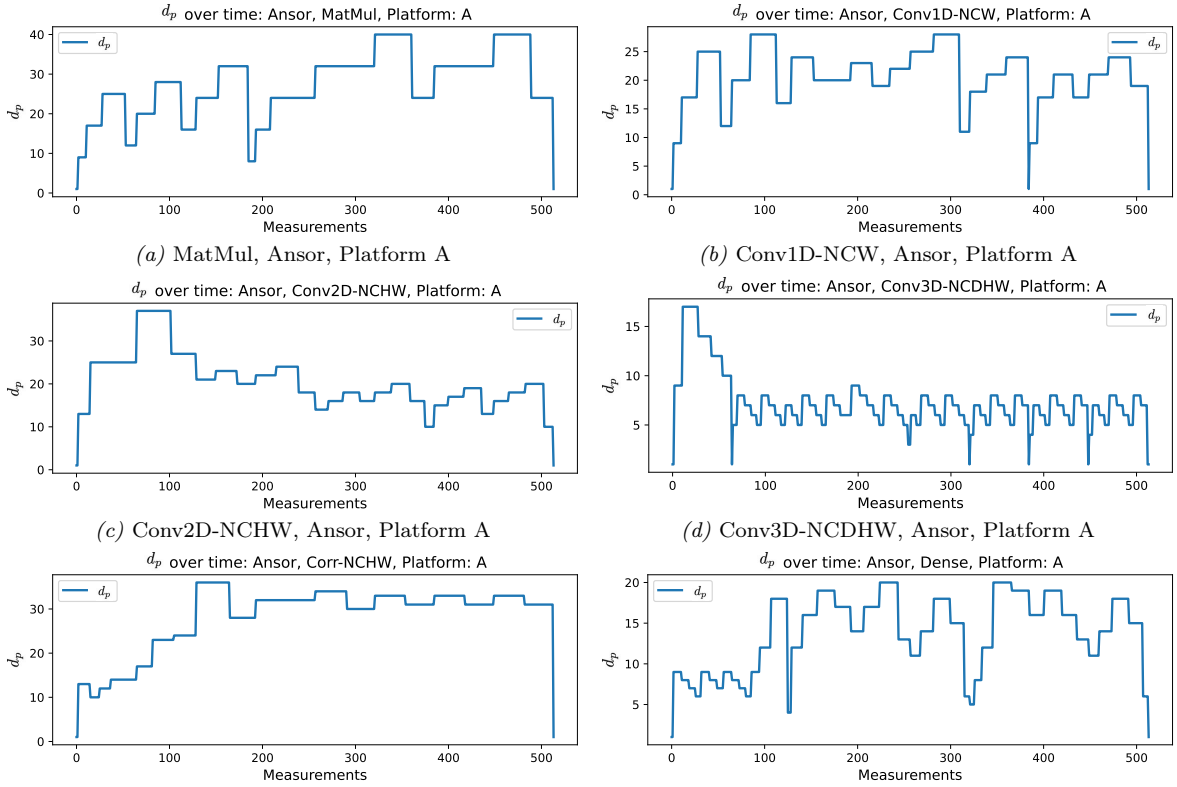


Figure H.11: Auto-tuning 15 groups of tensor operators using Chameleon auto-tuner on Platform C (Intel CPU + Nvidia GTX 1080 GPU) until 500 hardware measurements are performed.

