©Copyright 2022
Steven Solomon Lyubomirsky

Compiler and Runtime Techniques for Optimizing Deep Learning Applications

Steven Solomon Lyubomirsky

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2022

Reading Committee:

Zachary Tatlock, Chair

Luis Ceze

Kevin Jamieson

Program Authorized to Offer Degree:

Computer Science and Engineering

University of Washington

Abstract

Compiler and Runtime Techniques for Optimizing Deep Learning Applications

Steven Solomon Lyubomirsky

Chair of the Supervisory Committee:
Associate Professor Zachary Tatlock
Computer Science and Engineering

As the scaling and performance demands for deep learning systems have grown, system designers have struggled to incorporate innovations at opposite ends of the system stack: more varied and complex deep learning models and specialized hardware accelerators. New models that use data structures and dynamic control flow to address new learning problems cannot immediately benefit from previous system-level optimizations, which are defined over static dataflow graphs. Meanwhile, many novel hardware accelerators for accelerating common deep learning operations present unusual computing models and often require manual modification of applications to use, demanding expertise in both the deep learning domain and in hardware. The challenges in adding support for accelerators in existing compiler stacks slow development cycles and constrain deep learning systems' capabilities and efficiency.

Following earlier work on the Relay IR for the TVM framework, this dissertation demonstrates that system design problems in the deep learning domain can be approached by formalizing deep learning models as programs broadly (rather than assuming a more specific structure like a graph) and applying traditional compiler engineering techniques, simplifying various optimizations and transformations. In particular, this work addresses the use of runtime systems to support optimizations for dynamic deep learning models and on systematically supporting accelerators through the use of a formal software/hardware interface. Traditional deep learning model optimizations have been conceived as transformations on

static dataflow graphs, but can be adapted to perform similar reasoning dynamically (and hence make no assumptions about control flow) by performing similar reasoning in a run-time system, guided by heuristics that depend on dynamically gathered information. This work explores the specific example of Dynamic Tensor Rematerialization, which is an online approach to the problem of gradient checkpointing (recomputing intermediate activations instead of storing them to reduce the memory required for training) that achieves results comparable to optimal static techniques but generalizes to arbitrarily dynamic models. In addressing the problem of supporting accelerators in deep learning compiler stacks, this work demonstrates that a formal software/hardware interface enables traditional compiler techniques like instruction selection to be adapted for accelerators. Namely, this work presents a methodology for implementing a compiler stack with extensible support for accelerators that uses term rewriting to automatically discover opportunities to apply accelerator operations and lays the foundations for extending formal verification to entire compilation stacks with support for accelerators.

TABLE OF CONTENTS

Page

List of Figures . . . iii

Chapter 1: Introduction . . . 1
- 1 Motivation . . . 1
- 2 Deep Learning Definitions . . . 3
- 3 Differentiable Programming . . . 5
- 4 Runtimes for Dynamic Models: Dynamic Tensor Rematerialization . . . 8
- 5 Supporting Diverse Hardware Back-Ends: 3LA . . . 9
- 6 Organization . . . 10

Chapter 2: Related Work . . . 11
- 1 Reducing Memory Required in Training . . . 11
- 2 Compiling to Accelerators . . . 18

Chapter 3: Relay: A High-Level IR for Deep Learning Applications . . . 28
- 1 Design of Relay . . . 28
- 2 Design Advantage: Type-Directed Relay Fuzzing . . . 32
- 3 Summary . . . 46

Chapter 4: Runtime Techniques: Dynamic Tensor Rematerialization . . . 47
- 1 Problem Description . . . 47
- 2 Design Overview . . . 49
- 3 Formal Bounds . . . 54
- 4 Heuristic Evaluation . . . 72
- 5 Prototype Implementation . . . 91
- 6 Summary . . . 99

Chapter 5: Semantics-Based Hardware Search: 3LA 100
1 Problem Description 100
2 Overview 105
3 The 3LA Methodology 108
4 Prototype Implementation 118
5 Case Studies and Evaluation 122
6 Discussion and Future Work 134
7 Summary 137

Chapter 6: Conclusion 139

LIST OF FIGURES

Figure Number | Page

3.1 A diagram of TVM's architecture with Relay's position marked. 28

3.2 An example of Relay code, namely an implementation of a recurrent neural network following Olah (2015b). This example employs many of Relay's features. `@lin` is a linear layer, applying `nn.dense` (dense matrix multiplication) and elementwise addition. No tensor shapes are annotated, but Relay's type inference will infer the shape of the result using the operators' type relations (matrix multiplication's relation encodes broadcasting semantics (Contributors, 2019) and addition's relation is identity). `@relu_cell` defines a ReLU (rectified linear unit) RNN cell, taking in a tuple of weights and biases (the dots indicate tuple indexing) and returning a tuple literal. `@trained_relu_cell` returns a closure (note the lexical scoping). Note the parametric polymorphism employed in the definition of the `List` type and implementation of `@foldl`, which are common in functional programming languages (and part of Relay's standard library). `@encode` uses `@foldl` to implement a generic encoder RNN by folding a cell function over a list of inputs, demonstating how Relay's functional programming features can implement a deep learning model in a modular, reusable manner. 30

4.1 (Top) Pseudocode for DTR's basic logic (independent of heuristic), and (Bottom) DTR's sequence of events in an operator call. Note that `PerformOp()` may make further recursive calls in order to rematerialize arguments. 50

4.2 Visualization of the state of memory for DTR with $N = 200$, $B = 2\lceil\sqrt{N}\rceil$, and heuristic h_{e^*}. A value of 0 (black) indicates the tensor is evicted or banished, 1 (red) indicates the tensor is a forward value in memory, and 1.5 (white) denotes an in-memory gradient tensor corresponding to the forward tensor. The backward pass begins at the red vertical line; note the presence of evenly spaced *checkpoint tensors* (red horizontal lines) that persist in memory throughout the backward pass. Note also the recursive checkpointing behavior visible in the early gaps of the backward pass, and finally the completely red triangles of the later gaps, when there is enough free memory to avoid repeated rematerialization altogether. 58

4.3 An example construction of an adversarial graph. Gray tensors are in memory (t_0 must always be in memory). The initial tensor t_0 has B paths descending from it, so there is always some path from t_0 with no resident tensors. The adversarial construction chooses to place the next node at the end of such an entirely evicted path. 70

4.4 Simulated results comparing different heuristics on various models, showing the rate of computational slowdown for different budgets (fractions of the original peak memory usage). The black area in each graph corresponds to the memory required to store inputs and weights, while the gray area denotes the single operator requiring the most memory to be live at once. The dashed and dotted lines represent the last ratio before thrashing ($\geq 2\times$ slowdown) and out-of-memory errors, respectively. All logs were produced by running each model 50 times on a single input on a machine with an NVIDIA Titan V GPU (CUDA 10.1, CuDNN 7.6.4) and a 16-core AMD Ryzen Threadripper 1950X on Ubuntu 18.04, logging the final "warmed-up" run. 85

4.5 Results for fixed $c = e^*$, varying s and m. 87

4.6 Results for fixed $c =$ `EqClass`, varying s and m. 87

4.7 Results for fixed $c =$ `local`, varying s and m. 87

4.8 Results for fixed $c =$ no, varying s and m. 88

4.9 Results for the h_{DTR} heuristic, comparing banishing and eager evictions. . . 89

4.10 Total storages accesses incurred by heuristic evaluations and metadata maintenance, compared across different memory ratios, for the 3 main h'_{DTR} variants. 90

4.11 DTR's overhead from operators is competitive with Checkmate's, which uses ILP to produce an optimal rematerialization schedule. This comparison extends Figure 5 in Jain et al. (2019) by adding the DTR simulator as a "solver" that translates Checkmate's Keras-based graph representation into the DTR simulator's representation. To produce this comparison, we modified Jain et al. (2019)'s evaluation artifact because the PyTorch logs in Section 4.3.6 did not contain some information that past checkpointing techniques require (such as which backward operators correspond to which forward ones). Also included in the comparison (from the original experiment) are the Griewank and Walther (2000) Treeverse algorithm and variants of the Chen et al. (2016) checkpointing algorithm (modified to handle skip connections like those in ResNet). 92

4.12 We profiled the running time of our prototype for various models and memory budgets on a machine with an NVIDIA Titan V GPU (CUDA 10.1, CuDNN 7.6.4) and a 16-core AMD Ryzen Threadripper 1950X on Ubuntu 18.04. The red dotted lines correspond to trials that either ran out of memory or thrashed ($\geq 2\times$ unmodified PyTorch's time). Model batch sizes are given in parentheses. To ensure the accuracy of the DTR prototype's profiling, we used PyTorch's synchronous computation mode (see Section 5.1). Results (mean of 100 trials) are compared against unmodified PyTorch. "Cost compute" (computing heuristic scores) and "eviction loop" (comparing scores over tensors) correspond to overhead from the DTR *runtime* itself, which can be reduced by a more efficient implementation. "Unprofiled time" is the remainder of the time per batch; it may be due to runtime overhead from parts of PyTorch not modified in the prototype, like the operator dispatch system. The large proportion of unprofiled time in Unrolled GAN is likely due to its extensive use of Python reflection. The budgets with asterisks were run with the random sampling optimization (see Section 5.2) disabled, as sampling caused occasional failures at those budgets. 93

5.1 Snippet of the FlexASR device driver (Tambe et al., 2021). Through MMIO commands, the driver first stores input arguments, e.g., weights, in the accelerator's internal buffer (lines 10 to 13). It then sets up the configuration such as tensor dimension and vector size (lines 15 to 20). Finally, it triggers the operation (line 23) and retrieves the result (starting line 26). 102

5.2 3LA compilation flow overview. Note that the ILA modeling and the validation of the IR-accelerator mappings are omitted from this figure. 107

5.3 **FlexASR ILA model snippet.** Lines 5-18 define the FlexASR ILA model, its input and architectural states variables. Lines 20-32 shows an example of an ILA instruction named "`pe_0_cfg_mngr`," which corresponds to line 6 in Figure 5.4 (c). In each ILA instruction, we specify its decode condition and state update functions. For example, in this instruction, the decode condition (line 24-25) is when there is write instruction at the top interface to the address associated with the configuration of the PE's management configuration. Lines 27-32 show this instruction's state update functions for the architectural states. In this example, this ILA instruction models the behavior of storing the arguments from the input data at its interface into the FlexASR configuration registers. From this example, we can see that the ILA instructions provide an abstraction of the functionality of the accelerator corresponding to the MMIO instructions at its interface. 110

5.4 **IR-accelerator mapping for the FlexASR linear layer operation.** This shows a many-to-many mapping from Relay IR instructions to a sequence of FlexASR MMIO commands. (a) A Relay linear layer consists of a linear transformation operation `nn.dense`, followed by a bias addition operation `nn.bias_add`. (b) The compiler IR ILA instruction has a one-to-one mapping to the compiler IR instruction. (c) The FlexASR ILA program fragment in its assembly format: It includes: (1) writing instructions to transfer the data into FlexASR's memory; (2) setting up FlexASR `LinearLayer` configuration states, for example, the instruction at line 5 sets the states of FlexASR layer sizing information; (3) an instruction that triggers the FlexASR `LinearLayer` computation; and (4) reading data out from FlexASR's memory if needed. (d) The MMIO commands for FlexASR have a one-to-one mapping to its ILA. 112
5.5 A simplified version of Figure 5.4 highlighting the verification tasks in the IR-accelerator mapping for the FlexASR linear layer operation. 116
5.6 Prototype implementation of the 3LA compilation flow. 118
5.7 An illustration of the BYOC process, using an example similar to that in Chen et al. (2021). 120
5.8 An illustration of how 3LA offloads 2D maxpooling to FlexASR's temporal maxpooling operation. Note that (b) does not contain a match for the left-hand side of the IR-accelerator rewrite rule in (a). (c) is an equivalent rewritten IR program found by flexible matching, containing four instances of the left-hand side of the IR-accelerator rewrite rule. The result of the replacements is given in (d). Note that in this program, the initial store and the final load are needed to communicate with FlexASR; however, the intermediate loads/stores can be eliminated, since the output of one instance serves as input of another. (e) gives a rewrite rule for removing intermediate loads/stores and (f) shows the result of applying it. This program only performs a single (matrix) store at the start of the operation and a single (matrix) load to read the output at the end of the operation. In the future, we hope to generalize this example and consider memory organization in accelerators and data-movement for optimizing data transfers. 135

ACKNOWLEDGEMENTS

All the projects presented in this dissertation were supported in part by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA. Numerous other members of the ADA Center provided my collaborators and me with valuable feedback on earlier versions of the work as well. My participation in the ADA Center also greatly broadened my perspective on research problems related to hardware design and applying accelerators.

Many students of the PLSE and SAMPL groups at the UW Allen School with whom I did not directly collaborate nevertheless provided me with moral support and at times advice, for which I am very grateful. Their ranks include Eunice Jun, Sam Kaufman, Martin Kellogg, Chandrakana Nandi, Remy Wang, James Wilcox, Max Willsey, Eddie Yan, Zihao Ye, Amy Zhu, and Bill Zorn; any perceived omissions from this enumeration are unintentional.

Similarly, many faculty members with whom I did not directly collaborate on research were also influential on my intellectual development and my beliefs about the organization and design of research projects. Especially noteworthy in this regard have been René Just, whose classes on research in software engineering and on research methods directly guided the design of the evaluation in several of the projects of this dissertation and directly inspired the Relay fuzzer project, and Dan Grossman, whose eloquent articulation of the value of core programming languages concepts and functional programming (to which I had a close-up view as a TA for his undergraduate programming languages class) has helped ground my own views on the design of

compilers for deep learning applications.

This work would not have been possible without the faithful efforts of my many research collaborators. I had the honor to have worked with Jared Roesch, Marisa Kirisame, Logan Weber, Josh Pollock, Ziheng Jiang, Luis Vega, Thierry Moreau, and Tianqi Chen on various iterations of the Relay IR; with Edward Misback and Michael Flanders on the Relay fuzzer; with Marisa Kirisame again, Altan Haan, Jennifer Brennan, Mike (Deyuan) He, Jared Roesch again, and Tianqi Chen again on Dynamic Tensor Rematerialization; with Gus Smith, Andrew Liu, Scott Davidson, Joseph McMahan, Michael Taylor, and Luis Ceze on Glenside; and with Bo-Yuan Huang, Yi Li, Mike He again, Thierry Tambe, Gus Smith again, Akash Gaonkar, Vishal Canumalla, Gu-Yeon Wei, Aarti Gupta, and Sharad Malik on the 3LA methodology. (As well as, of course, my adviser, Zachary Tatlock.) We shaped our ideas, designed our evaluations, and prepared presentations over many meetings—across these hours of meetings, I learned countless technical and organizational lessons from my colleagues. The time spent on these projects is rich in pleasant memories, though even the difficult moments and setbacks from these projects have their own value.

I am especially grateful to my collaborators Jared Roesch and Tianqi Chen, to whom I owe the fact of my turning my research interests toward the domain of deep learning. They spent some time convincing me that my programming languages background would be applicable to deep learning and guided my initial steps into the literature on deep learning systems, helping me also to build an understanding of deep learning. I am also grateful to them for introducing me to the TVM project (which has since become part of the Apache Foundation), as working on a deep learning compiler stack proved to be a very effective means of acquainting me with the engineering challenges in that domain; it also gave me a very practical sense of how research ideas can be implemented and evaluated in real systems and eventually adopted. Jared's

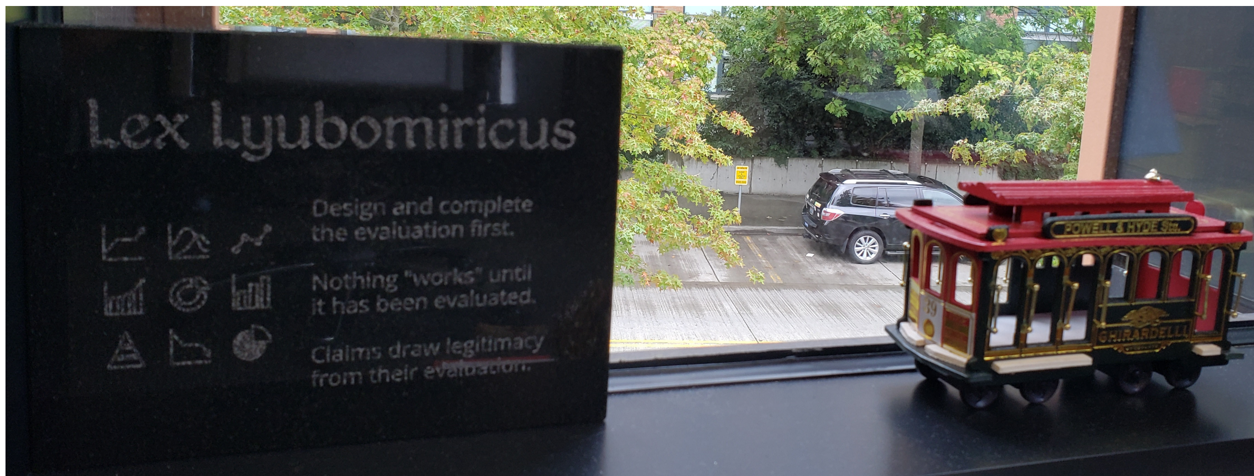
and Tianqi's initial vote of confidence was vital in starting me down the path that led to this dissertation.

Special acknowledgement is also due to some of the very talented undergraduates with whom I had an opportunity to work closely: Mike He, Altan Haan, and Marisa Kirisame (first as an undergraduate, then as a master's student). Marisa's encyclopedic knowledge of the functional programming literature was the source of many of our ideas for Relay and Dynamic Tensor Rematerialization; it was particularly challenging and gratifying to adapt these methods to the domain of deep learning systems. Mike and Altan also brought similar enthusiasm for functional programming and were particularly eager to contribute to the implementation and evaluation of Dynamic Tensor Rematerialization. The participation of all three of them in the Dynamic Tensor Rematerialization project made for a particularly energetic collaborative environment and was certainly a very warm memory during the otherwise sorrowful period of the coronavirus pandemic, when we could not work together in person. For my part, I would like to think that I was able to provide these exemplary students with at least some mentorship regarding the design, evaluation, and presentation of research projects as well as on technical matters. I hope one day to read their doctoral dissertations.

My faculty collaborators, including Michael Taylor and Luis Ceze of UW and Aarti Gupta and Sharad Malik of Princeton University, gave me much perspective not only on intellectual matters (especially when it comes to hardware—Sharad Malik's undergraduate logic design course still accounts for most of what I know about hardware!) but also on organizational skills. I have had the privilege of directly witnessing many different methods of managing research projects, assembling and running a research group (the SAMPL group at UW), preparing publications, and interacting with professional organizations—they all set a fine example, as do their students.

The greatest acknowledgement of all is due to my doctoral adviser, Zachary Tatlock, who was present throughout all these efforts. I knew very little of research in computer science when I began my graduate studies—perhaps he wouldn't phrase it like that, but I would say that Zach took a big risk in taking me on as a student! I had done some projects on program verification in Coq and had the vague sense that that was something I wouldn't mind doing for longer, but getting papers published takes more than just technical aptitude and gumption. Fortunately, I have had the great fortune to have a very calm and patient adviser, who has been willing to take the time to walk me through the steps of mapping out a research projects, writing papers, and navigating the peer review process, including the particular challenges of resubmission cycles. Zach seems to have a preternatural ability to identify gaps in our research ideas and articulate precisely what we must clarify; I have never come out of a meeting with Zach unsure of what to do next. Besides being willing to take the time to explain matters to me that I always thought came as second nature to my colleagues or helping to refine my research ideas, Zach has also encouraged me to have confidence when articulating my ideas or voicing objections. A specific instance is illustrated below: With Zach's encouragement, I wrote a webpage detailing my approach to systems research projects (which boils down to "keep your eyes on the prize" with respect to completing the evaluation). He called my doctrine "Lyubomirsky's Law," and I obnoxiously Latinized it as *Lex Lyubomiricus.* While I was serious about my claims (even if they weren't peer-reviewed), I thought the webpage was the end of it—but one day, Zach began one of our meetings by revealing that *he had had it carved in stone*! I have never been more flattered in my life. I cannot think of a better example of Zach's confidence in his students. In addition to everything Zach has done for me and his other students, he is also a social pillar of the PLSE group in particular and the Allen School in general, often taking the initiative in organizing social events

and contributing to our organizational culture (his enthusiasm is contagious). He has set a fine example of professionalism, and I consider it a privilege to have been his student—I will always strive to mentor others as devotedly as Zach has mentored me.



DEDICATION

This dissertation is dedicated to my maternal grandparents, Sulima and Isaak Khimishman, both of blessed memory, who would have loved nothing more than to have attended my defense.

Chapter 1
INTRODUCTION

1 Motivation

Many computing tasks which were once considered only distant possibilities for artificial intelligence (AI) are today not only feasible but are, in fact, commonplace and used regularly in day-to-day work and life. Such tasks include high-quality machine translation, facial recognition, and increasing degrees of automation for driving tasks (among others). In contrast to the techniques of what is now called classical AI, which generally operate on an explicitly stated logical model of the world and apply inference rules, these tasks have recently and most successfully been addressed using deep learning (DL) techniques, which use data to learn the values of parameters needed to guide decisions like which pixels in a 2D image correspond to a human face. For a famous example of the degree to which DL techniques have succeeded in scaling in problems where classical AI has had difficulty, consider the contrast between chess and Go. In chess, a program based on the Minimax algorithm and α-β pruning techniques from classical AI defeated the human champion in 1997 (Campbell and Hoane, 1999). By contrast, Go has a much larger search space than chess; automated techniques could not defeat human players until the DL-based approach of AlphaGo Zero in 2017 (Silver et al., 2017), which used parameters learned from tens of millions of simulated games (rather than manually specified heuristics) to guide choices of moves. In recent years, DL techniques have been applied to address increasingly complex tasks, some of which entail a degree of understanding the world, like assigning descriptive text captions to images or answering questions in text. While many systems for these tasks are research prototypes that are being evaluated against benchmarks, the latest advances find their way quickly into practice, as with Google's use of the BERT model (Devlin et al.,

2018) for processing search queries (Nayak, 2019). The rapidity with which innovations in deep learning are adopted in practice highlights not only the impressive pace of progress in that field but also the economic importance of many of the problems that DL techniques help to address, particularly with respect to automating tasks historically done manually.

As DL applications become increasingly diverse and important, the infrastructure required to support deep learning at scale also increases in importance, presenting a variety of engineering challenges. Depending on the application, DL systems may run at large scales on large distributed systems in order to process vast amounts of data, or at smaller scales on common consumer hardware like laptops and smartphones; they may also have to run at low latency on low-powered embedded devices in applications where safety is important, such as automated driving. At the largest scales, the power demands of DL applications have demanded significant analysis of the environmental impact of running these systems and accordingly efforts to mitigate said impact (Patterson et al., 2022). The increasing complexity of DL applications and the diversity of the settings in which they are executed together contribute to what is *fundamentally a compilation problem*: Having to design languages and implement compilers capable of expressing the necessary computations and deploying code to the appropriate devices, while satisfying numerous constraints on performance and resource usage. Numerous specialized compilers have been and continue to be developed for DL applications, some providing end-to-end faculties for defining entire DL models and executing them on the desired hardware and others intended to fulfill specific tasks, like optimizing particular operators, and interoperate with other tools. Though these specialized tools are undoubtedly necessary to meet the desired standards of performance, this dissertation contends that the best way to support continued innovation in DL applications is, in a sense, *less specialization* and more generality.

In particular, this dissertation will explore how expressing DL applications as general programs allows for adapting traditional compilers techniques in order to address outstanding problems related to new applications and settings, focusing on two specific problems:

1. Adapting static graph optimizations to dynamic DL models through the use of runtime systems (in this case, for the problem of gradient checkpointing), and

2. Developing a methodology for compiling DL models and other applications to new specialized devices through the use of an instruction-like software/hardware interface and adapting traditional instruction selection techniques.

2 *Deep Learning Definitions*

This dissertation will focus on DL techniques as they pertain to supervised learning on classification problems. While there are other learning problems, these problems are very common in practice and provide simple parameters for the compilation problems discussed later. This discussion is accordingly simplified, as it is intended to provide the context for system-level optimizations discussed later rather than serve as a detailed or formal description of how DL models are designed or motivated.

In a classification problem, a model is given an input from a domain and seeks to infer a label that corresponds to some classification (such as whether a certain region of an image is a human face). Deep learning provides a general approach for solving such problems if the user has sufficiently many example inputs that have already been labeled. Namely, so long as the inputs and labels can be interpreted numerically, a *model* can be specified as a function from an input to a label, generally also taking *parameters* (also called weights) that do not depend on the input. Modifying the model's parameters allows for tuning what label is returned for a given input, so the same model can be effective in different inference problems. A model is specialized to a particular inference problem in a process called *training*, which entails finding an optimal choice of parameters for a given set of labeled examples (the *training set*). Training typically involves definining a *loss function* that corresponds to the error between the model's returned label and the actual label for a given example, so training can be defined as finding an assignment of parameters that minimizes the loss. Optimization algorithms like stochastic gradient descent (SGD) have proven very effective in practice at

finding good choices of parameters; notably these techniques require finding the gradient of the model for a given input, so models must be *differentiable*. In SGD and related processes, training is performed in multiple rounds by running inference on inputs, computing the loss using the inference results and the true labels, and using the gradient of the loss (usually for a batch of inputs) to adjust the parameters towards an optimal value (called *backpropagation*), continuing until the loss reaches a fixed point.

The implementation of DL models and the training process involves large scales in many regards, thus motivating the need for optimizations, including through the often exorbitantly expensive route of developing specialized hardware. The numerical interpretations of many problems often require that inputs be tensors of many dimensions; for example, problems in computer vision usually treat input images as tensors of the RGB pixel values. Accordingly, models must operate over large tensors, meaning that arithmetic operations over tensors are computationally expensive (and performance bottlenecks). The parameters to these operations are often also large tensors; a large model like Transformer uses over 100 MB for storing parameters alone (Sohoni et al., 2019) and more recent models like Megatron-Turing NLG (with half a trillion parameters) use over a terabyte (Smith et al., 2022). These parameters must be updated during a training loop; achieving acceptable levels of accuracy also typically requires a large training set, so training a model involves a very large number of inference calls and parameter updates.

Hence, training is a very large up-front expense to using a DL model and when running or training a model, tensor operations tend to dominate performance. Reducing these expenses has been the focus of optimization efforts. In some cases, model designs incorporate features that allow them to train more quickly (converge in fewer epochs), as in the case of ResNet's identity mappings (He et al., 2016b), which is a subject beyond the scope of this dissertation. At the system level, there have been many efforts to produce highly optimized implementations of tensor operations that exploit hardware-specific properties or structures and parallelism within the operations themsleves, either writing these implementations by hand, as in NVidia's CuDNN library, or generating them from specifications using systems

like Halide (Ragan-Kelley et al., 2013), Tensor Comprehensions (Vasilache et al., 2018), or TVM (Chen et al., 2018b). Conversely, hardware designers have approached the matter of optimizing tensor operations from the opposite direction by implementing these operations in hardware to improve performance or energy efficiency, making use of parallelism and data transfer that are seldom available in software; these new devices (called *accelerators*), however, must be engaged from software, requiring DL libraries to support their interfaces.

Due to the computational intensity of DL training and inference, optimizations on different levels are very important for both practical use of models and research. While many DL applications involve large online systems, where training may be performed in a data center with a large hardware budget, many applications require inference to be performed on low-power devices like mobile phones or microcontrollers in IoT devices, meaning that performance may be very poor without optimizations. Within a data center, economies of scale apply as well: Savings during training might mean using fewer machines or appreciable differences in energy consumption. Even in a scaled-down research setting, training a model to completion can still be expensive and the model's effectiveness cannot be evaluated until it has been trained. Developing and applying new optimizations for DL models thus has the potential of improving the performance of many applications and enabling further research.

3 Differentiable Programming

The historical development of DL models from linear classifiers and neural networks (linear classifiers with non-linear activation functions that feed into each other) meant that many models are structured similarly. Namely, many models are described as static dataflow graphs, which are directed acyclic graphs where nodes correspond to tensor operators and edges correspond to input and output tensors. Some DL frameworks, like TensorFlow (Abadi et al., 2016), have built such assumptions about model structures into their APIs. A graph-like structure for models is easy to reason about and is also easy to automatically differentiate, so accordingly many optimizations assume such a structure as well and are phrased as operations on graphs. However, in principle, models are not required to be static dataflow

graphs of tensor operators: TreeLSTM (Tai et al., 2015) is one example of a model that descends recursively down tree-structured input data, applying a function and passing hidden state between the data. More recent examples of models with dynamic features include the Neuro-Symbolic Concept Learner (NS-CL) (Mao et al., 2019) for visual question-answering problems, which builds up a tree-like data structure representing a scene and processes it recursively in its forward pass, and the aforementioned AlphaGo (Silver et al., 2017), which combines learned parameters with a more traditional game tree search algorithm. Even if the dynamic portions of the application do not themselves directly use learned parameters, there are possibilities for co-optimization that DL compilers would be unable to pursue without some ability to represent and reason about dynamic logic. The popular DL framework PyTorch (Paszke et al., 2019a) has been widely adopted by DL researchers and practioners precisely for its "eager mode" of execution: Tensor operations are presented as API calls that are executed immediately (rather than first building up a graph and compiling it, as in TensorFlow) and dynamic control flow can be handled entirely as ordinary logic in Python. Indeed, TreeLSTM and the NS-CL were themselves implemented in PyTorch. PyTorch's ability to support dynamic control flow has also provided some opportunities for optimization, as in the recent project TorchDynamo (Ansel, 2022), a JIT that is able to compile API calls into PyTorch into optimized binary code while allowing dynamic logic to proceed as normally in Python. Further possiblities remain for optimizing and supporting dynamic DL models, as is discussed in the dissertation of my frequent collaborator Jared Roesch (Roesch, 2020).

This dissertation contends that describing DL models as programs in a general programming language (e.g., as expressions with types), rather than as mathematical objects with a specific (namely, graph-like) structure, facilitates the adaptation of general-purpose compiling techniques to the DL domain, simplifying many optimizations. This idea is rooted in the concept of "differentiable programming," the notion that DL models should be thought of as programs in a language that, in addition to the features expected of general-purpose programming languages, is *differentiable* (in the sense that it is possible to find the deriva-

tives of numerical programs in that language). A widely quoted 2018 Facebook post by Yann LeCun popularized the term. LeCun notes that dynamic DL models formed "a new kind of software by assembling networks of parameterized functional blocks and by training them from examples using some form of gradient-based optimization," which could be conceived as a program "very much like a regular progam [*sic*], except it's parameterized, automatically differentiated, and trainable/optimizable" (LeCun, 2018). This view is elaborated upon by Erik Meijer in his FSE keynote address (Meijer, 2018), in which he encourages DL researchers to describe models less as graphs but more as the composition of functions. Meijer also observes that DL models could be written and described like ordinary programs in general-purpose languages, as is supported by the earlier observations of TensorFlow developer Christopher Olah, namely the correspondences of various DL models to functional programming combinators (Olah, 2015a,b).

This dissertation concerns itself with another consequence of the framing of "differentiable programming:" Not only can DL models be *written* much like general-purpose programs; they can also be *optimized* much like general-purpose programs. That is, problems of DL system design and optimization can be presented as compilation and language-level problems. Many recent tools have provided infrastructure for differentiable programming, including the Lantern (Wang et al., 2018a) and Flux (Innes, 2018; Innes et al., 2017) libraries and support for automatic differentiation in Swift (Wei et al., 2020). In particular, the work detailed in this dissertation draws inspiration from the work others and I did on Relay, a domain-specific language (DSL) for DL applications that is explicitly designed to be a general-purpose, differentiable language, which is discussed in detail in Chapter 3. Namely, this dissertation presents specific examples of adapting traditional compilers techniques to the domain of deep learning, demonstrating how viewing DL models as "differentiable programs" (as the Relay project does at every opportunity) allows for creating infrastructure that can better support current applications as well as fostering emerging applications. The following two sections detail the nature of these contributions.

4 Runtimes for Dynamic Models: Dynamic Tensor Rematerialization

While static analysis is attractive because it includes guarantees of generality and does not impose overhead at runtime, dynamic analyses can be more precise because they can rely on information gathered at run time and therefore draw further conclusions than their static counterparts (Ernst, 2003). The lack of control flow in many DL applications makes static analysis particularly favorable, since it allows for avoiding many of the complexities that render static analyses on general programs less precise. However, many more recent DL models do include dynamic control flow, precluding many of the past static analyses based on dataflow graphs. Per the view of Ernst (2003) that dynamic analyses complement static analyses, this dissertation proposes to address this shortcoming of static analyses for DL by considering the alternative of dynamic analyses in the form of runtime systems. Because operations on large tensors are expensive and dominate the execution time of DL models, this provides many opportunities for sophisticated runtime systems for DL models: any auxiliary information would likely be much smaller than a tensor and the analyses performed would likely be much cheaper than a tensor operation.

The first contribution of this dissertation is a specific runtime system demonstrating this principle. This system is Dynamic Tensor Rematerialization (DTR), in Chapter 4, which is intended to generalize gradient checkpointing to more dynamic DL models by performing the analysis dynamically. Past gradient checkpointing approaches seek to decrease the memory required for training a DL model by recomputing intermediate values during backpropagation instead of storing them, but these approaches assume that models lack control flow and therefore that all recomputations can be scheduled in advance. DTR instead describes how a comparatively lightweight runtime system can dynamically make decisions about which intermediate values to free and later recompute, thereby allowing for memory savings on more dynamic models, ultimately attaining results comparable to optimal static techniques. In casting checkpointing, which has traditionally been implemented as a static analysis, as a dynamic analysis, DTR also demonstrates many commonalities between checkpointing

and software caching techniques, as well as with the traditional optimization of register rematerialization (Briggs et al., 1992).

5 *Supporting Diverse Hardware Back-Ends: 3LA*

The second contribution of this dissertation is a methodology for adding compilation support for new accelerators, featuring automatic search for opportunities to apply these accelerators' operations, motivated specifically by the rise of accelerators for specific operations in DL models. This methodology proceeds by modeling the semantics of applications and accelerators and searching over mappings of the model's execution between the different available devices. Compared to the present approach of building specialized compilation stacks for new devices or requiring developers to manually call APIs specific to the accelerators (which are presented as "hardware function calls"), this approach not only automates much of the present process for incorporating accelerators into DL systems but is also easily extensible with respect to both additional models and accelerators. We call this methodology "3LA" and present it in Chapter 5.

The commonality with traditional compilers lies in the form of the mapping from high-level applications (such as DL models) to the operations supported by accelerators. Specifically, accelerator operations in the 3LA methodology are represented as abstract *instructions* using the Instruction-Level Abstraction (ILA) (Huang et al., 2018a), casting the problem of invoking accelerator operations as one of choosing instructions appropriately corresponding to semantics of the application, much as in traditional instruction selection (Blindell, 2016). With a representation of the semantics of accelerator operations, it is possible to adapt many techniques previously used in the compilers domain to accommodate the use of accelerators in applications, as is realized in the prototype presented for the 3LA methodology, a DL model compiler that automatically detects opportunities to invoke accelerators. The prototype takes advantage of the ILA's representation of the semantics to allow for the use of formal verification techniques like bounded model checking to verify the correspondences between accelerator operations and operations in the source language, which also under-

girds the use of term-rewriting techniques (namely equality saturation (Tate et al., 2011)) to search for possible accelerator invocations by encoding correspondences between accelerator operations and source language operations as rewrite rules. The system is easily extensible as well, since new accelerator operations can be added simply by adding new rewrite rules and verifying the correctness of the changes can be approached incrementally, since it would be necessary only to verify the new rewrite rules. The fact that various general programming languages apporaches like instruction selection, term rewriting, and formal verification techniques combine to better facilitate the use of accelerators for DL applications *without any assumptions specific to the domain* demonstrates the potential of the broad interpretation of "differentiable programming."

6 Organization

The remainder of this dissertation is structured as follows:

- Chapter 2 surveys related work on DL systems and compiling techniques, particularly as related to DTR and 3LA;
- Chapter 3 discusses my past work on the Relay DSL and how it has informed the DTR and 3LA projects;
- Chapters 4 and 5 provide detailed descriptions of Dynamic Tensor Rematerialization and the 3LA methodology, respectively, as well as evaluation of their prototypes; and
- Chapter 6 gives concluding remarks.

Chapter 2
RELATED WORK

This chapter details past work that provides context for the specific techniques and approaches that motivate and guide the work presented on dynamic checkpointing and compilation to accelerators.

1 *Reducing Memory Required in Training*

Note: This section is partly adapted from the previously published work Kirisame et al. (2021).

This section surveys many techniques used to reduce the amount of memory required to traing DL models. Checkpointing is the a major focus of the work presented in this dissertation, but other important techniques for reducing peak memory have also involved swapping tensors between devices and modifications to models.

1.1 Checkpointing in Reverse-Mode Automatic Differentiation

The technique of checkpointing and the wider problem of saving memory while computing derivatives stem from decades of research in automatic differentiation (AD). AD techniques apply to training DL models (obtaining gradients for backpropagation), but AD has also been used in numerous other domains for decades, including other optimization problems and physical simulations.

As described in Baydin et al. (2015), AD refers specifically to automatically computing the derivatives of numerical programs (programs whose inputs and outputs are numerical) by producing a modified program. Baydin et al. (2015) specifically contrast the approach of AD

with other approaches of computing derivatives, namely manual, numerical, and symbolic differentiation. Manual differentiation requires a programmer to reason about the arithmetic of a given program and simply write another program that computes its derivative, which can result in very efficient implementations but is error-prone and demands careful manual effort. Numerical differentiation approximates derivatives for a numerical program by running the program and using its outputs. This approach is automatic and conceptually simple but has poor numerical stability and is computationally expensive. Symbolic differentiation applies symbolic derivation rules to arithmetic expressions to produce transformed expressions that compute their derivatives, thus automatically producing expressions that efficiently compute derivatives. However, this approach requires input programs to be closed-form arithmetic expressions and is vulnerable to exponential increase in the expression's size due to the symbolic application of the chain rule.

By contrast, AD approaches transform programs to compute derivatives alongside their normal behavior, handling general programs and avoiding exponential blow-up in the program size. However, AD-generated programs typically contain additional data structures and computations. This is particularly the case in reverse-mode AD, the most common form presently used, which Baydin et al. (2015) note Speelpenning (1980) first presented in truly automatic form. Reverse-mode AD computes partial derivatives with respect to each argument by applying the chain rule "in reverse" starting from the program's output. This process is usually implemented by running a "forward" sweep through the program (running it normally) while keeping a "tape" (a record of each computation performed with its arguments), computing partial derivatives by tracing the tape in reverse order and propagating intermediate values' partial derivatives one step at a time. This avoids exponential blowup of expression sizes, but requires additional computation and can incur considerable memory costs. As Dauvergne and Hascoët (2006) discuss, values needed in computing derivatives (i.e., those used more than "linearly") *must* remain in the tape from when they occur in the forward sweep until their entry is reached in the reverse sweep. Hence, in the worst case, the tape will store values for each operation in it, using space proportional to the number of

operations in the forward sweep.

Checkpointing is an optimization for reverse-mode AD that reduces the peak memory usage of the tape, as the dataflow analysis of Dauvergne and Hascoët (2006) illustrates. Checkpointing in reverse-mode AD proceeds by marking points in the tape called *snapshots*: Forward computations would be replayed to recompute the values needed at the next snapshot, saving (checkpointing) only the values needed to begin replaying computations and freeing the rest. Taking many snapshots would thus result in more values being saved but fewer computations, while taking few snapshots would require more computations. Griewank (1994) is one of the first publications[1] to note that it is possible to bound the computation-memory tradeoff in checkpointing for reverse-mode AD. Griewank (1994) introduces an algorithm called Treeverse for marking snapshots by a binomial partitioning scheme that used $\mathcal{O}(\log(T))$ times more additional computations to compute gradients in $\mathcal{O}(\log(T))$ memory, for a tape of length T in the unmodified program. Griewank and Walther (1998) provide a more detailed description of the Treeverse algorithm as well as empirical evaluation of the algorithm's performance. Note, however, that Treeverse's tape segmentation scheme requires knowing the length of the tape in advance and thus cannot save memory on programs with unbounded loops.

