

Deep Learning for Compilers

Chris Cummins



Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2020

Abstract

Constructing compilers is hard. Optimising compilers are multi-million dollar projects spanning years of development, yet remain unable to fully exploit the available performance, and are prone to bugs. The rapid transition to heterogeneous parallelism and diverse architectures has raised demand for aggressively-optimising compilers to an all time high, leaving compiler developers struggling to keep up. What is needed are better tools to simplify compiler construction.

This thesis presents new techniques that dramatically lower the cost of compiler construction, while improving robustness and performance. The enabling insight for this research is the leveraging of *deep learning* to model the correlations between source code and program behaviour, enabling tasks which previously required significant engineering effort to be automated. This is demonstrated in three domains:

First, a generative model for compiler benchmarks is developed. The model requires no prior knowledge of programming languages, yet produces output of such quality that professional software developers cannot distinguish generated from hand-written programs. The efficacy of the generator is demonstrated by supplementing the training data of predictive models for compiler optimisations. The generator yields an automatic improvement in heuristic performance, and exposes weaknesses in state-of-the-art approaches which, when corrected, yield further performance improvements.

Second, a compiler fuzzer is developed which is far simpler than prior techniques. By learning a generative model rather than engineering a generator from scratch, it is implemented in $100\times$ fewer lines of code than the state-of-the-art, yet is capable of exposing bugs which prior techniques cannot. An extensive testing campaign reveals 67 new bugs in OpenCL compilers, many of which have now been fixed.

Finally, this thesis addresses the challenge of feature design. A methodology for learning compiler heuristics is presented that, in contrast to prior approaches, learns directly over the raw textual representation of programs. The approach outperforms state-of-the-art models with hand-engineered features in two challenging optimisation domains, without requiring any expert guidance. Additionally, the methodology enables models trained in one task to be adapted to perform another, permitting the novel transfer of information between optimisation problem domains.

The techniques developed in these three contrasting domains demonstrate the exciting potential of deep learning to simplify and improve compiler construction. The outcomes of this thesis enable new lines of research to equip compiler developers to keep up with the rapidly evolving landscape of heterogeneous architectures.

Acknowledgements

It has been my utmost pleasure and good fortune to have Hugh Leather and Pavlos Petoumenos as my advisers for the past four years. They have made my time as a PhD candidate a deeply gratifying trajectory of learning, curiosity, and occasional hard work. Thank you.

I would like also to thank Zheng Wang, with whom I have collaborated on projects which form parts of this thesis. Thank you to Michel Steuwer for providing unapologetically thorough feedback on select chapters of this thesis, improving them markedly.

I would like to thank my friends and colleagues at the University of Edinburgh, Google, DeepMind, and Codeplay Software whose company I have had the pleasure of keeping. Finally, I would like to thank my dear Ashleigh, who makes anything and everything possible. Hello to Jason Isaacs.

Declaration

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

- Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather “Synthesizing Benchmarks for Predictive Modeling”. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2017.
- Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather “End-to-end Deep Learning of Optimization Heuristics”. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017.
- Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather “Compiler Fuzzing through Deep Learning”. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2018.

(Chris Cummins)

Table of Contents

1	Introduction	1
1.1	Machine Learning for Compilers	2
1.2	Challenges in Machine Learning for Compilers	4
1.2.1	Scarcity of Data	4
1.2.2	Model and Feature Design	4
1.2.3	Adoption of Machine Learning Practices	5
1.3	Contributions	6
1.4	Publications	7
1.5	Structure	7
1.6	Summary	8
2	Background	9
2.1	Introduction	9
2.2	Machine Learning	9
2.2.1	Feed-forward Neural Networks	10
2.2.2	Recurrent Neural Networks	13
2.2.3	Decision Trees	16
2.3	Model Evaluation Techniques	16
2.3.1	ZeroR	17
2.3.2	Training, Validation, Test Data	17
2.3.3	K-Fold Cross-validation	17
2.3.4	Principal Component Analysis	18
2.4	Summary	19
3	Related Work	21
3.1	Introduction	21
3.2	Program Generation	21

3.2.1	Benchmark Generation for Performance Characterisation	21
3.2.2	Test Case Generation for Compiler Validation	24
3.3	Program Optimisation	30
3.3.1	Iterative Compilation and Auto-tuning	31
3.3.2	Machine Learning for Compiler Optimisations	35
3.4	Deep Learning over Programs	39
3.5	Summary	41
4	Improving the Performance of Predictive Models for Compiler Heuristics	43
4.1	Introduction	43
4.2	The Case for Generating Benchmarks	46
4.3	CLgen: A System for Generating OpenCL Benchmarks	48
4.3.1	Overview	50
4.3.2	An OpenCL Language Corpus	50
4.3.3	Learning OpenCL	55
4.3.4	Synthesising Source Code	55
4.4	CLdrive: A System for Driving Arbitrary OpenCL Kernels	56
4.4.1	Generating Data Payloads	56
4.4.2	Dynamic Checker	59
4.5	Qualitative Evaluation of Generated Programs	60
4.5.1	Methodology	60
4.5.2	Experimental Results	60
4.6	Experimental Methodology	61
4.6.1	Experimental Setup	61
4.6.2	Methodology	63
4.7	Experimental Results	64
4.7.1	Performance Evaluation	64
4.7.2	Extending the Predictive Model	64
4.7.3	Comparison of Source Features	68
4.8	Summary	71
5	Lowering the Cost of Compiler Validation	73
5.1	Introduction	73
5.2	DeepSmith: Compiler Fuzzing Through Deep Learning	75
5.2.1	Generative Model	75
5.2.2	Test Harness	79

5.2.3	Voting Heuristics for Differential Testing	80
5.3	Experimental Setup	83
5.3.1	OpenCL Systems	83
5.3.2	Testbeds	83
5.3.3	Test Cases	83
5.3.4	Bug Search Time Allowance	83
5.4	Evaluation	85
5.4.1	Compile-time Defects	85
5.4.2	Runtime Defects	92
5.4.3	Comparison to State-of-the-art	93
5.4.4	Compiler Stability Over Time	100
5.4.5	Extensibility of Language Model	101
5.5	Summary	103
6	Simplifying the Construction of Optimisation Heuristics	105
6.1	Introduction	105
6.2	DeepTune: Learning On Raw Program Code	108
6.2.1	Overview	108
6.2.2	Language Model	108
6.2.3	Auxiliary Inputs	111
6.2.4	Heuristic Model	113
6.2.5	Training the Network	113
6.3	Case Study A: OpenCL Heterogeneous Mapping	114
6.3.1	State-of-the-art	114
6.3.2	Experimental Setup	116
6.3.3	Experimental Results	117
6.4	Case Study B: OpenCL Thread Coarsening Factor	118
6.4.1	State-of-the-art	118
6.4.2	Experimental Setup	123
6.4.3	Comparison to Case Study A	124
6.4.4	Experimental Results	125
6.5	Transfer Learning Across Problem Domains	125
6.6	DeepTune Internal Activation States	128
6.7	Summary	130

7 Conclusions	133
7.1 Contributions	133
7.1.1 A Solution for Benchmark Scarcity	133
7.1.2 Low-cost and Effective Compiler Fuzzing	134
7.1.3 Automatic Compiler Optimisation Tuning	135
7.2 Critical Analysis	136
7.2.1 Generative Models for Source Code	136
7.2.2 Rejection Sampling for Program Generation	137
7.2.3 Characterisation of OpenCL Compiler Bugs	137
7.2.4 Driving arbitrary OpenCL kernels	138
7.2.5 Modelling Program Semantics from Syntactic Sequences	138
7.3 Future Work	139
7.3.1 Guided Program Synthesis to Minimise Benchmarking Cost	139
7.3.2 Neural Model Selection through Adversarial Games	140
7.3.3 Learning Representations for Dynamic Program Inputs	140
7.3.4 Towards General-Purpose Program Comprehension	141
7.4 Summary	141
Bibliography	143

List of Figures

2.1	Training and inference in machine learning	10
2.2	Structure of an artificial neural network	11
2.3	Activation functions for artificial neural networks	12
2.4	Recurrent Neural Network architecture	14
2.5	Long Short-Term Memory cell architecture	15
3.1	Generating and evaluating compiler test cases	26
4.1	Training a predictive model for compiler optimisations	44
4.2	Benchmark counts in GPGPU research papers	47
4.3	Identifying and correcting outliers in a benchmark suite	49
4.4	Benchmark synthesis and execution pipeline	51
4.5	Speedup of predictions with and without synthetic benchmarks	65
4.6	Speedups of predictions using extended model over state-of-the-art . .	69
4.7	Number of kernels matching benchmark features	70
5.1	DeepSmith system overview	76
5.2	Test case execution, and possible results	81
5.3	Example codes which crash parsers	86
5.4	Example OpenCL kernels which crash compilers	87
5.5	Example kernels which hang compilers	89
5.6	Example kernels which are miscompiled	90
5.7	Further example kernels which are miscompiled	91
5.8	Comparison of DeepSmith and CLSmith runtimes	96
5.9	Example kernels which crash Intel compiler passes	98
5.10	Further example kernels which crash Intel compiler passes	99
5.11	Kernels which expose errors exposed by CLSmith and DeepSmith . .	100
5.12	Crash rate of the Clang front-end	101

6.1	Using machine learning for compiler optimisations	106
6.2	DeepTune system overview	109
6.3	Deriving a vocabulary encoding from source code	112
6.4	DeepTune artificial neural networks	117
6.5	Accuracy of optimisation heuristics for heterogeneous device mapping	119
6.6	Speedup of predicted heterogeneous mappings	120
6.7	Predicting OpenCL thread coarsening factors.	121
6.8	Speedups of predicted thread coarsening factors on AMD	126
6.9	Speedups of predicted thread coarsening factors on NVIDIA	127
6.10	Visualising the internal state of DeepTune	129

List of Tables

1.1	Development history and costs of popular open-source compilers	2
4.1	Cross-validation of benchmark suites on a predictive model	47
4.2	Grawe, Wang, and O'Boyle features for heterogeneous device mapping	62
4.3	Benchmarks used in evaluation	62
4.4	Experimental platforms used in evaluation	63
5.1	OpenCL systems and the number of bug reports submitted to date . .	84
5.2	Results from 48 hours of testing using CLSmith	94
5.3	Results from 48 hours of testing using DeepSmith	95
5.4	Number of DeepSmith programs which trigger errors	102
5.5	DeepSmith programs which trigger Solidity compiler errors	102
6.1	Heterogeneous mapping model features	115
6.2	Benchmarks used in Case Study A	116
6.3	Benchmarks used in Case Study A	116
6.4	Magni, Dubach, and O'Boyle features for predicting thread coarsening	122
6.5	Benchmarks used in Case Study B	123
6.6	Experimental platforms used in Case Study B	123
6.7	DeepTune model parameters	124

List of Listings

1	The <i>shim</i> header file for compiling OpenCL from GitHub	53
2	Example OpenCL content file from GitHub	54
3	OpenCL content file after rewriting	54
4	Synthesised vector operation with branching and synchronisation . .	57
5	Synthesised zip operation	57
6	Synthesised partial reduction operation	58
7	AMD’s Fast Walsh Transform kernel	67
8	Synthesised program with same features as an AMD benchmark . . .	68

Chapter 1

Introduction

There has been an unprecedented increase in the scale and quantity of data-intensive workloads, with the energy consumption of the ICT industry estimated to rise to 20% of global output by 2030 [And19]. Fundamental shifts in both hardware and software are required to meet the demands of the transition to *big data* [GH15; Has+15].