Appendix I

DOPpler - Additional Results

I.1 d_p Over Time Across Tensor Operators and Tensor Programs



(a) MatMul, Ansor, Platform A

(b) Conv1D-NCW, Ansor, Platform A

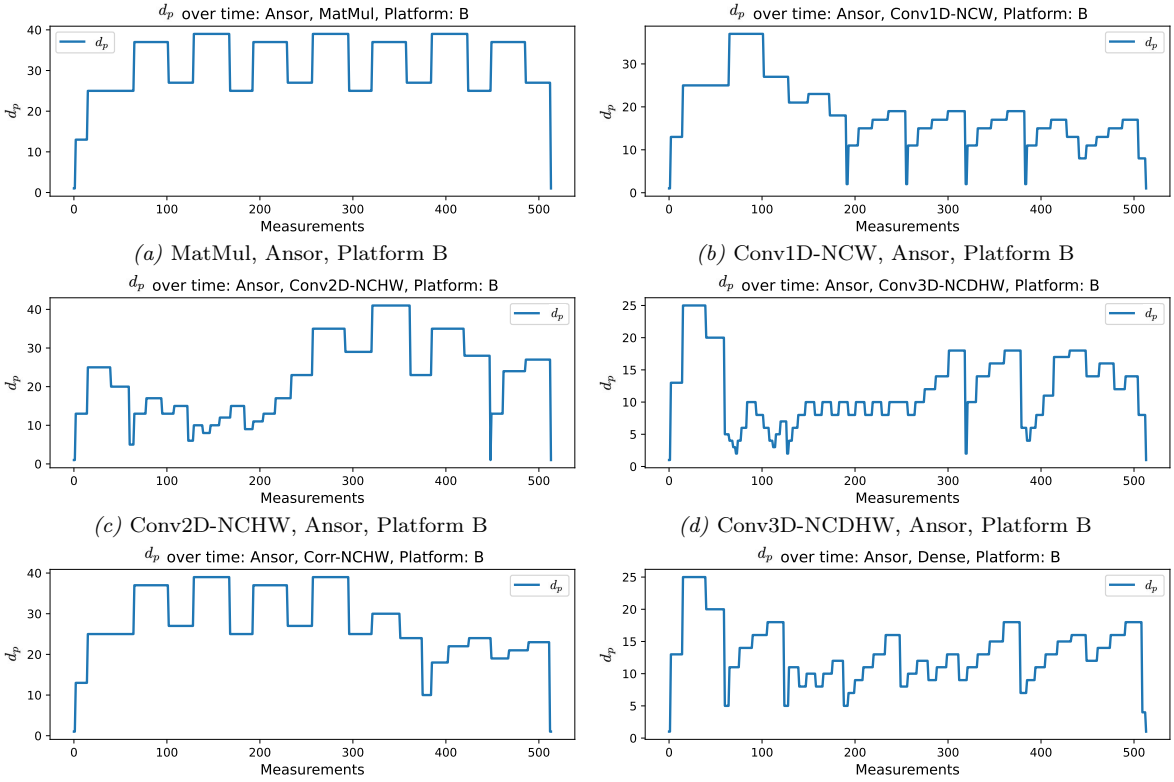
(c) Conv2D-NCHW, Ansor, Platform A

(d) Conv3D-NCDHW, Ansor, Platform A

(e) Corr-NCHW, Ansor, Platform A

(f) Dense, Ansor, Platform A

Figure I.1: d_p over time when auto-tuning with Ansor towards Platform A



(a) MatMul, Ansor, Platform B

(b) Conv1D-NCW, Ansor, Platform B

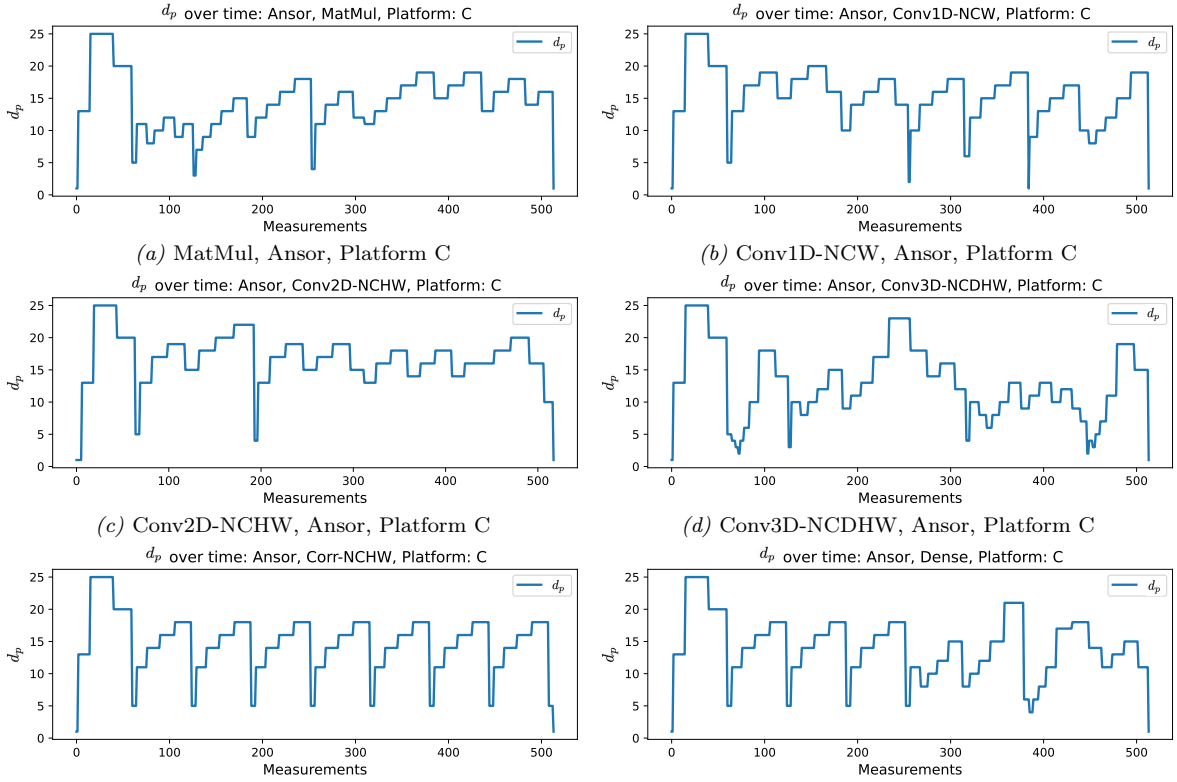
(c) Conv2D-NCHW, Ansor, Platform B

(d) Conv3D-NCDHW, Ansor, Platform B