Hascoet and Araya-Polo (2006) describe certain design choices in the checkpointing for the Tapenade AD tool that allowed it to achieve sublinear memory overhead and support arbitrary programs, albeit suboptimally. Hascoet and Araya-Polo (2006) note that there are situations where the Treeverse algorithm performs poorly in practice because the costs of taking snapshots and different operations in code can vary greatly. Among the reasons for the poor performance are that Treeverse makes no cost considerations between different operations when selecting snapshots and that the overhead of managing snapshots can be expensive in practice. As a remedy, the Tapenade AD tool described by Hascoet and Araya-

[1] Grimm et al. (1996) cites a 1991 preprint by John Abbott and André Galligo called "Reversing a Finite Sequence" as also proving a bound on the tradeoff; however, I could not find a published work from those authors by that title.

Polo (2006) allows for user annotations to denote particularly expensive operations and employ different strategies (e.g., marking an entire region as cheap enough to recompute that it should never be stored in the tape). Additionally, Tapenade provides heuristics for handling loops with non-constant bounds, including options like snapshotting before or after given loops or snapshotting every N iterations. Checkpointing by the method of Hascoet and Araya-Polo (2006) thus requires iterative experimentation and manual intervention to achieve the best performance.

Siskind and Pearlmutter (2018) describe a checkpointing technique capable of handling arbitrary programs that does not require manual configuation as in Hascoet and Araya-Polo (2006). The checkpointing scheme of Siskind and Pearlmutter (2018) is built into a differentiable programming language (in the style of Pearlmutter and Siskind (2008)) and relies on three capabilities provided as language primitives: interrupting an operation, saving an interrupted operation to a "capsule," and resuming a capsule. Siskind and Pearlmutter (2018) describe an algorithm for creating capsules in an arbitrary user program that corresponds to creating snapshots in checkpointing, allowing for these points to be chosen at any execution point, including within loops. The interruption and capsule abstractions can moreover be implemented using continuations, thereby allowing for producing compiled programs using a continuation-passing style transformation. The algorithm's implementations did not surpass the performance of Tapenade in practice but allows for specifying finer-grained checkpointing policies than Tapenade.

1.2 Checkpointing in Deep Learning

Checkpointing techniques from general reverse-mode AD have been adapted to the particular considerations of training DL models. Because of the monolithic nature of tensor operations within DL frameworks, these expensive operations provide natural boundaries for taking snapshots. Additionally, the fact that most DL models have been represented as static dataflow graphs means that approaches similar to that of Treeverse are indeed viable, since these do not have unbounded loops. The expense of tensor operations and the sizes of tensors

also means that the size of the instructions in a snapshot and the overhead of invoking a snapshot (typically a function pointer) are comparatively insignificant, eliminating the performance issue Hascoet and Araya-Polo (2006) identify in AD on scalars. Chen et al. (2016) implement a Treeverse-like algorithm specific to dataflow graphs. This algorithm, called Gradient Checkpointing, divides a neural network into segments that correspond to intervals to recompute during backpropagation, analogous to snapshots in general AD. Chen et al. (2016) define a segmentation scheme that can train a feedforward network of N layers using $\mathcal{O}(\sqrt(N))$ memory with one extra forward pass ($\mathcal{O}(N)$ tensor operations). Chen et al. (2016) also provide a greedy algorithm for selecting segments on any given memory budget, training an N-layer network with $\mathcal{O}(\log(N))$ memory with $\mathcal{O}(N\log(N))$ additional tensor operations (similar to the bound achieved by Treeverse).

Several other works have followed the approach of Chen et al. (2016) in developing checkpointing schemes specific to dataflow graph models. Two recent examples are Jain et al. (2019) and Kusumoto et al. (2019). Jain et al. (2019) reduce the problem of selecting snapshots in a DL model to integer linear programming (ILP). Their tool, Checkmate, generates ILP constraints corresponding to the computation and storage costs of different segmenting choices and uses an ILP solver to finding provably optimal (with the fewest recomputations) snapshots for a given budget. The subsequent work Shah et al. (2021) similarly uses an ILP encoding but also allows for choosing between different implementations of tensor operations according to the cost model to optimize for both memory use and computational overhead. Kusumoto et al. (2019) provide a generalization of the segmenting approach of Chen et al. (2016), defining it for arbitrary directed acyclic graphs (Gradient Checkpointing assumes each layer can only depend on the layer immediately before it), and an efficient dynamic programming algorithm for quickly approximating an optimal choice of snapshots.

Gruslys et al. (2016) provide a checkpointing scheme specific to recurrent neural networks (RNNs), which feature dynamic control flow. An RNN loops over a sequence of inputs, applying a cell function to an input and a hidden state. The cell function produces a corresponding output as well as a new hidden state to propagate to the next iteration.

Gruslys et al. (2016) adapt a Tapenade-like approach to the loops, proposing schemes for taking snapshots both between loop iterations (saving the hidden state) and within the RNN cell itself (segmenting it as in Chen et al. (2016)), also proving bounds for the recomputations these algorithms perform on a given budget. The resulting algorithms perform similarly to Gradient Checkpointing on "unrolled" RNNs for the same $\mathcal{O}(\sqrt{N})$ budget but use fewer recomputations on smaller budgets.

DTR differs fundamentally from those approaches because it handles arbitrary dynamic control flow in models (making no assumptions about the model's structure, like Gruslys et al. (2016)) and operates online, giving it access to dynamically gathered information. In principle, a static checkpointing technique could be applied to a dynamic model "just in time" by unrolling the model on the fly, but some static analyses (like an ILP solver) can be too expensive to run each epoch. Unlike static approaches, however, dynamic planning introduces overhead at run time, which limits the analyses that DTR's heuristics can feasibly perform. Note that the Chen et al. (2016) greedy scheme and the `GreedyRemat` baseline in Kumar et al. (2019) are similar to DTR in that they greedily place checkpoints using a heuristic (albeit statically). However, their heuristics only use the sizes of tensors.

1.3 Deep Learning Memory Managers

Other work has enable the training of DL models on lower memory budgets by swapping tensors between GPUs or to host RAM. Huang et al. (2020) use a genetic algorithm to plan swaps between devices on static computation graphs. Capuchin by Peng et al. (2020) and Superneurons by Wang et al. (2018b), like DTR, use runtime systems and incorporate checkpointing as well. Capuchin's checkpointing phase, which resembles DTR's, uses dynamically gathered information for checkpointing; it performs a single batch without checkpointing (only swapping) and uses the costs it measures to determine where to set checkpoints. However, Capuchin's and Superneurons's checkpointing schemes assume a static model architecture (inferred from an initial profiling batch), which they use to plan recomputations in advance. Rajbhandari et al. (2020) present ZeRO (part of the DeepSpeed framework), another system

designed to reduce peak memory during training. ZeRO is designed for large distributed systems and focuses on partitioning models across devices (aiming to minimize the amount of replicated state needed to achieve this model parallelism), offloading activations to CPU when necessary. The approach of partitioning reduces the overall peak memory across the devices due to sharing, at the expense of some communication.

These works highlight that swapping and rematerialization are complementary approaches, raising the question of whether DTR can be combined with swapping while maintaining performance, since swapping systems like Capuchin rely on interleaving communication and computation at a low level. One possibility would be to assume a fixed swapping schedule and use DTR to replace the rematerialization schemes used by systems like Capuchin (perhaps given a constraint like treating values to be swapped out as unevictable). Another intriguing possibility would be to use swapping as a form of "eviction" in DTR, where the "cost" for swapped-out values would be the communication time. Swapping presents interesting tradeoffs with rematerializations since it may scale better than some tensor operators. However, incorporating swapping into DTR's online approach presents the problem of efficiently overlapping computation and communication since the runtime would need to guarantee that a computation scheduled concurrently with a swap would not need to swap values back in. This could greatly complicate planning (e.g., requiring some lookahead to avoid missed swapping opportunities) and would be fertile ground for future work. The more recent work DELTA by Tang et al. (2022), which cites the original presentation of DTR, explores these possibilities by using a heuristic to decide on the fly whether a given tensor should be evicted or swapped to CPU, making use of prefetching and preemptive evictions (evicting after most memory is occupied, even if the current allocations still succeed) to allow for overlapping communication and computation.

1.4 Memory-Efficient DL Model Modifications

Some recent work manually modifies DL models to perform similar computations using less memory, which may be used alongside checkpointing and swapping approaches. One

example is the use of reversible layers, which enable recomputing a forward value during backpropagation using the result of the following layer, an approach rather similar to general checkpointing. The activations for reversible layers do not need to be stored so long as the activation for the next non-reversible layer is stored, thereby building a computation-memory tradeoff directly into the model. Gomez et al. (2017) present RevNet, a variant of ResNet with reversible residual layers that achieves nearly identical accuracy to a ResNet with a comparable number of layers but only uses a constant amount of storage (with a linear amount of additional forward computations). Kitaev et al. (2020) present Reformer, a Transformer variant which also uses reversible residual layers to save memory. While the use of reversible layers in these models requires manual changes to the models, future work can explore how these can potentially be used alongside DTR.

2 Compiling to Accelerators

Note: This section is partly adapted from the previously published work Huang et al. (2022).

The 3LA methodology presented in Chapter 5 is the first work to provide general, extensible support for accelerators within a compiler stack, but other works have supported specific classes of accelerators. Additionally, the 3LA methodology relies on past work in hardware verification, software/hardware co-design, and term rewriting (particularly as applied to domains similar to deep learning).

2.1 Hardware Verification

The 3LA methodology relies at its core on having a representation of the semantics of accelerator operations. The Instruction-Level Abstraction (ILA) introduced in Huang et al. (2018a) provides a means for uniformly reasoning about hardware devices of diverse capabilities, motivated explicitly by the development of accelerators for deep learning and other applications. The ILA enables reasoning about accelerators in terms of the operations they provide (treated as abstract instructions) by providing a functional specification for each,

defining the instructions in terms how they affect the architectural state of the device. In the ILA, a device specification is given in terms of an abstract machine that fetches, decodes, and executes a sequence of instruction opcodes with some initial state and inputs, intended to correspond to the programming model of general-purpose processors (directly analogous to the Instruction Set Architecture, ISA). Thus a device is specified by defining opcodes, a procedure for fetching an instruction based on the opcode and state, a decode function that determines if the instruction is valid, and a function that computes the next state based on the current state and instruction. The ILA's state is intended to model the contents of registers and the device's other buffers, so the state is typically a collection of bit vectors and instructions' effects are expressed as changes to these bit vectors. ILA specifications can be recursive, allowing for specifying a set of micro-instructions and defining macro-instructions as a sequence of micro-instructions.

Huang et al. (2018a) note that ILA specifications are effectively labeled state transition systems and that their specification allows for specifying parallel hardware behavior as a *sequence* of instructions, "*a key enabler for system design and verification*" (emphasis theirs). The representation of program state in terms of bit vectors allows for ILA states and state transitions to easily be lowered to SMT and thus for verifying conditions on ILA states using bounded model checking. Huang et al. (2018a) presents verification case studies that involve proving equivalence between two different ILA specifications for the same device (one modeling each instruction monolithically and the other as a sequence of micro-instructions) and an ILA specification and RTL model (by generating an ILA specifiction from the RTL). Another case study used ILA specifications to generate faithful cycle-accurate simulators.

Huang et al. (2019) present in further detail a tool called ILAng for constructing ILA specifications, translating ILA instructions into hardware descriptions (e.g., in Verilog), and specifying verification conditions to check between ILA instructions, with a case study that verifies instruction-by-instruction equivalence of two specifications of an AES accelerator.

Huang et al. (2018a) claim that ILA is the first language for specifying and reasoning about accelerators using a sequential instruction abstraction. However, they note that pre-

vious high-level hardware synthesis tools did introduce the notion of specifying hardware in terms of state changes, particularly Bluespec (Nikhil, 2004) (which provides an operational semantics), but they lacked a uniform representation for both general-purpose processors and accelerators. ILA's instruction-based specifications generalize previous verification efforts for general-purpose processors like Jhala and McMillan (2001), which also proceed using instructions.

2.2 Equality Saturation

Another important component of the 3LA methodology is the use of pattern matching and term rewriting to identify opportunities to identify possible accelerator operations in an input program, potentially choosing between different possible operations. Note that this section focuses on non-destructive term rewriting because it avoids issues of phase ordering or having to devise a canonical representation; in the context of compiling to accelerators, a non-destructive approach is useful because it ensures that no opportunities to invoke accelerators can be missed (as may happen in a situation involving phase ordering with destructive rewrites). However, there are many works, such as Newcomb et al. (2020) and Liu et al. (2022a), that successfully use destructive term rewriting for compilation and optimization purposes and avoid phase-ordering issues either by requiring rewrite rules to be manually applied or including carefully defined invariants that rewrite rules must uphold.

Searching over equivalent programs as a means of producing optimized code is the basis of the "superoptimizer" in Massalin (1987). This early superoptimizer takes a source program and uses a brute force search over all assembly code sequences up to a threshold of length and returns the shortest sequence that is equivalent to the source program (using either a probabilistic test or a boolean satisfiability test). The search space for this superoptimizer grows very quickly, so this approach can only generate very short sequences of instructions; Massalin (1987) intends it primarily for peephole optimizations.

Joshi et al. (2006) refine the approach of Massalin (1987) with an improved method of generating search candidates (avoiding many clearly unpromising programs, for which Massalin

(1987) only offers heuristics) and also generated programs that are known to be equivalent by construction instead of having to check equivalence. Joshi et al. (2006) present a tool called Denali, which uses algebraic identities between program operations as rewrite rules and identifies possible substitutions. In order to search efficiently over these possible rewritings, Denali uses a data structure introduced by Nelson (1981) called an e-graph, in which a program is represented as a DAG of terms and rewrite rules can be represented by the addition of nodes and edges corresponding to new instructions that compute equivalent values to a given subtree. An e-graph thus allows for non-destructively expressing all possible applications of rewrite rules, which can compound by taking advantage of other rewrites in the graph. Denali proceeds by applying rewrite rules until the e-graph no longer changes, thereby deriving all possible rewrites. Denali finally chooses an optimal rewriting by reducing the equivalence classes in the graph to SAT queries corresponding to the number of cycles needed to perform the corresponding operation, finding the largest k for which a query of the form "the encoded program cannot be computed in k cycles" is unsatisfiable.

Tate et al. (2011) develop an approach inspired by that of Joshi et al. (2006) that allows for optimizations to be performed across larger program structures, rather than within loop bodies or short segments of code (Denali's intended use), in addition to validating transformations performed by program optimizers. The approach of Tate et al. (2011) defines an IR for programs called a "Program Expression Graph" (PEG), an SSA-inspired representation of computations as dataflow graphs with control (Φ) nodes, and an e-graph over PEGs called an E-PEG that includes equality edges between equivalent sub-PEGs, allowing rewrite rules to be applied similarly. Because PEGs include control nodes, it is possible to define rewrite rules that act on loops and other structures, which Tate et al. (2011) develops in great detail. Crucially, Tate et al. (2011) notes that local transformations on an E-PEG can result in cascading non-local effects in terms of the potential code generated from the E-PEG, so the representation efficiently accumulates all possible rewrites and can examine transformations of large parts of a program. The algorithm described applies rewrite rules to an E-PEG until no new rules can be applied, thereby *saturating* the E-PEG, which is signified by the name

"equality saturation."

Willsey et al. (2021) present `egg`, a library providing a general and efficient library for applying equality saturation. `egg` is able to improve its performance in part due to a technique that Willsey et al. (2021) call "rebuilding," which involves allowing invariants to be temporarily violated when merging e-classes (transitive closures of nodes connected by equality edges) before later restoring them after all the merges are complete. (Crucially, `egg` separates the "read" phase, for detecting rule matches, from the "write" phase, in which new nodes and edges corresponding to rewrites are applied. This means that invariants may be violated in the write phase so long as they are restored before the next read phase, preventing incorrect applications of rewrite rules.) Separating the enforcement of invariants from adding and merging e-classes results in large speedups by eliminating repeated checks. `egg` also allows for domain-specific reasoning to be applied in the form of analyses on e-classes, which are functions that potentially modify e-classes (generally through the addition of nodes). Analyses in `egg` allow optimizations like constant folding to be implemented directly and act faster compared to relying on multiple rounds of rewritings to produce such behavior "emergently." `egg` has been used by many subsequent works involving equality saturation, including Ruler (Nandi et al., 2021), a system for inferring rewrite rules, and domain-specific compilation tools involving rewrite rules, which will be discussed below.

2.3 Term Rewriting for Tensor Programs

The use of term rewriting and equality saturation in the 3LA methodology relies on having a program representation over which rewrite rules can be defined. Many works, such as Nandi et al. (2020) for 3D model designs, have defined domain-specific representations intended to faciliate term rewriting. Though the broader 3LA project as described in Chapter 5 is not intended to target compilers of any one domain, in the DL context, the programs of interest operate over large tensors. Several past works have applied term rewriting and equality saturation for optimizing tensor programs and linear algebra kernels.

The Halide compiler for image processing kernels (Ragan-Kelley et al., 2013) is widely

used and is very well-known for its separation of "compute" (a definition of how each element of the output tensor should be computed based on elements of the input tensors) and "schedule" (the iteration order for elements), allowing for loop optimizations like reordering, unrolling, and fusion to be defined as schedule transformations. Note that the same approach has also been adopted by the TVM DL compiler stack (Chen et al., 2018b), since tensor operations in DL are in many regards similar to the image processing kernals that Halide optimizes. Halide includes a term-rewriting system for optimizing kernels (especially schedules), which is described extensively in Newcomb et al. (2020); the approach uses destructive rewrites with numerous invariants to ensure the rewrite system always terminates. These rewrites yield strong performance gains in practice. While Halide and many similar compilers attain success using destructive rewrites, SPORES (Wang et al., 2020) is an optimizing compiler for linear algebra kernels (also closely related to image processing) that uses equality saturation. SPORES relies on transforming linear algebra kernels into relational algebra, which has a smaller set of axioms and requires only thirteen rewrite rules, ultimately resulting in faster compilation and superior performance compared to the destructive rewriting baselines.

Note that while Halide uses term rewriting for simplifying arithmetic expressions or verifying that preconditions for optimizations have been met, Halide's scheduling directives like loop tiling and fusion are not themselves implemented using term rewriting. By contrast, Liu et al. (2022a) present a term rewriting system for tensor computations that is capable of implementing Halide's scheduling transformations through rewrites. Their representation is purely functional and relies on pipelining with a special tensor comprehension operator, which creates a tensor by mapping a function over indices; they formalized the language in the Coq proof assistant and proved the correctness of the rewrite rules.

In addition to optimizing individual linear algebra kernels like Halide and similar works, some DL compilers also apply rewrite-based optimizations over entire DL models; these are usually called "graph substitutions," since the models themselves are represented as dataflow graphs, where nodes are tensor operators. Jia et al. (2019) present TASO, a DL graph

optimizer that derives new graph substitutions through enumerating all operator graphs up to a small size bound, filtering out potential substitutions by testing if the two graphs produce the same result on random inputs, and finally verifying the most promising generated solutions against a set of 43 correctness properties using an SMT solver. The later work of Yang et al. (2021) applies equality saturation (implemented using `egg`) to the approach of TASO, yielding a tool called TENSAT that derives graph substitutions from a set of smaller graph rewrite rules (adapted from TASO's 43 correctness properties). TENSAT finds graph substitutions much more quickly than TASO because the enumeration is handled by the e-graph, hence avoiding any invalid substitutions and obviating the need for random tests.

The previously discussed works have considered widely available programmable devices (namely, CPUs and sometimes GPUs), but other works have also used term rewriting to compile code to accelerators. One such work is Diospyros (VanHattum et al., 2021), a compiler for linear algebra kernels to digital signal processors (DSPs), which, like many accelerators, provide vectorized instructions (individual instructions operating over entire vectors) and attain the best performance when loops in the source program are compiled to single vectorized instructions. Diospyros uses symbolic evaluation to extract abstract mathematical definitions of imperative input programs, converts those into a DSL of vector expressions, and uses equality saturation to explore the space of possible transformations within the vectorized DSL. The compiler includes a core set of rewrite rules likely to be useful in general, as well as rewrite rules specific to individual DSPs, which have differing buffer sizes and other hardware properties. The final result of the equality saturation (extracted using a cost function based on minimizing data movement) is then converted into device-specific C++ code; in the case of Diospyros, the extracted benchmark code closely matched the performance of previous baselines or exceeded it.

The tensor program DSL used in the 3LA prototype is Glenside (Smith et al., 2021), which is designed to faciliate equality saturation over tensor programs (implemented using `egg`), especially for the task of lowering code to accelerators. Like the kernel language of Liu et al. (2022a), Glenside defines tensor operations in terms of functional combinators

(such as maps and folds), but crucially restricts expressivity by avoiding bindings (wherever names are introduced, such as function arguments and `let` bindings), since tracking scope for names is difficult to perform efficiently in equality saturation. Despite these limitations, many common tensor operators can be concisely expressed in Glenside thanks to its core abstraction of *access patterns*, a feature of Glenside's type system that allows for easily expressing iteration domains over tensor indices without the use of bindings. An access pattern is a tuple of two tensor shapes (D, I), where D defines the dimensions of *iteration* and I defines the dimensions of *computation*. In particular, for each index in D, there is a tensor with shape I that can be processed; for example, a tensor of shape $(2, 3, 4)$ can be represented with (among others) the access patterns $((2), (3, 4))$, $((2, 3), (4))$, or $((2, 3, 4), ())$. The first of these access patterns corresponds to accessing two tensors of shape $(3, 4)$ (i.e., a 3×4 matrix) in sequence (i.e., arranged as a vector of length 2); the second, to accessing vectors of length 4 laid out like a tensor of shape $(2, 3)$; and the third, to accessing scalars laid out like a tensor of shape $(2, 3, 4)$ (i.e., iterating over an ordinary tensor of that shape). Using access patterns, many functional programming combinators can be restated in ways that make it easy to operate over tensors and still allow the combinators to easily compose. Glenside includes rewrite rules over these combinators and other built-in operations, which allows for many optimizations to be derived without explicitly defining them. In particular, Smith et al. (2021) use equality saturation to discover opportunities to apply an accelerator, namely by defining accelerator operations as Glenside expressions (adding a rewrite rule of the form `expression` $\Longrightarrow$ `accelerator_operation`), then using equality saturation to discover if there is any sequence of rewrites that yields uses of that operation. Smith et al. (2021) perform a case study with a rewrite rule for a systolic array (the TPU is a notable example) operation, which results in mapping a convolution operation to a systolic array by rederiving the `im2col` transformation (Chellapilla et al., 2006); the use of Glenside for "flexible matching" in Chapter 5 is a generalization of this case study.

2.4 Software/Hardware Co-design

Recent work on accelerator generation and integration (Bahr et al., 2020; Truong et al., 2020) has explored adding support compiler flow for specialized Coarse-Grained Reconfigurable Array (CGRA) accelerators in Halide. That work composes an impressive array of custom tools to generate and verify specialized CGRA accelerators and also map Halide program fragments down to accelerator invocations. HeteroCL (Lai et al., 2019) also provides a similar custom flow. By contrast, the 3LA methodology is designed to support software/hardware co-design by mapping from high-level DSLs and *near-arbitrary* accelerators; because of the flexibility of the ILA, the 3LA methodology is applicable to a broader class of compilers and accelerators.

2.5 Pattern Matching Accelerator Calls

In principle, many DSLs allow for supporting custom accelerators via bespoke translations from DSL operators to specific accelerator APIs, e.g., as in TVM's built-in support for VTA (Moreau et al., 2019). TVM's BYOC interface (Chen et al., 2021) eases incorporating custom accelerators by performing syntactic pattern matching to offload computations via user-provided code generators. However, BYOC leaves all matters of code generation to the user, while 3LA provides more structure to code generation via the ILA. In particular, the ILA provides useful simulation and verification capabilities. Additionally, BYOC's pattern matching cannot search the space of programs *equivalent* to the input, limiting the number of potential accelerator invocations compared to flexible matching in our 3LA prototype.

The MLIR framework (Lattner et al., 2021) provides a rich metalanguage and numerous tools for developing, optimizing, and translating between custom compiler IRs, but does not inherently provide direct support for 3LA's features, though it would be possible to do so using custom MLIR dialects. Past work has also explored rewrite-based techniques for automatically inferring instruction selection passes between ISAs (Ramsey and Dias, 2011). Rewriting in 3LA instead operates on a high-level DSL to expose opportunities to invoke

code generators, rather than performing low-level code generation directly.

2.6 Validating and Verifying Accelerator Calls

Tools like Verilator (Verilator, nd) and Cuttlesim (Pit-Claudel et al., 2021) enable efficient RTL-level simulation, but do not provide reusable interfaces or flexible matching to incorporate custom accelerators into existing compiler flows. Formally verified compilers such as CompCert (Leroy, 2006) and CakeML (Kumar et al., 2014) can rigorously establish end-to-end equivalence from high-level source code down to assembly for various CPU back-ends via machine-checkable proofs, but currently do not provide a general approach for integrating new accelerator support and provide no support for custom numerics. By contrast, 3LA enables validating accelerator mappings via end-to-end simulation handling custom numerics and formally verifying individual rewrite rules from compiler IR patterns to accelerator invocations.

Chapter 3

RELAY: A HIGH-LEVEL IR FOR DEEP LEARNING APPLICATIONS

My past work on the design, implementation, and evaluation of the Relay IR informed the motivation and approach of the primary subjects of this dissertation. While Relay itself is not a subject of this dissertation, the process of designing and presenting the language as well as the subsequent work in maintaining it in the context of the Apache TVM project, which has given Relay many industrial and academic users, have shaped my understanding of "differentiable programming." Hence, this chapter provides further background on the motivations and design of the Relay IR.

1 *Design of Relay*

Relay is the front-end intermediate representation (IR) for the TVM (Chen et al., 2018b) deep learning compilation stack, illustrated in Fig. 3.1. It is used to represent models directly defined by users or converted from interchange formats like ONNX and Keras. Relay

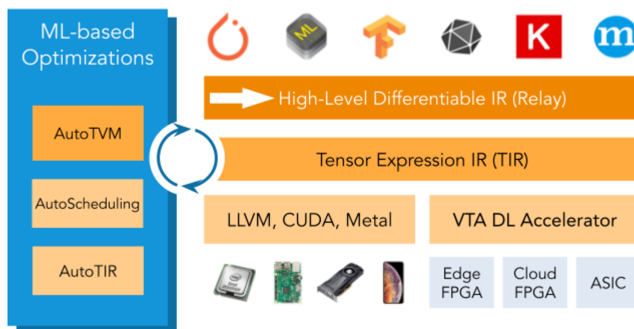


Figure 3.1: A diagram of TVM's architecture with Relay's position marked.

programs can, in turn, be deployed to various platforms by the lower levels of TVM: The programs can be interpreted through a "graph executor" runtime, compiled into bytecode for a virtual machine (Shen et al., 2021), or compiled directly into LLVM, CUDA, or C (for microcontrollers). The supported back-ends for Relay programs also include built-in support for deployment the VTA accelerator (Moreau et al., 2019) and support for user-provided accelerator extensions (Chen et al., 2021). The AutoTVM (Chen et al., 2018c) mechanism or successors like Ansor (Zheng et al., 2020) can also be used to optimize the tensor operators within a Relay program to the given hardware.

The design of Relay, introduced in Roesch et al. (2018) and described in further detail in Roesch et al. (2019) and Roesch (2020), explicitly takes inspiration from the discourse of "differentiable programming," providing an expressive language in order to separate low-level implementation details from the high-level descriptions of computations. Since the design of Relay has been extensively documented and described in the aforementioned works, I include an example of Relay in Figure 3.2 rather than reproduce the operational semantics and typing rules in full. In particular, in order to enable the greatest generality for expressing DL applications, Relay expresses models as programs in a functional programming language much resembling SML (Milner et al., 1997), with many features expected from a general-purpose programming language, such as recursion, first-class functions, and algebraic data types like lists and trees. Unlike a general-purpose functional language, however, Relay includes special support for tensors and tensor operators: all numerical values in Relay are TVM tensors and Relay programs can invoke TVM's predefined tensor operators by name. To support training for models with dynamic control flow, Relay implements reverse-mode automatic differentation (AD) as a source code transformation, similar to the approach of Lantern (Wang et al., 2018a), but using ML-style references to maintain a tape instead of `shift` and `reset` combinators. Relay's AD implementation is "higher-order" in two senses: the algorithm can handle branching, recursion, and *higher-order* functions and can also compute *higher-order* derivatives (by taking in a Relay program that computes a gradient), which are useful in meta-learning applications.

```
def @lin(%x, %w, %b) {
  nn.dense(%w, %x) + %b
}

def @relu_cell(%w, # weights
               %b, # offsets
               %s, # state
               %x  # input
) {
  let %x2 = @lin(%x, %w.0, %b.0);
  let %s2 = @lin(%s, %w.1, %b.1);
  (%s, nn.relu(%x2 + %s2))
}

def @trained_relu_cell(%w, %b) {
  fn(%x, %h) {
    @relu_cell(%w, %b, %x, %h)
  }
}

data List<a> {
  Nil: () -> List[a]
  Cons: (a, List[a]) -> List[a]
}
```

```
def @foldl<a, b>(
  %f : fn(b, a) -> b,
  %z : b, %l : List[a]) -> b {
  match(%l) {
    case Nil() { %z }
    case Cons(%h, %t) {
      @foldl(%f, %f(%z, %h), %t)
}}}

def @encode<state_t, in_t, out_t>(
  %cell : fn(state_t, in_t) -> (
     state_t, out_t),
  %input : List[in_t],
  %init : state_t) -> state_t {
  @foldl(
    fn(%state, %in) {
      %cell(%state, %in).0
    }, %init, %input)
}

@encode(
  @trained_relu_cell(
    %weights, %offsets),
    %input, %init)
```

Figure 3.2: An example of Relay code, namely an implementation of a recurrent neural network following Olah (2015b). This example employs many of Relay's features. `@lin` is a linear layer, applying `nn.dense` (dense matrix multiplication) and elementwise addition. No tensor shapes are annotated, but Relay's type inference will infer the shape of the result using the operators' type relations (matrix multiplication's relation encodes broadcasting semantics (Contributors, 2019) and addition's relation is identity). `@relu_cell` defines a ReLU (rectified linear unit) RNN cell, taking in a tuple of weights and biases (the dots indicate tuple indexing) and returning a tuple literal. `@trained_relu_cell` returns a closure (note the lexical scoping). Note the parametric polymorphism employed in the definition of the `List` type and implementation of `@foldl`, which are common in functional programming languages (and part of Relay's standard library). `@encode` uses `@foldl` to implement a generic encoder RNN by folding a cell function over a list of inputs, demonstating how Relay's functional programming features can implement a deep learning model in a modular, reusable manner.

Relay's type system is its main domain-specific adaptation, as it incorporates reasoning about tensor shapes in order to make shape information statically available for domain-specific optimizations. Hence, Relay's tensor types explicitly include the shapes and datatypes of tensors (rather than, for example, hiding size information from the type system as in some general-purpose languages like Java and C++).[1] DL applications present some complexity regarding tensor shapes, as commonly used tensor operators in DL frameworks may produce outputs of different shapes depending on the shapes of the inputs (for example, elementwise operations tend to have "broadcasting" semantics, meaning that if one input is larger than the other, the smaller input is copied out, or "broadcasted," to the larger size when possible). To support reasoning about the shapes of the results of tensor operators, Relay assigns *type relations* to all tensor operators, which declare the relationship between the input and output types (and therefore also the tensor shapes), constituting a lightweight form of dependent typing. The type relations are handled during type checking as constraints, which are first gathered during type inference (unification) and then passed to a constraint solver, which either finds a solution to all the constraints (success) or fails to solve some constraints (failure, meaning a shape is wrong or more annotations are needed). In principle, these constraints could be expressed in some logical domain for a solver; since most constraints in tensor operators are simple equalities or inequalities, the actual implementation of Relay's type checker simply includes imperative procedures (in C++) that examine the shapes of the arguments and results and either assign underspecified type variables to concrete ones or fail to unify and declare failure. While somewhat ad hoc, this implementation has sufficed for the dozens of tensor operators presently in Relay.

Language-level optimizations in Relay allow it to achieve performance comparable or superior to that of widely used frameworks on a variety of models in Roesch et al. (2019), despite Relay's additional complexity compared to static dataflow graphs. Rather, Relay's

---

[1] Relay also permits tensors with dynamically checked shapes, which include dimensions labeled `Any` to indicate that they should be checked at run time. This feature was not included in the initial presentation of the language, though it is discussed in Roesch (2020).

expressive features allow for many compile-time optimizations, preserving high-level program information for optimizations to use. Many optimizations use tensor shape information from the type system to change layouts for operators or perform memory planning. A particularly profitable optimization is Relay's automatic operator fusion pass, which recognizes chains of tensor operator calls and uses TVM's lower-level loop fusion faculty to replace these with a single fused operator. Relay is also designed to easily support writing further optimizations by providing a general interface for optimizations inspired by LLVM, namely as source-to-source program transformation passes. Since the output of an optimization is another Relay program, optimizations can easily compose with each other, allowing for adding optimizations to recognize specific structures in a program and handle those separately. For example, Relay's AD capability is implemented as an "ordinary" pass rather than as a language primitive—the fact the AD pass accepts any Relay program and returns a Relay program is a key reason for its generality. The extensible optimization interface thus provides a relatively simple means for adding program transformations to exploit domain-specific (or even model-specific) properties, allowing for improved performance without modifying the high-level descriptions of models, ensuring portability.

2 *Design Advantage: Type-Directed Relay Fuzzing*

Note: This section has been adapted from an unpublished class assignment Flanders et al. (2021).

While the generality and expressiveness of Relay's programming model has directly guided some later research, such as Nimble (Shen et al., 2021) or the 3LA work discussed in Chapter 5, the presentation of the language has also been helpful for the testing and development of the compiler itself. In particular, Relay's systematization of operators' tensor shape properties in terms of type relations allowed my collaborators and me to design and implement a fuzzer for the language (Lyubomirsky et al., 2021) that is able to generate Relay programs using almost all of the language's features, including operator calls. The programs generated

by the fuzzer are guaranteed to type check in Relay, which means that the fuzzer must ensure that the tensor shapes passed to operators are valid as programs are generated. While the task of reasoning about tensor shapes is, in principle, a complex one, the presentation of type relations in Relay for reasoning about tensor shapes suggested several possible approaches for automatically generating programs that fulfilled the constraints. The TVM community is in the process of discussing whether to adopt this fuzzing approach as part of its testing infrastructure (Lyubomirsky, 2022). The following sections give further background on fuzzing and details on the design of the fuzzer.

2.1 Fuzzing Overview

Debugging a compiler can pose significant difficulties because compilers make a strong guarantee to users—that they will accept any valid program and faithfully implement its semantics, including large programs with complex functionality. Identifying internal errors at an early stage in a compiler's development and narrowing down their causes can make it easier for compiler developers to fix them and devise regression tests for the future.

Though the space of potential programs is infinite, the programs themselves are structured and hence amenable to random generation. This approach is known as *language fuzzing* and is a commonly used test generation technique for compilers (Liang et al., 2018). CSmith (Yang et al., 2011) is a very well-known fuzzer for C and C++ programs, following the rules of the language specification to ensure that programs it generates will be standards-compliant. So long as the generated programs are valid, any errors detected during compilation indicate errors in the compiler rather than bugs in the programs; test oracles that might be applicable are simply ensuring that the compiler will not crash, that (if the language has strong typing) the type system's guarantees are never violated at run time, or that compiling the same program on different settings should not change the observable behavior.

A significant challenge for compiler fuzzing is "format validation"—initial compiler checks that the input is well-structured. Programs that fail such basic checks have no chance of revealing bugs deeper in the compiler. Any approach to language fuzzing must overcome this

barrier, either by starting with a valid program or by generating a program using a grammar and a set of generation rules. For example, some works like Holler et al. (2012) and Lemieux and Sen (2018) have taken approaches that combine or mutate known-buggy programs to produce new buggy programs. Other works, like Dewey et al. (2015) and Chandra and Bodik (2017), encode validity conditions in logical specification languages and employ solvers to ensure that generated programs fulfill the necessary criteria.

2.2 Reasoning about Tensor Shapes

In the case of DL frameworks, tensor shapes present a significant format validation challenge, since most operators will only accept shapes in particular formats and errors will typically only be detected at run time (generally because most DL frameworks have front-ends in dynamically typed languages like Python). Works such as ShapeFlow (Verma and Su, 2020) can detect shape errors using a dynamic analysis, but it would be expensive to generate programs entirely at random and filter them using such analyses afterwards. Generating *a priori* valid tensor programs is more challenging, since it requires tracking shape constraints as the program is built up (without full knowledge of the final program). The type relations in Relay, however, allow for statically determining if shapes are valid—this gives guidance for generating programs operating on tensors that are guaranteed to be valid.

However, even though Relay's type relations allow for statically reasoning about tensor shapes, they present other challenges for fuzzing. Namely, the type relations are opaque to the type system: They simply assert that some property holds over the input types, the output type, and potentially compile-time parameters (in real tensor operators, these parameters might be an axis dimension or a pooling policy), but provide no programmatic way of inspecting what that property is. As a result, the Relay type-checker makes no assumptions about what information type relations use or what properties they check—indeed, they are implemented using developer-written C++ functions that directly check properties and update any underspecified types. The Relay type-checker combines solving type relations with type inference by running the checkers continually until reaching fixpoint. This method

of solving poses a further difficulty to implementing a fuzzer, since this method assumes a complete program (whereas a fuzzer would be building up a program) and does not provide any way of determining whether adding a new relation would introduce an inconsistency, except for running all the gathered relations to fixpoint again. Another difficulty of this last scenario is that if adding another type relation introduces an inconsistency, the implementations of type relations do not give any information about what new type relations would succeed; they only reject those that do not. Generating valid Relay programs thus requires being able to reason about type relations and, in particular, constructively find expressions that will satisfy them.

2.3 Fuzzer Implementation

To implement the fuzzer, we rely on Relay's typing rules and follow the overall approach of Fetscher et al. (2015) to construct programs by following typing rules "in reverse." For dealing with the challenges posed by type relations and operators, we consider multiple approaches generally inspired by the cosntraint-based fuzzing approach detailed in Dewey et al. (2015). The initial prototype presented in Lyubomirsky et al. (2021) is publicly available at `https://github.com/slyubomirsky/relay_fuzzer` and an updated version being reviewed in TVM's RFC process is avilable at `https://github.com/slyubomirsky/tvm/tree/fuzzer-poc`.

The fuzzer proceeds by starting with a type and generating an expression that satisfies it, performing this procedure for each subexpression according to the typing rules. The program is thus built up "in reverse" starting from its return type—the expression generator is used to produce expressions matching the return type. This may require introducing more types. For example, if the generated return type of the program is `T1` and the expression generator intends to produce a let expression that fulfills this type, it will produce `let %a : T2 = ?b in ?c`, where `%a` is a fresh variable, `T2` is a newly generated type, `?b` is an expression of type `T2` (requiring another call to the expression generator), and `?c` is an expression of type `T1` in which `a` may appear (the expression generator must track which

variables are in scope).[2] The remainder of this section will discuss the implementation of the fuzzer in greater detail, as well as its approaches for handling type relations in Relay.