Compilers play an essential role in extracting software efficiency by transforming the input code to a set of machine instructions that best utilises the available resources of the target architecture. Modern compilers are multi-million dollar projects comprising millions of lines of code from hundreds of developers, shown in Table 1.1. Each architecture supported by a compiler requires extensive hand tuning by experts to extract good performance. Modern compilers are too complex to be fully understood by a single developer, yet optimising a single component requires knowledge of the full compiler and the interaction of its components.

Due to their high cost of development, compilers are unable to adapt to the rapidly changing hardware landscape arising from GPUs, FPGAs [Con+16; MS10], and ASICs [Jou+17; MS10]. When compilers are unable to keep up with the pace of change, this leads to wasted energy, slow performance, and buggy software. For the trend towards data-intensive workloads and heterogeneous devices to continue, new techniques are required to reduce the cost of compiler construction. This thesis aims to address the widening gap between the potential and achieved efficiency of software by developing new tools to simplify and improve compiler construction.

This thesis demonstrates the use of machine learning for lowering the cost of compiler construction. The remainder of this chapter describes the application of machine learning to compilers, followed by its challenges. Then the contributions of this thesis are detailed, followed by a description of the overall structure of the document.

Project	Started	Developers	LOC	Estimated Cost
FreePascal	2005	54	3,845,685	\$198,619,187
GCC 9.2.0	1988	617	5,591,759	\$425,747,278
Glasgow Haskell Compiler 8.8.1	2001	1,119	761,097	\$52,449,098
Intel Graphics Compiler 1.0.10	2018	149	684,688	\$46,934,626
LLVM 8.0.1	2001	1,210	6,887,489	\$529,895,190
OpenJDK 14+10	2007	883	7,955,827	\$616,517,786
Roslyn .NET 16.2	2014	496	2,705,092	\$198,619,187
Rust 1.37.0	2010	2,737	852,877	\$59,109,425
Swift	2010	857	665,238	\$45,535,689
v8 7.8.112	2008	736	3,048,793	\$225,195,832

Table 1.1: Development history, logical lines of code (LOC), and estimated cost of 10 popular open-source compiler projects. Estimated costs are calculated using a CO-COMO model [Dav01] with average 2019 US software developer salaries [Gla19].

1.1 Machine Learning for Compilers

Machine learning, the study of algorithms and systems capable of learning from data without being explicitly programmed, has been successfully applied across a broad range of fields and disciplines. Within compilers, there are many tasks for which machine learning may prove useful.

A common case where machine learning aids in compiler construction is in the labour-intensive process of optimisation heuristic construction. For example, suppose a developer is tasked with tuning the loop unrolling heuristic of a compiler¹. There is a multitude of factors that may influence the decision of whether to unroll a loop such as the size of the loop body, the number of loop iterations, and the number of registers required to execute the loop body. Determining which of these factors are most significant, and on what values the heuristic should differ, is an unwieldy task. The problem is worsened since the quality of the heuristic depends not just on the program being compiled, but on properties of the target hardware, such as the number of registers

¹Loop unrolling is a code transformation in which the body of a loop is duplicated so that fewer iterations of the loop must be executed. The idea is to reduce runtime by executing fewer loop control instructions, at the expense of increasing the size of the executable binary.

and the size of the instruction cache. In the face of these challenges, developing good heuristics is a huge undertaking, and it is unlikely that a developer will be able to craft a heuristic capable of extracting the best performance in all situations.

Instead, a developer may cast construction of the loop unrolling heuristic as a machine learning problem. Rather than expertly crafting a heuristic through intuition and manual experimentation, the idea is to use a learning algorithm to derive a heuristic from empirical data of the performance of loops under different configurations of unrolling. To do this, the developer profiles suites of benchmark programs repeatedly using different unrolling decisions to determine the best decision for each case, then combines this with a numerical representation of each program. A machine learning system then models the correlations between these numerical program representations — *features* — and the unrolling decisions that should be made. Using machine learning for this task reduces the need for domain expertise compared to the expert-driven approach, requiring less effort from the developer. Unlike expert-driven approaches, the system can easily be adapted to new architectures and changes in the compiler simply by repeating the data collection process and constructing a new model.

The appeal of machine learning is that it provides techniques to automatically understand the structure of data and how that structure relates to a specific goal. For unseen data which has similar structures to the training data, this enables accurate predictions to be made; all without the need for expert domain knowledge. In essence, machine learning negates the need for problem-domain expertise in cases where there is a ready supply of empirical observations.

The applicability of machine learning to a wide range of tasks in compiler construction has led to an established research direction. In previous studies machine learning has been shown to simplify the construction of compiler optimisations [Ash+18; WO18], often leading to higher quality heuristics that outperform those constructed by human experts. With the increasing demand for aggressively-optimising compilers across a range of heterogeneous hardware, it would appear that machine learning could provide a much-needed relief on the burden of compiler developers.

Yet, the integration of machine learning and compilers has remained a largely academic pursuit, with little progress towards adoption by industry. The following section summarises three key outstanding problems in applying machine learning to compilers.

1.2 Challenges in Machine Learning for Compilers

Machine learning techniques offer reduced compiler development costs and improved runtime performance compared to expert approaches, yet there are three significant challenges that must be overcome to realise these goals:

1.2.1 Scarcity of Data

In machine learning, a model is trained based on past observations to predict the values for future inputs. In order to be able to generalise well to unseen observations, plentiful training data must be provided, with a fine-grained overview of the feature space. In the case of compilers, training data is derived from benchmark programs, meaning that many benchmarks are needed to produce sufficient observations for training. Typical machine learning experiments outside of the compilation field train over thousands or millions of observations. However, there are typically only a few dozen common benchmarks available.

The small number of available benchmarks limits the quality of learned models as they have sparse training data. The problem is worsened by the exponential increase in feature space size with the addition of new features. Each additional feature makes the sparsity of training examples more pronounced, increasing the number of observations required.

To address this issue, there must be a sizeable increase in the availability of benchmarks for machine learning. Previously, researchers sought to provide this by randomly instantiating from hand-crafted benchmark templates [CGA15], but this is a challenging approach — the generator must be biased in such a way that the generated programs draw from a similar distribution to *real* programs so as to be useful for learning. It is not clear if such an approach could ever achieve parity with handwritten programs.

1.2.2 Model and Feature Design

Machine learning algorithms learn to correlate a set of explanatory variables, known as *features*, with a target value. Learning the correlation between features and target value requires that features must be chosen so as to be discriminative for the target value. Features that are not discriminative will prevent a machine learning system from being able to achieve the desired task.

Choosing a set of features to characterise a program so as to be discriminative for machine learning depends on the desired task to be learned, the type of model, and the environment from which training data was collected, e.g. the hardware and machine configuration. Many problems in compilers do not map directly to numeric attributes, so systems for extracting numeric representations from non-numeric inputs must be developed. For example, one might extract instruction counts from the input source code. Knowing which attributes to extract to represent a program is a hard problem; there is no one-size-fits-all approach that works for all cases. Further, features may only be rendered suitable for learning by transforming the initial values, such as through scaling or normalising. As a result, feature design is often an incremental process of trial and experimentation, and there are few clear signals when this iterative process is “done”.

1.2.3 Adoption of Machine Learning Practices

Machine learning systems are fundamentally probabilistic. This stands at odds with established compiler construction practice, which uses formal algorithms and systems to minimise the risk of errors and provide correctness. Given the high cost of developing compilers (Table 1.1), there is a significant effort and expertise invested in existing approaches. For the benefits of machine learning techniques for compilers to be realised, machine learning must provide significant benefits over existing approaches, without degrading the quality of the compiler.

Techniques must be developed for probabilistic machine learning systems to coexist with formal approaches. One approach is to limit machine learning only to use cases where correctness guarantees will not be violated, such as in replacing hand-crafted optimisation heuristics. This mitigates the risk of probabilistic systems introducing errors, but limits the applicability of the technique to only certain parts of the compiler. For other use cases with more stringent requirements for behaviour, for example when generating inputs to test compilers, techniques must be developed that enable probabilistic methods to be used without introducing errors or invalid behaviour.

For machine learning techniques to be widely adopted in compilers, they must be made significantly easier and cheaper to develop. The aim of this thesis is to research and develop machine learning techniques that simplify and lower the cost of compiler construction.