(e) Corr-NCHW, Ansor, Platform B

(f) Dense, Ansor, Platform B

Figure I.2: d_p over time when auto-tuning with Ansor towards Platform B



(a) MatMul, Ansor, Platform C

(b) Conv1D-NCW, Ansor, Platform C

(c) Conv2D-NCHW, Ansor, Platform C

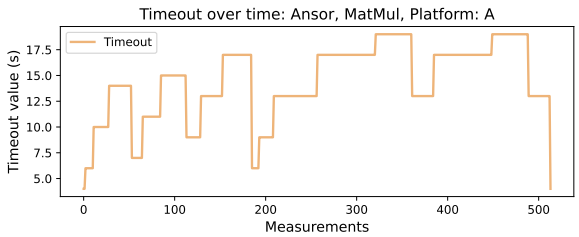
(d) Conv3D-NCDHW, Ansor, Platform C

(e) Corr-NCHW, Ansor, Platform C

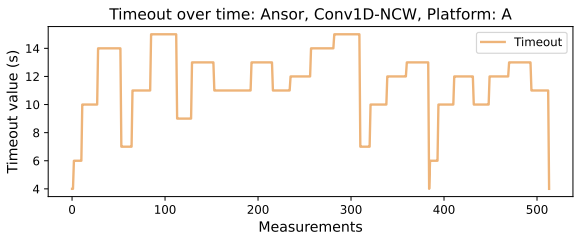
(f) Dense, Ansor, Platform C

Figure I.3: d_p over time when auto-tuning with Ansor towards Platform C

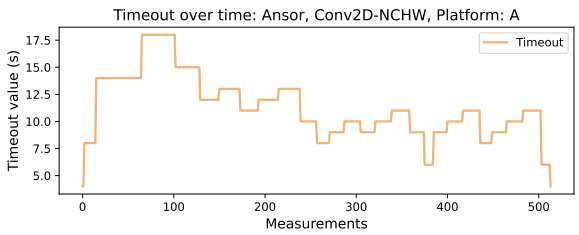
I.2 Timeout Setpoint Over Time Across Platforms and Tensor Operators