For expressions other than operators (therefore not requiring any type relations), generating expressions to match a given type by the method of Fetscher et al. (2015) is generally straightforward. All types other than algebraic data types (which will be discussed below) have literals, so for any type, there is always at least one possible expression of that type and therefore termination can be guaranteed by forcing the expression generator to produce a literal. Thus, for any types `T`, the expression generator may choose to produce a literal of that type, a variable that is in scope of type `T`, or a connective expression like a let binding, a conditional, etc., that has the final type `T`. Connective cases are generally recursive and require further calls to the type generator. Using the previous example of generating a let binding with a final type of `T1`, the expression generator proceeds by creating a fresh local variable `%a` that will be assigned to a new type `T2` (generated by the type generator). The final expression `let %a : T2 = ?b in ?c` will be produced by generating an expression `?b` of type `T2` and an expression `?c` of type `T1`, where `%a` is in scope and may be used. Below a few subtler cases are discussed in detail.

The fuzzer is capable of generating programs using all the language features of Relay, except for the following restrictions:

- All types are fully annotated.
- All shapes are concrete (do not contain dynamically checked `Any` dimensions).
- Only algebraic data types and polymorphic functions included in the standard library are included; the generated programs do not define new ones.

The below sections will briefly discuss how these restrictions could be addressed as well.

---

[2] Regarding notation, we will use `?` as a sigil to indicate a metavariable, one standing for an expression rather than a program variable (since Relay uses `%` and `@` sigils, the `?` sigil will adequately distinguish).

2.3.1 Algebraic Data Types (ADTs)

ADTs present some corner cases for expression generation. An ADT in Relay is defined as a sequence of constructors that take argument types and produce a member of the ADT. ADTs may take type parameters (for example, `List[a]`), which appear in the constructors, and, crucially, ADT constructors can be recursive, meaning that an argument to a constructor can be a member of the ADT (for example, a `List` ADT is built up by applying the `Cons` constructor to an existing list and a new member). Though an instance of an ADT (a "literal") in some sense can be given as a call to the constructor, it cannot always be guaranteed that a valid call can be constructed. For example, suppose there is an ADT `B` where all constructors of `B` take an argument of type `B`—it is impossible to construct an instance of `B` without already having one. Additionally, because ADT constructors are recursive, naively choosing constructors at random is not guaranteed to terminate: in the example of a `List` ADT, it is possible to generate an infinite stack of `Cons` constructors. Thus, ensuring termination will also require choosing a non-recursive constructor.

In principle, it would not be difficult to check whether a given ADT is possible to instantiate and only permit creating terms of that type if there is a variable in scope with that type, nor would it be difficult in general to find a non-recursive constructor to use for forcing termination. For simplicity, we considered only the ADTs in Relay's standard library (SML-style lists, option types, and Haskell-style trees), which can all be instantiated and for which we hard-coded the choices of non-recursive constructors (an empty list, an empty option, and a tree leaf).

2.3.2 Pattern-Matching

Relay pattern-matching expressions are based on those in SML and OCaml and consist of a value being matched and a series of clauses (pairs of patterns and expressions). Relay features a pattern language comprised of wildcards (match any value), pattern variables (match any value and bind it to a variable, scoped to the clause), a constructor-matcher (gives

an ADT constructor and subpatterns and matches an ADT with the same constructor for which all constructor arguments match the corresponding subpattern), and a tuple-matcher (checks that all subpatterns match all tuple members). A match in Relay by default must be *complete*, meaning that the matching patterns must be able to match any instance of the value type; match completeness is checked during type checking. This means the expression generator must be able to generate complete match expressions.

As a simplification, our prototype generates patterns at random from the pattern language (based on the given type: tuple patterns for tuple types and constructor patterns with the appropriate constructors for a given ADT) and finally appends a wildcard pattern to guarantee completeness. In general, it would be possible to generate patterns guaranteed to be complete using an approach similar to that of the Relay match-completeness checker, which enumerates possible pattern-match cases by "expanding" holes in the patterns if they are not specific enough to determine whether a given clause matches or rejects it. The completeness checker raises an error if some case is not matched by any pattern, but a generator could instead use such a case to create a new pattern.

2.3.3 Polymorphic Functions

The present generator does not produce new polymorphic functions (and the type generator does not output such types), but in principle, it would not be very difficult to produce them. The only difference from producing any other expressions is that there is no way to produce a literal for a type parameter: there must be a variable in scope that contains a value of that type (e.g., a tuple or ADT that has it for a member) or a function whose return type contains a value of that type; these conditions would be feasible to check, though the present fuzzer does not perform such reasoning.

However, the expression generator does produce calls to the polymorphic functions included in Relay's standard library (e.g., `map`, which is of type `fn<a, b>(f: fn(a) -> b, l: List[a]) -> List[b]`). This is implemented by checking whether it is possible to instantiate the function type against a given return type (whether there exists an assignment of type

variables that will produce that return type), where any unmatched variables are assigned to a new random type. To avoid type inference ambiguities, all type arguments to calls are specified explicitly.

2.3.4 Addressing Type Relations

To be compatible with the general scheme of following typing rules "in reverse," we must *solve* type relations in the following sense: Given a return type T and relation R, assuming that T is a possible return type R, we must find input types $I_1, \ldots, I_n$ such that $R(I_1, \ldots, I_n, T)$ holds. The solutions (the $I_1, \ldots, I_n$) need not be unique, though in practice there may be an interest in having policies to decide which solutions should be prioritized (this was not explored in the initial prototype).

For determining if an operator call is possible for a given type (i.e., if a type T is the possible return type of relation R for some input types), we found by inspection that TVM operators tend to output single tensors (or in the case of batch norm, a tuple of three tensors) of a specific rank (number of dimensions) and were generally capable of outputting a tensor of any shape with that rank. Thus, checking whether an operator call could be matched to a given type simply amounted to checking whether the type was a tensor type and the shape had the appropriate rank. For generality, the prototype allows for registering an arbitrary "recognizer" function that takes a type and determines if it is a valid return type for a given operator.

For the prototype, we considered considered several common Relay type relations that are associated with certain basic tensor operators and that are manipulated in TVM's standard optimization passes, such as elementwise multiplication, bias addition, matrix multiplication, and 2D convolutions. We also included batch norm as an example of a Relay operator that returns a tuple rather than a single tensor. Since this was intended only as an exploratory prototype, we implemented direct support only for 22 operators typed according to 7 relations (given in Table 3.1), chosen based on the complexity of their type relation implementations in C++ and on their being affected by the standard passes. This is only a fraction of the

Relation	Operators Supported
Identity	`nn.relu`, `ceil`, `floor`, `trunc`, `sign`, `logical_not`, `log`, `log10`, `log2`, `clip`
Broadcast	`add`, `subtract`, `multiply`, `divide`, `logical_add`, `logical_or`, `logical_xor` (all elementwise)
Dense	`nn.dense` (dense matrix multiplication)
Bias Add	`nn.bias_add`
Batch Matrix Multiplication	`nn.batch_matmul`
Batch Norm	`nn.batch_norm`
2D Convolution	`nn.conv_2d`

Table 3.1: Operators and type relations supported in the fuzzer prototype.

approximately hundred total operators in TVM, but does account for many of the most commonly used. The following discussion of the necessary steps for solving Relay type relations is based on the operators and relations in this group.

Formalization. One approach is to directly follow the approach of Dewey et al. (2015) and encode type relations in a logical domain. Type relations for operators tend to check only simple equalities between the result shape and argument shape, assuming a fixed rank and generally applying very simple rules for assigning the numerical representation (e.g., `float32` or `int8`). Because the relations tended to assume fixed ranks for result shapes and argument shapes, this made it very straightforward to encode the relations as integer linear programming (ILP) problems, where each shape dimension in the argument was a variable and the return shape was fixed to a constant. The ranks and numerical representations could easily be chosen outside the ILP encoding. For example, most unary operators in Relay use an identity relation, where the input shape exactly equals the output shape; many elementwise operators have a broadcast relation, where an argument with dimensions of size 1 or a smaller rank than another argument can be copied to be compatible with the larger argument (this simply required encoding conditional checks).

For example, here are the ILP encodings for the Identity, Broadcast, and Batch Norm type relations:

- **Identity**: For each argument (there can be any number, but all the Identity operators considered in the fuzzer are unary), add constraints that each dimension of each argument is equal to the corresponding dimension in the result shape (all shapes must be the same rank).

- **Broadcast**: There must be exactly two arguments, of which at least one must be the same rank as the result shape (let us call it r). The other may be of a smaller rank. Let a_1 be the argument with the greater rank and a_2 be the argument with lesser (or equal) rank (let us call it r_2). Let b be a shape vector where, for all indices i from 1 to r, $b_i = (a_1)_i$ if $i < r - r_2$ and if $i \geq r - r_2$, $b_i = (a_1)_i$ if $(a_1)_i$ is not equal to 1 and otherwise $b_i = (a_2)_{i-(r-r_2)}$. We constrain all dimension in b to be equal to the result shape. All dimensions in a_1 of index $i \leq r - r_2$ must be equal to b_i and for $i > r - r_2$, $(a_1)_i$ and $(a_2)_{i-(r-r_2)}$ must either be equal to b_i or 1.

- **Batch Norm**: The return type is a tuple of three tensors, (b_1, b_2, b_3). There are five arguments a_1, a_2, a_3, a_4, a_5. There is also a compile-time parameter called "axis," which in the current implementation is a fixed constant (eventually, it should be made an ILP variable), which we will denote as x (in the real implementation, it is a 0-based index, but for this description, let us consider it a 1-based index). b_1 can have any rank $r \geq x$ and b_2 and b_3 must both be vectors (rank 1). a_1 must be of rank r and all other arguments are vectors (rank 1). All dimensions of a_1 are constrained to be equal to the dimensions of b_1. All other arguments and b_2 and b_3 are constrained to have the shape $(b_1)_x$.

Since ILP is an NP-complete problem, it is possible to represent Boolean logic constraints like logical ors and implications (which are used in the encoding for Broadcast). This does

suggest that using a constraint logic programming (CLP) language as in Dewey et al. (2015) would likely be viable.

Notice that all reasoning about type constraints in this scheme is *local* (that is, solving one relation at a time without requiring the global gathering of type constraints). Locally reasoning about type constraints ensures that queries to constraint solvers will be kept small and likely tractable, which should allow programs to be constructed quickly while still guaranteeing the validity of generating programs.

Brute Force. If a maximum size is stipulated for shape dimensions, that makes the domain of possible solutions finite and therefore amenable to brute-force searching. Memoization can reduce the amount of searching, though brute force will still scale badly for large tensors (real DL models have tensors with dimensions of sizes greater than 1000). Nevertheless, an advantage of brute force is that it is not necessary to independently formalize the type relations as with the ILP-based approach—the C++ checkers from TVM can be directly called with the concrete candidate argument types and the return type.

Sampling Instead of Solving. To avoid the complexity of reformalizing type relations (as well as having to depend on a solver), another means of finding solutions by reframing the problem. Rather than starting from a concrete return type and finding argument types that satisfy the relation, it is instead possible to generate valid argument types and seed the type generator with the return type produced by the imperative checker. That is, the next time the expression generator needs a new type, it can (with some probability) pick a "sampler" to generate a return type and use the relation solution to produce an operator call.

One way of implementing forward solving is to use the same ILP encodings and leave the return type unconstrained, simply using the ILP solver's solution. However, for specific relations (those examined in the process of writing the prototype), it is often simple to write a procedure to generate random valid argument types and construct the return type from

those.

If we consider the same examples as before, we may note that the sampling procedures are even simpler than the ILP encodings:

- **Identity**: Generate a random shape for the return shape. Generate the desired number of arguments (1 for the operators considered here), which must match the generated return shape.

- **Broadcast**: Generate a random return shape (must be of rank at least 1, which we will denote r), which we will call b. Generate a_1, a random shape of rank r and a_2, a random shape of rank $0 \leq r_2 \leq r$. For all $i \in [1, r - r_2)$, set $(a_1)_i = b_i$. For all $i \in [r - r_2, r]$, choose a random bit: if it is 1, set $(a_1)_i = b_i$ and $(a_2)_{i-(r-r_2)} = 1$ and if it is 0, set $(a_2)_{i-(r-r_2)} = b_i$ and $(a_1)_i = 1$.

- **Batch Norm**: Generate a random rank $r \geq 1$. Generate a random axis parameter $x \leq r$ (again, assuming it is a 1-based index). Generate a return shape b_1 of rank r by choosing random dimensions. All other argument shapes and return shapes are vectors of dimension $(b_1)_x$.

Manual Solving. The original prototype included only ILP-based and brute-force solving and sampling, but subsequent experimentation in the process of preparing the RFC suggested another approach for reasoning about TVM's type relations would be viable—namely, simply writing manual procedures to generate argument types given a return type. We dismissed this approach out of hand for the original prototype as demanding too much manual engineering effort, but further reflection on the sampling approach revealed that sampling is, in reality, no less laborious. (Similarly, reformulating type relations to produce ILP encodings would also be quite laborious.) Notice that the above-described sampling procedure for broadcast begins by generating a return type *and then describing how to construct argument types from the return type.* The same was true of the other sampling procedures. In general, these

solving procedures for the type relations are simpler than the implementations of the type relations themselves since they have a simpler task: the type relations must ensure that the argument types and result type are compatible (some may be unconstrained; others, already specified), whereas this solver constrains only the result type and is free to generate *any* valid argument types.

Indeed, the simplicity of these manual solvers and the fact that they do not require an ILP or CLP solver led to my recommending this approach in my RFC to the TVM community (only brute force solving would entail less implementation work, at a great computational cost). While the type relation's implementation cannot be reused directly to implement this form of solver, the logic is similar, so it would not be a large imposition on contributors of new type relations to also implement a solver. Another practical advantage of having manually specified solvers is that it allows for more control over generation *policy*: solvers can be parameterized, in case only solutions of a particular form are desired for a specific fuzzing task (e.g., testing a specific kind of pass). There is only one invariant that must hold: as long as the "recognizer" for an operator accepts a given return type, the solver must be able to return input types for it.

2.4 Preliminary Results and Implications

We note that two interesting bugs were found during the development process for initial prototype. The first bug was uncovered in small-scale test runs of the fuzzer and was due to a missing base case in match exhaustion. The second bug was found by inspection while formalizing the type relation for `nn.bias_add` and was a missing bounds check. These bugs both related to type checking; fixes for them were accepted.[3] Large-scale runs also revealed a parser round-tripping bug (generating a program AST, using TVM's pretty-printer to express it in the text format, then trying to parse the program text, with the expectation of preserving the original AST) wherein `ref`s of `ref`s would fail to parse (this has not yet been

[3]`https://github.com/apache/tvm/pull/7459` and `https://github.com/apache/tvm/pull/7554`, respectively.

fixed).

Our trials at scale also yielded a generation rate of 26.4 KB/s, with no program ever failing to type check. (By contrast, a pure grammar-based fuzzer was able to reach 1225 KB/s, but only 0.082% of programs generated type checked and none of those included a single use of a tensor operator.) Interestingly, there was little variation in speed between the brute force, ILP-based, and sampling-based approaches (the manual solving approach had not been implemented at the time of our initial trials), though this may be attributable to the low maximum tensor sizes used in the trials (only a maximum dimension of 10). Solving operators may not have been a performance bottleneck at all; further profiling will likely be necessary. Additionally, the prototype was implemented in Python and could likely be sped up simply by reimplementing it using TVM's C++ APIs. Further exploration of performance tradeoffs for fuzzing Relay will be an interesting subject as the community discusses whether to employ it over the long term.

There are many further issues related to the fuzzer that would likely also be worth exploring. The fuzzer's ability to generate expressions of a given type can be similarly used for generating mutants (replacing a given term with another of the same type) and thus perform mutation-based fuzzing. We may also consider type-driven minimization by replacing subexpressions with literals of the same type. Depending on the fuzzer's success, we may need to implement fuzzer taming and test case minimization. It may also be possible to handle tensors with dynamically checked shapes in Relay's type system by "relaxing" concrete shapes, namely replacing concrete dimensions with `Any` (since it would be guaranteed that at least one concrete solution exists); type inference could also be tested by identifying cases where type annotations could be removed without introducing ambiguity. Additionally, given the present fuzzer's demonstrated ability to generate well-typed programs quickly, it would be of great interest to investigate whether the fuzzer can uncover bugs subtler than crashes at compile time, though this would require checking manually specified correctness conditions for compiler passes or execution results. It could potentially be fruitful to use a DART-like (Godefroid et al., 2005) white-box approach to harvest information about paths taken in

the compiler or executor and to thereby generate programs exercising more features, which has been pursued in the context of DL models (albeit testing lower-level kernels) by Liu et al. (2022b).

In the context of this dissertation's wider goals, it is most important to note that it was Relay's typing rules that allowed for the quick and relatively simple development of a fuzzer capable of exercising almost all of Relay's features. To the best of my knowledge, *no comparable tool exists for fuzzing the front end of any other deep learning framework*, certainly not one capable of reasoning about tensor shapes accepted by operators. It is my hope that the techniques used here for fuzzing Relay could be applied to the development of compilers in other domains with complex constraints and motivate the development of powerful type systems for safety and optimization purposes—indeed, powerful type systems that make the system itself easier to test.

3 Summary

Relay's principled approach to representing and optimizing deep learning models served as guidance for the work detailed in the chapters to follow, providing infrastructure and design approach for a view of DL models as programs; indeed, Relay itself was directly in the implementation of 3LA (Chapter 5). The enthusiastic adoption of Relay and TVM by some groups in industry, such as in Amazon's AI projects (Kim and Sharma, 2019) and by the many companies that have presented their use of TVM at TVMCon, stresses the practicality of providing a general programming model for deep learning programs and the application of traditional compilers techniques to this domain.

Chapter 4

RUNTIME TECHNIQUES: DYNAMIC TENSOR REMATERIALIZATION

Note: This chapter is adapted from the previously published work Kirisame et al. (2021).

This section focuses on the problem of reducing the amount of memory needed to train a DL model by exploring whether a runtime system can reduce the memory required to train dynamic models with limited performance overhead by taking advantage of domain-specific properties.

1 Problem Description

Though computation costs are the most common consideration when optimizing model performance, memory can also be a limiting factor. As state-of-the-art deep learning (DL) models continue to grow (Brown et al., 2020; Devlin et al., 2018; Brock et al., 2018), training them within the constraints of on-device memory becomes increasingly challenging. DL models require large amounts of memory to store input tensors, parameters, and intermediate values. Storing intermediate activations for backpropagation in particular is the largest source of memory usage in training, as Sohoni et al. (2019) report, noting that intermediate activations occupy 2.2GB in a batch of a Transformer model. Memory limitations prevent the training of larger and deeper models, which require storing more intermediate activations. Batch sizes are also limited by the available memory, potentially slowing down training runs overall, since it may be possible for the hardware to efficiently process a batch in parallel and thereby train a model using fewer batches (though batch size also affects convergence). Experimental higher-order learning techniques like Model-Agnostic Meta-Learning

(MAML) (Finn et al., 2017), which learns optimal hyperparameters for a model but requires storing multiple sets of intermediate activations, are particularly limited by the available memory. The great memory costs of training DL models also pose problems at the hardware level, as DL models can run on a variety of devices—including specialized accelerators, embedded devices, or simply older models of GPUs—and on-device memory cannot be easily expanded. While it may be possible to swap memory between devices to make the best use of the space available, communication is potentially expensive. Ultimately, the memory bottleneck limits what DL models can be run on various devices and limits researchers' ability to explore memory-intensive architectures and training techniques.

As noted in Chapter 2, the approach of checkpointing has been successful in reducing the memory costs of training. Checkpointing approaches trade computation time for space by freeing intermediate activations and recomputing them when they are later needed in backpropagation. Such an approach is particularly advantageous in DL because most tensor operators are pure, meaning that operators used to compute the intermediate values can be re-run without changing the behavior of the model. However, recent approaches to checkpointing in DL, including Jain et al. (2019), Kumar et al. (2019), and Shah et al. (2021), proceed by producing a fixed schedule of operations to perform, including which intermediate activations to free and later recompute. These approaches thus cannot directly support general dynamic models (e.g., Gruslys et al. (2016) only supports a specific form of RNN) or unusual training setups like in MAML. Additionally, optimal static planning in Jain et al. (2019) and Shah et al. (2021) is achieved by reducing checkpointing to ILP problems, often resulting in expensive queries to solvers (e.g., DenseNet-161 could not be solved within 24 hours in Jain et al. (2019)), which can limit the pace of development or prototyping.

The work detailed in this chapter, Dynamic Tensor Rematerializatino (DTR), demonstrates that static planning is *unnecessary* for DL checkpointing. DTR operates like a tensor-level cache: it collects metadata on tensors and operators as a model is trained and uses it to guide heuristics that choose which activations to free and later recompute. As a runtime system, DTR can utilize dynamically gathered information (e.g., measured operator costs) and

adapt to any given memory bound so long as there is enough at least enough memory to store the tensors needed for any single operator in the model. Additionally, its simple, cache-like approach requires no advance knowledge of the model or application, letting it immediately support arbitrarily dynamic models and applications featuring higher-order differentiation. For example, given a model with data-dependent control flow like TreeLSTM (Tai et al., 2015), DTR's runtime can simply evict tensors when memory runs out and rematerialize them as needed. By contrast, static planning techniques assume a static dataflow graph, which requires "unrolling" dynamic models and performing (potentially expensive) planning for every distinct input.

The following sections describe the design of DTR in further detail, highlighting the following contributions:

- We prove that DTR can train an N-layer linear feedforward network on an $\Omega(\sqrt{N})$ memory budget with only $\mathcal{O}(N)$ tensor operations (Section 3), which is within a constant factor of optimal and matches the offline bound of the Chen et al. (2016) static checkpointing technique.

- We formalize DL model checkpointing as an online rematerialization problem and define a greedy algorithm parameterized by caching-inspired heuristics. In simulated trials, our heuristic attains *near-optimal performance* on a variety of DL models (Section 4).

- We implement a DTR prototype by making only modest modifications to the PyTorch framework, enabling training under restricted memory budgets for both static and dynamic models and demonstrating the ease with which our algorithm can be incorporated into an existing DL framework (Section 5).

2 Design Overview

DTR follows the example of Jain et al. (2019) in comparing checkpointing to the older compilation technique of register rematerialization (Briggs et al., 1992) (hence freeing a tensor

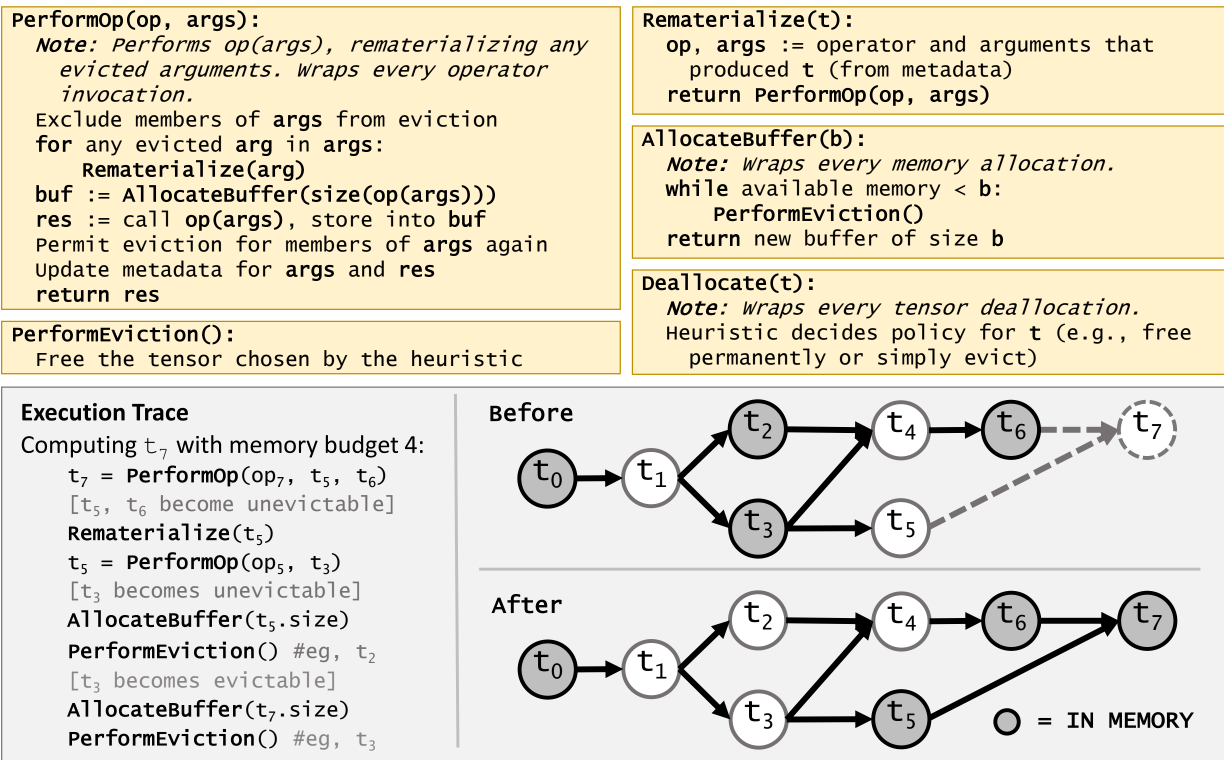


Figure 4.1: (Top) Pseudocode for DTR's basic logic (independent of heuristic), and (Bottom) DTR's sequence of events in an operator call. Note that `PerformOp()` may make further recursive calls in order to rematerialize arguments.

will be called "evicting" it and recomputing will be "rematerializing" it), but approaches the problem dynamically, like a domain-specific cache or memory manager. DTR incorporates similar reasoning to past checkpointing approaches, but avoids any inherent assumptions about the model or even the fact that gradients are computed. Hence, DTR operates as a thin runtime layer that intercepts tensor allocations, accesses, and deallocations.

Figure 4.1 shows DTR's high-level approach. In `AllocateBuffer`, DTR first checks if sufficient memory is available when a tensor allocation occurs. If so, it generates a fresh tensor identifier, initializes its metadata for future recomputation, allocates the requested memory, and returns a new tensor. If not, DTR heuristically selects and *evicts* resident tensors until the requested allocation can be accommodated. Constant tensors (loaded from external data) cannot be evicted since no corresponding operation rematerializes them. Upon tensor access, DTR first checks if the tensor is resident in memory. If so, it updates tensor metadata before returning the requested tensor. If the tensor has been evicted, DTR *rematerializes* it by replaying the *parent operation* that originally produced the tensor. Crucially, rematerialization can be recursive: if the arguments to an evicted tensor's parent operation have also been evicted, then *they* must first be rematerialized. Rematerialization may trigger more evictions if memory is exhausted during the potentially recursive process. Upon tensor deallocation (other than by evictions), the runtime is invoked again (`Deallocate`), letting it update tensor metadata and eagerly perform profitable evictions.

Assumptions. This description of DTR assumes that: tensors are accessed only by monolithic, opaque operators; tensors are either constants or produced by operators; operators produce individual tensors; and operators are pure (deterministic functions of their arguments). Under this model, a training epoch is simply a sequence of tensor operations without any inherent requirement to recognize training-specific structure, like the transition to the backward pass. DTR will evict as many tensors as necessary to avoid running out of memory. If all inputs and outputs of a single operation cannot fit into available memory, rematerialization will fail; therefore, on a given model and input, there may be a threshold

for the lowest budget DTR can support. The choice of heuristic can affect the likelihood of failure since different eviction choices can result in deeply nested rematerializations that require many tensors to remain in memory.

Heuristics. DTR is parameterized by heuristics that guide its eviction choices. As in caching, DTR's eviction heuristic *dynamically* predicts which resident tensors are least valuable. The choice of heuristic determines what metadata (additional runtime facts) must be tracked for each tensor and operator and thus affects DTR's runtime overhead. In our evaluation, we consider a runtime system that tracks the following metadata for each tensor t:

- **staleness**, $s(t)$, the time since last access;
- **memory**, $m(t)$, the size of the tensor; and
- **cost**, $c_0(t)$, the time required to compute t from its parent tensor(s).

We observe that the overhead of updating these metadata (recording access times, sizes, and computation times) is low relative to the cost of typical DL tensor operations.

We propose a rematerialization-specific heuristic that balances staleness, memory, and cost, evicting the tensor t that is stalest (least likely to be needed soon), largest (saves the most space), and cheapest (requires the least additional rematerialization if t is needed again). To capture the total amount of rematerialization required if t is evicted, we sum the costs over the tensor's *evicted neighborhood* $e^*(t)$, i.e., the set of evicted tensors that would either need to be rematerialized to recompute t or would need t to be resident to be recomputed. We define the *projected cost*, $c(t)$, of rematerializing tensor t as $c_0(t) + \sum_{t' \in e^*(t)} c_0(t')$. Using this definition, we define our heuristic, which evicts the tensor minimizing $h_{\mathrm{DTR}}(t) = c(t)/[m(t) \cdot s(t)]$. By including both forward and backward dependencies of t in $e^*(t)$, h_{DTR} penalizes creating long chains of evicted tensors (and hence potential recursive rematerializations) that could arise from t's eviction.

To illustrate evicted neighborhoods, suppose DTR is checkpointing the network shown in Figure 4.1, where the resident tensors are $\{t_0, t_2, t_3, t_6\}$. Before node t_7 is computed, we have $e^*(t_2) = \{t_1, t_4\}$ and $e^*(t_3) = \{t_1, t_4, t_5\}$. Since each new eviction can expand a given tensor's evicted neighborhood and each rematerialization can shrink it, dynamically tracking evicted neighborhoods can introduce further costs at run time. To decrease runtime overhead, we developed an approximation of e^* using an undirected relaxation tracked by a union-find data structure that uses a constant-time approximation for splitting. We use this approximation to define $h^{\text{eq}}_{\text{DTR}}$ analogously, which performs nearly as well as h_{DTR} in our evaluation but requires up to 2 orders of magnitude fewer metadata accesses per batch (see Section 4, especially Sections 4.2 and 4.5.3).

Our heuristic formalization in terms of s, m, and c_0 is sufficiently general to express several existing heuristics for caching and checkpointing. For example, the common LRU heuristic is "minimize $1/s(t)$," the `GreedyRemat` heuristic from Kumar et al. (2019) is "minimize $1/m(t)$," and the MSPS heuristic from Peng et al. (2020) is "minimize $c_R(t)/m(t)$" (where $c_R(t)$ sums c_0 over t's evicted ancestors). We compare h_{DTR} to these other heuristics inspired by recent work in our simulated evaluation (Sec. 4).

Deallocation. Deallocation policies present further tradeoffs since tensors marked as deallocated by the original program are still potential dependencies for rematerializations. In principle, DTR could simply disregard deallocations by the original program, but this would ignore potentially useful information about the deallocated tensors (viz., that the original program will not use them again). *Banishing* (permanently freeing) deallocated tensors can save memory immediately and is the only way to free constants (which cannot be evicted); however, it can prevent possible future evictions since the children of a banished tensor cannot be rematerialized. By contrast, evicting deallocated tensors does not prevent potential evictions, though it increases the runtime's management overhead and keeps constants in memory. In the heuristics we examined, we implemented an *eager eviction* mechanism, which evicts a tensor as soon as all external references to it are freed. This lets DTR adhere

to the garbage collection pattern of the underlying framework, preempting desirable evictions, which further reduces future runtime overhead. (See Section 4.5.2 for a comparison of deallocation policies.)

3 Formal Bounds

Following Chen et al. (2016), we prove a bound on DTR's checkpointing overhead (for a particular eviction heuristic) on a linear feedforward network of N nodes. Even without the ability to inspect the model, DTR requires only $\mathcal{O}(N)$ tensor operations under a $\sqrt{N}$ memory budget, the same bound (up to constant factors) as the Chen et al. (2016) static checkpointing technique and the optimal $\Theta(N)$ required by a memory-unconstrained algorithm. We also establish that DTR's dynamic approach cannot always match the overhead of static checkpointing: given N tensor operations and a memory budget of B, under any deterministic heuristic, an adversary could always construct a network where DTR would perform a factor of $\Omega(N/B)$ more tensor operations than a (potentially expensive, see Jain et al. (2019)) optimal static checkpointing algorithm.

3.1 Linear Feedforward Overhead

We assume that tensor computations dominate the run time and, as in prior work (Griewank and Walther, 2000; Chen et al., 2016; Binder et al., 1997; Beaumont et al., 2019), that each tensor is of unit space and time cost. For the proof below, we use the heuristic h_{e^*}, which evicts a resident tensor t with minimal $|e^*(t)|$.

Theorem 3.1. *Given an N node linear feedfoward network and a memory budget $B = \Omega(\sqrt{N})$, DTR with heuristic h_{e^*} can execute one forward and one backward pass in $\mathcal{O}(N)$ operations.*

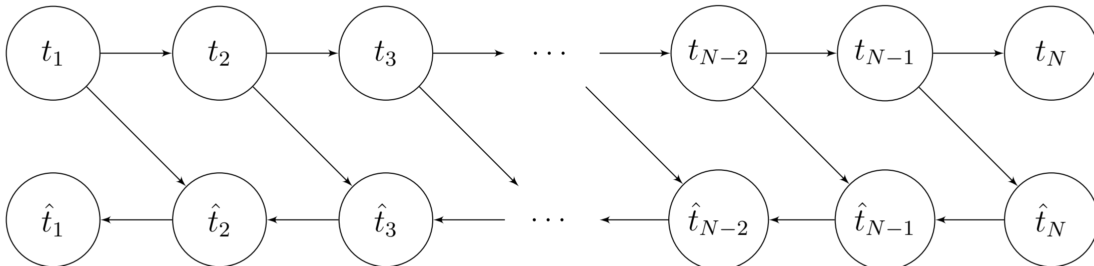
3.1.1 Proof Sketch.

During the forward pass, DTR performs exactly N tensor operations: since each node of the linear feedforward network depends only on the previous node, no rematerialization is necessary. Our heuristic h_{e^*}, which evicts tensors with the smallest evicted neighborhoods, ensures that the B tensors resident at the conclusion of the forward pass are evenly spaced throughout the network. In turn, these evenly spaced checkpoints ensure that DTR never has to successively rematerialize too many tensors. As the backward pass proceeds and checkpoint tensors are freed, the overhead to compute all gradients between the checkpoints k and $k+1$ shrinks as $\log(k)/k^2$, which sums to a constant. Below, we provide a full proof.

3.1.2 Network Definition

We assume the network consists of operators $f_1, \ldots, f_N$, where the tensor computed by the ith operator is given by $f_i(t_{i-1})$, with t_j denoting the tensor computed by the jth operator. Note that we consider t_0 to be the input tensor, which for simplicity will always reside in memory and not contribute to the active memory consumption. For this reason, we may consider f_1 to be a nullary operator. Additionally, we assume that the size of each tensor (denoted $m(t)$) is 1, and likewise for the compute time $c_0(f_i)$ for each operator f_i. Note that we may write $c_0(f_t)$ to mean the same as $c_0(f_i)$ for $t = t_i$, when the index i is not convenient.

For backpropagation, we assume each operator f_i has an associated *gradient* operator $\hat{f}_i$, which computes the result $\hat{t}_i = \hat{f}_i(t_{i-1}, \hat{t}_{i+1})$. We may consider $\hat{t}_{N+1} = \mathbf{1}$ to be an unevictable unit tensor, as is the case in automatic differentiation, but for simplicity we define $\hat{t}_1 = \hat{f}_1(\hat{t}_2)$ and $\hat{t}_N = \hat{f}_N(t_{N-1})$. As above, we assume unit memory and compute for each $\hat{f}_i$.



3.1.3 Liveness and Banishing

To optimize memory usage during computation, we introduce the notion of *liveness* and *banishing.* At a high level, liveness allows us to determine when a given tensor is no longer required for subsequent network computations, which in turn allows us to permanently free (banish) tensors to regain memory when certain conditions are met.

To be more precise, we formalize the network as a program:

```
let t_1 := f_1();
let t_2 := f_2(t_1);
...
let t_N := f_N(t_{N-1});
// Backpropagate.
let \hat{t}_N := \hat{f}_N(t_{N-1});
let \hat{t}_{N-1} := \hat{f}_{N-1}(t_{N-2}, \hat{t}_N);
...
let \hat{t}_2 := \hat{f}_2(t_1, \hat{t}_3);
let \hat{t}_1 := \hat{f}_1(\hat{t}_2);
```

We say a tensor t is *live* when there is a pending operation in the program that takes t as an input. When t is no longer live, *and* every tensor directly computed using t is in memory or banished, then we say t is *banished* and we reclaim the memory used by t. Banishing a tensor additionally makes its children unevictable.

Thus for example, t_N can be immediately banished after computing, t_{N-1} can be banished after $\hat{t}_N$, both t_{N-2} and $\hat{t}_N$ after $\hat{t}_{N-1}$, and so on. This will become important in the proof.

The analysis of liveness can be done statically for static models, and by reference counting for models with dynamism. In both cases, liveness information is fed to DTR online through deallocation events.

3.1.4 Heuristic Definition

Heuristic h_{e^*} is a reduced form of the DTR heuristic, as it does not account for tensor staleness. Here, we provide a detailed motivation of its definition.

Recall the *evicted neighborhood* $e^*(t)$ of tensor t, as described in Section 2.

Definition 3.1 (Projected Cost)**.** For a given tensor t, the *projected cost* of t is the value

$$c(t) = \sum_{t' \in e^*(t)} c_0(f_{t'})$$

Now, we define the reduced heuristic in full generality; the definition of h_{e^*} will be a consequence of the simplified setting we analyze.

Definition 3.2 (Compute-Memory Heuristic (general))**.** The *compute-memory* heuristic score for a resident tensor t is defined as

$$h_{e^*}(t) = \frac{c(t) + c_0(f_t)}{m(t)}$$

Corollary 3.1. *Under our simplified compute and memory constraints,* $h_{e^*}(t) = |e^*(t)| + 1$*. Since the heuristic is only used to rank tensors, the common additive constant* 1 *is unimportant. The heuristic* $|e^*(t)|$ *will have the same behavior as* $|e^*(t)| + 1$*.*

Note, importantly, that uncomputed tensors are not considered in any of the above definitions (as we do not know about their existence yet, from a dynamic execution perspective).

3.1.5 Proof of Theorem 3.1

Now we prove Theorem 3.1, which bounds the overhead of DTR on a linear feedforward network with N nodes and $\sqrt{N}$ memory by a constant factor of the runtime required by an algorithm with unlimited memory.