1.3 Contributions

This thesis presents machine learning-based techniques to simplify and accelerate compiler construction. The key contributions of this thesis are:

- The first application of deep learning over source codes to synthesise compilable, executable benchmarks. The proposed approach automatically enhances the predictive power of a state-of-the-art predictive model, improving the performance of heterogeneous workloads by $1.27\times$. Further, the additional benchmarks expose limitations in the feature design of the model which, after correcting, further increases performance by $4.30\times$. This addresses the scarcity of data challenge (Section 1.2.1) by enabling the generation of an unbounded quantity of training data for use in machine learning models.
- A novel, automatic, and fast approach for the generation of expressive random programs for compiler fuzzing. The proposed system *infers* programming language syntax, structure, and semantics from real-world examples, not through an expert-defined grammar. The system needs two orders of magnitude less code than the state-of-the-art. In modelling real handwritten code, the test cases are more interpretable than other approaches; the average test case size is two orders of magnitude smaller than the state-of-the-art. An extensive testing campaign reveals 67 new bugs in commercial and open source OpenCL compilers, exposing many bugs which prior work cannot, by covering more components of the compiler. This addresses the machine learning adoption challenge (Section 1.2.3) by demonstrating a far simpler method for developing compiler fuzzers, and proposing fast heuristic techniques that enable probabilistic machine learning systems in a domain which previously required formal approaches.
- A methodology for building compiler heuristics without the need for program feature engineering. In an evaluation of the technique it is found to outperform existing prior state-of-the-art predictive models by 14% and 12% in two challenging GPGPU compiler optimisation domains. The proposed approach enables the first application of *transfer learning* to compiler optimisations, improving heuristics by reusing training information across different optimisation problems, even if they are unrelated. This addresses the model and feature design challenge (Section 1.2.2) by circumventing the need for numeric representations of code, and enables a single model design to be applied to multiple problems.

1.4 Publications

This thesis is in part based on ideas and results which have been described in previous publications. The system for generating benchmarks which provides the basis of Chapter 4 is described in:

1. Cummins, C., Petoumenos, P., Zang, W., & Leather, H. (2017). *Synthesizing Benchmarks for Predictive Modeling*. In CGO. IEEE. [Cum+17b].

Chapter 5 describes an approach to compiler testing which was previously published in:

2. Cummins, C., Petoumenos, P., Murray, A., & Leather, H. (2018). *Compiler Fuzzing through Deep Learning*. In ISSTA. [Cum+17b].
3. Cummins, C., Petoumenos, P., Murray, A., & Leather, H. (2018). *DeepSmith: Compiler Fuzzing through Deep Learning*. In ACACES. [Cum+18]

The *DeepTune* system proposed in Chapter 6 was first published in:

4. Cummins, C., Petoumenos, P., Wang, Z., & Leather, H. (2017). *End-to-end Deep Learning of Optimization Heuristics*. In PACT. IEEE. [Cum+17a].

The experimental results in this thesis are reproductions of those in the above publications. What differentiates this work from prior publications is the addition of background material (Chapter 2) and a literature review (Chapter 3) which offers a more comprehensive overview of the relevant fields, and includes references to new works that have been published after the above publications, including works based on those publications.

1.5 Structure

This thesis is organised as follows:

Chapter 2 provides background. It defines terminology and describes the machine learning and evaluation techniques used in this work.

Chapter 3 surveys the relevant literature, divided into three categories: first program generation, then program optimisation, finally deep learning for programming languages.

Chapter 4 describes a novel technique for generating an unbounded number of executable benchmarks to augment the training data of a predictive model. A qualitative evaluation of the generated programs is presented, followed by a quantitative evaluation using a state-of-the-art OpenCL optimisation heuristic.

Chapter 5 extends the generator presented in Chapter 4 to the domain of compiler validation, presenting a low-cost technique for the inference of compiler fuzzers. Fast heuristic approaches are presented for false-positive prevention using probabilistic test case generators. An extensive testing campaign of OpenCL compilers is described, resulting in 67 bug reports.

Chapter 6 introduces a novel methodology for constructing optimising compiler heuristics without the need for code features. It presents two case studies of the technique: the first for learning a heterogeneous device mapping heuristic, the second for learning OpenCL thread coarsening.

Chapter 7 summarises the overall findings of the thesis, provides a critical review, and outlines potential avenues for future research.

1.6 Summary

The promise of machine learning techniques is reduced compiler development cost and improved performance. Achieving these goals requires overcoming three significant challenges: the data scarcity challenge, the feature design challenge, and the adoption challenge. The next two chapters discuss technical background knowledge and related work. Subsequent chapters describe novel techniques which address the three challenges.

Chapter 2

Background

2.1 Introduction

This chapter provides an overview of the techniques and theory used in this thesis. Sections 2.2 and 2.3 describe the machine learning and evaluation techniques used in this thesis, respectively. Section 2.4 concludes.

2.2 Machine Learning

Machine learning is a family of statistical models and algorithms used to infer functions to estimate future values given past observations and their *features*. Features, or explanatory variables, are a set of observable attributes used by machine learning algorithms to build the correlations required to predict unseen values, or *labels*. Figure 2.1 illustrates the process by which a model is fitted to past observations and used to infer the label of unseen data points. The n -dimensional space described by n features is known as a *feature space*. A *feature vector* is then the set of values describing a single point in this space.

Machine learning techniques are used in this thesis for classification and sequence modelling. Classification is the task of predicting the categorical label, or *class*, for a set of features, based on labelled training data, i.e. instances whose categories are known.

Sequence modelling is the task of capturing the underlying probability distribution describing a sequence of values. This section describes the classification and sequential modelling techniques used in this thesis.

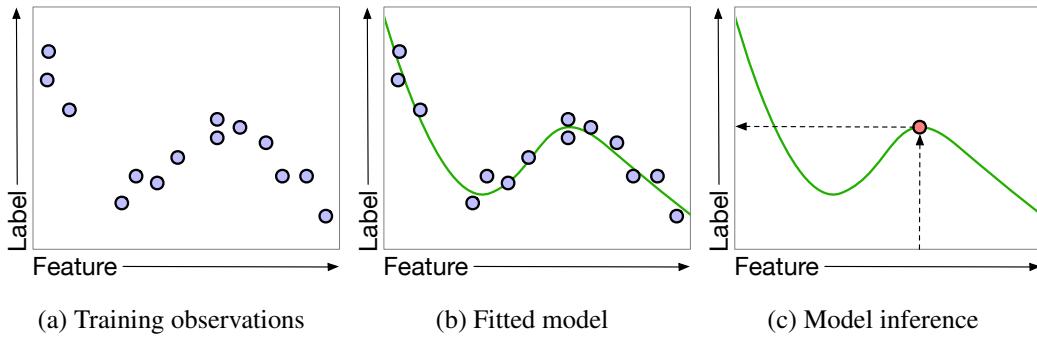


Figure 2.1: An illustration of the training and inference of a machine learning model. The x - and y -axes depict features and labels, which may be continuous or discrete multi-dimensional spaces. In (a), training observations have been collected, consisting of features and their corresponding labels. A model is then fitted to these observations, shown as the green curve in (b). The model can then be used to infer the label of unseen feature values, shown in (c).

2.2.1 Feed-forward Neural Networks

Artificial Neural Networks comprise a network of artificial neurons and weighted connections between them to map input variables to a response. Figure 2.2 shows the architecture of a feed-forward artificial neural network. Each node represents an artificial neuron. The neurons are grouped into layers. The signal of an artificial neuron at a given layer is connected to the input of each artificial neuron in the next layer.

Intermediate layers for which there are no ground truth values are known as *hidden* layers. Figure 2.2 depicts a three-layered network with three input values, a single hidden layer with four artificial neurons, and two outputs. Feed-forward multi-layered artificial neural networks are powerful *universal function approximators*, capable of learning any bounded continuous function to arbitrary precision [Hor91; Lu+17; Yar17].

The signal of an artificial neuron is the *activation*. For a given layer ℓ , the activations $\mathbf{a}^{[\ell]}$ are a function of the activations of the previous layer $\mathbf{a}^{[\ell-1]}$, the connection weights \mathbf{W} , biases \mathbf{b} , and an activation function $\phi(z)$:

$$\mathbf{a}^{[\ell]} = \phi\left(\mathbf{W}^{[\ell]T} \mathbf{a}^{[\ell-1]} + \mathbf{b}^{[\ell]}\right) \quad (2.1)$$

The connection weights and biases of an artificial neural network are adjusted during training such that, for a pair (\mathbf{x}, y) where \mathbf{x} is an input vector of features and y is an observed label, the difference between the final layer activations and y is minimised.

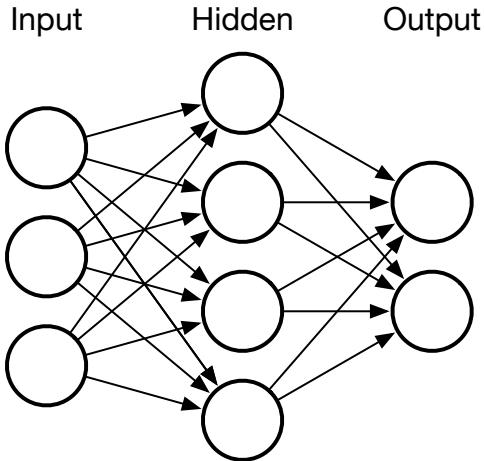


Figure 2.2: A feed-forward artificial neural network with a single hidden layer. Each node in the graph depicts an artificial neuron, and each edge represents the output of one neuron connected to the input of another.