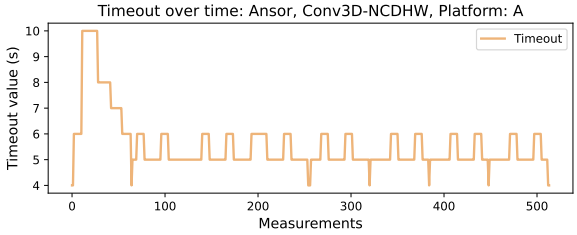
(a) MatMul, Ansor, Platform A



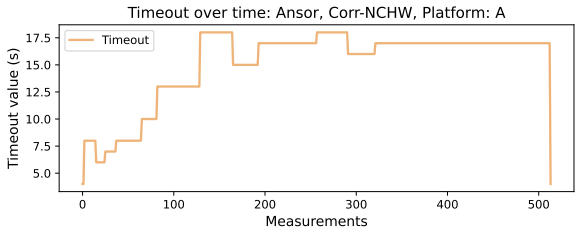
(b) Conv1D-NCW, Ansor, Platform A



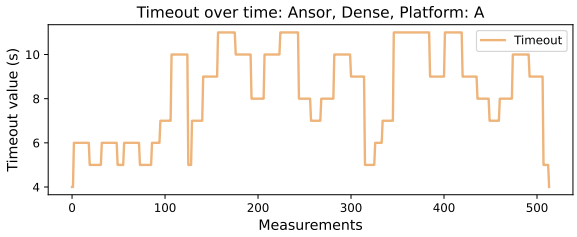
(c) Conv2D-NCHW, Ansor, Platform A



(d) Conv3D-NCDHW, Ansor, Platform A

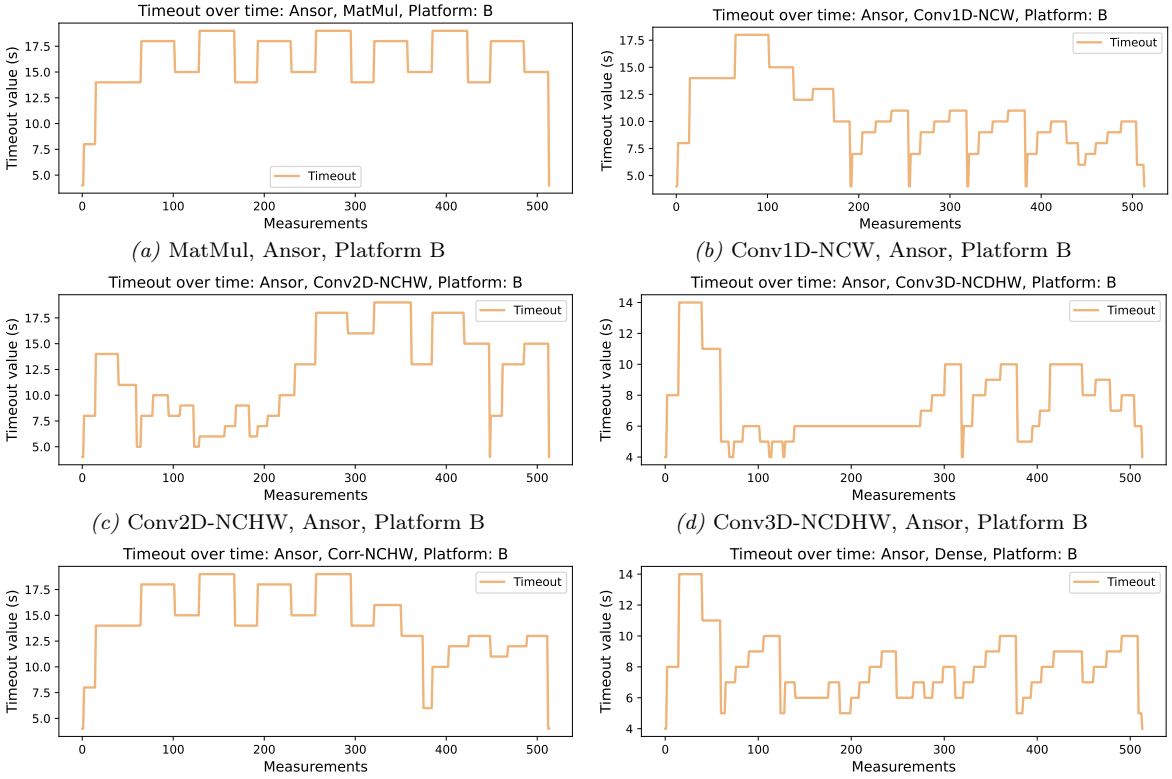


(e) Corr-NCHW, Ansor, Platform A



(f) Dense, Ansor, Platform A

Figure I.4: Timeout over time when auto-tuning with Ansor towards Platform A



(a) MatMul, Ansor, Platform B

(b) Conv1D-NCW, Ansor, Platform B

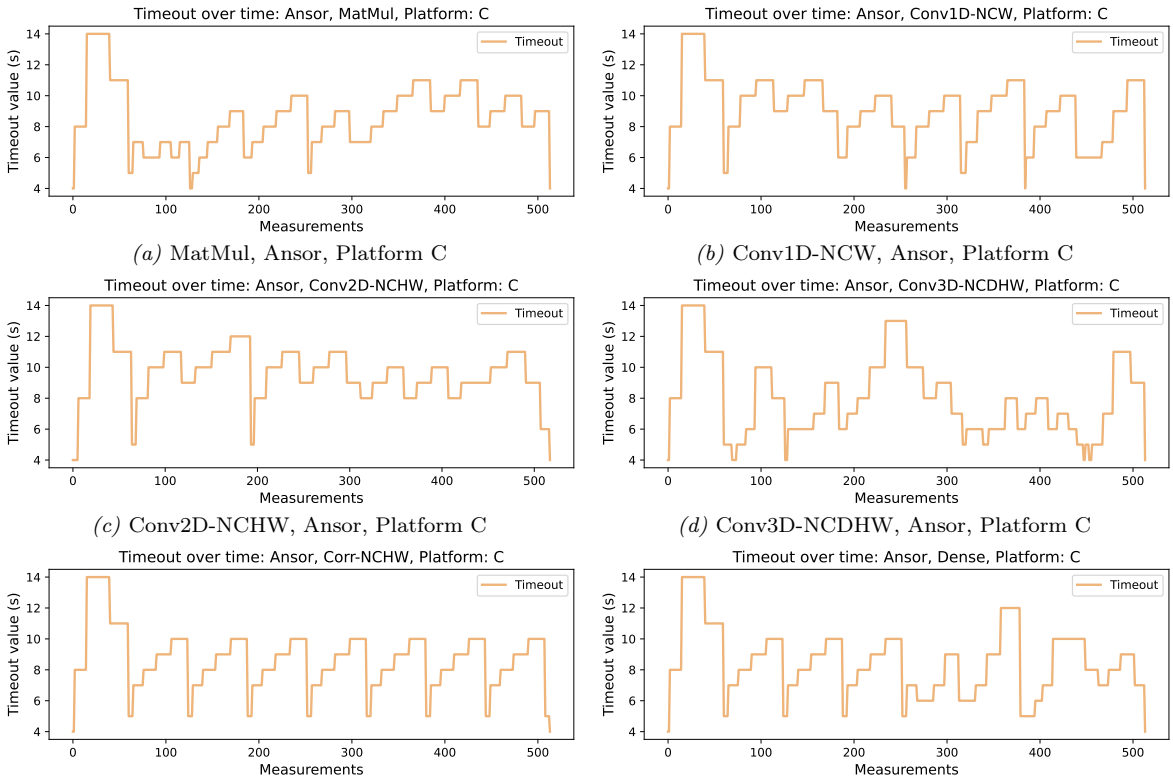
(c) Conv2D-NCHW, Ansor, Platform B

(d) Conv3D-NCDHW, Ansor, Platform B

(e) Corr-NCHW, Ansor, Platform B

(f) Dense, Ansor, Platform B

Figure I.5: Timeout over time when auto-tuning with Ansor towards Platform B



(a) MatMul, Ansor, Platform C

(b) Conv1D-NCW, Ansor, Platform C

(c) Conv2D-NCHW, Ansor, Platform C

(d) Conv3D-NCDHW, Ansor, Platform C

(e) Corr-NCHW, Ansor, Platform C

(f) Dense, Ansor, Platform C