Proof. To prove this claim, we will consider the forward pass and the backward pass separately. In the forward pass, we show that our algorithm only performs N computations, matching that of an algorithm with unlimited memory. Furthermore, upon completion of the forward pass, we tightly characterize the B tensors that remain in memory. We show that a set of evenly spaced *checkpoint tensors* remain in memory throughout the backward pass,

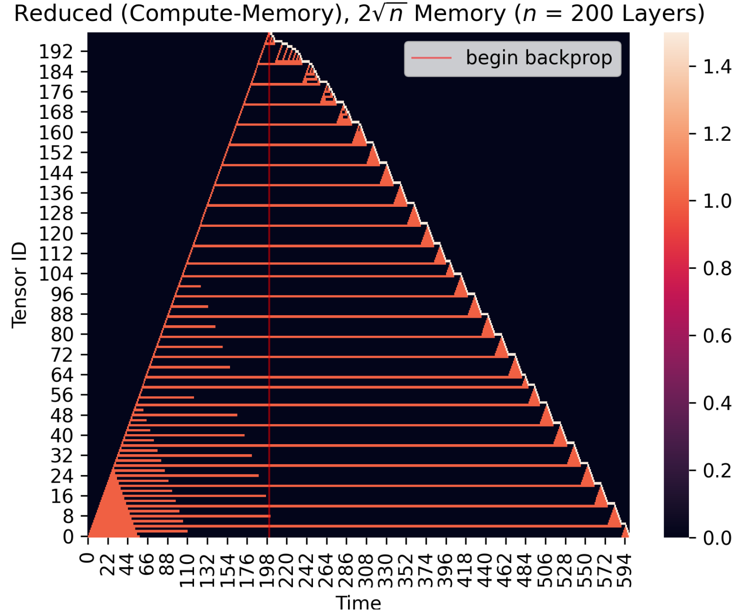


Figure 4.2: Visualization of the state of memory for DTR with $N = 200$, $B = 2\lceil\sqrt{N}\rceil$, and heuristic h_{e^*}. A value of 0 (black) indicates the tensor is evicted or banished, 1 (red) indicates the tensor is a forward value in memory, and 1.5 (white) denotes an in-memory gradient tensor corresponding to the forward tensor. The backward pass begins at the red vertical line; note the presence of evenly spaced *checkpoint tensors* (red horizontal lines) that persist in memory throughout the backward pass. Note also the recursive checkpointing behavior visible in the early gaps of the backward pass, and finally the completely red triangles of the later gaps, when there is enough free memory to avoid repeated rematerialization altogether.

until banishment. The presence of these checkpoint tensors allows us to argue that the algorithm never has to rematerialize too many tensors in a row. Furthermore, as the algorithm computes additional gradients, it banishes checkpoint tensors that are no longer needed, freeing more space for additional checkpoints. The overhead incurred by the algorithm can therefore be kept to a constant factor of the required $\Theta(N)$ time. This checkpointing behavior can be seen in the trace of the algorithm, visualized in Figure 4.2.

We now analyze each of the phases in detail.

Phase 1: Forward pass. Recall that in a feed-forward network, every computation depends only on the preceding one. Thus in our simplified network, we only ever need $B = 2$ units of memory to compute the forward pass without any rematerializations (furthermore, this is the minimum required memory). For this reason, the forward pass requires N computations.

After completing the forward pass, we can tightly characterize the tensors remaining in memory. In particular, Lemma 3.1 tells us that the maximum gap between resident tensors is bounded by

$$L \leq \frac{2(N-2)}{B-1}$$

We note that this bound is tight in an asymptotic sense: if we can keep B tensors in memory, and the forward pass is of length N, then the maximum gap must be at least N/B.

Next, we will analyze the backward pass. Key to this analysis is the claim that "not too many" of the tensors in memory at the beginning of the forward pass are evicted before banishment during the backward pass. The existence of these "checkpoint tensors" allows us to argue that we do not do too much rematerialization work.

Phase 2: Backward pass. During the backward pass, our algorithm computes gradients $\hat{t}_i$. Each gradient computation relies on two inputs: $\hat{t}_{i+1}$ and t_{i-1}. We show that neither input incurs too much rematerialization cost - $\hat{t}_{i+1}$ because it is pinned in memory, and t_{i-1} because the paths of evicted tensors are not "too long." The first condition follows from the fact that t_i is banished after computing $\hat{t}_{i+1}$, therefore forcing $\hat{t}_{i+1}$ to remain in memory until it is banished. The second condition is formalized in the following lemma, proved later in this section.

Lemma 3.1 (Checkpointing)**.** *Consider an execution of the DTR algorithm with B units of memory and heuristic h_{e^*}, applied to the graph described in section 3.1.2. Let S be the set of tensors in memory after computation of t_N in the forward pass. Then, $\mathcal{C} \subseteq S$ is a set of*

"checkpoint" tensors from the forward pass with the following properties:

1. *During the backward pass, each $c \in \mathcal{C}$ stays in memory until it is banished.*

2. *The gap between neighboring tensors in $\mathcal{C}$ satisfies*

$$L \leq \frac{4(N-2)}{B-1}$$

These $|\mathcal{C}|$ checkpoint tensors divide the n forward tensors into $|\mathcal{C}|$ groups, indexed by k, each of length $L_k \leq \frac{4(N-2)}{B-1}$. The total computational cost of the backward pass is equal to the sum of the computational cost for each group,

$$C = \sum_{k=1}^{|\mathcal{C}|} C_k.$$

The second key insight in the analysis of the backward pass is that, for every group that is processed, the algorithm banishes a checkpoint tensor $c \in \mathcal{C}$ and receives a unit of extra memory. In particular, at the start of processing group $|\mathcal{C}| - k$, the algorithm has $2 + k$ pieces of extra memory (two from banishing the most recently used gradient and forward tensor, and k from the banished checkpoint tensors). We can leverage this extra memory to process the gradients in later groups with less rematerialization overhead, using the k extra units of memory to create intermediate checkpoint tensors. The following lemma describes how the cost of computing all the gradients in a group decreases as we free more memory.

Lemma 3.2. *Suppose we have $2 + k$ pieces of free memory to compute all of the gradients associated with an evicted forward tensor path of length L_k. Then the number of rematerializations needed to compute all the gradients is of order*

$$C_k = \mathcal{O}\left(L_k + \frac{L_k^2}{k^2} \log k\right)$$

Applying this lemma, the total cost of the backward pass becomes

$$\begin{aligned}
C &= \sum_{k=1}^{|\mathcal{C}|} C_k \\
&\lesssim \sum_{k=1}^{|\mathcal{C}|} \left(L_k + \frac{L_k^2}{k^2} \log k \right) \\
&\leq \sum_{k=1}^{|\mathcal{C}|} L_k + \sum_{k=1}^{|\mathcal{C}|} \frac{\log k}{k^2} L_k^2 \\
&\leq |\mathcal{C}| \left(\frac{4(N-2)}{B-1} + 1 \right) + \sum_{k=1}^{|\mathcal{C}|} \frac{\log k}{k^2} \left(\frac{4(N-2)}{B-1} + 1 \right)^2 \\
&\lesssim |\mathcal{C}| \left(\frac{N}{B} \right) + \frac{N^2}{B^2} \sum_{k=1}^{|\mathcal{C}|} \frac{\log k}{k^2}
\end{aligned}$$

where $\lesssim$ hides constant factors. Note that $|\mathcal{C}| \leq B$, since $\mathcal{C} \in S$ where S is the set of tensors in memory at the end of the forward pass. Also note that $\frac{\log k}{k^2}$ is a convergent sequence, so its partial sums are bounded. Therefore, we can simplify the bound to

$$C \lesssim N + \frac{N^2}{B^2}$$

Since $B = \Omega(\sqrt{N})$, we conclude that the total cost of the backward pass is $\mathcal{O}(N)$. Adding this to the $\mathcal{O}(N)$ cost of the forward pass, we see the total compute is $\mathcal{O}(N)$, as desired.

□

3.1.6 Proofs of Intermediate Results

Here, we present intermediate results that we used in the proof of our main result.

Lemma 3.3. *Consider the DTR algorithm operating with heuristic h_{e^*}. Suppose we seek to (re)materialize forward tensor t_k for $k \leq N$, where the resident tensor preceding t_k is denoted by t_j (with $j < k$). Suppose also that t_j is not evicted during the computation of t_k. Then, if the algorithm begins with t_j in memory and with M units of memory, and runs*

until computing t_k, then the maximum length L of any evicted sequence of tensors between t_j and t_k is bounded by

$$L \leq 2((k - j) - 1)/(M - 1)$$

Proof. Proof by induction. We will show that, when the algorithm computes tensor $j + i$, for $i = 1, 2, \ldots, k - j$, the maximum length of an evicted sequence of tensors between t_j and t_{j+i} satisfies

$$L_i \leq 2(i - 1)/(M - 1)$$

Base case. When $i = 1$, both t_j and $t_{j+1} = t_k$ are resident tensors, so the gap is $L_1 = 0$.

Inductive step. Consider the contents of memory after computing t_{j+i}. We begin by partitioning tensors $t_j, \ldots, t_{j+i}$ into M segments $S_1, \ldots, S_M$, each ending in a resident tensor (note, the last segment must end on a resident tensor, since t_{j+i} was just computed). If $i < M$ so that there are not M resident tensors, then the length of each segment is zero and we are done. Otherwise, each segment corresponds to an evicted sequence of zero or more tensors (i.e., the tensors preceding the resident tensor). Let s_i denote the resident tensor that ends segment i.

Now, consider all adjacent pairs of segments (S_l, S_{l+1}) for $1 \leq l \leq M - 1$. The average

length of the pairs is given by

$$\begin{aligned}
\overline{L} &= \sum_{l=1}^{M-1} \frac{|S_l| + |S_{l+1}|}{M-1} \\
&= \left(2 \sum_{l=1}^{M} \frac{|S_l|}{M-1}\right) - \frac{|S_1| + |S_M|}{M-1} \\
&= \frac{2}{M-1} \left(\sum_{l=1}^{M} |S_l|\right) - \frac{|S_1| + |S_M|}{M-1} \\
&= \frac{2i}{M-1} - \frac{|S_1| + |S_M|}{M-1} \\
&\leq \frac{2(i-1)}{M-1}.
\end{aligned}$$

Let $(S_{l'}, S_{l'+1})$ be the pair of adjacent segments with minimum combined length. Since the average length is bounded by the inequality above, it follows that the length of $(S_{l'}, S_{l'+1})$ is also less than or equal to $2(i-1)/(M-1)$.

Since the heuristic evicts the tensor that results in the smallest gap, we conclude that the eviction will create a gap no larger than $2(i-1)/(M-1)$. By the inductive hypothesis, the largest previous gap was no larger than $2(i-2)/(M-1)$, so we conclude that the largest gap after this computation is no more than $2(i-1)/(M-1)$.

□

Proof of Lemma 3.1.

Proof. We will prove this lemma by dividing the backward pass into two phases. In the first phase, the first two gradient computations of the backward pass, we may be forced to evict some element of S. In the absence of further information on the evicted tensor, we upper bound the resulting gap by twice the maximum gap between tensors in S. This gives us the upper bound in Item 2 of the lemma.

In the second phase, the remaining $N-2$ gradient computations of the backward pass, we show that heuristic h_{e^*} never leads us to evict a tensor that would lead to a gap of more

than $\frac{4(N-2)}{B-1}$ among the tensors in memory. This allows us to conclude that the checkpoint tensors $\mathcal{C}$ remain in memory until eviction, as claimed.

We now elaborate on the two phases, as discussed above.

Phase 1: The first two gradient computations of the backward pass.

We present a detailed treatment of the first two gradient computations in the backward pass, $\hat{t}_N$ and $\hat{t}_{N-1}$. We will show that, during the course of these two computations, at most one tensor from S is evicted from memory. Since Lemma 3.3 tells us that the maximum gap in S satisfies $L_S \leq \frac{2(N-2)}{B-1}$, we conclude that removing a single tensor results in a gap in C of no more than $2L_S$. Additionally, we will show that after the computation of the first two gradients, there are at least two non-checkpoint tensors in memory. Since only two free units of memory are required to rematerialize a path of tensors, this sets us up for the analysis of the remaining gradient computations.

We begin by noting that, after the forward pass completes, t_N and t_{N-1} are both in memory (since t_N has just been computed, which requires t_{N-1}). Since t_N is no longer needed in subsequent computations, it is immediately banished. Assuming $B \leq N$, this leaves us with exactly one unit of free memory (if $B > N$, no elements of S are banished in the first two computations, and the $2L_s$ bound is trivial). This single unit of memory is then filled by the computation of $\hat{t}_N$, which only depends on t_{N-1}.

Now, t_{N-1} is no longer needed, so it is banished, and we have exactly one unit of free memory. To compute $\hat{t}_{N-1}$, we require t_{N-2} and $\hat{t}_N$ to be in memory. Since $\hat{t}_N$ was just computed, it is clearly in memory. However, t_{N-2} may or may not be in memory. We consider the two cases separately.

If t_{N-2} is in memory, then we immediately compute $\hat{t}_{N-1}$. Next, tensors t_{N-2} and $\hat{t}_N$ are banished, leaving us with the desired two free units of memory.

If, on the other hand, t_{N-2} is not in memory, we must rematerialize it. Let t_j be the resident tensor that terminates the evicted path of tensors containing t_{N-2}. We need to perform the sequence of computations $\{t_{j+1}, t_{j+2}, \ldots, t_{N-2}\}$. However, we only have one

unit of free memory, so after computing t_{j+1} we will need to evict some tensor from memory. The evicted tensor must be t_i for some $i \leq j$, as neither t_{j+1} nor $\hat{t}_N$ can be evicted (the former will be used for the next computation, and the latter is pinned in memory).

Regardless of which tensor t_i is evicted, the length of the evicted path it creates cannot exceed $2L_S$, where L_S is the length of the longest path in S. Lemma 3.3 bounds $L_S \leq \frac{2(N-2)}{B-1}$, so this step of the algorithm maintains Item 2 of the lemma.

It remains to show that the maximum gap in $\mathcal{C}$ does not become larger than $2L_S$ during the remaining steps of rematerialization, and that the computation of $\hat{t}_{N-1}$ ends with at least two units of free memory. To show the first claim, we note that the number of evicted tensors on the path to $\hat{t}_{N-1}$ does not exceed $2L_S$ (this is the maximum length possible, if t_j was evicted and its adjacent evicted paths were both of length L_S). Therefore, when performing the intermediate rematerializations necessary to rematerialize t_{N-2}, it is always possible to evict a tensor between t_j and t_{N-2}, with a heuristic value of less than $2L_S$. Since we evict the tensor with the smallest heuristic value, we will never create an evicted path of length greater than $2L_S$.

Finally, we note that, after computing $\hat{t}_{N-1}$, both t_{N-2} and $\hat{t}_N$ will be banished. This leaves us with the desired two units of free memory.

We have shown that, after computing $\hat{t}_{N-1}$, the algorithm has two units of free memory, and the checkpoint set $\mathcal{C}$ has a maximum gap of no more than $2L_S$. Next, we show that this set $\mathcal{C}$ is maintained throughout the remainder of the backward pass.

Phase 2: The remaining $N-2$ gradient computations.

The analysis for the remainder of the backward pass follows via induction, using the argument for rematerializing t_{N-2} above.

We have already shown a base case; we can maintain the desired properties of $\mathcal{C}$ when computing $\hat{t}_{N-2}$. For the inductive step, consider the computation of $\hat{t}_i$ for $1 < i < N-1$. Suppose we have at least two units of free memory, and $\hat{t}_{i+1}$ in memory. Furthermore, suppose that the set $\mathcal{C}$ satisfies the properties of the lemma. We need to rematerialize t_{i-1},

which terminates a path of evicted tensors of length no more than $2L_S$. As we rematerialize this path, it may require evicting tensors from memory. However, by the same logic we applied above, we know that the algorithm may always choose to evict a tensor resulting in a path of less than $2L_S$. The algorithm will always choose this option in favor of creating a longer evicted path. We conclude that the upper bound of $2L_S$ is preserved when computing $\hat{t}_i$. Furthermore, after $\hat{t}_i$ is computed, we may evict $\hat{t}_{i+1}$ and t_{i-1}, giving us two units of free memory. This proves the inductive step.

Note that, in the case that $i = 1$, the computation requires no rematerializations, as $\hat{t}_1$ only depends on $\hat{t}_2$, and the latter is in memory at the time of computing $\hat{t}_1$. □

Proof of Lemma 3.2.

Proof. Let $C_{i,k}$ denote the cost of processing gradient i in this group. Since there are L_k associated gradients, the total cost is

$$C_k = \sum_{i=1}^{L_k} C_{i,k}.$$

To compute each $C_{i,k}$ we note that computation of the gradients proceeds in phases. When the first gradient is computed (at cost $C_{0,k} = L_k$), two units of memory must be devoted to the current tensor computation, while the remaining k units of memory are used for intermediate rematerialized tensors. Applying the intermediate checkpointing lemma, 3.4, we conclude that some of these intermediate tensors will remain as checkpoints (indexed by j, with $j = 1$ indicating the highest-indexed tensor), with adjacent checkpoints separated by a distance at most $L_{k,j} = \frac{4(L_k-2)}{k-1}$. We can express the total cost of computing the gradients in this gap as

$$C_k = L_k + \sum_j \sum_{i \in \text{group } j} C_{i,k}$$

We begin by considering the first group to be processed, $j = 1$, associated with the last

path between checkpoints. Since it is the first group to be processed, it has no spare memory for intermediate checkpoints. Therefore, computing the first gradient requires rematerializing the entire group (with at most $L_{k,j}$ intermediate tensors), computing the second gradient requires rematerializing at most $L_{k,j} - 1$ tensors, and so on. This gives a total cost bounded as follows (using $\lesssim$ to denote inequality up to constant factors).

$$\begin{aligned}
\sum_{i \in \text{group } 1} C_{i,k} &\leq \sum_{l=0}^{L_{k,j}} L_{k,j} - l \\
&\lesssim (L_{k,j})^2 \\
&= \left(\frac{4(L_k - 2)}{k - 1} + 1 \right)^2 \\
&\lesssim \frac{L_k^2}{k^2}
\end{aligned}$$

Next, we compute the total cost of calculating all the gradients between checkpoints j and $j+1$. When the algorithm begins to compute group j, it has j pieces of extra memory, allowing it to further subdivide group j into $j+1$ intervals. By the intermediate checkpointing lemma, each of these intervals is of length at most $\frac{4(L_{k,j}-2)}{j-1} + 1$. We have

$$\begin{aligned}
\sum_{i \in \text{group } j} C_{i,k} &\leq j \sum_{l=0}^{\frac{4(L_{k,j}-2)}{j-1}+1} \frac{4(L_{k,j} - 2)}{j - 1} + 1 - l \\
&\lesssim j \left(\frac{4(L_{k,j} - 2)}{j - 1} + 1 \right)^2 \\
&\lesssim \frac{L_{k,j}^2}{j}.
\end{aligned}$$

Summing over the at most k checkpoints j, we conclude

$$\begin{aligned} C_k &\lesssim L_k + \sum_{j=1} \frac{L_{k,j}^2}{j} \\ &= L_k + L_{k,j}^2 H_k \\ &\lesssim L_k + \frac{L_k^2}{k^2} \log k \end{aligned}$$

where H_k is the kth harmonic number. □

Lemma 3.4 (Intermediate Checkpointing)**.** *Consider the behavior of the DTR algorithm using the heuristic h_{e^*}, when computing gradients for the backward pass. Suppose, immediately prior to the computation of gradient $\hat{t}_i$, we have $2 + k$ pieces of free memory ($k \geq 0$), and that $\hat{t}_{i+1}$ is in memory. Suppose also that forward tensor t_j is the first resident ancestor of $\hat{t}_i$, so that we will rematerialize t_{i-1} starting from t_j to compute $\hat{t}_i$. Finally, suppose that t_j is never evicted until it is banished.*

Then, immediately after computing $\hat{t}_i$, memory contains a set of "checkpoint" tensors $\mathcal{C}$ with the following properties:

1. *The tensors in $\mathcal{C}$ remain in memory until they are banished.*

2. *The gap between neighboring tensors in $\mathcal{C}$ satisfies*

$$L \leq \frac{2((i-j)-1)}{k+1}$$

Proof. We begin by analyzing the state of memory after computing $\hat{t}_i$. Since we started with $2+k$ pieces of free memory, and rematerialized t_{i-1} starting from t_j, Lemma 3.3 tells us that, after rematerializing t_{i-1}, the gaps in memory between t_j and t_{i-1} are all bounded by

$$L \leq \frac{2((i-j)-1)}{k+1}.$$

We need to evict one additional item from memory, in order to compute $\hat{t}_i$. After this single eviction, the maximum gap is no more than doubled. We conclude that, after computing the first gradient, the maximum gap is no more than $2L$.

It remains to show that the maximum gap in $\mathcal{C}$ does not become larger than $2L$ during the remaining steps of rematerialization. To show this, we first note that the computation of the next gradient, $\hat{t}_{i-1}$, begins with two units of free memory (having just banished $\hat{t}_{i+1}$ and t_i). We also note that the number of evicted tensors that need to be rematerialized for this gradient computation does not exceed $2L$. Therefore, when performing the intermediate rematerializations necessary to rematerialize t_{i-2}, it is always possible to evict a tensor with a heuristic value less than $2L$. Since we evict the tensor with the smallest heuristic value, we will never create an evicted path of length greater than $2L$.

This argument can be applied for every gradient computed between $\hat{t}_i$ and $\hat{t}_{j+1}$, which shows that the desired properties of $\mathcal{C}$ are maintained. □

3.2 Adversarial Overhead

Using a simple heuristic, DTR can match the performance of static checkpointing on linear feedfoward networks despite lacking advance knowledge of the architecture. However, we prove below that DTR cannot always match the performance of optimal static checkpointing on an arbitrary network because it cannot access or reorder the network. By contrast, an optimal static algorithm can reorder the same example to compute each feedforward network sequentially, requiring only N computations.

Theorem 3.2. *For any deterministic heuristic h, there exists an N-node network on which DTR with budget $B \leq N$ requires $\Omega(N/B)$ times more tensor computations than optimal static checkpointing.*

Proof. We will prove this theorem by designing an adversarially generated graph that forces DTR to repeatedly rematerialize evicted tensors. Our architecture simultaneously leverages the static planner's ability to reorder computations, to avoid repeated computation of evicted

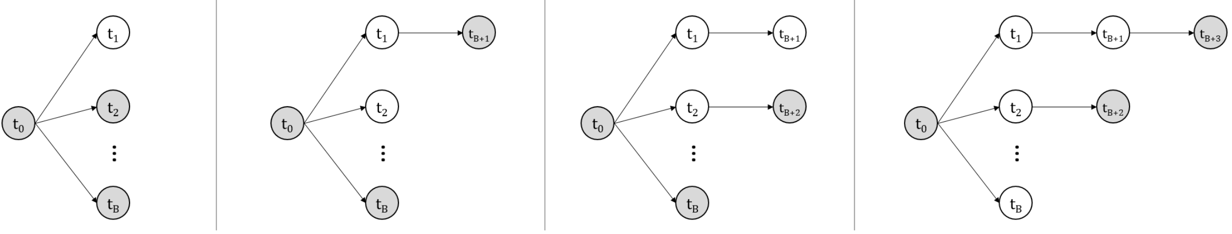


Figure 4.3: An example construction of an adversarial graph. Gray tensors are in memory (t_0 must always be in memory). The initial tensor t_0 has B paths descending from it, so there is always some path from t_0 with no resident tensors. The adversarial construction chooses to place the next node at the end of such an entirely evicted path.

tensors.

Since DTR is a dynamic algorithm, it must choose which tensor to evict at time T based only on the portion of the graph computed up to time T. Our adversarial architecture generator builds the network one node at a time, choosing the next node based on the previous choice of the DTR algorithm. The construction is as follows:

1. The graph begins with tensor t_0, which, by the behavior of DTR, must remain in memory. Tensor t_0 has B children, t_1 through t_B.

2. After step B of the computation, one of t_0's children must no longer be in memory. Call this evicted child t_* The next node revealed by the adversary is the child of t_*, causing DTR to rematerialize t_*.

3. The adversary continues to repeat this construction. Since t_0 has B children, but there are only $B-1$ units of memory to allocate among its descendants, there must be some path from t_0 that contains no resident tensors. The adversary reveals the next resident tensor on the end of that path, causing DTR to rematerialize the entire path. This repeats until we have revealed all N nodes of the graph.

An example construction of the adversarial architecture is given in Figure 4.3.

Next, we analyze the computation of DTR on this graph. To do this, we sum the cost of computing each tensor t_1 through t_N. Consider the architecture of the final revealed network, and let L_j denote the length of the path starting from t_j, where $j = \{1, \ldots, B\}$ so that t_j is a direct child of t_0. Since our adversary places the next node such that the entire path must be rematerialized, the total cost of computing this graph dynamically is

$$\begin{aligned} C &= \sum_{j=1}^{B} \sum_{i=1}^{L_j} i \\ &= \sum_{j=1}^{B} \frac{1}{2} L_j (L_j + 1) \\ &\approx \sum_{j=1}^{B} L_j^2 \end{aligned}$$

where $\approx$ hides constant factors. This sum is minimized when the L_j are all equal, which gives $L_j = (N-1)/B$. The cost of computing all the tensors is therefore at least

$$\begin{aligned} C &\gtrsim \sum_{j=1}^{B} N^2/B^2 \\ &= N^2/B \end{aligned}$$

To finish the proof, we upper bound the cost of the optimal static algorithm on this adversarial graph by exhibiting one static checkpointing algorithm and analyzing its behavior. The static algorithm may observe the entire structure of the N nodes, and rearrange the computation in any equivalent order.

Consider the static algorithm that computes the entire graph one path at a time. That is, the algorithm first computes t_1 and all its children (requiring only two units of memory, with no rematerializations), then computes t_2 and all its children (again, reusing the same two units of memory), until all B paths are computed. The total cost is therefore $\Theta(N)$.

We see that DTR requires $\Omega(N^2/B)$ computations to compute the tensors in this graph,

whereas a static checkpointing algorithm would only require $\Theta(N)$ computations. We conclude that when DTR is run with a deterministic heuristic, there exists an architecture on which it requires at least $\Omega(N/B)$ times the runtime of a statically checkpointed evaluation. □

4 Heuristic Evaluation

We simulated DTR on a variety of models to empirically evaluate its checkpointing performance across different heuristics and compare it to the static checkpointing schemes examined in Jain et al. (2019). DTR enables training under restricted memory budgets and closely matches the performance of an optimal baseline.

4.1 Experimental Setup

To model a realistic execution setting for DTR, we instrumented PyTorch (Paszke et al., 2019b) to log operations performed, metadata on tensors and operators (including sizes, compute times, and parent tensors), and deallocations during the execution of various models. We replayed the logs in a simulator that models the behavior of DTR in the style shown in Figure 4.1. The simulator tracks the tensors in memory at any given time, chooses tensors to evict per the heuristic when the memory budget is exceeded, and sums the total cost of the model operators and rematerializations. For verisimilitude, the simulator also models the semantics of various low-level PyTorch implementation details, including tensor aliasing, in-place mutation, and multi-output operations.

We gathered logs from several static models examined in recent work, such as Jain et al. (2019) and Peng et al. (2020), in addition to three dynamic models (LSTM, TreeLSTM, and Unrolled GAN); each log corresponds to an execution of the forward pass, computing the loss, and performing the backward pass. The simulator also enforces the additional condition that gradients for all trainable weights be resident at the end of the simulation in order to model the requirements for performing a full training step. See Section 4.3 for a full technical specification of the simulator and log format.

4.2 Heuristics Examined

We examine variants of the evicted neighborhood–based h_{DTR} heuristic described in Sec. 2 (on which we establish formal bounds) as well as heuristics inspired by past work in caching and checkpointing. All following heuristics are defined as a score function in terms of the metadata $m(t)$, $s(t)$, and $c_0(t)$, where the tensor with the minimum score is evicted.

In addition to h_{DTR}, we consider $h^{\text{eq}}_{\text{DTR}}$, which uses an equivalence class–based approximation $\tilde{e}^*$ for e^*, and $h^{\text{local}}_{\text{DTR}}$, which only uses individual tensors' costs instead of costs over evicted neighborhoods. We also include comparisons against other variants of h_{DTR} in Section 4.5, but here we focus primarily on these in particular because 1. $h^{\text{local}}_{\text{DTR}}$ lets us assess the importance of tracking evicted neighborhoods at run time and 2. $h^{\text{eq}}_{\text{DTR}}$ lets us evaluate how well $\tilde{e}^*$ approximates e^* in practice. We define the h_{DTR} variants as:

$$h_{\text{DTR}} \stackrel{\text{def}}{=} \frac{c_0(t) + \sum_{t' \in e^*(t)} c_0(t')}{m(t) \cdot s(t)}, \quad h^{\text{eq}}_{\text{DTR}} \stackrel{\text{def}}{=} \frac{c_0(t) + \sum_{t' \in \tilde{e}^*(t)} c_0(t')}{m(t) \cdot s(t)}, \quad h^{\text{local}}_{\text{DTR}} \stackrel{\text{def}}{=} \frac{c_0(t)}{m(t) \cdot s(t)}.$$

Rather than using directed dependencies, $\tilde{e}^*(t)$ treats the dependency graph of tensors as undirected (thus admitting some spurious dependencies), letting us decompose the graph into a set of disjoint evicted components. We can track these evicted components efficiently using a union-find data structure with a running sum for each component. When a tensor t is evicted, its component is unioned with those of any evicted neighbors and $c_0(t)$ is added to the component's running sum. Though this enables near-constant-time merging between components (by unioning and adding the sums), union-find does not support splitting. To efficiently split components, we make another approximation: when a tensor t is rematerialized, we simply subtract $c_0(t)$ from its component's running sum and map t to a new (empty) union-find component. Since this approach removes no connections, it produces "phantom dependencies" between some tensors. In practice, we find that despite these additional dependences, $h^{\text{eq}}_{\text{DTR}}$ closely matches the performance of h_{DTR} (Section 4.4) but requires fewer operations per eviction and rematerialization.

We also consider the following heuristics inspired by past work:

$$h_{\text{LRU}}(t) \stackrel{\text{def}}{=} \frac{1}{s(t)}, \quad h_{\text{size}}(t) \stackrel{\text{def}}{=} \frac{1}{m(t)}, \quad h_{\text{MSPS}}(t) \stackrel{\text{def}}{=} \frac{c_0(t) + \sum_{t' \in e_R(t)} c_0(t')}{m(t)},$$

where $e_R(t)$ is the set of evicted tensors that would have to be rematerialized in order to rematerialize t. h_{LRU} is based on the common "least-recently used" policy for caching, h_{size} is based on `GreedyRemat` from Kumar et al. (2019) (used in TensorFlow XLA), and h_{MSPS} is based on the MSPS heuristic from Peng et al. (2020). We also include a random baseline, $h_{\text{rand}}(t) \stackrel{\text{def}}{=} X \sim U(0,1)$, to assess how well a heuristic using no metadata whatsoever performs.

4.3 Simulator Specification

In this section, we provide a detailed technical specification of the DTR simulator. This includes fundamental abstractions, formal definitions of heuristics, pseudocode, runtime optimizations, and details about the log-replaying mechanism.

4.3.1 Fundamental Abstractions

We designed the simulator to support computations logged from PyTorch (see Section 4.3.6). In PyTorch, a tensor is a view (containing metadata) of a buffer; multiple tensors can point to a single buffer. This allows us to model the various aliasing relations between tensors in PyTorch (Paszke et al., 2017); other DL frameworks likely also use a similar representation.

Storage. At its core, DTR is a runtime system for reducing memory usage. As such, *storages* (i.e., buffers of memory) are the underlying unit which DTR operates on. They support the following operations:

- *size* : **Storage** $\to \mathbb{N}$: the size of the storage in bytes;

- *root* : **Storage** $\to$ **Tensor**: the tensor whose parent operation computes the contents of the storage (there is exactly 1 for each storage);

- *tensors* : **Storage** $\to$ **List**[**Tensor**]: all tensors which view the storage;

- *resident* : **Storage** $\to$ **bool**: true iff the storage is in memory;

- *locks* : **Storage** $\to$ $\mathbb{N}$: the number of locks on the storage held interally by DTR (indicating the storage is needed for pending rematerializations);

- *refs* : **Storage** $\to$ $\mathbb{N}$: the number of external references to the storage, i.e., those held by user code.

We say a storage S is *evictable* if and only if $\mathit{resident}(S) \land \mathit{locks}(S) = 0$.

Tensor. Each tensor t has an associated "parent" operation $\mathit{op}(t)$ which computes it (potentially along with $\mathit{storage}(t)$, its underlying storage).

Each tensor t also has an external reference count $\mathit{refs}(t)$; in particular, each storage S has $\mathit{refs}(S) = \sum_{t \in \mathit{tensors}(S)} \mathit{refs}(t)$. The external reference count is used to track whether a tensor is still live in the source program or whether it should be treated as having been deallocated by the source program. Additionally, t is an *alias* iff $t \neq \mathit{root}(\mathit{storage}(t))$, meaning that t is a view of a storage created by a different parent operator. For convenience, we define $\mathit{size}(t)$ to be 0 if t is an alias and $\mathit{size}(\mathit{storage}(t))$ otherwise (since the metadata will likely be on CPU).

Unlike storages, a tensor t is resident when $\mathit{storage}(t)$ is resident *and* $\mathit{op}(t)$ has been performed *after* $\mathit{storage}(t)$ last became resident. This condition is denoted as $\mathit{defined}(t)$, and models the behavior of our PyTorch prototype implementation where the whole tensor object is destroyed upon storage eviction (including metadata about the view, like striding

and offset).[1] Thus, before an operation depending on t can be executed, $\mathit{defined}(t)$ must be satisfied, given our assumption that views of a storage must be evicted once the underlying storage has been evicted. Note that for a non-alias tensor t, we have $\mathit{resident}(\mathit{storage}(t))$ if and only if $\mathit{defined}(t)$.

Operator. An *operator* represents a fundamental unit of computation in DTR. Operators are assumed to be pure functions of their arguments, not depending on any other external state (see Sec. 4.3.6 for our handling of mutation). As such, each operator f has an associated compute cost $\mathit{cost}(f) \in \mathbb{N}$. We assume each f has type $\textbf{List}[\textbf{Tensor}] \to \textbf{List}[\textbf{Tensor}]$ and define $\mathit{inputs}(f)$ and $\mathit{outputs}(f)$ to be the input and output tensors of f, respectively.

4.3.2 Formal Metadata Definitions

While our abstract description of DTR in Figure 4.1 is over tensors, the simulator operates over storages rather than tensors. Thus we must define the metadata our heuristics use over storages, providing notions of cost, staleness, and data dependencies for storages rather than for tensors.

Cost. For a given storage S, we define the compute cost of S as

$$\mathit{cost}(S) := \sum_{t \in \mathit{tensors}(S)} \mathit{cost}(\mathit{op}(t)).$$

This is a worst-case estimation: it represents the compute cost which is incurred when every tensor view of S needs to be rematerialized. An alternative definition is simply $\mathit{cost}(\mathit{op}(\mathit{root}(S)))$, which may be acceptable as aliasing operations are typically much cheaper than non-aliasing.

---

[1] The storage field in a PyTorch tensor is immutable. In principle, we could have changed this to permit reassigning views of evicted storages to `null` and ensure the storages be rematerialized when needed, but this would have required much more extensive modifications to the codebase, which may rely on the invariant of immutable storage pointers.

Staleness. We estimate the staleness of S by tracking the *last access* time of each $t \in tensors(S)$. The last access time $last_access(t)$ is defined as the most recent time when t was referenced by a queued operation. Naturally, we define

$$last_access(S) := \max_{t \in tensors(S)} last_access(t).$$

Staleness, given the current time $\mathcal{T}$, is then defined as

$$stale_{\mathcal{T}}(S) := \mathcal{T} - last_access(S).$$

Data dependencies. The dependencies of S are the set of storages

$$deps(S) := \{storage(u) \mid \exists t.\ t \in tensors(S) \wedge u \in inputs(op(t))\} \setminus \{S\}.$$

Note that we exclude S since it is not a true dependency (each alias tensor in $tensors(S)$ technically "depends" on S). Another possible approximation of the above is to simply take the dependencies of $root(S)$; although this ignores potential dependencies of aliasing operations, it is precise if all aliasing operations depend only on S.

We now define the *dependents* of S as the set $deps^{\top}(S)$ consisting of all T with $S \in deps(T)$. With this definition, DTR can operate over the dependency graph (V, E) where V is the set of storages and $(S, T) \in E$ iff $S \in deps(T)$. Note that (V, E) is implicitly indexed by time $\mathcal{T}$, with V being the set of non-banished but at-least-once computed storages at $\mathcal{T}$ and E being the dependency relations at $\mathcal{T}$.

Evicted neighborhood. The *evicted neighborhood* e^*, as defined in Section 2, works without modification over the storage dependency graph. We define it here for completeness. Let $deps_e(S)$ be the evicted subset of $deps(S)$, and likewise for $deps_e^{\top}(S)$. Now, let D_e and $D_e^{\top}$

be the transitive closures of the relations

$$\{(T,S) \mid T \in \mathit{deps}_e(S)\} \quad \text{and} \quad \{(S,T) \mid T \in \mathit{deps}_e^\top(S)\},$$

respectively. Then, $e^*(S) := \{T \mid (T,S) \in D_e\} \cup \{T \mid (S,T) \in D_e^\top\}$. Intuitively, $e^*(S)$ is the set of evicted storages that must be resident to compute all $t \in \mathit{tensors}(S)$, together with the set of evicted storages T that need S to be resident before all $t \in \mathit{tensors}(T)$ can be computed.

Relaxed (Union-Find) evicted neighborhood. Actually tracking $e^*(S)$ can be computationally expensive due to the directed and changing nature of the graph. For each S, $e^*(S)$ depends on its specific ancestors and descendants, which can vary as tensors are evicted and rematerialized. An exact solution would likely involve a dynamic graph connectivity data structure, which would greatly increase the complexity of the simulator's implementation.

We find an approximate solution by relaxing the definition of the evicted neighborhood. At a high level, our solution works as follows: given a storage dependency graph $G = (V, E)$, we first forget edge directions to obtain the undirected dependency graph $\tilde{G}$. Now, let $\tilde{G}_e$ be the subgraph obtained by removing all resident storages (and any edges including them). Each connected component of $\tilde{G}_e$ is then an *evicted component*, with each evicted $T \in V$ belonging to exactly one component $\epsilon^*(T)$.

Importantly, we track these evicted components using a *Union-Find* (UF) data structure, which efficiently supports merging and obtaining static set metadata. Each component tracks the sum of the compute costs of its elements (with the union of two components having the sum of each constituent cost). We denote the associated UF set for a storage T by $T.\mathit{set}$, which is mutable state.

We can now define the relaxed evicted neighborhood for a resident storage S as

$$\tilde{e}^*(S) := \left(\bigcup_{T \in deps_e(S)} T.set \right) \cup \left(\bigcup_{T \in deps_e^\top(S)} T.set \right).$$

Note that in practice, no UF unions are performed when querying this approximation. Instead, we collect and merge the set metadata separately, as otherwise we would erroneously merge evicted components during heuristic evaluation. This approximation reduces the worse-case time complexity of querying compute costs over the neighborhood to be linear in the number of adjacent storages, as opposed to all ancestor and descendant storages.

However, rematerializing a tensor in an evicted component creates a split in the component and *splitting* is not a supported operation on UF data structures.[2] Approaches to splitting would also need to recover the original compute costs of each set, which may require traversing the whole set if done naively. To handle splitting more efficiently, we use the following approximation: when a (previously) evicted storage S is rematerialized, we first set $S.set.cost := S.set.cost - cost(S)$, and then assign $S.set := \emptyset$ (*i.e.*, assign S to a new empty UF set). Note that when a storage is first computed, its evicted component is also initialized to be empty. While resident storages thus never count towards the compute cost of a component, "phantom connections" between evicted storages may accumulate over time (likely depending on the connectedness of the underlying dependency graph). Despite this limitation, this approximation worked well in practice, as seen in the simulated and prototype results.