Activation Function The activation function $\phi(z)$ is a non-linear differentiable function used to calculate the activation of an artificial neuron given a value $z \in \mathbb{R}$. A non-linear function is required to enable the “stacking” of artificial neuron layers to approximate non-linear functions. Commonly the *Sigmoid* logistic function is used:

$$\phi(z) = \frac{1}{1 + e^{-z}} \quad (2.2)$$

Sigmoid activations are bounded in the range $(0, 1)$, depicted in Figure 2.3a. For hidden layers, a disadvantage of sigmoid activation is that the output is not zero-centred. To address this, the *Hyperbolic Tangent* (tanh) activation may be used, which is a scaled sigmoid function with values in the range $(-1, 1)$:

$$\phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \quad (2.3)$$

Figure 2.3b illustrates the hyperbolic tangent activation. Logistic function-based activations suffer from a squashing effect with large positive and negative values, and they are *dense* activations in which every artificial neuron contributes to the output value. The *Rectified Linear Unit* (ReLU) activation function addresses both issues, with an unbounded range $[0, \infty)$:

$$\phi(z) = \max(z, 0) \quad (2.4)$$

The ReLU activation is shown in Figure 2.3c. If initialised with random weights in the range $[-1, 1]$, an average of 50% of ReLU activated neurons will not fire, so

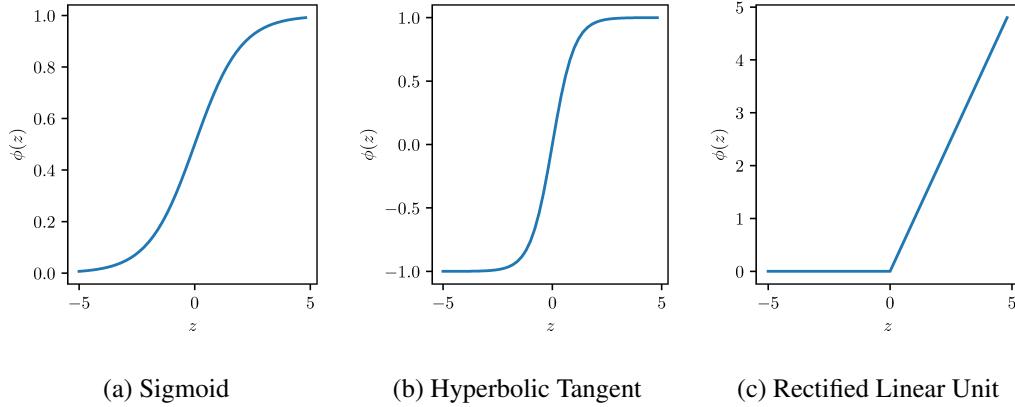


Figure 2.3: Three non-linear functions commonly used to determine the activation of artificial neurons in an artificial neural network.

that activations are sparse. The *Leaky ReLU* variation addresses the issue of large adjustments to parameters during training leading to neurons that are not activated for any input, effectively “dying”. Leaky ReLU prevents artificial neurons from becoming unresponsive to variations in input by introducing a small slope for negative values:

$$\phi(z) = \begin{cases} 0.01z, & \text{if } z < 0 \\ z, & \text{otherwise} \end{cases} \quad (2.5)$$

Typically a separate activation function is used for the output layer of an artificial neural network. For multi-class classification, *softmax* may be used, which, for K classes produces a vector:

$$\phi(\mathbf{z})^{(i)} = \frac{e^{\mathbf{z}^{(i)}}}{\sum_{j=1}^K e^{\mathbf{z}^{(j)}}}, i = 1, \dots, K \quad (2.6)$$

Where $\sum_{i=1}^K \phi(\mathbf{z})^{(i)} = 1$.

Backpropagation and Gradient Descent The most widely used technique to train artificial neural networks is backpropagation [RHW86]. Typically, the artificial neuron parameters are initialised with small random values. During training, a *mini-batch* of B observations is propagated through the network in a *feed-forward* stage. The final outputs of the network $\hat{\mathbf{y}}$ are then compared against the true values \mathbf{y} and used to compute an error $\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$. The appropriate error function depends on the task. For a classification task with K classes, *categorical cross-entropy* may be used:

$$\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) = - \sum_{i=1}^K \mathbf{y}^{(i)} \log(\hat{\mathbf{y}}^{(i)}) \quad (2.7)$$

For each layer ℓ , the average error of the mini-batch J is backpropagated through the network to update the connection weight $\mathbf{W}^{[\ell]}$ and bias parameters $\mathbf{b}^{[\ell]}$ based on a learning rate α :

$$J = \frac{1}{B} \sum_{i=1}^B \mathcal{L}^{(i)}(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)}) \quad (2.8)$$

$$\mathbf{W}^{[\ell]} = \mathbf{W}^{[\ell]} - \alpha \frac{\partial J}{\partial \mathbf{W}^{[\ell]}} \quad (2.9)$$

$$\mathbf{b}^{[\ell]} = \mathbf{b}^{[\ell]} - \alpha \frac{\partial J}{\partial \mathbf{b}^{[\ell]}} \quad (2.10)$$

Regularisation Techniques Neural networks are vulnerable to *over-fitting*, whereby the parameters of the model become specialised to the training observations, losing the ability to generalise to unseen data. Many *regularisation* techniques have been adopted to mitigate the risk of over-fitting.

Dropout is a regularisation technique in which a parameter in the range $[0, 1]$ is used to determine a proportion of artificial neurons to be removed. This helps training by preventing complex co-adaptations on training values [GBC16].

2.2.2 Recurrent Neural Networks

A Recurrent Neural Network (RNN) [Gra12] is an artificial neural network in which the connections between artificial neurons form a cycle, enabling the processing of arbitrary size sequences by maintaining and updating a *hidden state*. RNNs are a *deep learning* architecture, where deep learning is a loosely defined class of machine learning methods built on artificial neural networks¹. Figure 2.4 depicts an RNN, where $\mathbf{x}^{(t)} \in \mathbb{R}^d$ represents point $t \in \tau$ in a sequence of inputs $\mathbf{x} = (\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(\tau)})$. The hidden states $\mathbf{h}^{(t)} \in \mathbb{R}^h$ and predicted output $\hat{\mathbf{y}}^{(t)} \in \mathbb{R}^y$ are updated at each step using:

$$\mathbf{h}^{(t)} = \phi \left(\mathbf{U} \mathbf{x}^{(t)} + \mathbf{W} \mathbf{h}^{(t-1)} + \mathbf{b}_h \right) \quad (2.11)$$

$$\hat{\mathbf{y}}^{(t)} = \sigma \left(\mathbf{V} \mathbf{h}^{(t)} + \mathbf{b}_{\hat{\mathbf{y}}} \right) \quad (2.12)$$

¹Deep learning is distinct from *deep neural networks*, which are a specific deep learning architecture employing artificial neural networks with one or more hidden layers.

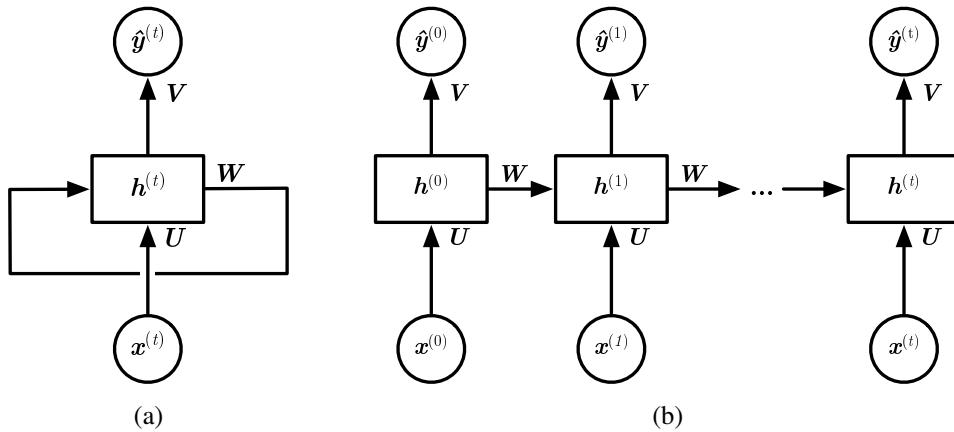


Figure 2.4: The computational graph of a Recurrent Neural Network, shown in (a) as a recurrence relation, and unfolded in (b). $\mathbf{x}^{(t)}$ is the input, $\mathbf{h}^{(t)}$ is the hidden state, and $\hat{\mathbf{y}}^{(t)}$ is the output. The network is parameterised through three weight matrices: inputs-to-hidden weights \mathbf{U} , hidden-to-hidden weights \mathbf{W} , and hidden-to-output weights \mathbf{V} . As can be seen, the parameters are shared across all points in the time series.

Where $\phi(\mathbf{z})$ and $\sigma(\mathbf{z})$ are non-linear activation functions, \mathbf{b}_h and \mathbf{b}_y are bias vectors, and $\mathbf{W} \in \mathbb{R}^{h \times d}$, $\mathbf{U} \in \mathbb{R}^{h \times h}$, and $\mathbf{V} \in \mathbb{R}^{y \times h}$ are matrices representing the hidden-to-hidden, input-to-hidden, and hidden-to-output weights respectively.

The recurrent structure of RNNs enables the modelling of patterns in data with a temporal domain, such as text or numerical time series. Whereas a feed-forward artificial neural network estimates a conditional distribution based on an instantaneous input $p(y|\mathbf{x})$, an RNN estimates a distribution conditioned on t prior observations $p\left(y^{(t)}|\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(t)}\}\right)$.

Ordinary backpropagation may be used on an RNN by unfolding the computation graph over time, shown in Figure 2.4b. This *backpropagation through time* [Wer90] enables the propagation of errors in the temporal domain in the same manner as through layers.

RNNs are universal, in that any function computable by a Turing machine can be computed by an RNN of finite size [SS91]. In practice, a significant obstacle to the performance of RNNs is the diminishing ability to learn connections between values over long sequences. This is caused by the exponential diminishing and enlarging of gradients as they are propagated through non-linear activation functions by the recurrence relation. This issue is known as the *vanishing gradients* problem [BSF94].

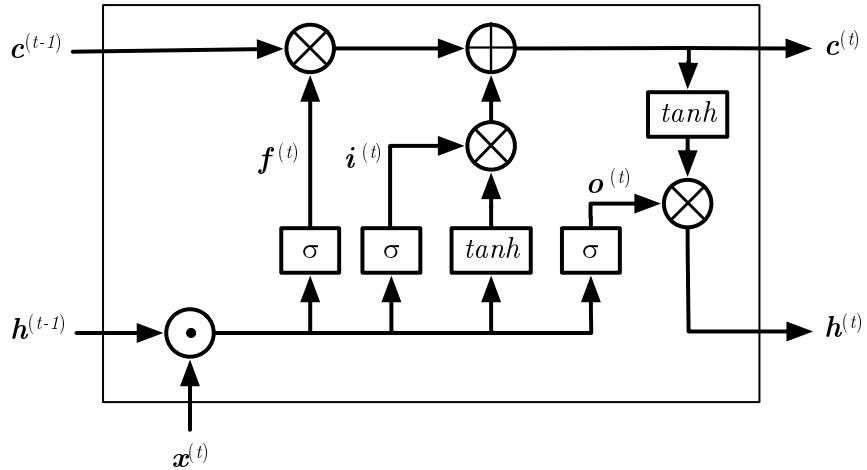


Figure 2.5: A Long Short-Term Memory cell block. Input $x^{(t)}$ is concatenated with prior hidden state $h^{(t-1)}$ and used along with prior cell state $c^{(t-1)}$ to compute a new hidden state $h^{(t)}$ and cell state $c^{(t)}$. Symbols \otimes denotes element-wise vector product, \oplus element-wise vector addition, and \odot vector concatenation.