Figure I.6: Timeout over time when auto-tuning with Ansor towards Platform C

I.3 Low-level View of the Candidate Measurement Procedure

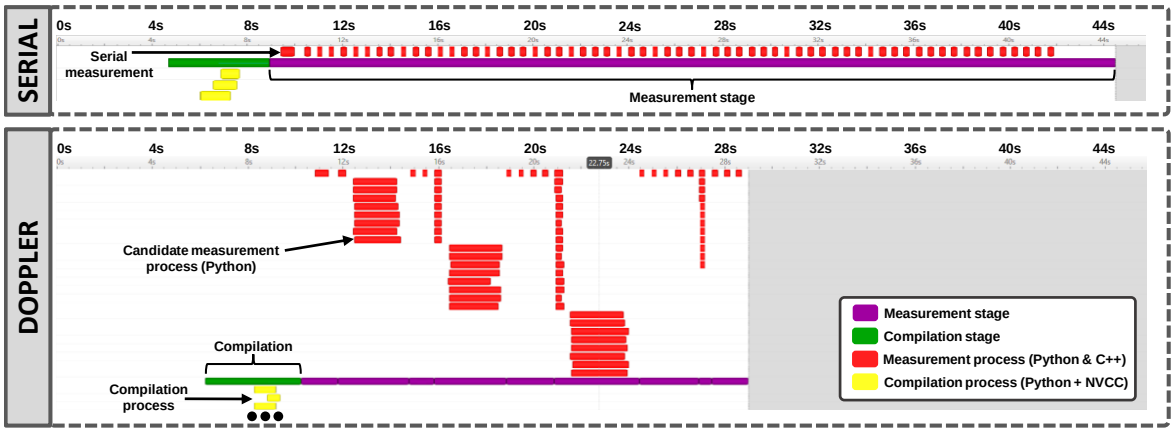


Figure I.7: Process-level view of auto-tuning MatMul with Serial and DOPpler (Grid-index, 64 measurements (1 batch of candidate schedules), 3 repeats (per candidate), Platform A). Stage executions traced using Nvidia NVTX [238], and Nsight Systems [236]

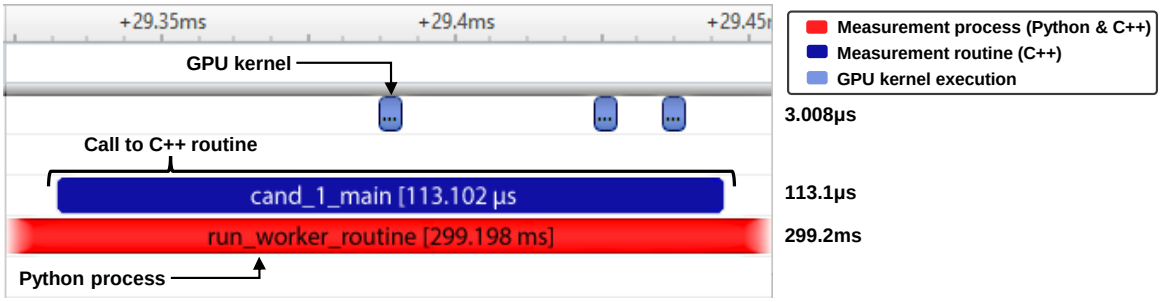


Figure I.8: NVTX [238] trace at the kernel level during candidate measurement. Python measurement process (red) calls TVM latency evaluation C++ runtime (blue), both of which are considerably slower than GPU kernel execution (pale-blue).

Table I.1: Auto-tuning scenario: MatMul, Ansor, Serial vs. DOPpler, 64 measurements, 3 Repeats / candidate. Timings data collected using Nvidia NVTX & Nsight Systems. *run_worker_routine* = total measurement time per candidate and *measure_candidates* = total auto-tuning time.