4.3.3 Formal Heuristic Definitions

Having defined the metadata above, we can now formally define the h_{DTR} variants examined. (Recall that h_{DTR} heuristics compute a score using measures of size, computational cost, and staleness and evict the tensor with the smallest score, corresponding to the intuition that

[2] This can be seen as a variant of the Union-Find-Split problem, which typically requires the use of more complex data structures such as link-cut trees.

the tensor evicted should be large, unlikely to be rematerialized, and cheap to rematerialize if it does need to be rematerialized.)

$$h_{\mathrm{DTR}}(S) := \frac{cost(S) + \sum_{T \in e^*(S)} cost(T)}{size(S) \cdot stale_{\mathcal{T}}(S)}.$$

$$h^{\mathrm{eq}}_{\mathrm{DTR}}(S) := \frac{cost(S) + \sum_{T \in \tilde{e}^*(S)} cost(T)}{size(S) \cdot stale_{\mathcal{T}}(S)} \approx \frac{cost(S) + cost^*(S)}{size(S) \cdot stale_{\mathcal{T}}(S)}$$

Note that the simulator implementation uses the splitting approximation described above, with $\tilde{e}^*(S)$ depending on the specific sequence of evictions and rematerializations. $cost^*(S)$ in the second expression is used to denote this statefulness.

$$h^{\mathrm{local}}_{\mathrm{DTR}}(S) := \frac{cost(S)}{size(S) \cdot stale_{\mathcal{T}}(S)}.$$

4.3.4 Implementation Details

Runtime state. In what follows, we denote the collective runtime state of the DTR simulator as R, and use the dot notation to indicate stateful reads and writes of runtime values. The simulator tracks the following runtime state:

- R.`heuristic` : $(\textbf{Storage}, \textbf{Metadata}) \to \mathbb{R}$, the eviction heuristic, interpreted as a score (the lowest-scored storage is evicted);
- R.`budget` : $\mathbb{N}$, the memory budget in bytes;
- R.`memory` : $\mathbb{N}$, the current memory usage in bytes;
- $R.\mathcal{T}$: $\mathbb{N}$, the current clock time in some unit of granularity, such as nanoseconds;
- R.`pool` : $\textbf{List}[\textbf{Storage}]$, list of all currently evictable storages.

Eviction and banishing. To evict a given storage S, we set all tensors in S to be undefined, remove S from the pool, and decrease R.`memory` by $\mathit{size}(S)$. Cached metadata are also updated as necessary.

Banishing (*permanent* eviction) is slightly subtler; in particular, it can be done for S only when $\mathit{deps}_e^{\top}(S) = \emptyset$. Banishing then proceeds by evicting S as above, but with the additional effect of removing S entirely from the dependency graph. Each $T \in \mathit{deps}^{\top}(S)$ is then locked (and effectively becomes an non-rematerializable constant). Storages locked in this way are said to be *pinned* (and have a special flag in the simulator), to distinguish them from those locked during rematerialization, and we permit them to be banished in the future. Note that banishing can be performed on evicted S when the above condition is met, in which case the eviction is skipped.

(Re)materialization. When a tensor t is to be (re)materialized, its parents' storages are first locked by incrementing the lock count (so that they don't get evicted while they are still needed) and undefined parents are recursively rematerialized. We then increment R.`memory` by $\sum_{u \in \mathit{outputs}(\mathit{op}(t))} \mathit{size}(u)$ (performing evictions as necessary), and move $R.\mathcal{T}$ forward by $\mathit{cost}(\mathit{op}(t))$. Multi-output operations must be handled carefully so as to not leak memory: we make sure to *decrease* R.`memory` by $\mathit{size}(u')$ for each $u' \in \mathit{outputs}(\mathit{op}(t))$ that was defined *prior* to the rematerialization. This models the immediate freeing of doubly computed ephemeral tensors in the PyTorch implementation. Lastly, locks on parent storages are freed and unlocked storages (including any newly rematerialized ones) are added back into R.`pool`.

Constants. The simulator models non-rematerializable constants like weights and inputs by creating dummy "constant" tensors using nullary operators with 0 cost and pinning the resulting storage. This allows the simulator to have a full picture of the computation graph. Furthermore, log-accurate banishing requires knowledge of constants (as PyTorch reference-counts constants).

4.3.5 Additional Runtime Optimizations

Banishing and eager eviction. When the final external reference to a storage S is lost, we know that the underlying DL framework would have reclaimed the memory used by S. To utilize this information as opposed to doing nothing, DTR can either banish S or simply evict S normally. When banishing, the runtime must first check that S has no evicted dependents; if it does, then we retry banishing each time a dependent is rematerialized. Banishing has the ability to free constants, but at the downside of pinning potentially exploding amounts of memory. The alternative (*eager eviction*) is easier to implement and simply involves evicting S normally (if possible). This prevents the problem of over-pinning memory, but with the downside that constants can never be evicted. In practice, eager evictions have allowed us to support lower budgets by pinning fewer values (see Section 4.5.2 for details).

Caching metadata. To avoid costly recomputations of metadata during heuristic evaluations, we cache the local cost $cost(S)$ for each S, as it only changes when new aliases are made. Additionally, for the h_{DTR} heuristic, we avoid recomputing $e^*(S)$ at each evaluation by caching and only recomputing it after evictions or rematerializations that directly affect $e^*(S)$. Such recomputations are further optimized by tracking the evicted ancestors and descendants separately (allowing them to be recomputed independently, depending on the position of the affected storage).

4.3.6 Log-Replaying Mechanism

Log format. We logged PyTorch operations as a sequence of abstract instructions corresponding to the semantics of the actions we were easily able to instrument in the framework. Every PyTorch tensor is given a unique identifier string upon creation, which is recorded and used in the log. In this section, each PyTorch tensor t corresponds to a simulator tensor $[\![t]\!]$.

The log contains the following instructions:

- `MEMORY`$(t, size)$: logs that t uses $size$ memory; treated as 0 if $[\![t]\!]$ is an alias.

- `ALIAS`(t_o, t_i): logs that $[\![t_o]\!]$ is an alias of $[\![t_i]\!]$, i.e., two different views of the same storage. t_i can either be a tensor identifier or $\perp$; if $t_i = \perp$, then t_o does not alias another tensor (t_o's parent operation created its storage).

- `CALL`$(\mathit{inputs}, \mathit{outputs}, \mathit{cost}, \mathit{op})$: logs the operator call $\mathit{outputs} = \mathit{op}(\mathit{inputs})$ with compute cost cost. This instruction is followed by $|\mathit{outputs}|$ `MEMORY` and `ALIAS` instructions to log information about each output. Each `CALL` corresponds to a simulator operator $[\![\mathit{op}]\!]$ with inputs $\{[\![i]\!] \mid i \in \mathit{inputs}\}$ and new simulator tensor outputs $\{[\![o]\!] \mid o \in \mathit{outputs}\}$.

- `MUTATE`$(\mathit{inputs}, \mathit{inputs}', \mathit{cost}, \mathit{op})$: logs the in-place (mutating) operator call $\mathit{op}(\mathit{inputs})$ with compute cost cost, which modifies $\mathit{inputs}' \subseteq \mathit{inputs}$.

- `CONSTANT`(t): logs that $[\![t]\!]$ is a constant, and is followed by a `MEMORY` instruction.

- `COPY`(t_o, t_i): logs a new identifier t_o with $[\![t_o]\!] = [\![t_i]\!]$. This increments $\mathit{refs}([\![t_i]\!])$. This happens when Python code like "`x = y`" is called where `y` is a PyTorch tensor and `x` is a fresh variable; this action neither creates a new storage nor a new view but only has `x` *point to* the same view as `y`.

- `COPYFROM`(t_o, t_i): logs the PyTorch code $t_o = t_i$ where each side is an existing tensor. This decrements $\mathit{refs}([\![t_o]\!])$, increments $\mathit{refs}([\![t_i]\!])$, and updates $[\![t_o]\!] \mapsto [\![t_i]\!]$. Intuitively, this corresponds to Python code like "`x = y`" where `y` is a PyTorch tensor and `x` was already assigned to a PyTorch tensor; in PyTorch, `x` is mutated to match `y`.

- `RELEASE`(t): logs the destructor of the PyTorch tensor t. This decrements $\mathit{refs}([\![t]\!])$.

Supporting mutation. To support mutation from in-place operators, the simulator adds a "reference layer" that mutates cloned tensors, allowing for a uniform interface for all operators. Given a mutation instruction `MUTATE`$(\mathit{inputs}, \mathit{inputs}', \mathit{cost}, \mathit{op})$, let i_{new} be a new unique

identifier for each $i \in inputs'$, and let $inputs'_{new} = \{i_{new} \mid i \in inputs'\}$. We then proceed by treating op as a pure operator from $inputs$ to $inputs'_{new}$, where each newly created simulated tensor $[\![i_{new}]\!]$ is non-aliasing and has size $size(storage([\![i]\!]))$. Lastly, we decrement $refs([\![i]\!])$ and update the mapping $[\![i]\!] \mapsto [\![i_{new}]\!]$. Intuitively, we are modeling the transformation

$$op(t) \rightsquigarrow \texttt{Tensor}\ t' = copy(t); op(t'); t = t'.$$

Note that in our prototype implementation, a mutation of i may produce incorrect results when $[\![i]\!]$ is an alias, since the mutation layer would create a clone but aliases would still point to the old storage. Potential solutions in real implementations would be to propagate the above rewrite to all aliases of a storage (costly) or to mutate storage pointers (which would have increased the complexity of our modfications to PyTorch).

Output condition. All live tensors at the end of a log (i.e., all t with $refs(t) > 0$) are treated as necessary outputs (namely, gradients, the loss value, and the prediction). They are thus rematerialized (if evicted) and locked to ensure they persist. This prevents the simulator from incorrectly reporting better results by evicting computed weight gradients and never rematerializing them. This permits the user to perform the weight update step outside of DTR immediately after the backward pass ends. Based on our observations of PyTorch's `optimizer` gradient updates, we could also support performing these updates within DTR, since a parameter update simply performs in-place mutating additions (`add_`) of scaled gradients to the parameters.

4.4 Comparing DTR Across Heuristics

For all models in Figure 4.4, DTR executed a training step using a small fraction of the normal memory required with limited compute overhead. Furthermore, unlike existing static approaches, DTR automatically supports models with arbitrary dynamism, though it thrashed at lower budgets for LSTM and TreeLSTM. In all cases, results show that heuristics in-

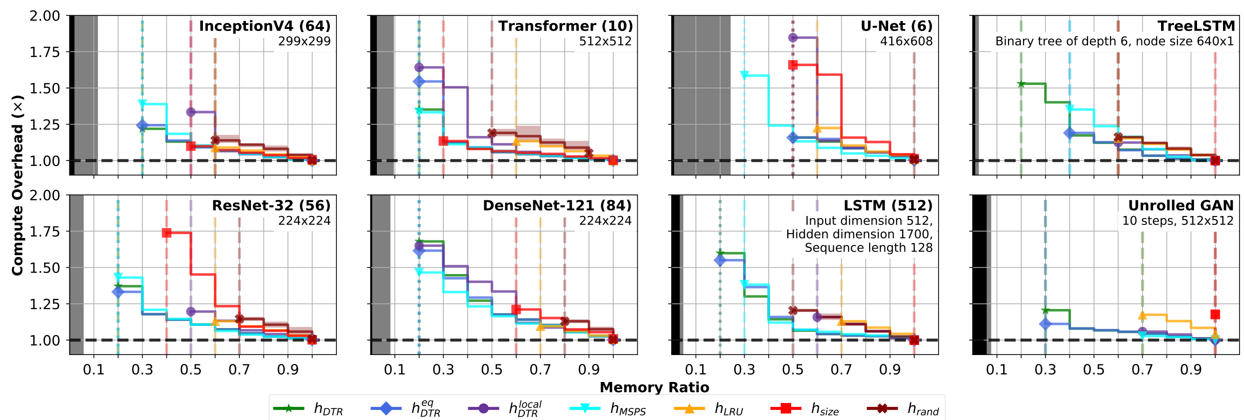


Figure 4.4: Simulated results comparing different heuristics on various models, showing the rate of computational slowdown for different budgets (fractions of the original peak memory usage). The black area in each graph corresponds to the memory required to store inputs and weights, while the gray area denotes the single operator requiring the most memory to be live at once. The dashed and dotted lines represent the last ratio before thrashing ($\geq 2\times$ slowdown) and out-of-memory errors, respectively. All logs were produced by running each model 50 times on a single input on a machine with an NVIDIA Titan V GPU (CUDA 10.1, CuDNN 7.6.4) and a 16-core AMD Ryzen Threadripper 1950X on Ubuntu 18.04, logging the final "warmed-up" run.

corporating more information about chain rematerializations (h_{DTR}, $h_{\mathrm{DTR}}^{\mathrm{eq}}$, and h_{MSPS}) can operate on lower budgets and perform fewer rematerializations than heuristics using less information. However, these complex heuristics also introduce more *runtime* overhead, which must be considered when implementing DTR. In particular, our simulations showed that h_{DTR} incurred up to 2 orders of magnitude more metadata accesses per batch compared to $h_{\mathrm{DTR}}^{\mathrm{eq}}$, and up to 3 orders of magnitude more compared to $h_{\mathrm{DTR}}^{\mathrm{local}}$ (see Section 4.5.3). The fact that $h_{\mathrm{DTR}}^{\mathrm{eq}}$ closely matches the performance of h_{DTR} while incurring much less runtime overhead suggests that it would be more effective in practice. Note that even simple heuristics like h_{LRU}, which require only modest runtime overhead, typically enabled training with 30% less memory, indicating that some memory savings from checkpointing can be very readily obtained.

4.5 Ablation Study of DTR Heuristics

In addition to comparing the overhead in terms of additional tensor computations, we also consider the runtime overhead of different h_{DTR} configurations in terms of the number of tensor accesses by heuristic computations and metadata updates. We also compare different eviction policies for the h_{DTR} heuristics: ignoring deallocations, eager eviction, and banishing.

4.5.1 Data Sources

First, we will analyze the three sources of information (metadata) for the h_{DTR} heuristic. Let us consider a parameterized version of h_{DTR} defined as $h'_{\mathrm{DTR}}(s, m, c)(t) = c(t)/[m(t) \cdot s(t)]$, where s is a measure of staleness, m is a measure of size, and c is a measure of compute cost. For this study, we take s and m to be the staleness and size functions defined in Section 4.3. For compute cost c, we compare the following alternatives: the full e^*, the approximation $\tilde{e}^*$, and the local cost (cost of the parent operator only). We allow each measure to be entirely ablated (e.g., $s(t) = 1$, which we denote $s = \mathrm{no}$).

In the following figures, we specifically have $s, m \in \{\mathrm{yes}, \mathrm{no}\}$ and $c \in \{e^*, \texttt{EqClass}, \texttt{local}, \mathrm{no}\}$. Each figure fixes a choice of c, varying s and m.

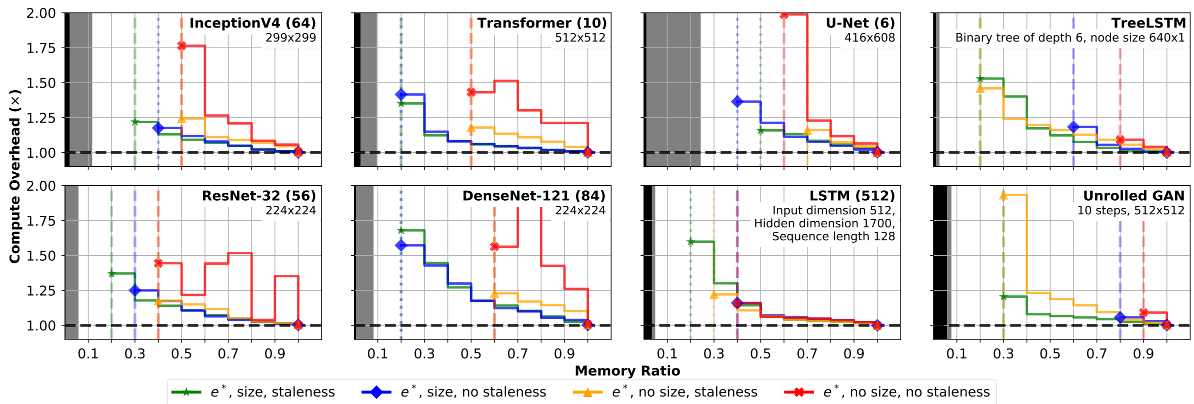


Figure 4.5: Results for fixed $c = e^*$, varying s and m.

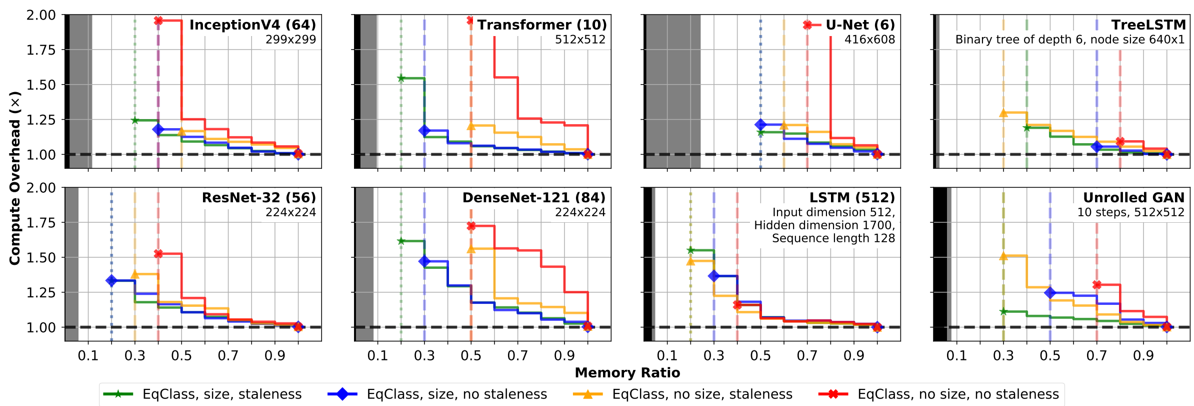


Figure 4.6: Results for fixed $c =$ `EqClass`, varying s and m.

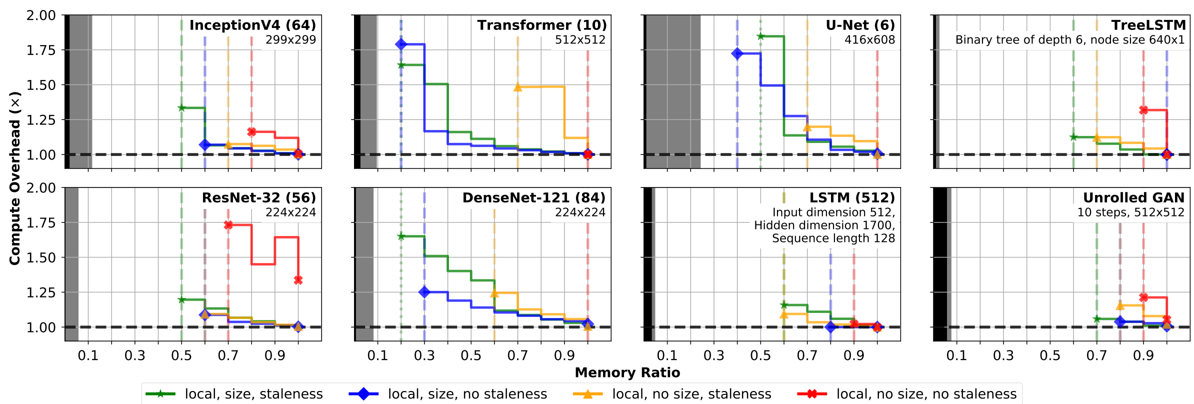


Figure 4.7: Results for fixed $c =$ `local`, varying s and m.

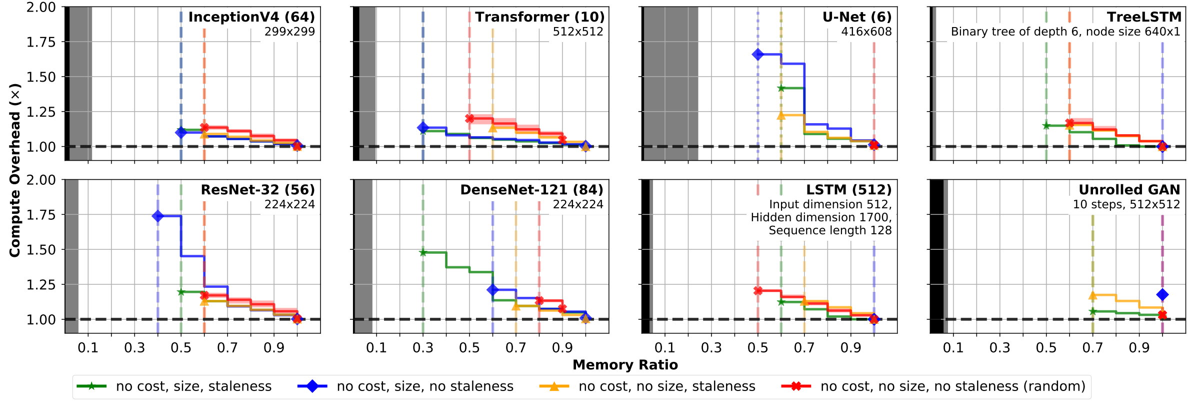


Figure 4.8: Results for fixed $c =$ no, varying s and m.

The general trend shown in Figures 4.5, 4.6, 4.7, and 4.8 is that higher metadata complexity (corresponding to more precise notions of the evicted neighborhood) enables more savings, while staleness and size are required for acceptable computational overhead. It is interesting to note that the importance of staleness and size depends on the specific model architecture. For example, cost and size alone each do far better than using both cost and staleness for the static models (DenseNet, ResNet, UNet), whereas the opposite is true for the dynamic models. This may be due to model depth or the distribution of tensor sizes or to the increasing impact of individual checkpoints at lower budgets; further research may shed more light on the influence of model-specific characteristics like these. Additionally, we may note that the $\tilde{e}^*$ approximate cost performs comparably to the e^* exact cost while requiring less information, validating our claim that the equivalence classes are a useful approximation.

In general, the best-performing of these heuristics were those with non-ablated choices of s, m, and c, hence our focus on the h'_{DTR} variants with e^*, $\tilde{e}^*$, and local cost (h_{DTR}, $h^{\text{eq}}_{\text{DTR}}$, and $h^{\text{local}}_{\text{DTR}}$, respectively).

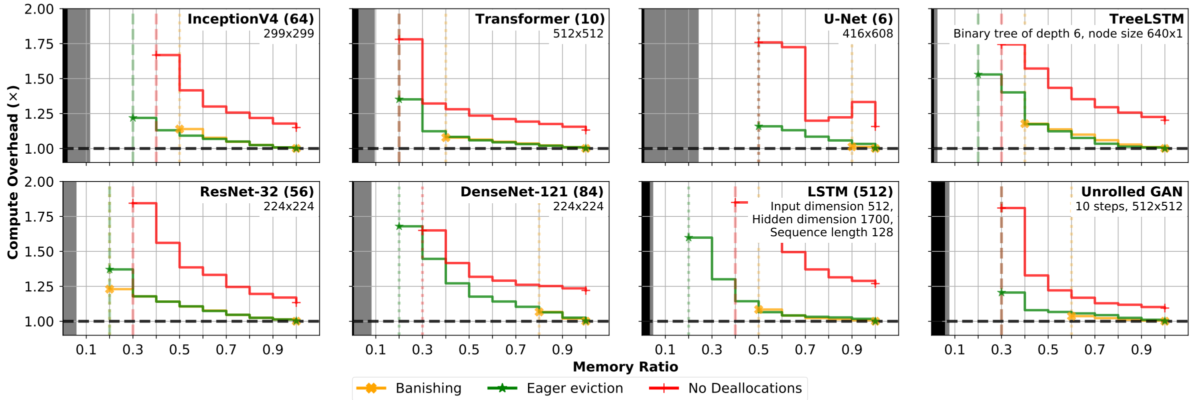


Figure 4.9: Results for the h_{DTR} heuristic, comparing banishing and eager evictions.

4.5.2 Banishing and Deallocations

For the following trial, we compared the h_{DTR} heuristic with banishing (permanent removal) against that with eager evictions, as described in Section 4.3.5. We also compare both deallocation-aware approaches against simply ignoring deallocations. We only used e^* cost because it performed much better than local cost and because it would have been more complicated to update the definition of $\tilde{e}^*$ to account for banished neighbors. The results are shown in Figure 4.9.

As the curves show, banishing is not able to achieve the same budgets across most models tested as eager eviction. For UNet, the difference is large: banishing can only support 90% of the baseline budget (and OOMs at 0.8 ratio), while eager eviction can support 50% of the baseline budget. However, banishing still attains low budgets on most models, even obtaining better computational overhead under the same budget and savings for ResNet. Since banishing potentially allows for greatly lowered runtime overhead, implementations of DTR can consider conditionally enabling it in situations where the tradeoff is more desirable.

Compared to ignoring deallocations, both banishing and eager eviction obtain noticeably lower rematerialization overhead. This shows that valuable information is captured by deallocations, and that DTR can make good use of it.

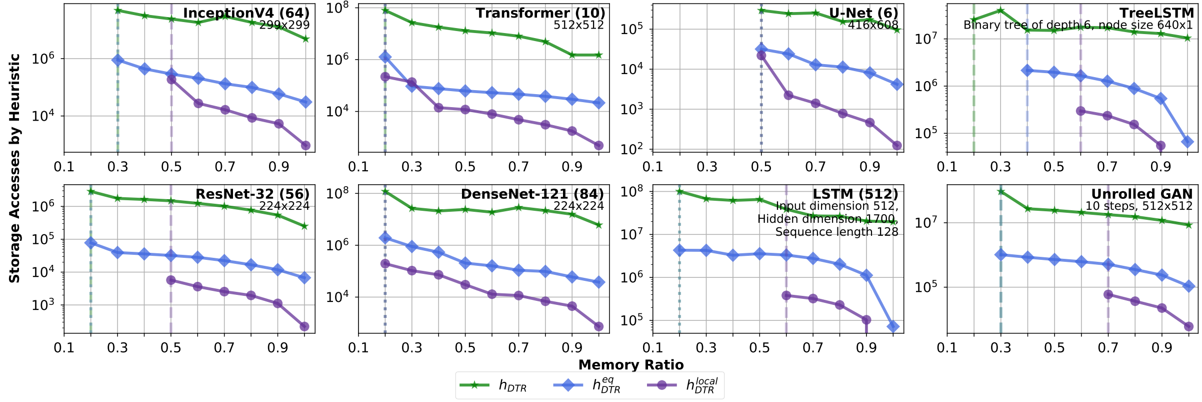


Figure 4.10: Total storages accesses incurred by heuristic evaluations and metadata maintenance, compared across different memory ratios, for the 3 main h'_{DTR} variants.

4.5.3 Runtime Overhead

For this experiment, we tracked the number of storage accesses made during evaluations of heuristics and maintenance of metadata. We chose this metric over wall-clock time, since our Python implementation of the simulator is not heavily optimized and may not accurately correspond to the real performance of the runtime. Storage accesses, on the other hand, do reflect operations that would be performed by a real implementation. For the h_{DTR} heuristic, this included each storage visited during the updating and rebuilding procedures for maintaining e^* for resident storages. For the $h_{\mathrm{DTR}}^{\mathrm{eq}}$ heuristic, this included each storage visited whenever the Union-Find data structure was traversed for each evicted component (which occurs mainly during merging and when reading the compute cost). The $h_{\mathrm{DTR}}^{\mathrm{local}}$ heuristic does not need to maintain any non-local metadata. For all heuristics, each heuristic evaluation counted as one storage access.

As Figure 4.10 shows, the accesses made by each heuristic are generally separated by at least an order of magnitude. This confirms our intuitions about the runtime overhead of each heuristic, and supports our choice of $h_{\mathrm{DTR}}^{\mathrm{eq}}$ as a good middle ground (in terms of both runtime and computational overhead). However, these overhead figures could be improved

with better-optimized implementations of the heuristics, as our implementation recomputes heuristics often, even when it may be possible to store the scores for tensors and maintain them in a sorted order. (Reformulating staleness to avoid having to use the current time might help.) Using persistent data structures that can be incrementally updated and maintain a sorted order will make these heuristics much more efficient, though this would also increase the complexity of the implementation.

4.6 Comparing DTR to Static Techniques

We compared the performance of DTR using h_{DTR}, $h_{\text{DTR}}^{\text{eq}}$, and (as a simple baseline) h_{LRU} against static checkpointing techniques, including the optimal Checkmate tool of Jain et al. (2019). As Figure 4.11 shows, DTR's h_{DTR} and $h_{\text{DTR}}^{\text{eq}}$ heuristics obtain performance remarkably close to Checkmate's optimal solutions; even the much simpler h_{LRU} heuristic obtains superior performance relative to the static baselines. While Checkmate requires full ahead-of-time knowledge of the model and seconds or minutes per budget to compute guaranteed-optimal solutions using an integer linear programming (ILP) solver, *DTR finds comparable solutions dynamically and in milliseconds* without ahead-of-time knowledge of the model.

5 Prototype Implementation

We implemented a DTR prototype[3] in PyTorch and evaluated its performance on a variety of models. We chose PyTorch because its eager mode of execution ("define by run") accomodates arbitrary control flow in models but makes static analysis more difficult; hence, it is a setting where DTR's online nature is an asset. Per the results in Sec. 4, we implemented $h_{\text{DTR}}^{\text{eq}}$ as the prototype's heuristic. *The core system was implemented in only 1,161 lines of code and made no deep modifications to PyTorch's memory management internals or tensor abstractions, illustrating the simplicity of our system.* The remaining 2,647 lines of changes were primarily

[3]Publicly available at `https://github.com/uwsampl/dtr-prototype`

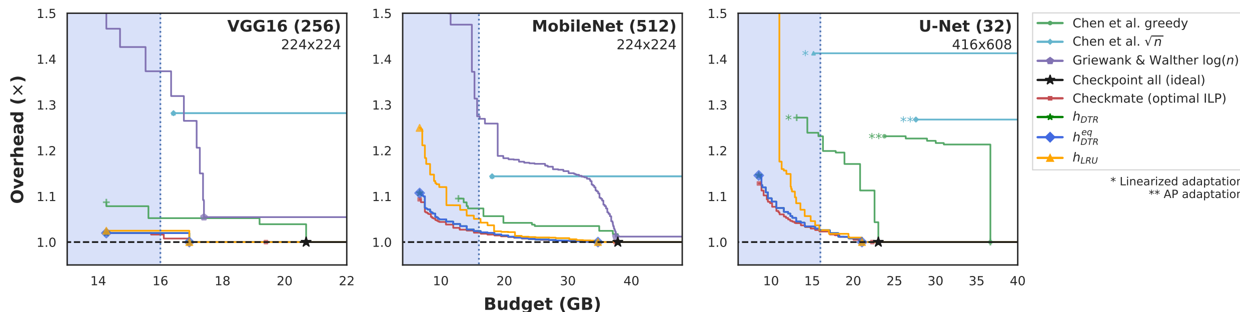


Figure 4.11: DTR's overhead from operators is competitive with Checkmate's, which uses ILP to produce an optimal rematerialization schedule. This comparison extends Figure 5 in Jain et al. (2019) by adding the DTR simulator as a "solver" that translates Checkmate's Keras-based graph representation into the DTR simulator's representation. To produce this comparison, we modified Jain et al. (2019)'s evaluation artifact because the PyTorch logs in Section 4.3.6 did not contain some information that past checkpointing techniques require (such as which backward operators correspond to which forward ones). Also included in the comparison (from the original experiment) are the Griewank and Walther (2000) Treeverse algorithm and variants of the Chen et al. (2016) checkpointing algorithm (modified to handle skip connections like those in ResNet).

boilerplate operator overloads used to dispatch tensor operations through DTR's core logic.

5.1 Integration into PyTorch

To avoid modifying PyTorch's core systems, our DTR prototype is implemented as a wrapper over PyTorch's existing tensor implementations. Namely, we add a new tensor representation into PyTorch called a `CheckpointTensor`, which is simply a wrapper over an existing PyTorch tensor that additionally tracks the tensor's parent operation and other metadata (such as the last access time and the cost of the parent operation, which is timed when the tensor is first created) and registers the tensor in the DTR runtime system. Timing operators for metadata purposes simply uses the system clock, hence to guarantee the correctness of these operator times, we force PyTorch into synchronous execution mode (which ensures that GPU operators are performed synchronously); we found that DTR was still able to execute models

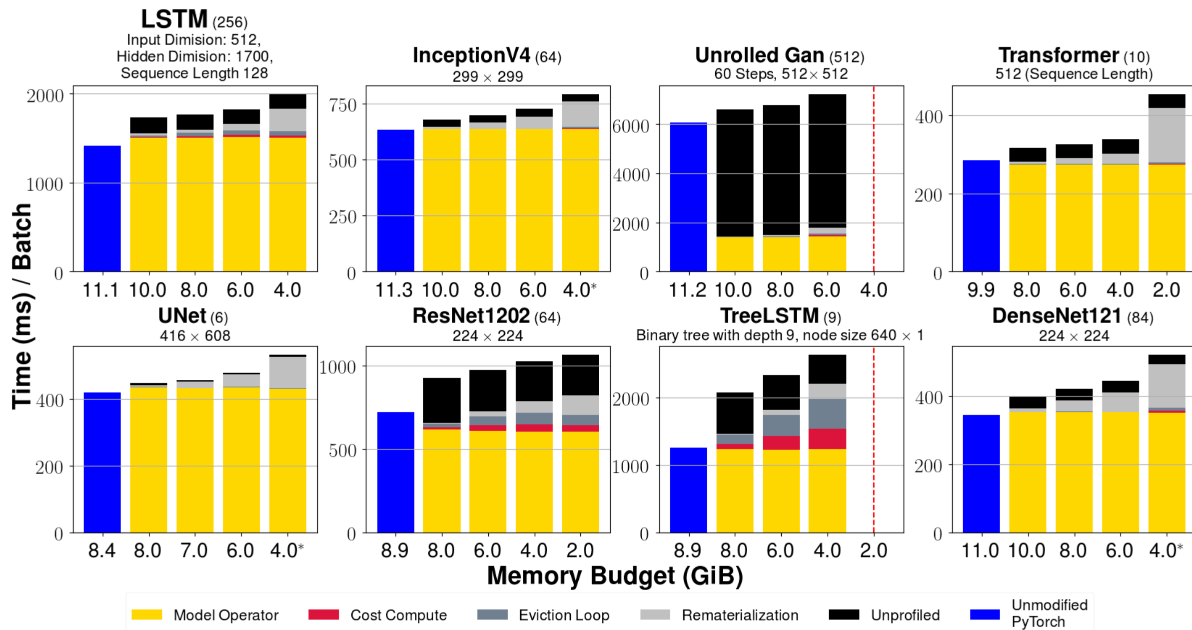


Figure 4.12: We profiled the running time of our prototype for various models and memory budgets on a machine with an NVIDIA Titan V GPU (CUDA 10.1, CuDNN 7.6.4) and a 16-core AMD Ryzen Threadripper 1950X on Ubuntu 18.04. The red dotted lines correspond to trials that either ran out of memory or thrashed ($\geq 2\times$ unmodified PyTorch's time). Model batch sizes are given in parentheses. To ensure the accuracy of the DTR prototype's profiling, we used PyTorch's synchronous computation mode (see Section 5.1). Results (mean of 100 trials) are compared against unmodified PyTorch. "Cost compute" (computing heuristic scores) and "eviction loop" (comparing scores over tensors) correspond to overhead from the DTR *runtime* itself, which can be reduced by a more efficient implementation. "Unprofiled time" is the remainder of the time per batch; it may be due to runtime overhead from parts of PyTorch not modified in the prototype, like the operator dispatch system. The large proportion of unprofiled time in Unrolled GAN is likely due to its extensive use of Python reflection. The budgets with asterisks were run with the random sampling optimization (see Section 5.2) disabled, as sampling caused occasional failures at those budgets.

on greatly reduced memory budgets without turning on synchronous execution mode, even though this should skew DTR's recorded operator times.

For evictions, `CheckpointTensor`s are capable of freeing their underlying tensor representation from memory; they keep a closure for replaying the parent operation, which the runtime can invoke when the tensor must be rematerialized. To handle deallocations by the original program, `CheckpointTensor`s also report increments and decrements to the reference count *of the underlying tensor* to the DTR runtime. We add a method to tensors called "`checkpoint()`" that lifts any tensor into a `CheckpointTensor` and a method "`decheckpoint()`" that extracts the underlying tensor from a `CheckpointTensor`, rematerializing it if necessary (we use the latter in our trials to ensure the loss and output are in memory at the end).

Our modified version of PyTorch dispatches any operation involving a `CheckpointTensor` to a specific implementation for `CheckpointTensor`s; this is the same mechanism that PyTorch uses, for example, to dispatch operations on GPU-managed tensors to CUDA implementations. Specifically, whenever PyTorch encounters an operator where an argument is a `CheckpointTensor`, its dispatch mechanism searches for a specific overload of that operator for `CheckpointTensor`s. Since a `CheckpointTensor` simply wraps the underlying PyTorch tensor, adding `CheckpointTensor` implementations for operators simply requires invoking the operator's existing implementation for the underlying tensor and wrapping the result in a `CheckpointTensor`. These overloads were essentially boilerplate code and it is likely possible to generate them automatically. As far as PyTorch's dispatch system is concerned, all tensor accesses occur through operators, so updating metadata like access time only requires invoking the DTR runtime inside the `CheckpointTensor` operator overloads.

The DTR runtime is simply a singleton that keeps a pool of all `CheckpointTensor`s created since the start of the program. The runtime is also responsible for maintaining the equivalence class data structure needed for $h_{\mathrm{DTR}}^{\mathrm{eq}}$ as described in Section 4.3.1 (updated each time a `CheckpointTensor` is evicted or rematerialized). Before each `CheckpointTensor` operation, the DTR runtime checks whether the memory budget has been exceeded; if it

has, the runtime searches over the pool of `CheckpointTensor`s, computing the heuristic score ($h_{\text{DTR}}^{\text{eq}}$) for each using their metadata, and evicting the least-scoring until either it is not possible to evict any more tensors or the budget has been met. (N.b., this means that the prototype permits exceeding the budget by exactly one tensor allocation. In principle, we can correct this by inserting a callback into PyTorch's GPU memory manager to call the DTR runtime as soon as an allocation is *requested*; we did not do this to simplify our implementation.) This method of searching is very simplistic; it is likely that redundant heuristic computations can be removed using data structures to keep `CheckpointTensor`s in a sorted order and incrementally update metadata, but the optimizations discussed below in Section 5.2 were very simple and helped to reduce some of the overhead from this naive method. The DTR runtime is also responsible for implementing the logging mechanism described in Section 4.3.6; this is accomplished by simply writing JSON records of events intercepted by the runtime (operator calls, reference count increments and decrements, etc.) to a file.

The DTR prototype supports PyTorch's implementation details like in-place mutations, aliasing, and multiple operator outputs, which are all discussed in Paszke et al. (2017), using the same methods as in Section 4.3. As in Section 4.3.6, the DTR prototype supports PyTorch operators that perform in-place mutations by introducing a copy-on-write mutation layer: The mutating operator is made pure (and therefore infinitely replayable) by copying the source tensor for the mutation and mutating the copy. (Similarly, impure operators like `batchnorm` and `dropout` are made pure by treating state like the PRNG seed as part of the input to the operators and the updated state as part of their output.) The DTR runtime performs these copies for `CheckpointTensor` operator overloads to mutating operators. To support operators whose results are aliases of their arguments, the DTR runtime groups together all `CheckpointTensor`s whose underlying tensors are aliases of each other into *alias pools*. When a member of an alias pool is evicted, all members of the alias pool are treated as evicted; aliases are, however, rematerialized separately, only as they are needed. For `CheckpointTensor`s produced by multi-output operations, the DTR runtime allows them to

be evicted separately but ensures that they are rematerialized together.

5.2 Runtime Optimizations

Searching for tensors to evict is a significant source of overhead for DTR's runtime because the runtime recomputes each tensor's staleness and equivalence class cost upon each eviction, rather than storing and incrementally updating this information. In principle, we could reduce this portion of the overhead by using more complex data structures to maintain an ordering of the tensors to avoid searching, though this would greatly increase the complexity of our implementation. As a simpler means of reducing the DTR runtime's overhead from searching and computing heuristic scores, we added two approximate optimizations to reduce the search space: ignoring small tensors (less than 1% of the average size) and only searching over a random sample of $\sqrt{n}$ tensors from the pool of n evictable tensors. This greatly reduces the number of tensors that the runtime needs to check upon evicting. Even though this improves the search overhead considerably, searching and computing costs still present considerable DTR-specific overhead, as the profiling breakdown in Figure 4.12 shows. Additionally, random sampling caused occasional failures at low budgets or very large inputs due to excluding good eviction candidates from the search space, which led us to deactivate that optimization in certain trials. (At low budgets, individual eviction choices are very impactful, so removing tensors from the search space completely at random can dramatically affect the results.)