Long Short-Term Memory Long short-term memory (LSTM) [HS97] is an RNN architecture designed to overcome the vanishing gradients problem. The LSTM augments RNN design with the addition of a *cell* for storing information, and three gates which control the flow of information into and out of the cell.

Figure 2.5 depicts the structure of an LSTM cell. In addition to the recurrent connections for hidden states, a cell state vector $c^{(t)} \in \mathbb{R}^h$ is propagated through time. The signal flow through the cell is controlled by three sigmoid activated gates: the forget gate $f^{(t)} \in \mathbb{R}^h$, input gate $i^{(t)} \in \mathbb{R}^h$, and output gate $o^{(t)} \in \mathbb{R}^h$. At each time step, the values of the gate vectors are updated using:

$$f^{(t)} = \sigma(\mathbf{W}_f x^{(t)} + \mathbf{U}_f h^{(t-1)} + \mathbf{b}_f) \quad (2.13)$$

$$i^{(t)} = \sigma(\mathbf{W}_i x^{(t)} + \mathbf{U}_i h^{(t-1)} + \mathbf{b}_i) \quad (2.14)$$

$$o^{(t)} = \sigma(\mathbf{W}_o x^{(t)} + \mathbf{U}_o h^{(t-1)} + \mathbf{b}_o) \quad (2.15)$$

$$(2.16)$$

Where weight matrices $\mathbf{W} \in \mathbb{R}^{h \times d}$ and $\mathbf{U} \in \mathbb{R}^{h \times h}$, and bias vectors $\mathbf{b} \in \mathbb{R}^h$ are subscripted for the activation being calculated: f forget gate, i input gate, or o output gate. The cell state and hidden state are then updated using:

$$\mathbf{c}^{(t)} = \mathbf{f}^{(t)} \otimes \mathbf{c}^{(t-1)} + i^{(t)} \otimes \tanh(\mathbf{W}_c \mathbf{x}^{(t)} + \mathbf{U}_c \mathbf{h}^{(t-1)} + \mathbf{b}_c) \quad (2.17)$$

$$\mathbf{h}^{(t)} = \mathbf{o}^{(t)} \otimes \tanh(\mathbf{c}^{(t)}) \quad (2.18)$$

Where \otimes denotes element-wise vector product, and subscript c denotes cell state weight matrices and bias vectors. The gated cell enables LSTMs to build connections over long sequences in data. The LSTM architecture (and its many variants [Gre+15a]) have been responsible for breakthrough results in a number of areas, for example machine translation [SVL14], speech recognition [GS05], and weather prediction [Shi+15].

2.2.3 Decision Trees

Decision trees are an intuitive form of classifier whereby a tree structure of decision nodes are used to predict the class for a given set of features. Decision trees are built using binary recursive partitioning: by creating an axis-parallel decision node for the feature which provides the highest gain, creating new child nodes for each possible outcome, splitting the data amongst these child nodes, and continuing recursively. The gain of a feature is found by first computing the entropy of the data set. Given a set of data points D where $p_{(+)}$ is the number of positive examples in D and $p_{(-)}$ is the number of negative examples in D :

$$H(D) = -p_{(+)} \log_2 p_{(+)} - p_{(-)} \log_2 p_{(-)} \quad (2.19)$$

The gain of a feature x is found using:

$$\text{Gain}(D, x) = H(D) - \sum_{V \in \text{Values}(x)} \frac{|D_V|}{|D|} H(D_V) \quad (2.20)$$

Decision trees are a popular and low overhead form of classification. Once trained, a decision tree can be implemented as a set of nested conditional statements. This makes the learned model easily interpretable, simplifying debugging.

2.3 Model Evaluation Techniques

This section describes the techniques used to evaluate the machine learning methods described in the previous section.

2.3.1 ZeroR

A *ZeroR* model is used in classification tasks to provide a baseline to evaluate the performance of classifiers. A ZeroR model represents the simplest approach to classification, its output is the mode of the training data labels, irrespective of the input. For example, given training data with the labels $\mathbf{y} = \{A, B, B, C\}$, a ZeroR model will produce output $\hat{\mathbf{y}} = B$ for all future inputs. A ZeroR model has no power of prediction since its output is not conditioned on input features.

2.3.2 Training, Validation, Test Data

The data used to train a machine learning model must not be used to evaluate it. To evaluate a machine learning model, data must be divided into disjoint *training* and *testing* partitions. For a set of N pairs of input and output observations $\mathcal{D} = (\mathbf{x}^{(i)}, \mathbf{y}^{(i)}), i = 1, \dots, N$, let $k : 1, \dots, N \rightarrow 1, \dots, K$ be an indexing function that indicates the partition to which the observation $i \in N$ is allocated by the randomisation. The model is then trained with the k th part of the data removed, yielding fitted function $\hat{f}^{-k}(\mathbf{x})$. The quality of the model $V(\hat{f}^{-k})$ is then evaluated using the loss between the expected values and predicted values in the unseen test set:

$$\mathbf{y}_k = \{\mathbf{y}^{(i)} \in \mathbf{y} | i \in k(\mathbf{y})\} \quad (2.21)$$

$$V(\hat{f}^{-k}) = \mathcal{L}(\mathbf{y}_k, \hat{f}^{-k}(\mathbf{x}_k)) \quad (2.22)$$

Employing a second indexing function u to further divide the data $\not\in k$ into disjoint *training* and *test* sets creates a *validation* set. A validation set can be used to automatically tune model parameters. Given a model f_p parameterised by p , the quality of the model evaluating using the parameters \bar{p} that provide the smallest loss on the validation set can be found from a set of parameters P using:

$$\bar{p} = \arg \min_p \mathcal{L}(\mathbf{y}_u, \hat{f}_p^{-(k \cup u)}(\mathbf{x}_u)), \quad p \forall P \quad (2.23)$$

$$V(\hat{f}_{\bar{p}}^{-k}) = \mathcal{L}(\mathbf{y}_k, \hat{f}_{\bar{p}}^{-(k \cup u)}(\mathbf{x}_k)) \quad (2.24)$$

2.3.3 K-Fold Cross-validation

Dividing the dataset into fixed training and test sets is problematic if it results in the test set being small. A small test set implies statistical uncertainty around the estimated

average test error, making it difficult to compare models.

Ideally, sufficient data would exist to be set aside for testing. When not practically possible, cross-validation may be used. Cross-validation enables the use of all observations in a data set in the estimation of the mean test error, at the expense of increased computational cost through repeated evaluations.

K-Fold cross-validation sets aside part of the available data to fit the model, and a different part to test it. Data is split into K roughly equal-sized parts.

$$CV(\hat{f}) = \frac{1}{K} \sum_{i=1}^K \mathcal{L}(\mathbf{y}^{(i)}, \hat{f}^{-K^{(i)}}(\mathbf{x}^{(i)})) \quad (2.25)$$

Where for a set of n observations, $K \leq n$. If $K = n$ this is known as *leave-one-out* cross-validation.

For a classification task, if observations are distributed amongst the K partitions such that the distribution of observed classes in all partitions is equal, this is called *stratified K-fold* cross-validation.

2.3.4 Principal Component Analysis

Principal Components Analysis PCA [Jol11] is a statistical procedure for identifying the underlying components responsible for variance in data. Given n dimensions of observations, PCA produces a k -dimensional subspace through a linear transformation, where $k < n$. Each component is ordered by decreasing variance, $Var(k^{(i)}) \leq Var(k^{(i-1)})$. This dimensionality-reduction is useful for exposing underlying correlations in machine learning feature design, and to aid in visually inspecting high-dimensional spaces where $n \geq 3$.

To compute the principal components, one begins with a normalised matrix of observations $\mathbf{X} \in \mathbb{R}^{m \times n}$ of m rows and n columns (components). The data is normalised to the range $[-1, 1]$ by subtracting the mean and scaling each component by its variance. Given eigenvectors $\mathbf{v}^{(1)}, \dots, \mathbf{v}^{(m)}$, the matrix of k principal components $\hat{\mathbf{X}} \in \mathbb{R}^{m \times k}$ can be computed using:

$$\hat{\mathbf{X}} = [\mathbf{v}^{(1)}, \dots, \mathbf{v}^{(k)}]^T \cdot \mathbf{X} \quad (2.26)$$

2.4 Summary

This chapter provides background on the machine learning techniques used in this thesis to develop low cost techniques for compiler construction, and the techniques used in evaluating their effectiveness. The following chapter surveys research literature relevant to this work.

Chapter 3

Related Work

3.1 Introduction

This chapter surveys the literature in areas relevant to this thesis. Section 3.2 reviews research in program generation, focusing first on generating benchmarks for performance characterisation, then compiler test case generation. Section 3.3 reviews the literature of empirical program optimisation, covering iterative compilation and machine learning. Section 3.4 surveys the literature of related works in deep learning over programs. Finally Section 3.5 concludes.

3.2 Program Generation

In this thesis, program generation is used as a mechanism to simplify the development of compiler heuristics and to validate the behaviour of compilers. The generation of artificial programs is a broad field with a wide range of applications; this section categorises the literature in the two areas that are relevant to this thesis: program generation for performance characterisation, and program generation for compiler validation.

3.2.1 Benchmark Generation for Performance Characterisation

Benchmark suites serve a wide variety of uses from compiler optimisations to hardware design. The challenge in creating a benchmark suite is to capture a diverse set of workloads that is both representative of real-world usage while providing adequate coverage of the program space. Achieving either of these two goals is a challenging task, and efforts towards one goal may hamper the other. As such, there is no “one size

fits all” approach to assembling benchmark suites.