	Serial			DOPpler		
Range	**Total Time**	**Instances**	**Avg**	**Total Time**	**Instances**	**Avg**
compile_candidates	3.296 s	1	3.296 s	3.055 s	1	3.055 s
measure_candidates	33.192 s	1	33.192 s	19.189 s	9	2.132 s
build_worker_routine	31.968 s	64	499.497 ms	32.599 s	64	509.363 ms
run_worker_routine	15.642 s	64	244.404 ms	68.859 s	78	882.807 ms
cand_0_main	37.886 µs	1	37.886 µs	51.573 µs	1	51.573 µs
cand_1_main	55.237 µs	1	55.237 µs	56.363 µs	1	56.363 µs
cand_2_main	75.351 µs	1	75.351 µs	60.705 µs	1	60.705 µs
cand_3_main	104.914 µs	1	104.914 µs	156.318 µs	2	78.159 µs
cand_4_main	52.754 µs	1	52.754 µs	40.475 µs	1	40.475 µs
cand_5_main	90.648 µs	1	90.648 µs	154.676 µs	2	77.338 µs
cand_6_main	104.210 µs	1	104.210 µs	78.978 µs	1	78.978 µs
cand_7_main	68.402 µs	1	68.402 µs	71.210 µs	1	71.210 µs
cand_8_main	90.787 µs	1	90.787 µs	79.337 µs	1	79.337 µs
cand_9_main	54.282 µs	1	54.282 µs	50.662 µs	1	50.662 µs
cand_10_main	111.115 µs	1	111.115 µs	65.891 µs	1	65.891 µs
cand_11_main	94.577 µs	1	94.577 µs	173.491 µs	2	86.745 µs
cand_12_main	55.020 µs	1	55.020 µs	92.877 µs	1	92.877 µs
cand_13_main	54.821 µs	1	54.821 µs	66.336 µs	1	66.336 µs
cand_14_main	110.808 µs	1	110.808 µs	204.611 µs	1	204.611 µs
cand_15_main	56.072 µs	1	56.072 µs	72.345 µs	1	72.345 µs
cand_16_main	121.208 µs	1	121.208 µs	103.836 µs	1	103.836 µs
cand_17_main	113.870 µs	1	113.870 µs	72.305 µs	1	72.305 µs
cand_18_main	108.994 µs	1	108.994 µs	75.513 µs	1	75.513 µs
cand_19_main	119.754 µs	1	119.754 µs	49.756 µs	1	49.756 µs
cand_20_main	74.024 µs	1	74.024 µs	176.912 µs	2	88.456 µs
cand_21_main	110.730 µs	1	110.730 µs	75.401 µs	1	75.401 µs
cand_22_main	114.193 µs	1	114.193 µs	207.953 µs	2	103.976 µs
cand_23_main	68.692 µs	1	68.692 µs	35.935 µs	1	35.935 µs
cand_24_main	114.957 µs	1	114.957 µs	405.785 µs	2	202.892 µs
cand_25_main	115.852 µs	1	115.852 µs	42.507 µs	1	42.507 µs
cand_26_main	68.845 µs	1	68.845 µs	71.151 µs	1	71.151 µs
cand_27_main	51.498 µs	1	51.498 µs	67.860 µs	1	67.860 µs
cand_28_main	116.209 µs	1	116.209 µs	55.012 µs	1	55.012 µs
cand_29_main	106.520 µs	1	106.520 µs	95.270 µs	1	95.270 µs
cand_30_main	116.503 µs	1	116.503 µs	312.540 µs	2	156.270 µs
cand_31_main	118.382 µs	1	118.382 µs	62.012 µs	1	62.012 µs
cand_32_main	79.117 µs	1	79.117 µs	186.879 µs	2	93.439 µs
cand_33_main	112.753 µs	1	112.753 µs	53.782 µs	1	53.782 µs
cand_34_main	110.261 µs	1	110.261 µs	57.144 µs	1	57.144 µs
cand_35_main	113.912 µs	1	113.912 µs	70.529 µs	1	70.529 µs
cand_36_main	69.266 µs	1	69.266 µs	210.705 µs	1	210.705 µs
cand_37_main	112.165 µs	1	112.165 µs	184.692 µs	2	92.346 µs
cand_38_main	117.646 µs	1	117.646 µs	184.771 µs	2	92.385 µs
cand_39_main	66.436 µs	1	66.436 µs	60.563 µs	1	60.563 µs
cand_40_main	54.024 µs	1	54.024 µs	71.109 µs	1	71.109 µs
cand_41_main	115.262 µs	1	115.262 µs	301.617 µs	2	150.808 µs
cand_42_main	65.308 µs	1	65.308 µs	74.169 µs	1	74.169 µs
cand_43_main	52.079 µs	1	52.079 µs	74.251 µs	1	74.251 µs
cand_44_main	108.586 µs	1	108.586 µs	76.008 µs	1	76.008 µs
cand_45_main	115.650 µs	1	115.650 µs	88.433 µs	1	88.433 µs
cand_46_main	111.522 µs	1	111.522 µs	48.056 µs	1	48.056 µs
cand_47_main	52.613 µs	1	52.613 µs	74.799 µs	1	74.799 µs
cand_48_main	115.446 µs	1	115.446 µs	68.579 µs	1	68.579 µs
cand_49_main	114.742 µs	1	114.742 µs	78.050 µs	1	78.050 µs
cand_50_main	114.828 µs	1	114.828 µs	96.261 µs	1	96.261 µs
cand_51_main	113.627 µs	1	113.627 µs	71.761 µs	1	71.761 µs
cand_52_main	112.560 µs	1	112.560 µs	103.580 µs	1	103.580 µs
cand_53_main	119.490 µs	1	119.490 µs	276.753 µs	2	138.376 µs
cand_54_main	113.671 µs	1	113.671 µs	149.204 µs	1	149.204 µs
cand_55_main	112.558 µs	1	112.558 µs	47.917 µs	1	47.917 µs
cand_56_main	52.039 µs	1	52.039 µs	316.021 µs	2	158.010 µs
cand_57_main	67.278 µs	1	67.278 µs	68.823 µs	1	68.823 µs
cand_58_main	111.210 µs	1	111.210 µs	52.847 µs	1	52.847 µs
cand_59_main	51.677 µs	1	51.677 µs	57.651 µs	1	57.651 µs
cand_60_main	107.627 µs	1	107.627 µs	210.557 µs	2	105.278 µs
cand_61_main	105.448 µs	1	105.448 µs	203.352 µs	1	203.352 µs
cand_62_main	114.796 µs	1	114.796 µs	89.601 µs	1	89.601 µs
cand_63_main	66.537 µs	1	66.537 µs	61.615 µs	1	61.615 µs