There are also several possible sources of runtime overhead that could potentially be improved by making deeper modifications to PyTorch's core systems. For example, we introduced an overload layer that results in many more layers of callbacks. The mutation layer also clones tensors (even though it frees the necessary space immediately), resulting in additional overhead. Further modifications to the framework could allow for more optimizations, particularly by reducing the number of heap allocations and conversions between tuples and lists. PyTorch's define-by-run nature and shallow embedding into Python also meant that much of DTR's metadata, such as the parent operator of a tensor, needed to be

computed at run time (such as by creating a closure). In other frameworks that feature a compilation step, such as Glow (Rotem et al., 2018), it may be possible to eliminate much of this overhead by generating these structures in a compiler pass. We may also note that all the bookkeeping for DTR takes place on CPU while operators are generally offloaded to other devices, so an implementation could interleave these updates with GPU operations.

5.3 Handling Errors in Trials

As discussed in Table 4.1 and Figure 4.12, the DTR prototype encountered errors on certain models when running on low budgets or on large input sizes. These errors were primarily CUDA out-of-memory errors (OOMs), but in some cases, the trial simply hung, neither crashing nor terminating. For CUDA OOMs, disabling the random sampling optimization eliminated the errors in most cases, suggesting that the OOMs were due to excluding useful eviction candidates. For the hanging trials, we were not able to determine whether the root cause was DTR thrashing (being trapped in a very deep recursive rematerialization, as occurred in some of the simulated trials on certain heuristics) or an infinite loop or deadlock elsewhere in PyTorch; we can investigate the cause by further instrumenting the implementation, but we have been unable to consistently reproduce hanging trials and they seem to occur less frequently than OOMs.

In the largest two batch sizes for UNet in Table 4.1, disabling sampling did not eliminate all OOMs or hanging trials. Thus, for the large-input trials in Table 4.1, we employed a procedure for retrying upon encountering an OOM or a hang. First (as with all other GPU measurements), we perform some untimed “warm-up” trials to allow for CUDA initialization and caches to be populated and then begin timing the trials. If a trial raises a CUDA OOM or hangs (which we define as taking twice as long as the trial before it), we keep the measured times from that point in the trial and then restart (doing another warm-up), collecting the remaining number of measurements. Restarting the measurement run was the only way to ensure that all memory allocated during the trial would be collected in the event of an OOM (attempts to proceed simply by resetting the PyTorch allocator’s cache resulted in

memory accumulating between trials regardless). Our experimental setup automates this process of retrying failed trials and reports the total number of retries. Note that we treat failures during warm-up runs the same as failures in timed runs, since recovering from an OOM would require exiting the process running PyTorch and reinitializing CUDA. In the Table 4.1 results, there was 1 failed run for UNet on batch size 9 and 10 failures on batch size 10; most of the latter were during warm-up runs.

A possible reason for the occasional failed trails in UNet may be variance in operator timings, which affect the metadata and may be influencing rematerialization decisions. One way to control for this possibility in a static model like UNet would be to use a DTR simulation to produce a static rematerialization schedule and therefore have a known, safe execution schedule for operators. For a dynamic model, a static plan is not an option, but variations in operator timings could be reduced by using a fixed cost model for operators instead of timing them dynamically. That is, the DTR heuristics employed could be defined to use proxy measures that are less subject to variation (e.g., defining staleness in terms of a counter incremented by operations rather than wall-clock time) or less likely to be influenced by specific system implementation details in order to have more predictable and reproducible behavior.

5.4 Empirical Results

Our empirical evaluation demonstrates that DTR can efficiently train models under restricted memory budgets using the $h^{\text{eq}}_{\text{DTR}}$ heuristic. We used the same models and experimental setup as in Section 4, timing the forward pass, loss computation, and backward pass. Table 4.1 presents several cases where DTR trains models on much larger input sizes than unmodified PyTorch, including a dynamic model, TreeLSTM. This highlights that *DTR enables exploration of models that push the boundaries of existing deep learning architectures*. While the simulated trials in Section 4 consider the slowdown due only to rematerializations but not overhead from managing metadata and computing heuristics, Figure 4.12 measures the time per batch required to train eight DL models on a variety of restricted memory bud-

	ResNet-1202 (Batch Size)				Transformer (Batch Size)				UNet (Batch Size)				TreeLSTM (Tree Nodes)			
	64	100	120	140	30	70	80	90	7	8	9	10	2^6-1	2^7-1	2^8-1	2^9-1
DTR	0.974s	1.18s	1.28s	1.39s	367ms	830ms	950ms	1079ms*	566ms	684ms	822ms*	1170ms*	0.486s	1.05s	2.50s	7.89s*
PT	0.712s	✗	✗	✗	331ms	✗	✗	✗	481ms	✗	✗	✗	0.431s	✗	✗	✗

Table 4.1: Median execution times per batch (out of 100 runs) for various models, giving both the largest input size that unmodified PyTorch ("PT") could support on our GPU and larger input sizes DTR could support. Input sizes are as in Figure 4.12, except for TreeLSTM (complete binary trees with nodes of size 1024×1024) and Transformer (sequence length 256). Asterisks indicate inputs on which the random sampling optimization was disabled due to occasional failed trials. Even without sampling, DTR still occasionally failed on UNet (see Section 5.3 for details). This behavior may be due to PyTorch memory allocator implementation details or poor rematerialization decisions influenced by variance in individual operator times.

gets, profiling the time spent by the runtime system. Among the models is Unrolled GAN, which uses higher-order partial derivatives and Python reflection extensively; *the DTR prototype supported these unusual features, underscoring its generality.* Despite our prototype's simplicity—it merely loops through all tensors when searching for an eviction candidate and recomputes the heuristic scores from scratch each time—on most models, *its overhead due to searching and computing heuristics remains low for most memory budgets.*

6 Summary

DTR provides a simple, customizable approach to checkpointing for DL models. It supports a broad range of applications without the need for any ahead-of-time analyses, manual annotations, or modifications. Our formal results establish that DTR can match the same asymptotic bounds as recent static checkpointing approaches for linear feedforward networks. In simulation, it enables training for a range of both static and dynamic models under various restricted memory budgets and closely matches the performance of optimal checkpointing. The DTR prototype in PyTorch demonstrates how our approach can be incorporated into existing frameworks with modest, non-invasive changes by simply interposing on tensor allocations and operator calls and collecting lightweight metadata on tensors.

Chapter 5

SEMANTICS-BASED HARDWARE SEARCH: 3LA

Note: This section is adapted from the previously published work Huang et al. (2022).

This chapter addresses the problem of utilizing new accelerators in DL systems, approaching it by an approach we call *the 3LA methodology*, by introducing an IR for reasoning about the semantics of accelerator operations and using that to define a search over possible mappings from the source DSL to hardware operations. This mode of searching allows for automatically exposing opportunities to invoke accelerator operations that can be easily extended with new rewrite rules and can be formally verified, allowing for end-to-end functional verification that includes accelerators.

1 Problem Description

The early success of the GPU in efficiently implementing tensor operations in DL workloads (Krizhevsky et al., 2012) underscored the possibilities for hardware to exploit the parallelism and data transfer behavior of DL models. The increasing economic importance and growing scales of DL applications have since driven further investment in developing DL-specific accelerators, notably the Tensor Processing Unit (TPU) (Jouppi et al., 2017), multiple versions of which have entered production use at Google. By customizing compute engines, memory hierarchies, and data representations (Chan et al., 2014; Fang et al., 2019; Lai et al., 2021), hardware accelerators provide efficient computation in various application domains like artificial intelligence, image processing, and graph analysis (Han et al., 2016; Chen et al., 2017; Reagen et al., 2016; Zhang et al., 2016; Hameed et al., 2010; Ham et al., 2016). Accelerators like the TPU tend to have specialized instruction sets intended to best

utilize their carefully designed data pipelines, such as commands for reading and writing tensors and for specifying weights and inputs to the operations implemented in the device. Executing a complete model using such an accelerator typically requires interacting with a host device that must invoke the accelerator's specialized instructions and load in the appropriate data to the accelerator; invoking such low-level instructions from the high-level DSLs used to define DL models poses a considerable challenge. Indeed, large industrial teams invest substantial resources to address DSL-to-accelerator mapping challenges via bespoke infrastructure (Jouppi et al., 2017, 2020). For smaller teams, like in the academic research community (both DL researchers and accelerator researchers), these gaps make end-to-end evaluation of new accelerator designs prohibitively difficult; many recent papers on accelerator designs only evaluate on small application snippets, e.g., individual layers of deep neural networks (Tambe et al., 2021; Jia et al., 2020; Park et al., 2021; Rossi et al., 2021; Schmidt et al., 2021; Whatmough et al., 2019; Fujii et al., 2018; Cao et al., 2020; Giordano et al., 2021; Saito et al., 2021; Wei et al., 2019; Garofalo et al., 2021). (Note, however, that, as we explore in Section 5, an accelerator's results for application snippets are often *not* predictive of its influence on overall system behavior, e.g., due to cumulative effects of custom numerical representations, further motivating the need to more easily enable end-to-end execution.)

Incorporating an accelerator into a complete DL workload requires mapping the computations within a model to those supported by the accelerator, if possible, which requires developing a compiler from an IR to that device. In current practice, this is generally accomplished by manually crafting device drivers to provide "hardware function calls" for specific operations. This essentially provides an application-programmer interface (API) for accelerators, where each hardware function call consists of low-level accelerator invocation commands that configure, initiate, validate, and return the results. Figure 5.1 shows an example hardware function call, in which a layer reduction operation is implemented by a sequence of memory-mapped input/output (MMIO) loads/stores from the host processor to invoke the FlexASR accelerator (Tambe et al., 2021). Such MMIO-based APIs are the prevalent mechanism for accelerator invocation.

```
#define HWREG(addr) (*((volatile uint128_t*)(addr)))
union buffer128 {
uint128_t v128;
int64x2_t v64;
} buf;

// FlexASR API: Layer Recude (max/min/mean pooling)
bool FlexAsrLayerReduce(uint64_t* arg1, /* ... */) {
// set up inputs and arguments
buf.v64[0] = 0xC9A8070100CA8801;
buf.v64[1] = 0x09D1008100810000;
HWREG(0xA0500000) = buf.v128; // internal buffer[0]
// ...
// configure and invoke operation
buf.v64[0] = 0x0101000000010001;
buf.v64[1] = 0x0000000200000001;
HWREG(0xA0700010) = buf.v128; // global buffer control
buf.v64[0] = 0x0000000000000001;
buf.v64[1] = 0x0000000000000000;
HWREG(0xA0400010) = buf.v128; // memory manager config
buf.v64[0] = 0x0000000000000000;
buf.v64[1] = 0x0000000000000000;
HWREG(0xA0000010) = buf.v128; // invoke operation
// wait and retrieve results from the buffer
sleep(5);
buf.v128 = HWREG(0xA0500000);
// ...
}
```

Figure 5.1: Snippet of the FlexASR device driver (Tambe et al., 2021). Through MMIO commands, the driver first stores input arguments, e.g., weights, in the accelerator's internal buffer (lines 10 to 13). It then sets up the configuration such as tensor dimension and vector size (lines 15 to 20). Finally, it triggers the operation (line 23) and retrieves the result (starting line 26).

Generally, hardware function calls are manually added by application programmers, or they can be added by compilers via handcrafted accelerator-specific extensions. Such bespoke compiler extensions demand tedious effort and deep expertise in the hardware and the compilation stack. Large differences among accelerators' supported operations, performance characteristics, choices of numerical representation, and memory capacities demand even greater expertise in order to achieve some degree of interoperability and ensure a system can utilize different devices. Such implementation hurdles could prevent optimal use of available accelerators and slow the integration of new accelerators into existing systems, stifling further potential innovation in hardware and potentially restricting model performance in production systems; as it stands, only large enterprises that can afford teams of hardware, software, and systems experts for high-value applications can readily make use of diverse accelerators (Caulfield et al., 2016; Fowers et al., 2018; Jouppi et al., 2017).

The approach of API-based hardware function calls presents two fundamental challenges:

1. **Lack of portability.** Each API call is specialized to a particular device, whose interfaces may differ significantly even from other devices that may implement similar functionality due to hardware implementation details. These differences in interface must also be managed between the source DSL and the target device.

2. **Lack of integration into standard compiler flows.** The MMIO interfaces for accelerators are typically not as well-defined as traditional software/hardware interfaces; API calls are opaque to the compiler stack. For example, standard techniques for instruction selection (Blindell, 2016) become challenging with a "black-box" API, which provides little flexibility in selecting and reusing parts of these APIs. The fixed API also limits automation in exposing potential optimizations, e.g., operator fusion that may minimize data transfers.

As a consequence of these challenges, it is additionally difficult to validate compilation results, which results in development issues of its own. Validating the compilation results requires

(manually) porting over entire applications to use the appropriate API calls, which incurs some development effort (as opposed to portable code that is well-supported by compilers, which would allow the original program implementation to be reused verbatim). Moreover, even after the application is ported, the costs of hardware manufacturing mean that early testing must be done using either an FPGA emulation of the accelerator or a low-level register-transfer level (RTL) simulation; the former requires further engineering effort, while the latter is often prohibitively slow. Furthermore, the need for a fully functional RTL model limits early-stage software/hardware co-design, i.e., software development before the hardware is implemented.

At the heart of these issues in compiling to accelerators is the difficulty in reasoning about the semantics of accelerator operations. We propose to address this problem through the introduction of a new IR that will serve as a formal model for accelerators, rather than ad hoc API-supported hardware function calls. To motivate this approach, consider the more restricted problem of portably compiling programs to different CPUs, which was addressed by LLVM (Lattner and Adve, 2004). With LLVM, programs can more easily be compiled to different CPUs by first compiling programs into a common representation, leaving the less laborious task of writing compilers from LLVM to the CPUs. However, the reason the task of compiling from LLVM to different CPUs was comparatively simple is that LLVM itself was designed to be similar to the instruction sets of various CPUs. In the case of accelerators, however, the large differences among devices suggest the need for a representation sufficiently general to account for those differences—a formal model for the semantics.

The analogous representation can therefore be ceonceived as an *accelerator instruction set*, providing an ISA-like formal model for accelerator operations, providing finer-grained, instruction-level control over accelerator operations (in contrast to API calls). The formal model for the accelerator operations can be combined with a semantics for the program-level IR to reason about the end-to-end computation, suggesting the possibility of automatic analysis of the input program's semantics to identify opportunities to invoke accelerators. We term the use of an instruction-level formal representation for compiling to accelerators

the 3LA methodology. In the following sections, we explore how this explicit model for accelerator operations facilitates standard compilation flows, portability, and validation of compilation results.

2 Overview

The 3LA methodology uses an instruction-level formal software/hardware interface specification for accelerator operations, which abstracts away low-level implementation details while providing a formal hardware semantics. This representation enables flexible mappings between instruction-level program fragments, easy integration into compiler flows, and end-to-end validation of compilation results that take the accelerator operations' semantics into account.

2.1 Key Ideas

Identifying the computation in the application that can be offloaded to the accelerators is effectively seeking *mappings* between compiler IR intrinsics and the accelerator operations such that they are functionally equivalent and lead to performant code. (Note that in certain domains like machine learning, small numerical differences may not affect the application-level results, such as a classification category, so notions of "equivalence" should reflect this property.) In the 3LA methodology, we use the Instruction-Level Abstraction (ILA) (Huang et al., 2018b), for this purpose. The ILA, like the ISA for processors, provides a software/hardware interface specification for accelerators. It bridges the gap between the compiler IR intrinsics on one hand and the accelerator operations on the other—providing the basis for specifying *IR-accelerator mappings*.

The ILA model of an accelerator provides formal semantics at its software/hardware interface through a *lifting* of the accelerator operations in the form of a set of abstract instructions, each of which reads and updates the accelerator arthiectural state according to transition functions. Similarly, the compiler IR intrinsics can also be defined as ILA instructions that update program state. This provides a uniform model for both the compiler

IR and the accelerator operations. We can therefore define the mappings of compiler IR instructions to accelerator operations as pairs of *program fragments*, where a program fragment comprises a sequence of instructions. Unlike API calls or RTL models, program fragments using accelerator ILA instructions capture the underlying accelerator operation semantics while abstracting over low-level hardware implementation details.

Given the definitions of IR-accelerator mappings in terms of program fragments, we can approach the problem of instruction selection as *term rewriting* (Dershowitz, 1993; Baader and Nipkow, 1998; Blindell, 2016), namely by replacing a compiler IR fragment with the corresponding accelerator operation fragment. Specifically, we utilize equality saturation (Tate et al., 2011) to optimally[1] match all the different possible rewrites of the program and therefore reduce the need for manual program restructuring (improving portability and increasing the amount of compiler automation). Additionally, since the accelerator ILA models the semantics of accelerator calls and captures their interface, the 3LA methodology is also capable of implementing optimizations that require understanding of accelerator behavior, e.g., operator fusion for reducing data movement.

Because the ILAng (Huang et al., 2019) toolchain allows for synthesizing functional simulators for ILA instructions, modeling accelerator operations using the ILA means that it is possible to efficiently (in comparison to RTL simulation) simulate accelerator operations and therefore validate the results of end-to-end applications earlier in development. This capability is useful not only for validating the results of end-to-end applications, but also for prototyping and exploring software/hardware *co-design*, since an ILA specification can be written and analyzed while the hardware design is still in progress. The formal semantics of ILA instructions also allow for formal verification of mappings and instructions by techniques like bounded model checking, which we will explore in Section 5.

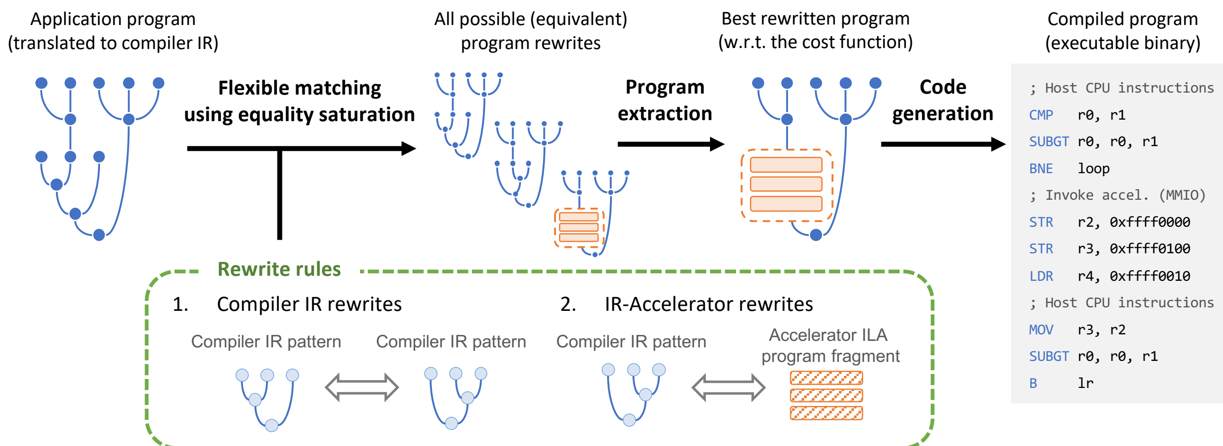


Figure 5.2: 3LA compilation flow overview. Note that the ILA modeling and the validation of the IR-accelerator mappings are omitted from this figure.

2.2 Overall Compilation Flow

The overall compilation flow using the 3LA methodology is shown in Figure 5.2. We first translate the application program, provided in a high-level DSL, into the compiler IR. Next, we perform equality saturation to search a large space of equivalent programs given the compiler IR-accelerator mappings (along with general-purpose rewrite rules). Based on a given cost function, we then extract the lowest-cost program and pass it on for code generation, where each accelerator instruction is subsequently lowered to the corresponding MMIO command. This generated program executes on the host processor and invokes accelerator operations through MMIO commands.

2.3 Prototype Implementation

As a demonstration of the 3LA methodology, we have implemented an end-to-end compilation flow for deep learning (DL) applications by integrating it with an existing compiler flow. Specifically, our prototype compiler utilizes the DSL front-end and the code generation

[1]In terms of a provided cost function, and with respect to the rewrites *actually performed* in cases where the e-graph is not saturated.

capabilities provided by the TVM framework (Chen et al., 2018a). For instruction selection, it leverages the rewrite rules and the equality saturation engine provided by Glenside and `egg` (Smith et al., 2021; Willsey et al., 2021).

We show the generality of the 3LA methodology through multiple case studies. At the front-end, we consider six DL applications for language processing and image recognition, e.g., Transformer (Vaswani et al., 2017) and ResNet (He et al., 2016a). For the target accelerators, we add support for three custom accelerators that provide hardware operations at different levels of granularity: VTA (Moreau et al., 2019) is a fine-grained accelerator for general tensor operations; HLSCNN (Whatmough et al., 2019) is a coarse-grained accelerator providing 2D convolutions; FlexASR (Tambe et al., 2021) is an accelerator for speech recognition, specializing in coarse-grained operations like long short-term memory (LSTM) layers.

We added support for the three accelerators—developed their ILA models, provided compiler IR-accelerator mappings for operations supported by them, and validated all the mappings. Note that this work for supporting a new accelerator is a one-time effort that can be reused across different applications. Our prototype compiler successfully compiled all six DL applications (developed by different teams and programmed in different DSLs) for exploiting the three custom accelerators. Our prototype and case studies demonstrate the key ideas in the 3LA methodology for end-to-end compilation with validated results. We do not claim this prototype provides a complete, fully optimized compiler for custom accelerators; rather, it establishes the foundations for validated compilation for such targets through the use of a formal instruction-level software/hardware interface for accelerators.

3 The 3LA Methodology

In this section, we explain in detail the three key aspects of the 3LA methodology:

1. adding accelerator support by specifying mappings between compiler IR intrinsics and accelerator operations,

2. compiling applications by searching within input programs for computations supported by accelerators, and

3. ensuring compilation result validation and supporting early-stage software/hardware co-design.

3.1 Specifying IR-Accelerator Mappings using ILAs

The ILA is an ISA-like formal model for specifying the functional behavior of accelerators. It generalizes the notion of instructions to accelerators and provides a modular functional specification as a set of instructions. Like processor ISAs, it does so by specifying how each instruction updates software-visible (viz., architectural) state while abstracting out implementation details.

3.1.1 Accelerator ILA

We develop the ILA formal model for an accelerator by following the methodology proposed in prior work (Huang et al., 2018a). Each instruction of an accelerator ILA corresponds to a command at the accelerator interface, i.e., an MMIO load or store command. The ILA captures formal semantics of accelerator behavior by specifying how each instruction reads/updates the architectural state variables. Essentially, the ILA is a modular (per-instruction) operational specification of an accelerator. Figure 5.3 provides an accelerator ILA example.

3.1.2 Compiler IR ILA

While the ILA is primarily intended to serve as the formal model for accelerators, it is convenient to also use it to formally model compiler IR intrinsics. We develop the compiler IR ILA by following the approach used in prior work for the NVidia parallel execution thread (PTX) programming model (Xing et al., 2018). Each instruction of a compiler IR ILA corresponds to an IR intrinsic and specifies its operational behavior in terms of how it updates program

```
#include <ilang/ilang++.h>

int main() {
// declare an ILA model module
auto m = ilang::Ila("flexasr-ila");

// declare model interface input ports
m.NewBvInput("top_if_wr", TOP_IF_WR_BITWIDTH);
m.NewBvInput("top_if_rd", TOP_IF_RD_BITWIDTH);
m.NewBvInput("top_addr_in", TOP_ADDR_IN_BITWIDTH);
m.NewBvInput("top_data_in", TOP_DATA_IN_BITWIDTH);

// declare architectural states
m.NewBvState("pe_0_is_valid", PE_VALID_BITWIDTH);
m.NewBvState("pe_0_is_bias", PE_IS_BIAS_BITWIDTH);
...
m.NewMemState("gb_large_buffer", TOP_ADDR_IN_BITWIDTH, TOP_DATA_IN_BITWIDTH);
...
// define ILA instructions
{ // ILA instruction for configuring pe_cfg_mngr
auto instr = m.NewInstr("pe_0_cfg_mngr");

// define decode condition for this instruction
auto is_write = (m.input("top_if_wr") == 1) & (m.input("top_if_rd") == 0);
instr.SetDecode(is_write & (m.input("top_addr_in") == PE_0_CFG_MNGR_ADDR));

// define state update functions for this instruction
auto is_valid = ilang::SelectBit(m.input("top_data_in"), PE_IS_VALID_BIT_IDX);
instr.SetUpdate(m.state("pe_0_is_valid"), is_valid);
auto is_bias = ilang::SelectBit(m.input("top_data_in"), PE_IS_BIAS_BIT_IDX);
instr.SetUpdate(m.state("pe_0_is_bias"), is_bias);
...
}
// other ILA instructions
...
}
```

Figure 5.3: **FlexASR ILA model snippet.** Lines 5-18 define the FlexASR ILA model, its input and architectural states variables. Lines 20-32 shows an example of an ILA instruction named "`pe_0_cfg_mngr`," which corresponds to line 6 in Figure 5.4 (c). In each ILA instruction, we specify its decode condition and state update functions. For example, in this instruction, the decode condition (line 24-25) is when there is write instruction at the top interface to the address associated with the configuration of the PE's management configuration. Lines 27-32 show this instruction's state update functions for the architectural states. In this example, this ILA instruction models the behavior of storing the arguments from the input data at its interface into the FlexASR configuration registers. From this example, we can see that the ILA instructions provide an abstraction of the functionality of the accelerator corresponding to the MMIO instructions at its interface.

state. Modeling both the compiler IR and accelerators using ILAs provides a uniform model on both sides, and enables the use of the ILAng toolkit for their verification/validation.

3.1.3 IR-Accelerator Mappings

Due to the granularity gap between the IR instrinsics and the accelerator operations, it is often not possible to construct a one-to-one mapping between the compiler IR and the accelerator operations. Instead, on each side (the compiler IR and the accelerator), we consider a program fragment that comprises a sequence of instructions defined by the associated ILA model. The program fragments provide a basis for *many-to-many instruction mappings between the two sides*, providing flexibility that is key to addressing the granularity mismatch challenge.

The specification of the mappings starts from the accelerators—based on the given accelerator, we provide an IR-accelerator mapping for each accelerator operation. This bottom-up approach is a one-time effort for each accelerator, requires no knowledge of the input program, and is the key for modular and extensible compilation.

Figure 5.4 shows an example of an IR-accelerator mapping for a linear layer operation for the FlexASR accelerator (Tambe et al., 2021). The program fragments of the compiler IR and the accelerator are shown in parts (b) and (c), respectively. As discussed, each instruction of the compiler IR ILA corresponds to one IR intrinsic, which is reflected in parts (a) and (b). Similarly, each instruction of the accelerator ILA corresponds to one command at its MMIO interface, as shown in parts (c) and (d).

3.2 Flexible Matching using Equality Saturation

Given the IR-accelerator mappings and an input program, the next step in the 3LA methodology is to identify computations in the input program that can be offloaded to equivalent accelerator operations. We approach this task by utilizing term rewriting techniques—given a set of syntactic rewrite rules ($\ell \Longrightarrow r$), rewrite instances of pattern ℓ in the input program with pattern r where applicable (Dershowitz, 1993; Baader and Nipkow, 1998; Blindell, 2016;

(a) Compiler IR instructions

```
ComILA.relay_nn_dense
ComILA.relay_bias_add
```

(b) Compiler IR ILA program fragment

```
// 1. writing data into the accelerator memory
FlexASR_ILA.write_v
...
// 2. configuring the accelerator for executing linear layer operation
FlexASR_ILA.pe_cfg_rnn_layer_sizing
FlexASR_ILA.pe_cfg_mngr
FlexASR_ILA.pe_cfg_act_mngr
FlexASR_ILA.pe_cfg_act_v
FlexASR_ILA.gb_cfg_mmngr_gb_large
FlexASR_ILA.gb_cfg_gb_control
// 3. triggering the accelerator linear function
FlexASR_ILA.fn_start
// 4. reading data out from the accelerator memory (if needed)
FlexASR_ILA.read_v
...
```

(c) FlexASR ILA program fragment

```
// 1. writing data into the accelerator memory
Write, addr=0xA4500000, data=0x0F0EFFBF8F746F9FB58D148E0EB7BFDAD
...
// 2. configurating the accelerator states for linear layer operation
Write, addr=0xA4400010, data=0x0010101000001
Write, addr=0xA4400020, data=0x000000010000000102020200
Write, addr=0xA4800010, data=0x0000000000102050001
...
// 3. triggering the accelerator function
Write, addr=0xA3000010, data=0x1
// 4. reading data out from the accelerator memory (if needed)
Read, addr=0xA3500200, data=0x0
...
```

(d) FlexASR MMIO commands

Figure 5.4: **IR-accelerator mapping for the FlexASR linear layer operation.** This shows a many-to-many mapping from Relay IR instructions to a sequence of FlexASR MMIO commands. (a) A Relay linear layer consists of a linear transformation operation `nn.dense`, followed by a bias addition operation `nn.bias_add`. (b) The compiler IR ILA instruction has a one-to-one mapping to the compiler IR instruction. (c) The FlexASR ILA program fragment in its assembly format: It includes: (1) writing instructions to transfer the data into FlexASR's memory; (2) setting up FlexASR `LinearLayer` configuration states, for example, the instruction at line 5 sets the states of FlexASR layer sizing information; (3) an instruction that triggers the FlexASR `LinearLayer` computation; and (4) reading data out from FlexASR's memory if needed. (d) The MMIO commands for FlexASR have a one-to-one mapping to its ILA.

Tate et al., 2011). In term rewriting systems, the application of rewrite rules is correct by construction as long as the rules preserve semantic equality. This provides for modular correctness checking through checking the individual rewrite rules, and allows for easy extension by adding in new rewrite rules (which automatically take advantage of the existing ones).

Classic term rewriting systems often suffer from the phase ordering problem (i.e., the order in which rewrites are applied affects final performance) and thus require careful ordering (Whitfield and Soffa, 1997; Newcomb et al., 2020).

In 3LA, we utilize the technique of equality saturation to mitigate phase ordering problems (Tate et al., 2011). Given an input program p, equality saturation repeatedly applies the given rewrite rules to explore all equivalent ways to express p (with respect to the rules). It utilizes the *e-graph* data structure to efficiently represent an exponentially large set of equivalent program expressions (Nelson and Oppen, 1980; Nieuwenhuis and Oliveras, 2005). Upon reaching a fixed point, i.e., when no application of any rewrite rule can introduce a new program expression, or upon reaching some time or resource threshold, it extracts the optimal rewritten program according to a given cost function. This provides for searching over a large space of rewrites (a complete space if a fixed point was reached) and finding the representative most suitable for the given purpose without sophisticated ordering considerations.

3.2.1 IR-Accelerator Rewrites

Based on the provided IR-accelerator mappings, we derive a set of rewrite rules where the left-hand side of the rule is the compiler IR pattern and the right-hand side is the corresponding accelerator instructions. Applying these rules, which we call the *IR-accelerator rewrites*, allows replacing the computations that are exact syntactic matches to the compiler IR pattern specified in the mappings by the corresponding accelerator operations. We call the direct, destructive application of rewrite rules *exact matching*.

3.2.2 Flexible Matching and Compiler IR Rewrites

Exact matching provides a baseline matching capability but may be limited in practice because there is often no canonical IR expression to represent a program. Therefore, the input program can have constructs that are syntactically different from the left-hand side of the IR-accelerator rewrites but are semantically equivalent to the pattern. For example, in the IR-accelerator mapping shown in Figure 5.4, we specify the compiler IR pattern for a linear layer (as an S-expression):

```
(bias_add (nn_dense %a %b) %c).
```

However, a linear layer can be equivalently expressed in Relay as

```
(add (reshape (nn_dense %a %b) %s) %c)
```

when `%c` is a vector, for certain shapes `%s`. This prevents exact matching from identifying potential accelerator calls.

A possible means of dealing with equivalent patterns as in the above case may be to enumerate all the different variants of interest and search for all of them in exact matching. While this approach would be feasible for the given linear layer example, larger and more complex patterns (such as LSTM operations) would be likelier to have many equivalent variants and would be tedious to manually enumerate. In general, enumerating equivalent rewrites is error-prone and not guaranteed to be complete. Instead of this approach, we propose to use rewrite rules to reason about what patterns are equivalent. Namely, we include another set of rewrite rules that we call *compiler IR rewrites*. Each compiler IR rewrite transforms an IR pattern into another IR pattern, e.g., from the second to the first linear layer IR pattern above, without replacement by accelerator instructions. These general-purpose rewrite rules do not depend on the input program nor on the accelerator, but help expose more potential matches, as with the “exploratory rewrites” in Smith et al.

(2021). We refer to this as *flexible matching*, as it enables finding matches that may be missed by exact matching.

Flexible matching takes advantage of the ability of equality saturation to non-destructively search over different applications of rewrite rules. Compiler IR rewrites would, in the exact matching setting, be prone to phase ordering issues; however, in an e-graph, many different applications of these exploratory rewrites can be considered simultaneously during the extraction phase. This is particularly important in the case where there are multiple different IR-accelerator rewrites in the resulting e-graph (especially if compiler IR rewrites reveal some of these opportunities): The e-graph representation allows for explicitly choosing between the calls based on the cost function, rather than implicitly via the phase ordering.

3.3 ILA-Based Compilation-Results Validation

The formal semantics of ILA instructions provides the foundation for validating compilation results. The 3LA methodology does compilation results validation at two levels.

3.3.1 Checking IR-Accelerator Mappings

The use of term rewriting provides for modular validation in the form of checking end-to-end compilation correctness by verifying individual rewrite rules. In 3LA, we focus on the verification of IR-accelerator mappings from which IR-accelerator rewrites are derived. Checking and inferring rules between compiler IR patterns is not the focus of this methodology (Nandi et al., 2021; Bansal and Aiken, 2006; Menendez and Nagarakatte, 2017).

Verifying an IR-accelerator mapping consists of three verification tasks, as illustrated in Figure 5.5. In VT1 and VT3, we directly compare the compiler IR and the accelerator IR against the compiler implementation and the accelerator implementation, respectively. Between the compiler IR and accelerator IR, VT2 checks the equivalence between a compiler IR ILA program fragment and an accelerator ILA program fragment. This is an instruction-sequence-to-instruction-sequence verification, typically over short program fragments that correspond to operations instead of a whole application.

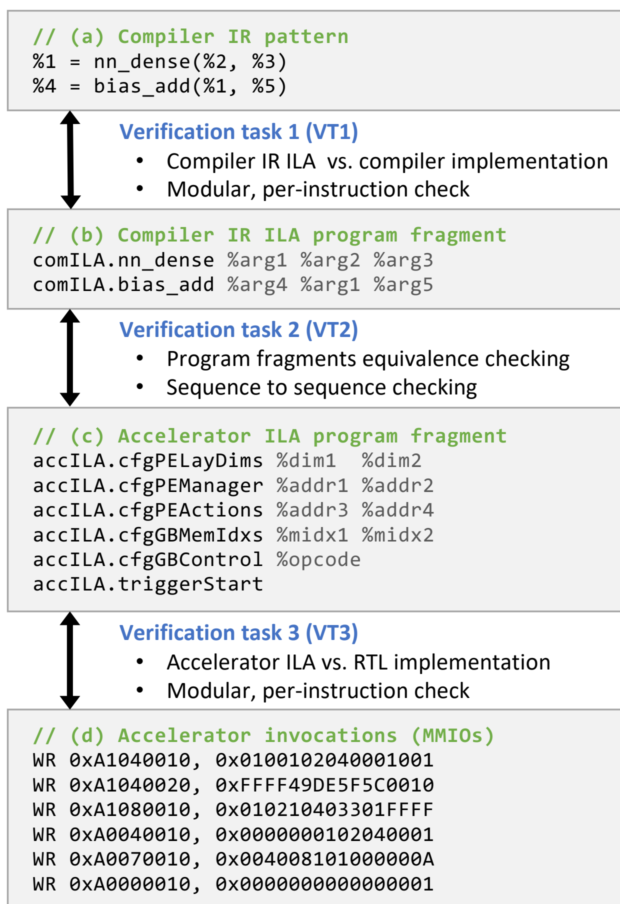


Figure 5.5: A simplified version of Figure 5.4 highlighting the verification tasks in the IR-accelerator mapping for the FlexASR linear layer operation.

Proof-Based Formal Verification The formal semantics of ILA instructions allows for formally verifying the IR-accelerator mappings. For VT1, the equivalence between compiler IR intrinsics and compiler IR ILA instructions can be checked using software model checking tools (e.g., CBMC and SeaHorn (Clarke et al., 2004; Gurfinkel et al., 2015)) by translating ILA models into software models, as has been done in prior work (Huang et al., 2018a). For VT2, the two program fragments can be encoded into Satisfiability Modulo Theories (SMT) formulas (e.g., via unrolling the instructions) and their equivalence checked using an SMT solver such as Z3 (de Moura and Bjørner, 2008). For VT3, the refinement checking between the accelerator ILA (specification) and the accelerator RTL (implementation) has been shown successfully in prior work (Huang et al., 2018a) that leverages processor verification techniques (Burch and Dill, 1994; Manolios and Srinivasan, 2008). We provide a case study for VT2 in Section 5 that focuses on verifying equivalence of the operator definitions in the two program fragments over abstract data types, thus avoiding dealing with differences in numerics which are the focus of the simulation-based validation.

Simulation-Based Validation The 3LA methodology also supports checking simulation-based validation. This is highly automated as the ILAng platform (Huang et al., 2019) can automatically generate an executable software model (in C++/SystemC) of a program of ILA instructions. These executable models capture the precise definitions of the numerics used by the accelerator. For VT1, the simulation of ILA instructions is checked against the execution of the corresponding IR intrinsics (i.e., the original compiler-generated code). For VT2, we compare the simulation of two ILA program fragments. For VT3, the ILA simulation can be checked against RTL simulation of the accelerator implementation.

3.3.2 Application-Level Co-Simulation

Verification at the application level, e.g., examining the final accuracy of an inference model instead of its individual layers, is especially critical for exploiting accelerators that utilize custom data representations. While such accelerators gain power-performance efficiency by

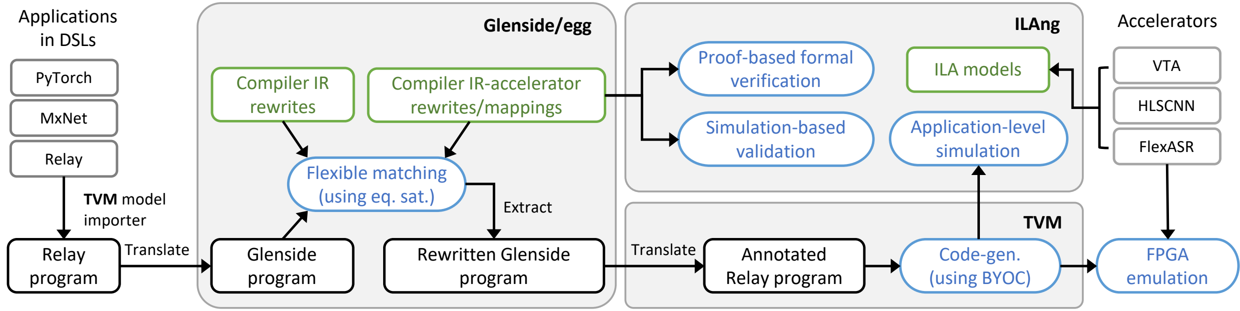


Figure 5.6: Prototype implementation of the 3LA compilation flow.

leveraging custom data types, they may introduce modest numerical mismatches at every operation that can accumulate and alter the application-level result. Thus, the IR-accelerator mapping validation for a pair of program fragments (Section 3.3.1) can at best check that the numerical differences for the computation in these fragments is within a certain range. Unfortunately, this provides no guarantee of the correctness at the application level and, therefore, requires the application-level co-simulation capability enabled by the use of the ILA models. In the 3LA methodology, application-level co-simulation can be achieved simply by executing the full application's code invoking the ILAng-generated simulator whenever an accelerator call is desired.