Given its importance, benchmark suite characterisation has been the subject of much research. An evaluation of popular GPGPU benchmark suites [Ryo+15] reveals there are important parts of the program space left untested. Xiong et al. [Xio+13] demonstrate that workload behaviour is highly input dependent, and argue that benchmarks created for academic research cannot represent the cases of real-world applications. A review of big data benchmarks [Fer+12] found many to be unrepresentative, and that current hardware designs, while optimised for existing benchmark suites, are inefficient for true workloads.

Benchmark suites should be *diverse*, with each benchmark within a suite occupying a distinct point in the program space, else the benchmark may be redundant. Ould-Ahmed-Vall et al. [Oul+08] show that statistical models trained on 10% of SPEC CPU 2006 data are transferable to the remaining data. Goswami et al. [Gos+10] evaluate the diversity of 38 GPGPU benchmark workloads, finding that *Similarity Score*, *Scan of Large Arrays*, and *Parallel Reduction* benchmarks show significantly different behaviour due to their large number of diverse kernels, but the remaining 35 benchmarks provide similar characteristics. Phansalkar, Joshi, and John [PJJ07] show that a subset of 14 SPEC CPU 2006 programs yields most of the information of the entire suite, and Panda et al. [Pan+18] find that SPEC CPU 2017 contains workloads that, when removed, do not degrade the suite’s coverage of the program space.

Researchers have turned to *synthetic* benchmarks to address the coverage and diversity challenges. The use of synthetic benchmarks is not new, with an early example from 1976 using a synthetic benchmark to compare the computing power of processors for scientific workloads [CW76]. Bell and John [BJ05] pose the *synthesis* of synthetic benchmarks as a test case generation problem, using hardware counters to validate the similarity of synthesised benchmarks to a target workload.

A popular use of synthetic benchmark generation techniques is to aid microprocessor design. Joshi et al. [Jos+08] use micro-architecture-independent characteristics such as basic block sizes and data footprint to summarise workloads. Their benchmark generator, *BenchMaker*, then generates a linear sequence of basic blocks and randomly populates them with assembly instructions to match the desired workload characteristics. *MicroProbe* [Ber+12] uses feedback-directed micro-benchmark generation to perform a systematic energy characterisation of a processor.

GENESIS [CGA15] is a language for generating synthetic training programs. The essence of the approach is to construct a probabilistic grammar with embedded seman-

tic actions that define a language of possible programs. New programs may be created by sampling the grammar and, through setting probabilities on the grammar productions, the sampling is biased towards producing programs from one part of the space or another. Theoretically, this technique is completely general since a grammar may be constructed to match any desired program domain. However, despite being theoretically possible, it is not easy in practice to construct grammars which are both suitably general and also produce programs that are in any way similar to what humans write. Such grammar-based benchmark generators have been shown to be successful only over a highly restricted space of stencil benchmarks with little control flow or program variability [Cum+16a; FE15; GA15]. Restricting the domain of programs that can be generated limits the usefulness of this approach for many problems. It remains unclear how much effort it would take to define grammars capable of producing realistic programs in more complex domains.

Interesting recent developments in synthetic benchmark generation have combined elements from feedback-directed test case synthesis (reviewed in the next section) with synthetic benchmarking for the purpose of generating *adversarial* benchmarks that expose performance issues in systems.

Dhok and Ramanathan [DR16] apply mutation techniques to an initial set of coverage-driven inputs to expose inefficiencies in loops. *SlowFuzz* [Pet+17] uses resource-usage to guide evolutionary search over program inputs to expose performance bottlenecks that could be exploited by attackers to produce Denial-of-Service attacks. It considers the input to a program as a byte sequence and performs mutations to find the byte sequence within a fixed input size that maximises slowdown. Similarly, *Singularity* [Wei+18] uses an evolutionary search over the space of program inputs but using input *patterns* to find the input with the worst case performance.

PerfSyn [TPG18] tackles the related problem of exposing performance bottlenecks from API usage. For a method under test, it starts with a minimal example input and applies a sequence of mutations that modify the original code. *PerfFuzz* [Lem+18] uses feedback-directed program mutation to generate programs which maximise execution counts at program locations. Pedrosa et al. [Ped+18] applies this adversarial benchmark approach to network functions. Their tool, *CASTAN*, takes as input the code for a network function and outputs packet sequences that trigger slow execution paths.

Ding, Mishra, and Hofmann [DMH19] propose an alternative approach to addressing data scarcity in machine learning. Instead of generating benchmarks whose features must be evaluated and labelled to derive training data, they instead generate new

feature values and labels based on the distributions in the training set. In contrast to the methods proposed in this work, their dataset generator — based on a Gaussian Mixture Model — is explicitly designed to amplify unusual behaviour within the training set. The generative model proposed in this work is intended to produce new data that is representative of the training set.

In contrast to prior works, the benchmark generation technique proposed in this thesis provides *general-purpose* program generation over unknown domains, in which the statistical distribution of generated programs is automatically inferred from a corpus of real-world code. This allows *representative* benchmarks to be generated automatically without expert guidance. To the best of my knowledge, no prior work has tackled the problem of undirected benchmark generation from example code.

3.2.2 Test Case Generation for Compiler Validation

Compilers are a fundamental trusted technology, and their correctness is critical. Errors in compilers introduce security vulnerabilities and catastrophic runtime failures. Therefore, ensuring that a compiler behaves as expected is of utmost importance.

Approaches to assure compiler behaviour are divided into *verification* and *validation*. The complexity of optimising compilers and the programming languages they compile renders formal verification of the entire compiler prohibitively expensive. Efforts have been made in this direction, for example, CompCert [Ler17], a formally verified compiler for the C programming language, but this comes at the cost of supporting only a subset of the language features and with significantly lower performance compared to unverified compilers such as GCC [Daniel2017]. Still, even CompCert is not fully verified, and errors have been discovered in the unverified components [Yan+11].

Because of the difficulties of *verification*, compiler developers turn to *validation*, in which the behaviour of a compiler is validated using a set of hand crafted input programs, or *test cases*. For each test case, the expected outcome (determined by the specification of the compiler) is compared against the observed outcome to validate that the compiler conforms to the specification, for those inputs. However, the absence of errors during the validation process does not prove that the compiler is free from errors unless all possible inputs are tested exhaustively, and the input space for compilers is huge¹. As such, hand-designed suites of test programs, while important, are

¹In theory, the input space of a compiler is infinite. In practice, however, constraints such as maximum input file size bound the space, though these bounds far exceed what is computationally tractable with current technology.

inadequate for covering such a large space and will not touch all parts of the compiler.

The random generation of programs to test compilers is a well-established approach to the compiler validation problem. The main question of interest is in how to efficiently generate programs which expose bugs. There are two complementary approaches: *program generation*, where inputs are synthesised from scratch; and *program mutation*, where existing inputs are modified so as to identify anomalous behaviour. This section reviews each approach in turn.

3.2.2.1 Program Generation for Compiler Testing

The idea of generating programs to test compilers is well-established. The majority of test case generation approaches are based on a formal specification of the programming language syntax and grammar. An early approach is presented by Hanford [Han70], which randomly enumerates programs described by a context-free grammar to produce an inexhaustible supply of new programs. While the generated programs are syntactically valid, they are meaningless, and many cannot be executed. This limits their value only to testing the compiler front end.

Deeper testing of compiler components is enabled by extending test case generators to produce both a syntactically correct program and a corresponding *gold standard output* that would be produced by a conformant compiler. The compiled program is then executed and its output compared against this gold standard. Figure 3.1a shows the process. Gold standard-based approaches are surveyed by Boujarwah and Saleh [BS97] and Kossatchev and Posypkin [KP05]. The challenge of the approach is in generating the corresponding gold standard output for each program.

Differential testing, illustrated in Figure 3.1b, accelerates testing by enabling many compilers to be tested at once. In differential testing, a single program is fed through multiple compilers to produce corresponding executables. Each of these executables are run and their outputs recorded. Since the executables were compiled from the same input program, if any of the outputs differ, a bug in one of the compilers has been exposed. The advantage of differential testing over prior approaches is that it does not require a gold standard for the expected behaviour of a conformant compiler. As such, any well-formed program may be used as a test. Even malformed inputs may be used to identify anomalies in the error handling logic of compilers. While lacking a gold standard for behaviour makes differential testing theoretically insufficient to prove that a compiler with a minority output is at fault, in practice the likelihood of the majority consensus being incorrect is extremely unlikely, and I have not been able to find any

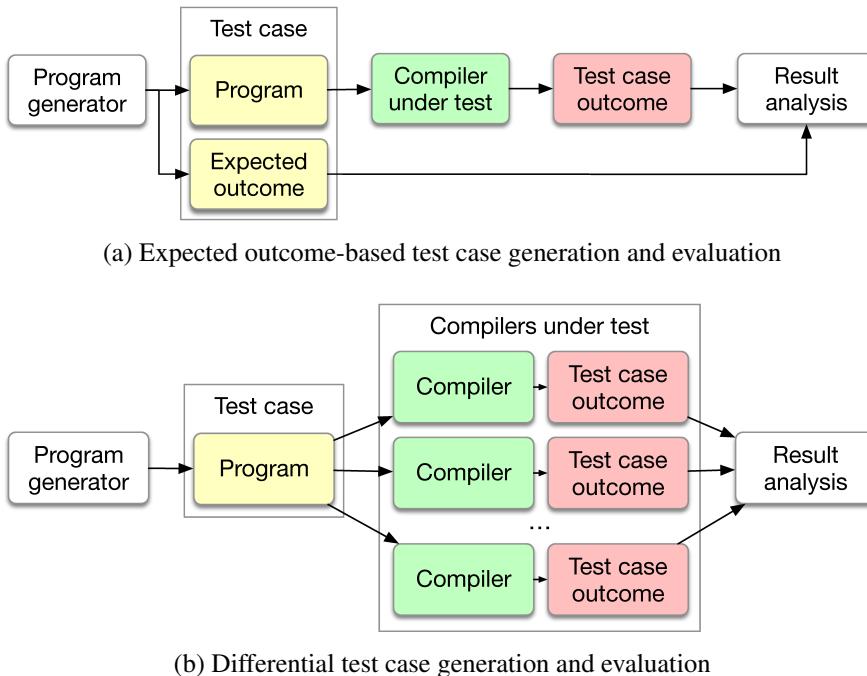


Figure 3.1: Two approaches to addressing the *compiler validation* problem through test case generation. In (a), a test case is composed of a program and a summary of its expected behaviour. In (b), only a program is required, and the expected outcome is determined by majority voting on the observed outcomes across multiple compilers.

work in the literature that reports such issues.