Glossary

Accelerator Dedicated co-processor to the main machine's CPU, used to perform a select subset of operations more efficiently than via the use of CPU's instructions.

Activation In the context of DL, an activation is an output of a single neuron's activation function. ANN layers produce activations.

Architecture In the context of DNNs, architecture describes the organisation of the DNN graph and choice of tensor operators / layers that populate each node within the graph. In the context of processors, an architecture describes the composition of the chip, for example, the number and type of cores, the interconnects between the cores or their association with different levels of caches / DRAM memory.

Auto-tuning Automated discovery of high-performance program implementations.

Autoscheduler In the context of low-level DL optimisation, an autoscheduler is an auto-tuner that does not rely on schedule templates, instead generatively constructing schedules using rules, steps or other forms of patterns, whereby a schedule is iteratively constructed. Generally, autoschedulers do not require a DL compiler engineer to provide a template that matches a tensor operator class and target-device class for the combination to be optimisable.

Batch In the context of DL training, a batch denotes the number of individual training samples being process before updating the model's parameters. In the context

of Trimmer, a batch denotes a partial auto-tuning session performed for a tensor operator. In the context of DOPpler, a batch denotes a subset of candidate tensor programs being measured simultaneously across one or more target-devices.

Black box A component of a system is said to be a black box when it produces expected results as per specification without revealing its inner workings.

Bottleneck A design factor within a system that limits its capacity/performance.

Candidate In the context of DL auto-tuning, a candidate is a configuration, that when used to parameterise a schedule, produces a unique tensor program.

Cloud Distributed computing paradigm with on-demand access to compute resources.

Cluster Collection of more than one machine, interconnected via a common network.

Configuration In the context of broader computer science, a configuration describes parameterisation of a system component that in some way modifies its behaviour or desired functions. In the context of DL compilation and auto-tuning-based optimisation, a configuration may refer to the parameters that are used to construct a schedule for a given tensor operator.

Convergence In the context of ML or optimisation problems, convergence is a desirable state where the prediction error or search progress is satisfactory.

Dense In terms of NN layers, dense may refer to a dense layer - a fully connected layer where each neuron connects to every other neuron in the preceeding layer. In terms of entire NNs, dense may refer to dense networks - entire networks composed of dense layers.

Edge In the context of distributed computing, Edge can refer to edge devices - devices that are part of a distributed system that deliver computation and request serving close to their location to leverage data locality and reduced network overheads.

Embedding In the context of DL, an embedding is a construct that enables a high-dimensional vector space to be mapped to low-dimensional representation (for example, a vector). An embedding captures input semantics and can be learned.

End-to-end When referring to DL model optimisation, end-to-end describes a situation where the entire model is optimised - that is all tensor operators.

Engine A core software component that may be included in applications to facilitate some functionality. For example, an *inference engine* facilitates DL inference.

Epoch In the context of DL training, an epoch denotes a single pass through the entire training dataset. In the context of Trimmer, an epoch denotes a single partial auto-tuning pass through all tensor operators within a model.

Feature An attribute of a data point.

Filter In the context of Convolutions, a filter is the same as kernel (i.e. a window that moves across the layer's inputs). In the context of systems, a filter may refer to a component that filters entities according to rules, patterns or policies.

Gradient A derivative calculated with respect to an ML model parameter (weight) during training, typically within the process of backpropagation.

Graph-level When referring to DL optimisation, graph-level is commonly used to group optimisations that transform the DL model graph structure.

Hyperparameter In the context of DL model training, a parameter that is not part of the internal (learned) model parameters (weights), and is used to configure the process of training. The term is also used across the thesis to describe configuration parameters of the Trimmer and DOPpler systems.

Inference The process of producing a prediction by a trained ML model.

Inter-device Across multiple devices - for example, performing multiple simultaneous candidate measurements across multiple devices.

Interference Degradation of execution performance (for example, increase in latency), when two or more workloads simultaneously share the same computational resource.

Intra-device In the context of measurement parallelism - within a single device - for example, performing simultaneous candidate measurements within a single device.