4 *Prototype Implementation*

As a demonstration of the 3LA methodology, we have implemented an end-to-end compilation flow for DL applications by integrating with existing compiler frameworks. Figure 5.6 shows the workflow of our prototype.

4.1 DSL Front-End

TVM is a compiler framework for DL applications that provides various capabilities for expressing and optimizing DL applications (Chen et al., 2018a). Here, we make use of TVM's model importer as the front-end for DSL programs. The importer supports taking programs

written in mainstream DL frameworks and interchange formats (e.g., ONNX (ONNX, 2019), PyTorch (Paszke et al., 2019a), and TensorFlow (Abadi et al., 2016)) and translating them into Relay, the top-level IR used in TVM (Roesch et al., 2019).

4.2 Flexible Matching

We leverage the `egg` library for equality saturation in our prototype (Willsey et al., 2021). First, the input program is translated from Relay to Glenside, a pure (side effect–free) tensor program representation that supports specifying rewrite rules for tensor programs (Smith et al., 2021). Next, with both the compiler IR rewrites and IR-accelerator rewrites provided in Glenside, the equality saturation engine explores the space of possible rewrites as discussed in Section 3.2. Upon reaching a fixed point, a rewritten program is extracted based on a given cost function. Here, as a proof of concept, we implemented a cost function that maximizes the number of accelerator operations; cost functions that correspond to measures of real-world performance are out of scope for the initial prototype of 3LA, since there are many more complicated considerations that would be difficult to model (such as timing properties, cache sizes, and communication costs).

4.3 Code Generation

Once flexible matching completes, the extracted rewritten program is translated back to Relay where accelerator instructions are specially annotated. In our prototype, we use TVM's Bring Your Own Codegen (BYOC) interface to implement the generation of those accelerator instructions (Chen et al., 2021). BYOC allows for invoking the target interface of a custom execution mechanism (e.g., an accelerator's MMIO loads/stores) by having TVM's runtime defer execution to a user-specified runtime when it reaches an annotated portion of the program, following the process illustrated in Figure 5.7.

In the first step, shown in part (a), user-specified syntactic patterns are matched against an input Relay program. The patterns correspond to operations supported by the target

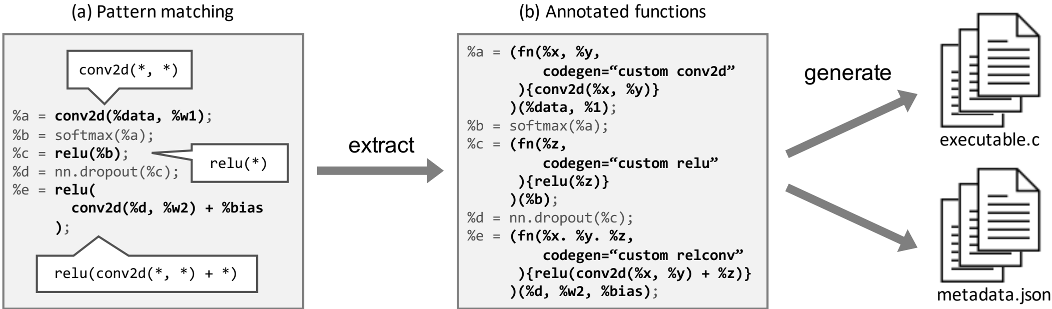


Figure 5.7: An illustration of the BYOC process, using an example similar to that in Chen et al. (2021).

device. In Figure 5.7, we assume that the device supports ReLU and 2D convolution operators, as well as a special combination of ReLU and a 2D convolution as one operation (hence, the pattern-matching step can match AST subtrees comprised of more than a single operator call). In the second step, each AST subtree matched is extracted as a specially annotated Relay function, as shown in Figure 5.7(b). In the third step, when the Relay program is compiled, the annotated functions are passed to the custom code generator. BYOC supports two usage modes for the generated code:

1. The generated file is the source of an executable (e.g., in C) that will invoke the target device's interface to implement the semantics of the matched function. The files will be compiled into binaries, which the TVM runtime will invoke when hitting an annotated function, thereby deferring execution to the generated code.

2. The generated file is device- or library-specific metadata (e.g., in JSON) that will be passed to a user-defined custom runtime. When the TVM runtime hits an annotated function, it will defer execution to the custom runtime. In our prototype implementation, we use this execution mode to implement just-in-time (JIT) code generation to allow for conveniently passing data values to ILA simulators.

Here, we implemented a custom runtime that invokes the accelerator implemented on an

FPGA (Section 5.3.2) as well as the ILAng-generated simulators (see below), producing the necessary ILA instructions at run time using the second execution mode. The JIT approach helps prototyping as it allows for easily inspecting intermediate values in the program and simplifies the implementation, but introduces overhead from communication between the TVM runtime and our custom 3LA runtime. In principle, this overhead can be eliminated by using BYOC's ahead-of-time compilation mode.

Note that BYOC's pattern-matching phase is handled by a dataflow-based DSL provided in TVM (Contributors, 2020), essentially corresponding to the exact syntactic matching strategy. This language allows for specifying patterns containing operator calls, function literals, tuple literals, and wildcards, as well as the disjunction of multiple patterns; for example, the pattern `conv2d(*, *)` will match any call to `conv2d` in the source program. It is also possible to match for the disjunction of two patterns, `p1 | p2`. The pattern language can also check for particular operator attributes (e.g., the stride length for a convolution) or numerical representations to allow for conditional matches. In Figure 5.7, the patterns matched are `conv2d(*, *)`, `relu(*)`, and `relu(conv2d(*, *) + *)`. Patterns in Relay are matched by converting Relay programs into a dataflow graph representation, annotating the entry and exit points of the subgraphs described by the patterns, and subsequently extracting those subgraphs into Relay functions with annotations that indicate to TVM's runtime when to defer execution to the custom-generated code.

While we implemented our exact matching trials using a slightly modified version of BYOC's normal pattern-matching interface, we implemented flexible matching by using Glenside to perform all the necessary rewrites and then converting the final Glenside program into a Relay program with BYOC annotations.

4.4 ILA Modeling and Correctness Verification

We utilize ILAng, an open-source platform for ILA-based modeling and verification, for developing the ILA models and performing ILA-based verification/validation (Huang et al., 2019). ILAng provides support for the following capabilities:

Table 5.1: End-to-end compilation statistics.

Application Statistics							
1	Application	EfficientNet	LSTM-WLM	MobileNet V2	ResMLP	ResNet-20	Transformer
2	Source DSL	MxNet	PyTorch	PyTorch	PyTorch	MxNet	PyTorch
3	#Relay Ops	232	578	757	343	494	872
Number of Static Accelerator Invocations using Exact Matching/Flexible Matching							
4	FlexASR	0/35	1/1	0/41	0/38	2/22	0/66
5	HLSCNN	35/35	0/0	40/40	0/0	21/21	0/0
6	VTA	0/35	36/36	1/41	38/38	2/22	66/66

1. (a) manually specifying and (b) semi-automatically synthesizing an ILA model (Subramanyan et al., 2018).

2. refinement checking between an ILA specification and an RTL implementation.

3. automatic translation from semantics of ILA instructions to SMT formulas.

4. generating a sound executable simulator based on the operational semantics defined by the ILA model.

We use 1(a), 3, and 4 in this work.

5 *Case Studies and Evaluation*

We show the generality of the 3LA methodology through multiple case studies.

5.1 Target Accelerators

We added support for three accelerators specialized for DL applications that provide hardware operations at different levels of granularity:

5.1.1 FlexASR

FlexASR is an accelerator optimized for speech and natural language processing (NLP) tasks that supports various recurrent neural networks (Tambe et al., 2021). It uses a custom numeric datatype, *AdaptivFloat*, for boosting the accuracy of quantized computations (Tambe et al., 2020).

Our compiler supports two of FlexASR's operations: linear layers (illustrated in Figure 5.4) and LSTM layers. For simplicity, the pattern we match for the LSTM layer in exact matching is precisely the formulation of an LSTM produced by TVM's PyTorch importer, which is "unrolled" to the correct number of timesteps (35 in the case of our LSTM-WLM application). For flexible matching, we translate the "unrolled" LSTM in Relay into Glenside and also match it in the target Glenside program. In principle, it would be possible to define a rewrite rule corresponding to a single LSTM timestep, and another rewrite to "fold" adjacent timesteps into a single FlexASR LSTM layer invocation (one ILA instruction) with the total number of timesteps.

5.1.2 HLSCNN

HLSCNN is an accelerator optimized for 2D convolutions (Whatmough et al., 2019). It is designed to operate on 8/16-bit fixed point numbers, and the feature map tensors are expressed in the NHWC layout format for providing better performance through parallelization.

Our HLSCNN specification has only one operation, a non-grouped 2D convolution. In Relay and Glenside, we map any non-grouped 2D convolution (`nn.conv2d`) to the HLSCNN convolution operation. Note that the Relay convolution operation allows for padding an input before convolving it; our implementation pads on the host before invoking the accelerator. In principle, it would be possible to rewrite a grouped convolution into a concatenation of non-grouped convolutions, but the number of groups in the models example tended to be large (960 groups in MobileNet), which would blow up the programs and be impractical to run on a single device.

5.1.3 VTA

VTA is a parameterizable accelerator for tensor operations featuring a processor-like design with an ISA and configurable parameters (Moreau et al., 2019). It provides efficient hardware implementations of element-wise arithmetic operations as well as generalized matrix multiplication (GEMM).

Unlike the above accelerators, VTA is a fine-grained programmable accelerator with a defined ISA. Hence, "operators" in VTA are really sequences of VTA instructions that implement the semantics of a tensor operator in Relay. As discussed in Huang et al. (2018a), for processor-like designs, the ILA can be based on the ISA. TVM has a built-in bespoke code generator for VTA, which operates on TVM's lower-level Halide-like DSL and directly implements arbitrary tensor operations for VTA. In principle, it would be possible for us to adapt the existing VTA code generator to instead output VTA ILA instructions, resembling traditional instruction selection. For simplicity, our prototype implements matrix multiplication and addition as fixed sequences of VTA ILA instructions.

5.1.4 Implementation Complexity

The ILAs for FlexASR, HLSCNN, and VTA are approximately 5600, 1600, and 2100 lines of code, respectively—note that these ILAs serve the dual purposes of enabling compilation via the 3LA methodology as well as validating the RTL design. Additionally, the BYOC-based code generators and runtimes for these devices are approximately 450, 300, and 900 lines of code, respectively. For comparison, the existing VTA compilation stack in TVM is about 5500 lines of code (though it supports many more features than our code generator).

5.2 Target Applications

For our experiments, we considered six DL applications corresponding to common neural network models for language and vision tasks that contain operators supported by the three target accelerators. We selected applications with reasonable size for human inspection and

in-depth analysis.

With the exception of the minor change for LSTM-WLM, all applications were mapped to accelerators *without any manual modifications*.

5.2.1 EfficientNet

EfficientNet is a recent convolutional neural network (CNN) designed for image classification that uniformly scales network width, depth, and resolution (Tan and Le, 2019). We chose it because it contains convolutions that could be accelerated by VTA and HLSCNN. We used a publicly available implementation of EfficientNet[2] in MxNet, pretrained on ImageNet (image size 224×224, with 1000 classes). We imported it through TVM's MxNet importer.

5.2.2 LSTM-WLM

LSTM-WLM is a simple text generation application (PyTorch, 2020) implemented using an LSTM recurrent neural network architecture (Graves and Jaitly, 2014). We chose this model because it contains an LSTM layer that could be accelerated by FlexASR.

We used the LSTM model implementation from the official PyTorch examples repository,[3] training on WikiText-2 with the provided script on the following settings: 40 epochs, a batch size of 20, sequence length of 35, and an initial learning rate of 20, with a single layer for the LSTM. We imported it through TVM's PyTorch importer with one simplification in the importer: for simplicity, our FlexASR LSTM layer integration only returned the LSTM's sequence output but not the final hidden and cell states (even though the device itself supports this). In order to match the semantics between our integration and the LSTM, we modified the imported LSTM not to return the final hidden and cell states either. In future work, it would be feasible for us to support returning the final hidden and cell states and eliminate this simplification.

---

[2] `https://github.com/mnikitin/EfficientNet`, accessed Nov. 18, 2021.

[3] `https://github.com/pytorch/examples/tree/master/word_language_model`, accessed Nov. 18, 2021.

5.2.3 MobileNet V2

MobileNet V2 is a commonly used CNN, designed for mobile and embedded vision applications, that uses depth-wise separable convolutions (Howard et al., 2017; Sandler et al., 2019). We chose MobileNet due to its wide use, especially on embedded devices.

We used an open-source implementation of MobileNetV2[4] in PyTorch and used the implementation's provided script to train on CIFAR-10, training for 200 epochs with a learning rate of 0.01 and a batch size of 128. We imported it using TVM's PyTorch importer.

5.2.4 ResMLP

ResMLP is a recent residual network, designed for image classification, comprised only of multi-layer perceptrons and no convolutional layers (Touvron et al., 2021). We chose this model because its preponderance of linear layers means it could be accelerated by VTA as well as by FlexASR (despite not being a language model).

We used an open-source ResMLP implementation[5] in PyTorch. We used similar parameters as those reported in Touvron et al. (2021) for training on CIFAR: 384 features, 12 layers, and a patch size of 16. We trained it on CIFAR-10 for 100 epochs with a learning rate of 0.01, though in Table 5.4, we note that we obtained a lower reference accuracy on CIFAR-10 than that reported in Touvron et al. (2021). We are not certain that we trained using all the same settings as in the original work and (in order to reduce the load on the simulator) we trained and evaluated on 32×32 images, whereas the original work scaled the images up to 256×256. We imported the model through TVM's PyTorch importer.

5.2.5 ResNet-20

ResNet-20 is a CNN designed for image classification that applies identity mapping (He et al., 2016a). As with MobileNet, we chose ResNet for its common use in practice.

---

[4] `https://github.com/kuangliu/pytorch-cifar`, accessed Nov. 18, 2021.

[5] `https://github.com/lucidrains/res-mlp-pytorch`, accessed Nov. 18, 2021.

We used the Gluon Model Zoo's implementation of ResNet-20 in MxNet,[6] pretrained on CIFAR-10. We imported it unmodified through TVM's MxNet importer.

5.2.6 Transformer

Transformer is a language representation model comprised primarily of attention mechanisms (Vaswani et al., 2017). We chose Transformer as a representative of recent NLP models in common use.

We used the `nn.Transformer` implementation from PyTorch, with 8 heads, 6 encoder layers, 6 decoder layers, and 256 features (left untrained). These settings were based on a TVM PyTorch importer unit test.

5.3 Evaluation: Compilation

We examined the portability provided by 3LA through end-to-end compilation using our compiler prototype. We took the previously described six DL applications, developed by different teams in different DSLs, and compiled them for the three target accelerators. Our compiler prototype successfully generated code that exploits the accelerators for supported computations. Furthermore, we demonstrate full-system deployment, running 3LA-generated code on physical hardware through FPGA emulation.

5.3.1 Portability and Flexible Matching

Table 5.1 shows the compilation statistics of using exact matching and flexible matching. It describes the source language in which the application is programmed (Row 2) and the program complexity using the number of Relay operators as a proxy (Row 3). It reports the number of invocations to FlexASR, HLSCNN, and VTA when using exact/flexible matching in Rows 4-6. Note that some invocations (IR-acclerator mappings) correspond to multiple Relay operators; in particular, the 35-step FlexASR LSTM in LSTM-WLM is 566 operators

---

[6] `https://cv.gluon.ai/api/model_zoo.html#gluoncv.model_zoo.cifar_resnet20_v1`, accessed Nov. 18, 2021.

and maps to *one* FlexASR LSTM instruction (a dramatic granularity mismatch between DSL and accelerator operations).

Our results demonstrate portability with the successful exploitation of accelerators for supported operations and provide evidence for the utility of flexible matching. For example, flexible matching revealed several offloads to FlexASR's linear layer in MobileNet V2 by rewriting `nn.dense` to `nn.dense` followed by an add of a zero tensor. Also note that certain Glenside rewrites that implement the `im2col` optimization (Chellapilla et al., 2006) result in many more offloads to VTA in that table; this is due to 2D convolutions being rewritten into matrix multiplications. Hence, flexible matching allowed us to support 2D convolutions on VTA even though our prototype code generator did not explicitly implement 2D convolutions via VTA instructions. This is an example of *emergent effects* resulting from simple, reusable rewrite rules.

5.3.2 System Deployment and FPGA Emulation

To demonstrate the 3LA methodology on a real hardware platform, we synthesized, placed-and-routed the FlexASR accelerator on a Xilinx Zynq ZCU102 FPGA, which consumed 86% of the available LUT resources.[7] We utilized the Xilinx SDK (XilinxSDK, nd) to pass the accelerator instructions (MMIO commands) to the accelerator interface for invoking the supported operations. Specifically, we compiled and executed synthetic application programs in which LSTM layers and linear layers are offloaded to the FlexASR accelerator. This case study demonstrates the applicability of 3LA in actual system deployment on a commodity hardware platform. Further, it shows that in the absence of compilation-results validation enabled by the 3LA methodology, the need for a fully functional RTL model and the significant engineering overhead indeed limit early-stage software/hardware co-design.

---

[7] Due to the significant engineering overhead of FPGA emulation, FlexASR is the only accelerator we deployed on an FPGA.

Table 5.2: **Simulation-based validation results of checking IR-accelerator mappings (partial).** The average relative error (Avg. Err.) and its standard deviation (Std. Dev.) are measured over 100 simulated test inputs.

	Accelerator	Operation	Avg. Err.	Std. Dev.
1	VTA	GEMM	0.00%	0.00%
2	HLSCNN	Conv2D	1.78%	0.16%
3	FlexASR	LinearLayer	0.84%	0.29%
4	FlexASR	LSTM	1.21%	0.19%
5	FlexASR	LayerNorm	0.27%	0.20%
6	FlexASR	MaxPool	0.00%	0.0%
7	FlexASR	MeanPool	1.79%	0.28%
8	FlexASR	Attention	4.22%	0.09%

5.4 Evaluation: Compilation-Results Validation

An important criterion of correct compilation is that the compiled program, in which parts of the computation are offloaded to accelerators, must retain the same functionality as intended with the IR semantics. Thus, we use the IR ILA specification as the reference for formal verification of the IR-accelerator mappings, and use an IR interpreter as the reference when running simulation at two levels: the operation level (for checking the IR-accelerator mappings) and the application level.

5.4.1 Checking IR-Accelerator Mappings

Modern accelerators often adopt custom numerics for achieving better power-performance efficiency. For example, in our cases, FlexASR and HLSCNN use the AdaptivFloat and 8/16-bit fixed point data types, respectively. This means checking the IR-accelerator mappings must account for the numerical differences. To separate the effect of numerics and focus on the definition of operations, we use abstract data types to formally verify the equivalence (demonstrated through a proof-of-concept case study). In addition, to precisely capture the numerical differences, we use simulation to compare accelerator executions (using an ILA

simulator) against the IR semantics (using an IR interpreter).

Simulation-Based Validation For checking the mappings through simulation, we generated 100 random test inputs and compared the outputs of the accelerator ILA simulator and that of an IR interpreter. The accelerator ILA simulators precisely model the data types used by the accelerators. Specifically, VTA, HLSCNN, and FlexASR use 8-bit integer, 8/16-bit fixed point, and AdaptiveFloat, respectively. For the IR interpreter, as a reference, we use 8-bit integers for VTA and 32-bit floating point for HLSCNN and FlexASR. These are respectively the closest standard datatypes to those used by these accelerators. The relative errors are measured using the standard Frobenius norm (Anderson et al., 1999) for the tensors: $Error = \|Out_{ref} - Out_{acc}\|_F \,/\, \|Out_{ref}\|_F$.

Table 5.2 shows a selected subset of the validation results: four IR-accelerator mappings (Rows 1-4) that are used in the end-to-end compilation (Table 5.1) and four additional mappings for non-trivial operations (Rows 5-8). We omit validation results of other mappings, e.g., for trivial operations like *add* and *max*. Columns 1 and 2 indicate the accelerator and the supported operation for each IR-accelerator mapping, respectively. Columns 3 and 4 provide the average relative error and the standard deviation, respectively, over the 100 test inputs. For mappings that are not affected by numerical differences, e.g., the VTA-supported GEMM, we see exact matches in the results. For other mappings, we see deviations caused by the custom numerics, especially for complex operations such as the attention operation for FlexASR.

Proof-Based Formal Verification The key challenges in formally verifying mappings between fragments that represent DL computations include handling nested loops that iterate through tensor elements (in both the compiler IR and accelerator specification) and relating tensor variables between the two sides which may employ various tiling mechanisms. As a proof of concept, we considered the Relay and FlexASR fragments for the FlexASR MaxPool IR-accelerator mapping. These fragments both have three or more nested loops, and the

relation between the two fragments must account for a special customized tiling provided by FlexASR (Tambe et al., 2021). For this study, we considered equivalence of the fragments over fixed-sized tensors with symbolic data[8] and implemented verification using two methods, bounded model checking (BMC) (Biere et al., 2003) and by employing a solver for constrained Horn clauses (CHCs) (Komuravelli et al., 2016).

The BMC-based method unrolls all the loops in both fragments, which is straightforward but may fail to scale for large-sized tensors. The CHC-based method is given a product program of the two fragments and uses *relational* loop invariants, i.e., formulas that relate the two fragments at intermediate loop boundaries. This avoids loop unrolling and can handle large tensors. Our implementation uses Z3 (de Moura and Bjørner, 2008) as the underlying SMT solver in both approaches.

While ILAng directly supports BMC, we manually created CHCs for the CHC-based method. We also supplied the relational invariants that capture the customized tiling of FlexASR. In future work, we plan to automate CHC generation, which will allow formal verification of other IR-accelerator mappings used in this paper. Table 5.3 shows the results for this case study for various dimensions of the 2D input matrix (Column 1), with runtimes of the BMC-based and CHC-based verification methods in Columns 2 and 3, respectively. The BMC-based method was able to verify equivalence of mappings with small-sized matrices, but timed out (with a 3-hour time limit) on the 16×64 matrix that was used for simulation-based validation. In contrast, the CHC-based method was faster than BMC and successfully verified mappings with larger matrices. These results are encouraging and demonstrate how the 3LA methodology enables formal verification of key steps in the compilation flow.

5.4.2 Application-Level Co-Simulation

With the validated IR-accelerator mappings, we want to check if minor deviations at the operation level will influence application-level behavior. Therefore, we performed application-

[8]Formally modeling custom numerics at the bit level is left to future work.

Table 5.3: **Formal verification case study: results for verifying the IR-accelerator mapping for FlexASR MaxPool.** Experiments were run on an Intel Core i7-5500U CPU (two 2.40GHz cores) with 8 GB RAM.

Matrix dim.	BMC verif. time (s)	CHC verif. time (s)
2×16	443	38
4×16	1976	37
4×32	7954	146
8×64	Timeout (>3 hrs)	1831
16×64	Timeout (>3 hrs)	5177

Table 5.4: **Application-Level Co-Simulation Results**. In each validation, we evaluated 2000 images (for vision tasks) or 100 sentences (for text generation) that were evenly sampled from the corresponding dataset. Results for the vision models ResMLP, ResNet-20, and MobileNet V2 are given in terms of classification accuracy, while results for LSTM-WLM (a text generation model) are given in terms of perplexity. The original result is measured using original accelerator designs. The updated result is measured using modified designs provided by the accelerator developers.

Application	Processing Platform	Reference Res.*	Original Res.	Updated Res.	Time†
LSTM-WLM	FlexASR	122.15 (perp.)	257.39 (perp.)	Reported	1m10s
ResMLP	FlexASR	69.65% (acc.)	10.65% (acc.)	Reported	19m15s
ResNet-20	FlexASR & HLSCNN	91.55% (acc.)	29.15% (acc.)	91.85% (acc.)	14m23s
MobileNet V2	FlexASR & HLSCNN	92.40% (acc.)	10.35% (acc.)	91.20% (acc.)	42m26s

∗ The reference result does not represent the best achievable accuracy/perplexity of the model on the given dataset. This table is intended for comparing the application-level results on different processing platforms.

† Average simulation time of running one data point (e.g., an image or a sentence) on a 2.4GHz AMD EPYC 7532 core.

level co-simulation on several applications which offload various computations to FlexASR and HLSCNN, the two accelerators that utilize custom numerics. Specifically, we examined LSTM-WLM and ResMLP, which offload to FlexASR the LSTM layer and linear layer operations, respectively. We also considered MobileNet and ResNet, which both offload 2D convolution and linear layer operations to HLSCNN and FlexASR, respectively. (We compiled to two target accelerators by including the IR-accelerator rewrite rules for both accelerators.)

We trained and validated the LSTM-WLM model using the WikiText-2 dataset (Merity et al., 2016). The image classification models (MobileNet V2, ResMLP, and ResNet-20) were trained and validated using the CIFAR-10 dataset (CIFAR, 2009). Table 5.4 shows the application-level co-simulation results. Columns 1 and 2 describe the application and the target processing platform under evaluation, respectively. We provide a reference result (perplexity for the text-generation task and inference accuracy for vision tasks) in Column 3 by running the application on the host processor, i.e., not offloading to accelerators. The validation result using original accelerator designs, labeled "Original Result," is provided in Column 4. For cases where the original result was significantly poorer than the reference result, we reported it to the accelerator developers for their further investigation. When they provided an accelerator design modification to address this, we provide an updated result, using this modified accelerator, in Column 5. The average simulation time is reported in Column 6.

Case Study: ResNet-20 and MobileNet V2 We reported the original validation results of ResNet-20, which were far from the 91.55% reference result, to the accelerator developers. We also provided statistics for each accelerator invocation (e.g., error accumulation, input and output ranges, etc.), gathered by our compiler prototype and ILA simulators. With the information, the accelerator developers were able to identify the root cause: weight data values in HLSCNN's 2D convolutional layers were heavily quantized by its 8-bit fixed point data type due to a narrower value range. After updating the design by expanding the original

8-bit representation to 16 bits, the application-level result matched up to the reference result. The same tuning approach also resulted in improved accuracy for MobileNet V2.

The results in Table 5.4 reaffirm the need for application-level validation, especially for accelerators utilizing custom numerics. However, without a portable end-to-end compilation flow, such application-level validation is prohibitively difficult for new accelerators. Through our case studies, we demonstrate how 3LA provides systematic and automatic compilation-results validation at the application level and also show its usefulness in software/hardware co-design. Specifically, with the ILA, 3LA provides quick design space exploration and numerics tuning without hardware engineering overhead (e.g., deploying to FPGA) in each hardware design iteration. Further, it provides handy debugging information and efficient simulation. (For FlexASR, we see a 30× speedup on average with the ILA simulator compared to RTL simulation using a commercial Verilog simulator.)

6 Discussion and Future Work

The 3LA methodology establishes a foundation for an end-to-end, extensible compilation flow for utilizing accelerators. Our prototype not only provides a working implementation, but also an experimental framework for future research in this area. Thus far, we have demonstrated how flexible matching, mapping validation, and application-level co-simulation are enabled by 3LA.

Below we discuss two example near-term extensions ripe for further exploration and inclusion in 3LA-based frameworks.

6.1 Optimizing Data Transfers

As a motivating example, we consider an image processing application with a 2D maxpooling layer that we would like to offload to FlexASR. Suppose that the maxpooling layer uses a window with shape $(4, 4)$ and stride $(2, 2)$ and is used to downsample a 128×128 matrix into a 64×64 matrix. However, FlexASR does not directly support this window or stride size. Instead, it supports a related operation called *temporal maxpooling*, which corresponds to 2D

```
; (a) IR-accelerator rewrite rule for FlexASR's temporal maxpooling
; (which corresponds to 2D maxpooling with window shape and stride (2, 1))
(map reduceMax (windows (2, 1) (2, 1) ?T)) -> (fasrMaxpLoad (fasrMaxpool (
    fasrMaxpStore ?T)))

; (b) Initial Glenside code for a 2D maxpool with window shape (4, 4)
; and stride (2, 2). We use T to denote the input of the 2D maxpooling layer.
(map reduceMax (windows (4, 4) (2, 2) T))

; (c) A rewritten IR program found via Glenside
; T denotes the input of the 2D maxpooling layer,
; and S is the shape of the output of the 2D maxpooling layer
(reshape (map reduceMax (windows (2, 1) (2, 1)
(map reduceMax (windows (2, 1) (2, 1)
(map reduceMax (windows (2, 1) (2, 1)
(map reduceMax (windows (2, 1) (2, 1)
(map flatten (windows (4, 4) (2, 2) T)) )))))))) S)

; (d) 2D maxpooling using FlexASR
(reshape (fasrMaxpLoad (fasrMaxpool (fasrMaxpStore
(fasrMaxpLoad (fasrMaxpool (fasrMaxpStore
(fasrMaxpLoad (fasrMaxpool (fasrMaxpStore
(fasrMaxpLoad (fasrMaxpool (fasrMaxpStore
(map flatten (windows (4, 4) (2, 2) T)) )))))))))))) S)

; (e) IR-accelerator rewrite rule to remove redundant Store-Loads
(fasrMaxpStore (fasrMaxpLoad ?T) -> ?T

; (f) Optimized 2D maxpooling using FlexASR
(reshape (fasrMaxpLoad (fasrMaxpool (fasrMaxpool (fasrMaxpool (fasrMaxpool (
    fasrMaxpStore
(map flatten (windows (4, 4) (2, 2) T)) )))))) S)
```

Figure 5.8: An illustration of how 3LA offloads 2D maxpooling to FlexASR's temporal maxpooling operation. Note that (b) does not contain a match for the left-hand side of the IR-accelerator rewrite rule in (a). (c) is an equivalent rewritten IR program found by flexible matching, containing four instances of the left-hand side of the IR-accelerator rewrite rule. The result of the replacements is given in (d). Note that in this program, the initial store and the final load are needed to communicate with FlexASR; however, the intermediate loads/stores can be eliminated, since the output of one instance serves as input of another. (e) gives a rewrite rule for removing intermediate loads/stores and (f) shows the result of applying it. This program only performs a single (matrix) store at the start of the operation and a single (matrix) load to read the output at the end of the operation. In the future, we hope to generalize this example and consider memory organization in accelerators and data-movement for optimizing data transfers.

maxpool with a fixed window of shape $(2, 1)$ and stride $(2, 1)$. The following IR-accelerator rewrite rule represents an offloading of the temporal maxpooling operation, where the IR fragment is shown on the left of the arrow and the FlexASR fragment, on the right (`?T` in the pattern denotes the input matrix; fragments are shown as S-expressions).

```
(map reduceMax (windows (2, 1) (2, 1) ?T)) ⟹
(fasrMaxpLoad (fasrMaxpool (fasrMaxpStore ?T)))
```

By using flexible matching, our prototype found the following rewritten IR program for the 2D maxpooling layer, with a (4, 4) window of shape and a stride of 2 on both axes and where `T` denotes the input to the layer and `S` is the shape of the layer's output:

```
(reshape (map reduceMax (windows (2, 1) (2, 1)
(map reduceMax (windows (2, 1) (2, 1)
(map reduceMax (windows (2, 1) (2, 1)
(map reduceMax (windows (2, 1) (2, 1)
(map flatten (windows (4, 4) (2, 2) T))
)))))))) S)
```

Then the IR-accelerator rules above rewrite each of these four `map reduceMax` instances to the FlexASR fragment. The listings for this example are in Figure 5.8, with the result of applying the IR-accelerator mapping in part (d).

Note that each of the `map reduceMax` instances in the rewritten IR program is mapped to a composition of three FlexASR instructions (as shown in the IR-accelerator mapping rewrite rule), where `fasrMaxpStore` stores the input data into FlexASR, `fasrMaxPool` performs the maxpool computation in FlexASR, and `fasrMaxpLoad` loads the output result from FlexASR. When four of these instances are composed, the initial store and the final load are needed to communicate with FlexASR; however, the other intermediate transfers can be eliminated, since the output of one instance serves as input of another. We plan to enhance our prototype to cancel such redundant transfers, with the final optimized result shown in part (f) of Figure 5.8.

This example illustrates the importance of minimizing data transfers while offloading

operations to accelerators. Note that a fixed set of accelerator APIs may not allow such optimizations, whereas 3LA provides this flexibility through individual accelerator instructions. In future work, we would like to consider more general optimizations on the accelerator side that account for memory organization and data movement, potentially leveraging standard register allocation as well as recent DL operator fusion techniques (Niu et al., 2021).

6.2 Extending Formal Verification of Mappings

In Section 5.4.1, we explored CHC-based verification of IR-accelerator mappings for fixed-size tensors (with symbolic data) and supplied relational loop invariants to the verifier. In future work, we would like to add support for symbolic-sized tensors and automatic inference of relational loop invariants. Additionally, our simulation validation reveals that custom numerics can significantly impact model accuracy. We would like to extend our verification to account for custom numerics and check or derive error bounds.

7 Summary

The 3LA methodology addresses key challenges presently complicating effective accelerator utilization under the prevalent API-based approach. Specifically, we highlight the lack of portability, the inability to integrate accelerators into existing optimizing compilers, and the difficulty in validating generated code. The 3LA methodology alleviates these issues by introducing a formal software/hardware interface for accelerators, using the recently developed ILA representation to fulfill this purpose. Using term rewriting to implement flexible matching allows for easily, extensibly, and verifiably exposing opportunities to apply accelerator operations, in a process resembling traditional instruction selection. The 3LA prototype serves as a proof of concept of such a compilation flow, indeed incorporated into an existing DL compiler stack and able to simulate entire DL models end to end (exposing, in the process, potential issues with the numerical representations used by the devices). While the prototype lacks some of the capabilities of propertietary tools developed by accelerator manufacturers, it demonstrates that adding accelerator support to an existing compiler stack is

no longer the exclusive province of large enterprises that can afford entire teams of hardware and compilers experts.

Chapter 6

CONCLUSION

The main contributions of this dissertation, DTR and the 3LA methodology, apply techniques from the broader programming languages literature in order to satisfy the needs of DL applications. These contributions were in turn motivated by the earlier work on the Relay language, to which I contributed, which allowed for expressing DL models as programs in a general functional programming language with support for automatic differentiation, ultimately achieving performance superior to other DL frameworks while generalizing many past DL optimizations into compiler optimizations. In DTR, we approach the technique of checkpointing as a dynamic analysis by taking an approach inspired by software caching and register rematerialization, ultimately developing a checkpointing algorithm that not only attains near-optimal performance on static models but also generalizes to DL models with arbitrary dynamic control flow. The 3LA methodology presents a technique for integrating support for new accelerators into an existing compiler stack, culminating in a prototype that extends the TVM compiler stack for DL with support for greatly varying accelerators, applying equality saturation in order to apply more automation. These systems approach distinct problems in the DL domain by taking more abstract views of DL models, namely as general programs which might use the features provided by typical programming languages, ultimately adapting general compilers techniques to this particular domain without imposing assumptions of a more restricted computing model.

While the systems presented in the dissertation achieve tangible results with applicability to deep learning, there are many possibilities for further work along these lines. DTR only considers a limited set of metadata for formulating its heuristics and relies on manually specified heuristics. We may consider taking further metadata into account, as in the

MegEngine DL framework (MegEngine, 2021), which includes an implementation of DTR whose heuristic takes memory fragmentation into account and prioritizes evictions that reduce fragmentation, allowing for better utilization of GPU memory. Another possibility for DTR would be to investigate whether heuristics could be learned from multiple training runs rather than manually specifying them, which may provide more confidence that the heuristics chosen are optimal for the specific applications chosen. As discussed in Chapter 2, works such as that of Tang et al. (2022) also raise the possibility of combining DTR's dynamic evictions with swapping, suggesting that the runtime might choose when to free a tensor from memory or when simply to send it to CPU or another device, providing more flexibility though posing challenges in ensuring optimal performance.

The 3LA methodology presents even more possibilities, as the prototype discussed is only a proof of concept meant to demonstrate an end-to-end compilation pipeline but without any guarantees of performance. Applying the methodology to write a production-ready compiler that could achieve performance comparable to present-day custom accelerator integrations would likely require the addition of numerous new layers that provide information about timings and other hardware performance characteristics, suggesting the need to add further intermediate layers between the pattern-matching phase in the high-level DSL and the generation of ILA instructions. The same will likely also be necessary to properly generate performant code for accelerators that have expressive instruction sets, like VTA. One implication of more sophisticated performance modeling in 3LA would also be that the cost function for flexible matching would have to be sufficiently sophisticated to correspond to true likely performance, which is likely to be a challenging future problem given the difficulty of accurately modeling hardware performance. Additionally, the aspects of formal verification explored in the 3LA prototype do not amount to true end-to-end verification of the compiler stack, as in CompCert (Leroy, 2006)—besides verifying more accelerator operations and their correspondence to language constructs, many more intermediate steps would have to be formally specified and verified, including the individual rewrite rules as well as the correctness of the flexible matching step (recent progress on proof artifacts in `egg` should

be of assistance).

Beyond the future possibilities of the specific systems detailed in this dissertation, I would like to emphasize the broader trend that they and related systems signify. DTR, 3LA, and my broader work on Relay have addressed with all levels of the DL system stack: the representation and expression of programs and reasoning about them, system-level optimizations, and hardware-level optimizations. The contributions I have detailed in this dissertation form a small part of the vast infrastructure that is continually being developed to support emerging DL applications, not only the models in use today but supporting more expressive features in the hope that future models will make use of them. The need for such systems highlights the present importance, economies of scale, and constantly shifting frontiers of the DL domain: New, greatly varying applications are being developed and new hardware is being developed at an astonishing pace in order to support those applications, all the while motivating the development of systems-level optimizations in order to ensure the best use of computing resources for DL applications.

The systems discussed in this dissertation metonymize a new compiler stack that has emerged to support DL applications, or indeed "differentiable programming" writ large—what is essential to note is that this new compiler infrastructure resembles in great deal traditional compilers used to support general-purpose languages on general-purpose programmable devices. It is my hope that these systems will find use not only for DL applications but may serve as a guide for adapting well-studied principles from the compilers literature to future domains and future problems that may demand similarly great allocation of computing resources.

BIBLIOGRAPHY

Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283. `https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf`

E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, Jack J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. 1999. *LAPACK Users' Guide (Third Ed.)*. Society for Industrial and Applied Mathematics, USA.

Jason Ansel. 2022. TorchDynamo. `https://github.com/facebookresearch/torchdynamo` Accessed Apr. 5, 2022.

Franz Baader and Tobias Nipkow. 1998. *Term Rewriting and All That*. Cambridge University Press, USA.