Differential testing has been applied across different compilers [Che+16b; Lid+15] and using the same compiler with different configurations [Kyl+15; Pał+11] (and combinations of the two). Chen et al. [Che+16a] empirically contrasts the two approaches, along with a comparison to Equivalence Modulo Inputs testing (described in the following subsection).

In the foundational work on differential testing for compilers, McKeeman [McK98] presents generators capable of enumerating programs of a range of qualities, from random ASCII sequences to C model conforming programs. Subsequent works have presented increasingly complex generators which improve in some metric of interest, generally expressiveness or probability of correctness.

CSmith [Yan+11] is a widely known and effective generator which enumerates programs by pairing infrequently combined language features. In doing so, it produces correct programs with clearly defined behaviour but extremely unlikely functionality, increasing the chances of triggering a bug. Achieving this required extensive engineering work, most of it not portable across languages, and ignoring some language features. Lidbury et al. [Lid+15] extend CSmith to the OpenCL programming language, a superficially simple task, yet this required 9 man-months of development and 8000 lines of code. Subsequent generators influenced by CSmith focus on compiler features and bug types beyond the scope of CSmith, such as Orange3 [NHI13] which targets arithmetic bugs.

Similar to program generation for testing compilers, prior work has focused on input generation for testing programs [CDP15; DJW12; Fet+15; GKS05; RNL08]. Glade [Bas+17] derives a context-free grammar for structured program inputs from a corpus of examples. The derived grammar is enumerated to produce new inputs, though no distribution is learned over the grammar; enumeration is uniformly random.

Programs generated by grammar-based approaches are often unlike real handwritten code, and are typically large. As such, once a bug has been identified, test case reduction [Reg+12] is required to minimise the size of the program and isolate the code of interest. Automated test case reduction does not scale to the rate at which effective compiler fuzzers produce programs of interest, often taking minutes or hours for each test case [PDL16].

Entirely machine learning-based approaches to program generation have recently been proposed. They are reviewed in Section 3.2.2.3.

3.2.2.2 Program Mutation and Feedback-directed Compiler Testing

An alternative method for generating programs to use as compiler test cases is to mutate a seed input. Equivalence Modulo Inputs (EMI) testing, introduced by Le, Afshari, and Su [LAS14], starts with an existing program and inserts or deletes statements that will not be executed so that the functionality of the code is unchanged. If the functionality of the compiled code is affected, it is due to a bug in the compiler. In the original work [LAS14], code from dead regions is randomly deleted. *Athena* [LSS15a] guides the approach using a Markov Chain Monte Carlo method, and supports dead code insertion as well as removal. *Proteus* [LSS15b] applies the EMI technique to test link-time optimisers. *Hermes* [SLS16] extends EMI to permit the mutation of *live code* regions, not just dead code. This greatly increases the expressiveness of the generated programs.

LangFuzz [HHZ12] also uses program mutation but does this by inserting code segments which have previously exposed bugs. This increases the chances of discovering vulnerabilities in scripting language engines. Starting with a coverage-guided set of inputs, *T-Fuzz* [PSP18] uses dynamic tracing to detect input checks in programs and selectively removes them to expose defects. *Skeletal program enumeration* [ZSS17] again works by transforming existing code. It identifies algorithmic patterns in short pieces of code and enumerates all the possible permutations of variable usage. *pFuzzer* [Mat+19] targets input parsers, using dynamic tainting to produce a set of legal inputs that cover all conditions during parsing. Coverage-directed mutation techniques have been used for differential testing the Java Virtual Machine [Che+16b].

Machine learning has been used to guide test case mutation. Cheng et al. [Che+19] construct artificial neural networks to discover correlations between PDF test cases and their execution in the target program. The correlations are then leveraged to generate new paths in the target program. *NEUZZ* [She+18] learns a differentiable neural approximation of target program logic, then uses Stochastic Gradient Descent to guide program mutation. *Skyfire* [Wan+17] learns a probabilistic context-sensitive grammar over a corpus of programs to generate input seeds for mutation testing. The generated seeds are shown to improve the code coverage of AFL [Zal14] when fuzzing XSLT and XML engines. The seeds are not directly used as test cases.

EMI and feedback-directed approaches rely on having a large number of seed programs to mutate. As such, they may still require an external code generator. Similarly to grammar-based approaches, these methods often tend to favour large test programs.

3.2.2.3 Neural Program Generation

Recently, machine learning methods have been proposed for generating test cases. These differ from prior works that use machine learning to *guide* the generation of test cases. Methods have been proposed based on the success of Recurrent Neural Networks (RNNs) at modelling sequential data [Joz+16]. RNNs have been successfully applied to a broad range of generative tasks in other domains, including image captioning [Vin+15], colourising black and white photographs [ZIE16], artistic style [GEB15], and image generation [Gre+15b].

The proficiency of RNNs for sequence modelling is well demonstrated [SVL14]. Sutskever, Vinyals, and Le apply two RNN networks to translate first a sequence into a fixed length vector, then to decode the vector into an output sequence. This architecture achieves state-of-the-art performance in machine translation. The authors find that reversing the order of the input sequences markedly improves translation performance by introducing new short term dependencies between input and output sequences.

Although nascent, the use of artificial neural networks to generate programs is evolving rapidly. *Neural Programmer* [NLS16] is an early example of program generation through the latent representation of an artificial neural network.

Learn&Fuzz [GPS17] and *IUST DeepFuzz* [NPK18] use LSTM networks, trained on a corpus of PDF files, to generate test inputs for PDF renderers. In the case of *Learn&Fuzz*, they uncover a bug in the Microsoft Edge renderer. Unlike compiler testing, PDF test cases require no inputs and no pre-processing of the training corpus.

Jitsunari and Arahori [JA19] incorporate coverage directed feedback during the training of generative models for PDFs. They train a model on an initial corpus, then sample it. They then evaluate the samples using code coverage and select those that provide the best coverage to be used as additional training data to fine-tune the model. Doing so improves the code coverage of the final generative model.

Most similar to the work presented in this thesis is *DeepFuzz* [Liu+19], in which an LSTM network is used to generate fragments of C programs that are inserted into GCC unit tests. The mutated unit tests are then used for differential testing. They propose a character-level model with three sampling methodologies: one where a sample is made for every character, one where no sampling is made (so the generated fragment is conditioned solely on the sample prefix), and a hybrid approach in which sampling occurs only on white-space. In their evaluation, they uncover 8 bugs in GCC, and achieve up to an 82.63% rate of syntactically valid samples. The work presented in this thesis

differs in several ways. The key difference is the use of a recurrent neural network to generate only *fragments* of a program, derived from an existing GCC unit test. In contrast, the approach presented in this thesis generates entire programs. Further, the approach presented in this thesis is trained on a wide range of handwritten programs with the intent of emulating *natural* programming styles.

Kosta, Seaman, and Xi [KSX19] generate C functions to augment the training dataset of machine learning for static analysis, using vulnerability injection to produce positive examples. They generate programs using a Grammar Variational Autoencoder, an artificial neural network architecture which decomposes a program to a sequence of context-free grammar production rules. Unlike the syntactic-level approach presented in this work, this grammar-based approach guarantees the generation of syntactically correct code by masking production rules that would lead to invalid programs. The authors combine this with semantic repair to further reduce the chance of a sample being invalid.

A drawback of Grammar Variational Autoencoder approaches is the explosion in vocabulary size arising from using a context-free grammar to model a programming language. To mitigate this, the authors must limit the expressiveness of the generator by using only a small subset of the programming language’s features. Kosta, Seaman, and Xi achieve this by selecting a handful of C functions to derive a grammar from, followed by further manual pruning of the derived grammar based on what the authors felt would be used most often by programmers. It is unclear whether such an approach could be extended to match the expressiveness of the method presented in this work.

Neural program generation has been used for purposes other than compiler testing, reviewed in Section 3.4.

3.3 Program Optimisation

Modern compilers are complex, typically containing dozens or hundreds of independent optimisation passes. Determining which optimisation passes to apply, and in what order, is a challenge that depends on a variety of factors from the properties of the program being compiled to the target hardware. Current state-of-practice is for compilers to use a fixed ordering of optimisations, and for each optimisation to contain heuristics to determine when the optimisation is applied, and with what parameters. Such heuristics require expert design at the expense of great effort and manual tweaking. Still, they rarely are capable of extracting all of the available performance.

Extracting the maximum performance of a program is not simply a case of enabling more optimisations, but in identifying which, out of a set of candidate optimisations, will provide the best performance for the current case. A recent study by Georgiou et al. [Geo+18] illustrates the scale of the challenge. Using two modern releases of the industry-standard LLVM compiler, they obtain an average 3.9% performance improvement across 71 benchmarks on embedded processors by selectively *disabling* optimisations enabled at the standard `-O2` optimisation level.

Selecting the right optimisations is critical. In some domains, the margin of performance to be gained is significant. For example, Ryoo et al. [Ryo+08a] find speedups of up to $432\times$ through the appropriate selection and use of tiling and loop unrolling optimisations on a GPU matrix multiplication implementation.

Given the challenges of heuristic and analytical methods for extracting performance, researchers have turned to empirical methods such as iterative compilation.

3.3.1 Iterative Compilation and Auto-tuning

Iterative compilation is a method of performance tuning in which a program is compiled and profiled using multiple optimisation configurations to find the configuration which provides the best performance. Unlike analytical methods which attempt to predict the parameters that produce good performance, iterative compilation is empirical. A set of candidate configurations are selected, and for each, the program is compiled and profiled. The configuration that minimises the value of a suitable cost function (such as program runtime) is selected. Pioneered by Bodin et al. [Bod+98], the technique was initially demonstrated to find good configurations in the non-linear three-dimensional optimisation space of a matrix multiplication benchmark. By exhaustively enumerating the optimisation space they were able to find the global minima of the cost function; however, the authors state that in practice this may not be possible. In cases where an exhaustive enumeration of the optimisation space is infeasible, the process may be cast as a search problem.