Isolation In relation to DL compilation and optimisation, a situation where the target-device is prevented from executing workloads other than those originating from the auto-tuning process and only one at a time.

Kernel In the context of CUDA, a kernel is a self-contained routine that executes multiple threads on the GPU. In the context of Convolution operators, a kernel describes a window used to multiply inputs and weights to produce activations.

Latency Latency describes the total time taken to perform the action from start to finish. For example, inference latency of an end-to-end DL model, captures the time between the input reaching the model and the model producing a prediction.

Layer In the context of DL models, a layer is a logical component of a model consisting of learnable weights and an operator.

Loop nest Set of loops embedded into each-other's bodies. For example, inner and outer loops.

Loss The measure of error during ML model training with respect to some objective function.

MatMul Short for Matrix Multiplication.

Middleware A software system or a set of systems that facilitate applications in utilising system resources. Oftentimes, refers to any distributed systems software that facilitates communication, data and process management.

Neural Adjective describing ML or DL entities that rely on artificial neurons.

Padding In the context of Convolutions, padding refers to input or output padding - a 0-based frame that is applied to a matrix / tensor data intended to preserve spatial dimensions as the input is moving along the network layers.

Parallelisation In the context of DL compilation, the process of mapping individual computation patterns such as inner loops or sub-loops to different parallelism constructs such as threads or individual cores.

Pipeline In the context of ML or DL, a pipeline is a set of software constructs that facilitate the inflow of inputs and outflow of outputs to and from a DL model or multiple models. In the case of training, a pipeline contains dataset management routines, the training loop, validation subroutines and/or testing scripts.

Platform In this thesis, platform refers to a unique class of machines that contain identical hardware (CPU, GPU, amount of DRAM memory etc.).

Plug and play Two or more operationally compatible systems without the need to substantially modify either of the systems to facilitate their co-operation.

Policy In the context of DL optimisation and more specifically DOPpler, the term policy is used to describe a decision-making subroutine.

Polyhedral A form of program representation as parametric polyhedra, which enables geometric optimisations to be performed on constructs such as loop nests, with the aim of improving data locality or reuse.

Quality In the context of DL optimisation, quality refers to the ability of the optimiser (high-level or auto-tuner) to improve execution performance of individual tensor operators or end-to-end DL models.

Round In the context of DOPpler, a round refers to a cyclic process where a subset (batch) of candidates is measured on target-device, followed by calibration of the degree of parallelism. Multiple DOPpler rounds are performed across a single batch of candidates proposed by the auto-tuner for measurements. For example, DOPpler may perform eight measurement rounds with eight simultaneous candidate measurements across a batch of 64 candidate schedules provided by the auto-tuner.

Schedule In the context of DL compilation, schedule is a set of program transformations that describes how a given computation should be performed.

Scheduling In the context of general computing, the process of allocation of resources to workloads. In the context of DL compilation, the process of determining an implementation for a given computation definition (for example, constructing a tensor program from a tensor operator given some intermediate transformations).

Serial In DL auto-tuning candidate measurements, "serial" describes the process of performing one candidate measurement at a time within a single target-device.

Shape When referring to tensors, a shape describes the dimensionality of a tensor.

Stack In the context of software, a collection of independent systems, programming languages and libraries that enable execution of an application.

Stride In Convolution operators, a stride denotes how far the convolution kernel (filter) moves in some direction across the input tensor when performing convolution.

Target-device In the context of DL optimisation, a target-device is a hardware device towards which the optimisations of workloads are being performed.

Template In the context of DL auto-tuning, a template is a pre-defined skeleton of a schedule that requires parameters (schedule configuration) to construct a complete tensor program for a given tensor operator.

Tensor operator In the context of DL models, a tensor operator specifies the logical operations necessary to perform actions associated with a model's layer.

Tensor program In the context of DL models and DL systems, a tensor program is a standalone program that executes computation associated with a tensor operator. Alternatively, a tensor program can be thought of as a program that performs operations on tensors as inputs and outputs.

Tensorisation In the context of DL compilation, the process of mapping individual computation patterns such as inner loops or sub-loops to hardware intrinsics - instructions that let the program leverage specialised silicon such as tensor cores in recent Nvidia GPUs.

Throughput In the context of DL model execution, throughput describes the number of inferences that can be performed within a given period of time. This definition extends to tensor operators and tensor programs that exhibit execution latency.

Tiling In the context of DL compilation, the process of dividing loop nests into tiles/blocks that can be individually mapped to parallelisation primitives or to exploit data locality via more efficient cache alignment.

Timeout Duration of time a given action is permitted to take before being cancelled.

Training The process during which ML model's parameters are iteratively updated with respect to training data until convergence.

Utilisation In the context of processors, the portion (i.e. a percentage) of the peak compute capabilities utilised by the workload at any measurable point in time.

Vectorisation In the context of DL compilation - mapping of computation patters such as loops to singular SIMD instructions.

Weight A parameter within an ML model that represents learned regularities.

This page is left intentionally blank