Rick Bahr, Clark Barrett, Nikhil Bhagdikar, Alex Carsello, Ross Daly, Caleb Donovick, David Durst, Kayvon Fatahalian, Kathleen Feng, Pat Hanrahan, Teguh Hofstee, Mark Horowitz, Dillon Huff, Fredrik Kjolstad, Taeyoung Kong, Qiaoyi Liu, Makai Mann, Jackson Melchert, Ankita Nayak, Aina Niemetz, Gedeon Nyengele, Priyanka Raina, Stephen Richardson, Raj Setaluri, Jeff Setter, Kavya Sreedhar, Maxwell Strange, James Thomas, Christopher Torng, Leonard Truong, Nestan Tsiskaridze, and Keyi Zhang. 2020. Creating an Agile Hardware Design Flow. In *Proceedings of the 57th ACM/EDAC/IEEE Design Automation*

Conference (DAC '20). IEEE Press, New York, NY, USA, Article 142, 6 pages. https://doi.org/10.1109/DAC18072.2020.9218553

Sorav Bansal and Alex Aiken. 2006. Automatic Generation of Peephole Superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. Association for Computing Machinery, New York, NY, USA, 394–403. https://doi.org/10.1145/1168857.1168906

Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2015. Automatic differentiation in machine learning: a survey. *CoRR* abs/1502.05767 (2015). arXiv:1502.05767 http://arxiv.org/abs/1502.05767

Olivier Beaumont, Julien Herrmann, Guillaume Pallez, and Alena Shilova. 2019. Optimal Memory-aware Backpropagation of Deep Join Networks. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 378 (01 2019). https://doi.org/10.1098/rsta.2019.0049

Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. 2003. Bounded model checking. *Adv. Comput.* 58 (2003), 117–148. https://doi.org/10.1016/S0065-2458(03)58003-2

John Binder, Kevin Murphy, and Stuart Russell. 1997. Space-Efficient Inference in Dynamic Probabilistic Networks. In *Proceedings of the Fifteenth International Joint Conference on Artifical Intelligence - Volume 2 (IJCAI'97)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1292–1296.

Gabriel Hjort Blindell. 2016. *Instruction Selection - Principles, Methods, and Applications*. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-319-34019-7

Preston Briggs, Keith D. Cooper, and Linda Torczon. 1992. Rematerialization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and*

Implementation (PLDI '92). Association for Computing Machinery, New York, NY, USA, 311–321. https://doi.org/10.1145/143095.143143

Andrew Brock, Jeff Donahue, and Karen Simonyan. 2018. Large Scale GAN Training for High Fidelity Natural Image Synthesis. arXiv:cs.LG/1809.11096

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:cs.CL/2005.14165

Jerry R. Burch and David L. Dill. 1994. Automatic Verification of Pipelined Microprocessor Control. In *Proceedings of the 6th International Conference on Computer Aided Verification (CAV '94)*. Springer-Verlag, Berlin, Heidelberg, 68–80. https://dl.acm.org/citation.cfm?id=735662

Murray S. Campbell and A. Joseph Hoane. 1999. Search control methods in Deep Blue. In *In AAAI Spring Symposium on Search Techniques for Problem Solving Under Uncertainty and Incomplete Information*. AAAI Press, pages.

Ningyuan Cao, Baibhab Chatterjee, Minxiang Gong, Muya Chang, Shreyas Sen, and Arijit Raychowdhury. 2020. A 65nm Image Processing SoC Supporting Multiple DNN Models and Real-Time Computation-Communication Trade-Off Via Actor-Critical Neuro-Controller. In *Proceedings of the 2020 IEEE Symposium on VLSI Circuits*. IEEE, New York, NY, USA, 1–2. https://doi.org/10.1109/VLSICircuits18222.2020.9162878

Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd

Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. 2016. A Cloud-Scale Acceleration Architecture. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. IEEE Press, New York, NY, USA, Article 7, 13 pages. https://doi.org/10.1109/MICRO.2016.7783710

Wei-Ting Jonas Chan, Andrew B. Kahng, Siddhartha Nath, and Ichiro Yamamoto. 2014. The ITRS MPU and SOC system drivers: Calibration and implications for design-based equivalent scaling in the roadmap. In *Proceedings of the 32nd IEEE International Conference on Computer Design (ICCD '14)*. IEEE Computer Society, New York, NY, USA, 153–160. https://doi.org/10.1109/ICCD.2014.6974675

Kartik Chandra and Rastislav Bodik. 2017. Bonsai: Synthesis-Based Reasoning for Type Systems. *CoRR* abs/1708.00551 (2017). arXiv:1708.00551 http://arxiv.org/abs/1708.00551

Kumar Chellapilla, Sidd Puri, and Patrice Simard. 2006. High Performance Convolutional Neural Networks for Document Processing. In *Proceedings of the Tenth International Workshop on Frontiers in Handwriting Recognition*, Guy Lorette (Ed.). Université de Rennes 1, La Baule (France), 6. https://hal.inria.fr/inria-00112631

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018a. TVM: End-to-End Compilation Stack for Deep Learning. In *SysML 2018*. https://arxiv.org/abs/1802.04799

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018b. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation*

(OSDI 18). USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. *CoRR* abs/1604.06174 (2016). arXiv:1604.06174 http://arxiv.org/abs/1604.06174

Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018c. Learning to Optimize Tensor Programs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 3393–3404.

Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. 2017. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE J. Solid State Circuits* 52, 1 (2017), 127–138. https://doi.org/10.1109/JSSC.2016.2616357

Zhi Chen, Cody Hao Yu, Trevor Morris, Jorn Tuyls, Yi-Hsiang Lai, Jared Roesch, Elliott Delaye, Vin Sharma, and Yida Wang. 2021. Bring Your Own Codegen to Deep Learning Compiler. arXiv:cs.LG/2105.03215

CIFAR 2009. The CIFAR-10 dataset. Retrieved Nov. 15, 2021 from http://www.cs.toronto.edu/~kriz/cifar.html

Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004) (Lecture Notes in Computer Science)*, Kurt Jensen and Andreas Podelski (Eds.), Vol. 2988. Springer, Berlin, Heidelberg, 168–176. https://doi.org/10.1007/978-3-540-24730-2_15

Torch Contributors. 2019. Broadcasting Semantics. https://pytorch.org/docs/stable/notes/broadcasting.html

TVM Contributors. 2020. Pattern Matching in Relay. https://tvm.apache.org/docs/langref/relay_pattern.html. Accessed Apr. 9, 2021.

B. Dauvergne and L. Hascoët. 2006. The Data-Flow Equations of Checkpointing in reverse Automatic Differentiation. In *International Conference on Computational Science, ICCS 2006, Reading, UK*.

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008) (Lecture Notes in Computer Science)*, C. R. Ramakrishnan and Jakob Rehof (Eds.), Vol. 4963. Springer, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Nachum Dershowitz. 1993. A Taste of Rewrite Systems. In *Functional Programming, Concurrency, Simulation and Automated Reasoning (Lecture Notes in Computer Science)*, Peter E. Lauer (Ed.), Vol. 693. Springer, Berlin, Heidelberg, 199–228. https://doi.org/10.1007/3-540-56883-2_11

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:cs.CL/1810.04805

Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2015. Fuzzing the Rust Typechecker Using CLP (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. https://doi.org/10.1109/ase.2015.65

Michael D. Ernst. 2003. Static and dynamic analysis: Synergy and duality. In *WODA 2003: Workshop on Dynamic Analysis*. Portland, OR, USA, 24–27.

Zhenman Fang, Farnoosh Javadi, Jason Cong, and Glenn Reinman. 2019. Understanding Performance Gains of Accelerator-Rich Architectures. In *Proceedings of the 30th IEEE International Conference on Application-specific Systems, Architectures and Processors*

(ASAP '19). IEEE, New York, NY, USA, 239–246. https://doi.org/10.1109/ASAP.2019.00013

Burke Fetscher, Koen Claessen, Michał Pałka, John Hughes, and Robert Bruce Findler. 2015. Making Random Judgments: Automatically Generating Well-Typed Terms from the Definition of a Type-System. In *Programming Languages and Systems*, Jan Vitek (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 383–405.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. 2017. Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks. In *Proceedings of the 34th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Doina Precup and Yee Whye Teh (Eds.), Vol. 70. PMLR, International Convention Centre, Sydney, Australia, 1126–1135. http://proceedings.mlr.press/v70/finn17a.html

Michael Flanders, Steven Lyubomirsky, and Edward Misback. 2021. Constraint-Based Fuzzing for Deep Learning Applications. (2021). Unpublished class assignment (CSE 503 taught by René Just).

Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. 2018. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA '18)*. IEEE Press, New York, NY, USA, 1–14. https://doi.org/10.1109/ISCA.2018.00012

Taro Fujii, Takao Toi, Teruhito Tanaka, Katsumi Togawa, Toshiro Kitaoka, Kengo Nishino, Noritsugu Nakamura, Hiroki Nakahara, and Masato Motomura. 2018. New Generation Dynamically Reconfigurable Processor Technology for Accelerating Embedded AI Applications. In *Proceedings of the 2018 IEEE Symposium on VLSI Circuits*. IEEE, New York, NY, USA, 41–42. https://doi.org/10.1109/VLSIC.2018.8502438

Angelo Garofalo, Gianmarco Ottavi, Alfio Di Mauro, Francesco Conti, Giuseppe Tagliavini, Luca Benini, and Davide Rossi. 2021. A 1.15 TOPS/W, 16-Cores Parallel Ultra-Low Power Cluster with 2b-to-32b Fully Flexible Bit-Precision and Vector Lockstep Execution Mode. In *Proceedings of the 47th European Solid State Circuits Conference (ESSCIR 2021)*. IEEE, New York, NY, USA, 267–270. https://doi.org/10.1109/ESSCIRC53450.2021.9567767

Massimo Giordano, Kartik Prabhu, Kalhan Koul, Robert Radway, Albert Gural, Rohan Doshi, Zainab F. Khan, John W. Kustin, Timothy Liu, Gregorio B. Lopes, Victor Turbiner, Win-San Khwa, Yu-Der Chih, Meng-Fan Chang, Guénolé Lallement, Boris Murmann, Subhasish Mitra, and Priyanka Raina. 2021. CHIMERA: A 0.92 TOPS, 2.2 TOPS/W Edge AI Accelerator with 2 MByte On-Chip Foundry Resistive RAM for Efficient Training and Inference. In *Proceedings of the 2021 Symposium on VLSI Circuits*. IEEE, New York, NY, USA, 1–2. https://doi.org/10.23919/VLSICircuits52068.2021.9492347

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 213–223. https://doi.org/10.1145/1065010.1065036

Aidan N Gomez, Mengye Ren, Raquel Urtasun, and Roger B Grosse. 2017. The Reversible Residual Network: Backpropagation Without Storing Activations. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 2214–2224.

Alex Graves and Navdeep Jaitly. 2014. Towards End-To-End Speech Recognition with Recurrent Neural Networks. In *Proceedings of the 31st International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Eric P. Xing and Tony Jebara (Eds.), Vol. 32. PMLR, Bejing, China, 1764–1772. https://proceedings.mlr.press/v32/graves14.html

Andreas Griewank. 1994. Achieving Logarithmic Growth Of Temporal And Spatial Com-

plexity In Reverse Automatic Differentiation. *Optimization Methods and Software* 1 (04 1994). https://doi.org/10.1080/10556789208805505

Andreas Griewank and Andrea Walther. 1998. Treeverse: An Implementation of Checkpointing for the Reverse or Adjoint Mode of Computational Differentiation. (03 1998).

Andreas Griewank and Andrea Walther. 2000. Algorithm 799: Revolve: An Implementation of Checkpoint for the Reverse or Adjoint Mode of Computational Differentiation. *ACM Trans. Math. Software* 26, 1 (mar 2000), 19–45. http://doi.acm.org/10.1145/347837.347846

José Grimm, Loïc Pottier, and Nicole Rostaing-Schmidt. 1996. *Optimal Time and Minimum Space-Time Product for Reversing a Certain Class of Programs.* Technical Report RR-2794. INRIA. https://hal.inria.fr/inria-00073896

Audrunas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. 2016. Memory-Efficient Backpropagation Through Time. *CoRR* abs/1606.03401 (2016). arXiv:1606.03401 http://arxiv.org/abs/1606.03401

Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. 2015. The SeaHorn Verification Framework. In *Proceedings of the 27th International Conference on Computer Aided Verification (CAV 2015) (Lecture Notes in Computer Science)*, Daniel Kroening and Corina S. Pasareanu (Eds.), Vol. 9206. Springer, Berlin, Heidelberg, 343–361. https://doi.org/10.1007/978-3-319-21690-4_20

Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A High-Performance and Energy-Efficient Accelerator for Graph Analytics. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. IEEE Press, New York, NY, USA, Article 56, 13 pages.

Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. 2010. Under-

standing Sources of Inefficiency in General-Purpose Chips. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. Association for Computing Machinery, New York, NY, USA, 37–47. https://doi.org/10.1145/1815961.1815968

Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, New York, NY, USA, 243–254. https://doi.org/10.1109/ISCA.2016.30

Laurent Hascoet and Mauricio Araya-Polo. 2006. Enabling user-driven Checkpointing strategies in Reverse-mode Automatic Differentiation. arXiv:cs.DS/cs/0606042

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016a. Deep Residual Learning for Image Recognition. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, New York, NY, USA, 770–778. https://doi.org/10.1109/CVPR.2016.90

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016b. Identity Mappings in Deep Residual Networks. In *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part IV*. 630–645. https://doi.org/10.1007/978-3-319-46493-0_38

Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *In Proc. USENIX Security*. 445–458.

Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiv:cs.CV/1704.04861 http://arxiv.org/abs/1704.04861

Bo-Yuan Huang, Steven Lyubomirsky, Yi Li, Mike He, Thierry Tambe, Gus Henry Smith, Akash Gaonkar, Vishal Canumalla, Gu-Yeon Wei, Aarti Gupta, Zachary Tatlock, and Sharad Malik. 2022. Specialized Accelerators and Compiler Flows: Replacing Accelerator APIs with a Formal Software/Hardware Interface. arXiv:cs.AR/2203.00218

Bo-Yuan Huang, Hongce Zhang, Aarti Gupta, and Sharad Malik. 2019. ILAng: A Modeling and Verification Platform for SoCs Using Instruction-Level Abstractions. In *Tools and Algorithms for the Construction and Analysis of Systems*, Tomáš Vojnar and Lijun Zhang (Eds.). Springer International Publishing, Cham, 351–357.

Bo-Yuan Huang, Hongce Zhang, Pramod Subramanyan, Yakir Vizel, Aarti Gupta, and Sharad Malik. 2018a. Instruction-Level Abstraction (ILA): A Uniform Specification for System-on-Chip (SoC) Verification. *ACM Trans. Des. Autom. Electron. Syst.* 24, 1, Article Article 10 (Dec. 2018), 24 pages. https://doi.org/10.1145/3282444

Bo-Yuan Huang, Hongce Zhang, Pramod Subramanyan, Yakir Vizel, Aarti Gupta, and Sharad Malik. 2018b. Instruction-Level Abstraction (ILA): A Uniform Specification for System-on-Chip (SoC) Verification. *ACM Trans. Des. Autom. Electron. Syst.* 24, 1, Article 10 (Dec. 2018), 24 pages. https://doi.org/10.1145/3282444

Chien-Chin Huang, Gu Jin, and Jinyang Li. 2020. SwapAdvisor: Pushing Deep Learning Beyond the GPU Memory Limit via Smart Swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1341–1355. https://doi.org/10.1145/3373376.3378530

Mike Innes. 2018. Flux: Elegant Machine Learning with Julia. *Journal of Open Source Software* (2018). https://doi.org/10.21105/joss.00602

Mike Innes, David Barber, Tim Besard, James Bradburyand Valentin Churavy, Simon Danisch, Alan Edelman, Stefan Karpinski, Jon Malmaud, Jarrett Revels, Viral Shah, Pon-

tus Stenetorp, and Deniz Yuret. 2017. On Machine Learning and Programming Languages. https://julialang.org/blog/2017/12/ml&pl.

Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Kurt Keutzer, Ion Stoica, and Joseph E. Gonzalez. 2019. Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization. arXiv:cs.LG/1910.02653

Ranjit Jhala and Kenneth L. McMillan. 2001. Microarchitecture Verification by Compositional Model Checking. In *Computer Aided Verification*, Gérard Berry, Hubert Comon, and Alain Finkel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 396–410.

Tianyu Jia, Yuhao Ju, and Jie Gu. 2020. 31.3 A Compute-Adaptive Elastic Clock-Chain Technique with Dynamic Timing Enhancement for 2D PE-Array-Based Accelerators. In *Proceedings of the 2020 IEEE International Solid- State Circuits Conference (ISSCC 2020)*. IEEE, New York, NY, USA, 482–484. https://doi.org/10.1109/ISSCC19947.2020.9063062

Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 47–62. https://doi.org/10.1145/3341301.3359630

Rajeev Joshi, Greg Nelson, and Yunhong Zhou. 2006. Denali: A Practical Algorithm for Generating Optimal Code. *ACM Trans. Program. Lang. Syst.* 28, 6 (Nov. 2006), 967–989. https://doi.org/10.1145/1186632.1186633

Norman P. Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. 2020. A Domain-Specific Supercomputer for Training Deep Neural Networks. *Commun. ACM* 63, 7 (jun 2020), 67–78. https://doi.org/10.1145/3360307

Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, Richard C. Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. *CoRR* abs/1704.04760 (2017). arXiv:1704.04760 `http://arxiv.org/abs/1704.04760`

Sukwon Kim and Vin Sharma. 2019. AWS launches open source Neo-AI project to accelerate ML deployments on edge devices. `https://aws.amazon.com/blogs/machine-learning/aws-launches-open-source-neo-ai-project-to-accelerate-ml-deployments-on-edge-devices/`

Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. 2021. Dynamic Tensor Rematerialization. In *International Conference on Learning Representations*. `https://openreview.net/forum?id=Vfs_2RnOD0H`

Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. 2020. Reformer: The Efficient Transformer. In *International Conference on Learning Representations*. `https://openreview.net/forum?id=rkgNKkHtvB`

Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2016. SMT-based model checking for

recursive programs. *Formal Methods Syst. Des.* 48, 3 (2016), 175–205. https://doi.org/10.1007/s10703-016-0249-4

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 1097–1105. http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. Association for Computing Machinery, New York, NY, USA, 179–191. https://doi.org/10.1145/2535838.2535841

Ravi Kumar, Manish Purohit, Zoya Svitkina, Erik Vee, and Joshua Wang. 2019. Efficient Rematerialization for Deep Networks. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d Alché-Buc, E. Fox, and R. Garnett (Eds.). Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2019/file/ffe10334251de1dc98339d99ae4743ba-Paper.pdf

Mitsuru Kusumoto, Takuya K Inoue, Gentaro Watanabe, Takuya Akiba, and Masanori Koyama. 2019. A Graph Theoretic Framework of Recomputation Algorithms for Memory-Efficient Backpropagation. In *NeurIPS*.

Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. 2019. HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '19)*. Association for Computing Machinery, New York, NY, USA, 242–251. https://doi.org/10.1145/3289602.3293910

Yi-Hsiang Lai, Ecenur Ustun, Shaojie Xiang, Zhenman Fang, Hongbo Rong, and Zhiru Zhang. 2021. Programming and Synthesis for Software-Defined FPGA Acceleration: Status and Future Prospects. *ACM Trans. Reconfigurable Technol. Syst.* 14, 4, Article 17 (Sept. 2021), 39 pages. https://doi.org/10.1145/3469660

Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques A. Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO '21)*, Jae W. Lee, Mary Lou Soffa, and Ayal Zaks (Eds.). IEEE, New York, NY, USA, 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

Yann LeCun. 2018. Untitled. Retrieved Mar. 2, 2020 from https://www.facebook.com/yann.lecun/posts/10155003011462143

Caroline Lemieux and Koushik Sen. 2018. FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Sep 2018). https://doi.org/10.1145/3238147.3238176

Xavier Leroy. 2006. Formal Certification of a Compiler Back-End or: Programming a Compiler with a Proof Assistant. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*. Association for Computing Machinery, New York, NY, USA, 42–54. https://doi.org/10.1145/1111037.1111042

Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. 2018. Fuzzing: State of the Art. *IEEE Transactions on Reliability* 67, 3 (Sept. 2018), 1199–1218. https://doi.org/10.1109/tr.2018.2834476

Amanda Liu, Gilbert Louis Bernstein, Adam Chlipala, and Jonathan Ragan-Kelley. 2022a. Verified Tensor-Program Optimization via High-Level Scheduling Rewrites. *Proc. ACM Program. Lang.* 6, POPL, Article 55 (jan 2022), 28 pages. https://doi.org/10.1145/3498717

Jiawei Liu, Yuxiang Wei, Sen Yang, Yinlin Deng, and Lingming Zhang. 2022b. Coverage-Guided Tensor Compiler Fuzzing with Joint IR-Pass Mutation. arXiv:cs.SE/2202.09947

Steven Lyubomirsky. 2022. [RFC] Type-Directed Relay Fuzzing Library. https://discuss.tvm.apache.org/t/rfc-type-directed-relay-fuzzing-library/12234

Steven Lyubomirsky, Michael Flanders, and Edward Misback. 2021. Fuzzing TVM Relay. https://www.youtube.com/watch?v=JRlqfFs_NMs TVMCon.

Panagiotis Manolios and Sudarshan K. Srinivasan. 2008. A Refinement-Based Compositional Reasoning Framework for Pipelined Machine Verification. *IEEE Trans. Very Large Scale Integr. Syst.* 16, 4 (2008), 353–364. https://doi.org/10.1109/TVLSI.2008.918120

Jiayuan Mao, Chuang Gan, Pushmeet Kohli, Joshua B. Tenenbaum, and Jiajun Wu. 2019. The Neuro-Symbolic Concept Learner: Interpreting Scenes, Words, and Sentences From Natural Supervision. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rJgMlhRctm

Henry Massalin. 1987. Superoptimizer: A Look at the Smallest Program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems (ASPLOS II)*. IEEE Computer Society Press, Washington, DC, USA, 122–126. https://doi.org/10.1145/36206.36194

MegEngine 2021. Reduce GPU memory usage by Dynamic Tensor Rematerialization. https://github.com/MegEngine/MegEngine/wiki/Reduce-GPU-memory-usage-by-Dynamic-Tensor-Rematerialization accessed Apr. 6, 2022.

Erik Meijer. 2018. Behind Every Great Deep Learning Framework is an Even Greater Programming Languages Concept (Keynote). In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 1. https://doi.org/10.1145/3236024.3280855

David Menendez and Santosh Nagarakatte. 2017. Alive-Infer: Data-Driven Precondition Inference for Peephole Optimizations in LLVM. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 49–63. https://doi.org/10.1145/3062341.3062372

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer Sentinel Mixture Models. arXiv:cs.CL/1609.07843

Robin Milner, Mads Tofte, and David Macqueen. 1997. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA.

Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Yan, Lianmin Zheng, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2019. A Hardware–Software Blueprint for Flexible Deep Learning Specialization. *IEEE Micro* 39, 5 (2019), 8–16. https://doi.org/10.1109/MM.2019.2928962

Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. 2020. Synthesizing Structured CAD Models with Equality Saturation and Inverse Transformations. In *Proceedings of the 41st ACM SIGPLAN*

Conference on Programming Language Design and Implementation (PLDI 2020). Association for Computing Machinery, New York, NY, USA, 31–44. https://doi.org/10.1145/3385412.3386012

Chandrakana Nandi, Max Willsey, Amy Zhu, Yisu Remy Wang, Brett Saiki, Adam Anderson, Adriana Schulz, Dan Grossman, and Zachary Tatlock. 2021. Rewrite Rule Inference Using Equality Saturation. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 119 (oct 2021), 28 pages. https://doi.org/10.1145/3485496

Pandu Nayak. 2019. Understanding searches better than ever before. https://blog.google/products/search/search-language-understanding-bert/ Accessed Apr. 4, 2022.

Greg Nelson. 1981. *Techniques for Program Verification.* Ph.D. Dissertation. University of California at Berkeley.

Greg Nelson and Derek C. Oppen. 1980. Fast Decision Procedures Based on Congruence Closure. *J. ACM* 27, 2 (April 1980), 356–364. https://doi.org/10.1145/322186.322198

Julie L. Newcomb, Andrew Adams, Steven Johnson, Rastislav Bodik, and Shoaib Kamil. 2020. Verifying and Improving Halide's Term Rewriting System with Program Synthesis. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 166 (Nov. 2020), 28 pages. https://doi.org/10.1145/3428234

Robert Nieuwenhuis and Albert Oliveras. 2005. Proof-Producing Congruence Closure. In *Proceedings of the 16th International Conference on Term Rewriting and Applications (RTA 2005) (Lecture Notes in Computer Science)*, Jürgen Giesl (Ed.), Vol. 3467. Springer, Berlin, Heidelberg, 453–468. https://doi.org/10.1007/978-3-540-32033-3_33

R. Nikhil. 2004. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.* 69–70. https://doi.org/10.1109/MEMCOD.2004.1459818

Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. 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)*. Association for Computing Machinery, New York, NY, USA, 883–898. https://doi.org/10.1145/3453483.3454083

Christopher Olah. 2015a. Calculus on Computational Graphs: Backpropagation. Retrieved Mar. 2, 2020 from http://colah.github.io/posts/2015-08-Backprop/

Christopher Olah. 2015b. Neural Networks, Types, and Functional Programming. Retrieved Mar. 2, 2020 from http://colah.github.io/posts/2015-09-NN-Types-FP/

ONNX 2019. ONNX: Open Neural Network Exchange. Retrieved Apr. 21, 2021 from https://onnx.ai/

Jun-Seok Park, Jun-Woo Jang, Heonsoo Lee, Dongwoo Lee, Sehwan Lee, Hanwoong Jung, Seungwon Lee, Suknam Kwon, Kyung-Ah Jeong, Joon-Ho Song, SukHwan Lim, and Inyup Kang. 2021. 9.5 A 6K-MAC Feature-Map-Sparsity-Aware Neural Processing Unit in 5nm Flagship Mobile SoC. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC 2021)*. IEEE, New York, NY, USA, 152–154. https://doi.org/10.1109/ISSCC42613.2021.9365928

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017). https://openreview.net/pdf?id=BJJsrmfCZ

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019a. PyTorch: An Imperative Style, High-Performance Deep Learning Library. arXiv:cs.LG/1912.01703

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019b. PyTorch: An Imperative Style, High-Performance Deep Learning Library. arXiv:cs.LG/1912.01703 https://arxiv.org/abs/1912.01703

David Patterson, Joseph Gonzalez, Urs Hölzle, Quoc Hung Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeffrey Dean. 2022. The Carbon Footprint of Machine Learning Training Will Plateau, Then Shrink. (3 2022). https://doi.org/10.36227/techrxiv.19139645.v3

Barak A. Pearlmutter and Jeffrey Mark Siskind. 2008. Reverse-mode AD in a Functional Framework: Lambda the Ultimate Backpropagator. *ACM Trans. Program. Lang. Syst.* 30, 2, Article 7 (March 2008), 36 pages. https://doi.org/10.1145/1330017.1330018

Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. 2020. Capuchin: Tensor-Based GPU Memory Management for Deep Learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 891–905. https://doi.org/10.1145/3373376.3378505

Clément Pit-Claudel, Thomas Bourgeat, Stella Lau, Arvind, and Adam Chlipala. 2021. Effective Simulation and Debugging for a High-Level Hardware Language Using Software Compilers. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 789–803. https://doi.org/10.1145/3445814.3446720

PyTorch 2020. Word-level language modeling RNN. Retrieved Nov. 18, 2021 from https://github.com/pytorch/examples/tree/master/word_language_model

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. 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)*. ACM, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: Memory optimizations Toward Training Trillion Parameter Models. *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis* (Nov 2020). https://doi.org/10.1109/sc41405.2020.00024

Norman Ramsey and João Dias. 2011. Resourceable, Retargetable, Modular Instruction Selection Using a Machine-Independent, Type-Based Tiling of Low-Level Intermediate Code. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. Association for Computing Machinery, New York, NY, USA, 575–586. https://doi.org/10.1145/1926385.1926451

Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. 2016. Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, New York, NY, USA, 267–278. https://doi.org/10.1109/ISCA.2016.32

Jared Roesch, Steven Lyubomirsky, Marisa Kirisame, Josh Pollock, Logan Weber, Ziheng Jiang, Tianqi Chen, Thierry Moreau, and Zachary Tatlock. 2019. Relay: A High-Level IR for Deep Learning. *CoRR* abs/1904.08368 (2019). arXiv:1904.08368 http://arxiv.org/abs/1904.08368

Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. 2018. 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)*. Association for Computing Machinery, New York, NY, USA, 58–68. https://doi.org/10.1145/3211346.3211348

Jared G. Roesch. 2020. *Principled Optimization of Dynamic Neural Networks*. Ph.D. Dissertation. University of Washington.

Davide Rossi, Francesco Conti, Manuel Eggimann, Stefan Mach, Alfio Di Mauro, Marco Guermandi, Giuseppe Tagliavini, Antonio Pullini, Igor Loi, Jie Chen, Eric Flamand, and Luca Benini. 2021. 4.4 A 1.3TOPS/W @ 32GOPS Fully Integrated 10-Core SoC for IoT End-Nodes with 1.7μW Cognitive Wake-Up From MRAM-Based State-Retentive Sleep Mode. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC 2021)*. IEEE, New York, NY, USA, 60–62. https://doi.org/10.1109/ISSCC42613.2021.9365939

Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Satish Nadathur, Jakob Olesen, Jongsoo Park, Artem Rakhov, and Misha Smelyanskiy. 2018. Glow: Graph Lowering Compiler Techniques for Neural Networks. *CoRR* abs/1805.00907 (2018). arXiv:1805.00907 https://arxiv.org/abs/1805.00907

D. Saito, T. Kobayashi, Hiroki Koga, Nicolo Ronchi, K. Banerjee, Y. Shuto, Jun Okuno, K. Konishi, Luca Di Piazza, A. Mallik, Jan Van Houdt, M. Tsukamoto, K. Ohkuri, Taku Umebayashi, and Takayuki Ezaki. 2021. Analog In-memory Computing in FeFET-based 1T1R Array for Edge AI Applications. In *Proceedings of the 2021 Symposium on VLSI Circuits*. IEEE, New York, NY, USA, 1–2. https://doi.org/10.23919/VLSICircuits52068.2021.9492479

Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2019. MobileNetV2: Inverted Residuals and Linear Bottlenecks. arXiv:cs.CV/1801.04381

Colin Schmidt, John Charles Wright, Zhongkai Wang, Eric Chang, Albert J. Ou, Woo-Rham Bae, Sean Huang, Anita Flynn, Brian C. Richards, Krste Asanovic, Elad Alon, and Borivoje Nikolic. 2021. 4.3 An Eight-Core 1.44GHz RISC-V Vector Machine in 16nm FinFET. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC 2021)*. IEEE, New York, NY, USA, 58–60. https://doi.org/10.1109/ISSCC42613.2021.9365789

Aashaka Shah, Chao-Yuan Wu, Jayashree Mohan, Vijay Chidambaram, and Philipp Kraehenbuehl. 2021. Memory Optimization for Deep Networks. In *International Conference on Learning Representations*. https://openreview.net/forum?id=bnY0jm4l59

Haichen Shen, Jared Roesch, Zhi Chen, Wei Chen, Yong Wu, Mu Li, Vin Sharma, Zachary Tatlock, and Yida Wang. 2021. Nimble: Efficiently Compiling Dynamic Neural Networks for Model Inference. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 208–222. https://proceedings.mlsys.org/paper/2021/file/4e732ced3463d06de0ca9a15b6153677-Paper.pdf

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. 2017. Mastering the game of Go without human knowledge. *Nature* 550, 7676 (01 Oct 2017), 354–359. https://doi.org/10.1038/nature24270

Jeffrey Mark Siskind and Barak A. Pearlmutter. 2018. Divide-and-conquer checkpointing for arbitrary programs with no user annotation. *Optimization Methods and Software* 33, 4-6 (Sep 2018), 1288–1330. https://doi.org/10.1080/10556788.2018.1459621

Gus Henry Smith, Andrew Liu, Steven Lyubomirsky, Scott Davidson, Joseph McMahan, Michael Taylor, Luis Ceze, and Zachary Tatlock. 2021. Pure Tensor Program Rewriting via Access Patterns (Representation Pearl). In *Proceedings of the 5th ACM SIGPLAN Inter-*

national Symposium on Machine Programming (MAPS 2021). Association for Computing Machinery, New York, NY, USA, 21–31. https://doi.org/10.1145/3460945.3464953

Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, Elton Zhang, Rewon Child, Reza Yazdani Aminabadi, Julie Bernauer, Xia Song, Mohammad Shoeybi, Yuxiong He, Michael Houston, Saurabh Tiwary, and Bryan Catanzaro. 2022. Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model. arXiv:cs.CL/2201.11990

Nimit Sharad Sohoni, Christopher Richard Aberger, Megan Leszczynski, Jian Zhang, and Christopher Ré. 2019. Low-Memory Neural Network Training: A Technical Report. *CoRR* abs/1904.10631 (2019). arXiv:1904.10631 http://arxiv.org/abs/1904.10631

Bert Speelpenning. 1980. *Compiling Fast Partial Derivatives of Functions given by Algorithms*. Ph.D. Dissertation. USA. AAI8017989.

Pramod Subramanyan, Bo-Yuan Huang, Yakir Vizel, Aarti Gupta, and Sharad Malik. 2018. Template-Based Parameterized Synthesis of Uniform Instruction-Level Abstractions for SoC Verification. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 37, 8 (2018), 1692–1705. https://doi.org/10.1109/TCAD.2017.2764482

Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. *CoRR* abs/1503.00075 (2015). arXiv:1503.00075 http://arxiv.org/abs/1503.00075

Thierry Tambe, En-Yu Yang, Glenn G. Ko, Yuji Chai, Coleman Hooper, Marco Donato, Paul N. Whatmough, Alexander M. Rush, David Brooks, and Gu-Yeon Wei. 2021. 9.8 A 25mm^2 SoC for IoT Devices with 18ms Noise-Robust Speech-to-Text Latency via Bayesian Speech Denoising and Attention-Based Sequence-to-Sequence DNN Speech Recognition in 16nm FinFET. In *Proceedings of the IEEE International Solid-*

State Circuits Conference (ISSCC '21). IEEE, New York, NY, USA, 158–160. https://doi.org/10.1109/ISSCC42613.2021.9366062

Thierry Tambe, En-Yu Yang, Zishen Wan, Yuntian Deng, Vijay Janapa Reddi, Alexander Rush, David Brooks, and Gu-Yeon Wei. 2020. Algorithm-Hardware Co-Design of Adaptive Floating-Point Encodings for Resilient Deep Learning Inference. In *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference (DAC '20)*. IEEE Press, USA, Article 51, 6 pages.

Mingxing Tan and Quoc V. Le. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning (ICML '19)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.), Vol. 97. PMLR, Atlanta, Georgia, USA, 6105–6114. http://proceedings.mlr.press/v97/tan19a.html

Yu Tang, Chenyu Wang, Yufan Zhang, Yuliang Liu, Xingcheng Zhang, Linbo Qiao, Zhiquan Lai, and Dongsheng Li. 2022. DELTA: Dynamically Optimizing GPU Memory beyond Tensor Recomputation. arXiv:cs.LG/2203.15980

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2011. Equality Saturation: A New Approach to Optimization. *Logical Methods in Computer Science* Volume 7, Issue 1 (March 2011). https://doi.org/10.2168/LMCS-7(1:10)2011

Hugo Touvron, Piotr Bojanowski, Mathilde Caron, Matthieu Cord, Alaaeldin El-Nouby, Edouard Grave, Armand Joulin, Gabriel Synnaeve, Jakob Verbeek, and Hervé Jégou. 2021. ResMLP: Feedforward networks for image classification with data-efficient training. arXiv:cs.CV/2105.03404 https://arxiv.org/abs/2105.03404

Lenny Truong, Steven Herbst, Rajsekhar Setaluri, Makai Mann, Ross G. Daly, Keyi Zhang, Caleb Donovick, Daniel Stanley, Mark Horowitz, Clark W. Barrett, and Pat Hanrahan. 2020. fault: A Python Embedded Domain-Specific Language for Metaprogramming

Portable Hardware Verification Components. In *Proceedings of the 32nd International Conference on Computer Aided Verification (CAV 2020) (Lecture Notes in Computer Science)*, Shuvendu K. Lahiri and Chao Wang (Eds.), Vol. 12224. Springer, Berlin, Heidelberg, 403–414. https://doi.org/10.1007/978-3-030-53288-8_19

Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. 2021. *Vectorization for Digital Signal Processors via Equality Saturation.* Association for Computing Machinery, New York, NY, USA, 874–886. https://doi.org/10.1145/3445814.3446707

Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. arXiv:1802.04730 https://arxiv.org/abs/1802.04730

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. arXiv:cs.CL/1706.03762

Verilator n. d.. Verilator. https://www.veripool.org/verilator/ Accessed Nov. 18, 2021.

Sahil Verma and Zhendong Su. 2020. ShapeFlow: Dynamic Shape Interpreter for TensorFlow. arXiv:cs.LG/2011.13452

Fei Wang, Xilun Wu, Grégory M. Essertel, James M. Decker, and Tiark Rompf. 2018a. Demystifying Differentiable Programming: Shift/Reset the Penultimate Backpropagator. *CoRR* abs/1803.10228 (2018). arXiv:1803.10228 http://arxiv.org/abs/1803.10228

Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018b. Superneurons. *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Feb 2018). https://doi.org/10.1145/3178487.3178491

Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. 2020. SPORES. *Proceedings of the VLDB Endowment* 13, 12 (Aug 2020), 1919–1932. https://doi.org/10.14778/3407790.3407799

Richard Wei, Dan Zheng, Marc Rasi, and Bart Chrzaszcz. 2020. Differentiable Programming Manifesto. Retrieved Mar. 2, 2020 from https://github.com/apple/swift/blob/master/docs/DifferentiableProgramming.md

Xuechao Wei, Yun Liang, and Jason Cong. 2019. Overcoming Data Transfer Bottlenecks in FPGA-Based DNN Accelerators via Layer Conscious Memory Management. In *Proceedings of the 56th Annual Design Automation Conference 2019 (DAC '19)*. Association for Computing Machinery, New York, NY, USA, Article 125, 6 pages. https://doi.org/10.1145/3316781.3317875

Paul N. Whatmough, Sae Kyu Lee, Marco Donato, Hsea-Ching Hsueh, Sam Likun Xi, Udit Gupta, Lillian Pentecost, Glenn G. Ko, David M. Brooks, and Gu-Yeon Wei. 2019. A 16nm 25mm^2 SoC with a 54.5x Flexibility-Efficiency Range from Dual-Core Arm Cortex-A53 to eFPGA and Cache-Coherent Accelerators. In *Proceedings of the 2019 Symposium on VLSI Circuits*. IEEE, New York, NY, USA, 34. https://doi.org/10.23919/VLSIC.2019.8778002

Deborah L. Whitfield and Mary Lou Soffa. 1997. An Approach for Exploring Code Improving Transformations. *ACM Trans. Program. Lang. Syst.* 19, 6 (Nov. 1997), 1053–1084. https://doi.org/10.1145/267959.267960

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and Extensible Equality Saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434304

XilinxSDK n. d.. The Xilinx Software Development Kit (XSDK). Retrieved Apr. 24, 2021 from https://www.xilinx.com/products/design-tools/embedded-software/sdk.html

Yue Xing, Bo-Yuan Huang, Aarti Gupta, and Sharad Malik. 2018. A Formal Instruction-Level GPU Model for Scalable Verification. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD '18)*. Association for Computing Machinery, New York, NY, USA, Article 130, 8 pages. https://doi.org/10.1145/3240765.3240771

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. *SIGPLAN Not.* 46, 6 (jun 2011), 283–294. https://doi.org/10.1145/1993316.1993532

Yichen Yang, Phitchaya Mangpo Phothilimtha, Yisu Remy Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. 2021. Equality Saturation for Tensor Graph Superoptimization. arXiv:cs.AI/2101.01332

Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-x: An Accelerator for Sparse Neural Networks. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. IEEE Press, New York, NY, USA, Article 20, 12 pages. https://doi.org/10.1109/MICRO.2016.7783723

Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 863–879. https://www.usenix.org/conference/osdi20/presentation/zheng