While conceptually simple, the empirical nature of iterative compilation yields good results. Iterative compilation has since been demonstrated to be a highly effective form of empirical performance tuning for selecting compiler optimisations. In a large scale evaluation across 1000 data sets, Chen et al. [Che+10] found iterative compilation to yield speedups in GCC over the highest optimisation level (`-O3`) of up $2.23\times$.

The greatest challenge of iterative compilation is the exponential blow-up of optimisation space size with the addition of independent optimisations. The hundreds of discrete optimising transformations found in modern compilers render an exhaustive search of the optimisation space infeasible. This has driven the development of methods for reducing the cost of evaluating configurations. These methods reduce evaluation costs either by pruning the size of the optimisation space and performing a random or exhaustive enumeration or by guiding a directed search to traverse the optimisation space while evaluating fewer points.

3.3.1.1 Pruning the Iterative Compilation Search Space

Triantafyllis and August [TA03] propose using feedback during the evaluation of configurations to prune the optimisation space. This is combined with a static performance estimator to obviate the need to run each configuration of a program. Pan and Eigenmann [PE06] formalise the iterative compilation problem as: given a set of compiler optimisation options, find the combination that minimises the program execution time efficiently, without a priori knowledge of the optimisations and their interactions. Their technique, *Combined Elimination*, iteratively prunes the search space, reducing the tuning time to 57% of the closest alternative. Posing the problem as a subset search negates the challenge of optimisation *ordering*, though this challenge has been the focus of other work [KC12; PJ13].

Ryoo et al. [Ryo+08b] prune the optimisation space for GPGPU workloads using the common subset of optimal configurations across a set of training examples. This technique reduces the search space by 98%. The trade-off of this approach is that there is no guarantee that for a new program, the reduced search space will include the optimal configuration. Similarly, Purini and Jain [PJ13] identify a set of good optimisation sequences offline that is small enough for each new program to be tried with all sequences in the set. They find that a sequence set size of 10 yields 13% speedups on PolyBench and MiBench programs. Although this does not reduce the cost of finding the set of good sequences, the process need only be performed once per platform, so the cost may be amortised by reusing the same sequence set.

Frameworks for iterative compilation offer mechanisms to abstract the iterative compilation process from the optimisation space. These lower the cost of adopting iterative compilation techniques by providing reusable logic to search optimisation spaces. Examples include *OpenTuner* [Ans+13] which provides domain-agnostic ensemble search techniques and *CLTune* [NC15] for iterative compilation in OpenCL.

A complementary approach to search space pruning is knowledge sharing. The idea is that, since software has many users, knowledge of the optimisation space can be shared between users rather than having each redundantly perform their own exploration of the optimisation space from scratch. Such “big data” approaches to auto-tuning have been variously proposed as *Collective Optimization* [FT10], *Crowdtuning* [MF13], and *Collective Mind* [Fur+14]. Fursin et al. argue that the challenges facing widespread adoption of iterative compilation techniques are attributable to: a lack of common, diverse benchmarks and data sets; a lack of common experimental methodologies; problems with continuously changing hardware and software; and the difficulty in validating techniques due to a lack of sharing in publications. They propose systems for addressing these concerns which provide a modular infrastructure for sharing iterative compilation performance data and related artefacts across the internet [Fur+14]. In past work [Cum+16b], a domain-specific implementation of knowledge sharing was used to accelerate tuning of stencil codes on GPUs by sharing iterative compilation data between users across the internet.

Other challenges facing iterative compilation are the lack of portability and the inability to respond to change. Any change to the exacting combination of program, input data, and hardware may impact the results of optimisations, invalidating any prior exploration of the optimisation space and requiring a new iterative compilation process to be started from scratch. *Online* techniques attempt to mitigate these issues.

3.3.1.2 Online Iterative Compilation

The expensive optimisation space exploration required by iterative compilation has spurred development of online iterative compilation that interleaves the exploration of the optimisation space with regular program use. This is a challenging task, as a random search of an optimisation space may result in many configurations with performance far from optimal, degrading the quality of service for the user. In a real-world system, evaluating many sub-optimal configurations may cause a significant slowdown of the program. Thus a requirement of dynamic optimisers is that convergence time towards optimal parameters is minimised. Further, *exploration* and *exploitation* must be balanced so as to maintain an acceptable quality of service.

Tartara and Crespi Reghizzi [TC13] propose a technique for *long-term learning* of compiler heuristics without an initial training phase. They treat the continued optimisation of a program over its lifetime as an evolutionary process with the goal of finding the best set of compiler heuristics for a given binary.

Ansel and Reilly [AR12] present an adversarial approach to online evolutionary performance tuning. At runtime, the available parallel resources of a device are divided between two partitions. Two configurations of the application are then executed simultaneously, one on each partition. One of the configurations is chosen to be “safe”, the other, experimental. The configuration which yields the best performance is retained as the “safe” choice for future iterations, and the process repeats.

Mpeis, Petoumenos, and Leather [MPL16] combine online and offline iterative compilation for mobile devices. They capture slices of user behaviour on a device online during use, which are then replayed offline for iterative compilation. This has the advantage of specialising the performance tuning of software to the behaviour of the individual user.

Related to online iterative compilation is dynamic optimisation. *Dynamo* [BDB00] is a dynamic optimiser which performs binary level transformations of programs using information gathered from runtime profiling and tracing. This provides the ability for the program to respond to changes in dynamic features at runtime using low-level binary transformations.

3.3.1.3 Algorithmic Choice and Rewriting

Complementary to iterative compilation is *algorithmic choice*. Like iterative compilation, the goal is to find the configuration of a program that maximises performance. However, whereas iterative compilation indirectly affects the program by selecting compiler optimisations to produce different configurations, algorithmic choice selects between permutations of semantically equivalent algorithms, typically explicitly provided by the user.

PetaBricks [Ans+09] is a language and compiler for algorithmic choice. Users provide multiple implementations of algorithms, optimised for different parameters or use cases. This creates a search space of possible execution paths for a given program. This has been combined with auto-tuning techniques for enabling optimised multigrid programs [Cha+09], with the wider ambition that these auto-tuning techniques may be applied to all algorithmic choice programs [Ans14]. While this helps produce efficient programs, it places the burden of producing each algorithmic permutation on the developer, requiring them to provide enough contrasting implementations to make a search of the implementation space fruitful.

Halide [Rag+13] alleviates the burden of algorithmic rewriting by providing a high-level domain-specific language that allows users to express pipelines of stencil com-

putations succinctly. The *Lift* framework [SD17] uses a set of semantic-preserving rewrite rules to transform high-level Halide-like expressions to candidate low-level implementations, creating a space of possible implementations.

3.3.2 Machine Learning for Compiler Optimisations

Machine learning has emerged as a viable means for automatically constructing heuristics for code optimisation. Its great advantage is that it can adapt to changes in the software and hardware environments as it has no a priori assumptions about their behaviour. This section provides a brief overview of the field. Comprehensive reviews by Ashouri et al. [Ash+18] and Zhang et al. [Zha+18] provide further detail.

Pioneered by Agakov et al. [Aga+06], the application of machine learning to compiler optimisation uses iterative compilation to evaluate a collection of training programs offline and gather features describing the distinguishing properties of the programs. The program features and the optimisation decisions which yield the greatest performance are combined and a model is learned. This model is then used to make predictions on unseen programs by extracting the features describing the program. Agakov et al. [Aga+06] use machine learning to guide the iterative compilation search. Stephenson, Martin, and Reilly [SMR03] use “meta optimisation” to tune compiler heuristics through an evolutionary algorithm to automate the search of the optimisation space. Kulkarni and Cavazos [KC12] formulate the phase-ordering problem as a Markov process and construct artificial neural networks to predict beneficial optimisation orderings given a characterisation of the state of code being optimised. Ashouri et al. [Ash+17] approach the phase-ordering problem by clustering optimisations and using machine learning to predict the speedup of sequences of optimisation clusters.

Lutz, Fensch, and Cole [LFC13] and past work [Cum+16a] develop domain-specific machine learning systems to optimise stencil computations on GPUs. Restricting the domain of optimisations to a single class of algorithm simplifies the learning task by limiting variance in the range of inputs. Ganapathi et al. [Gan+09] present an auto-tuner for stencil codes which achieves performance up to 18% better than that of a human expert. From a space of 10 million configurations, they evaluate the performance of a randomly selected 1500 combinations and use Kernel Canonical Correlation Analysis to build correlations between tunable parameter values and measured performance targets. Performance targets are L1 cache misses, TLB misses, cycles per thread, and power consumption. The use of KCAA restricts the scalability of their system as the

complexity of model building grows exponentially with the number of features. In their evaluation, the system requires two hours of compute time to build the KCAA model for only 400 seconds of benchmark data.

A domain-specific machine learning based auto-tuner is presented for the SkePU library in [DEK11]. SkePU is a C++ template library for data-parallel computations on GPUs. The auto-tuner predicts optimal device mapping (i.e. CPU, GPU) for a given program by predicting execution time and memory copy overhead based on problem size. Similarly, in this thesis machine learning is used to predict optimal heterogeneous device mapping, though the system is capable of making predictions for arbitrary GPU programs, it is not bound to a single template library. Moren and Gohringer [MG18] also tackle the task of mapping arbitrary OpenCL kernels to CPU/GPU using dynamic features extracted from the kernel at runtime.

Milepost GCC [Fur+11] is the first practical attempt to embed machine learning into a production compiler. It adds an interface for extracting program features and controlling optimisation passes, combined with a knowledge sharing system to distribute training data over the internet. The embedded interface exposes candidate features which may be used to apply machine learning to optimisations in GCC.

Ogilvie et al. [Ogi+17] use active learning to reduce the cost of iterative compilation by searching for points in the optimisation space which are close to decision boundaries. This reduces the cost of training compared to a random search. The approach complements the techniques presented in this thesis, enabling more efficient use of training data.

Besides compilers, there is a broad range of applications for machine learning in improving software performance. Surprising applications include the use of machine learning to replace conventional hash functions in key-value stores. Kraska et al. [Kra+18] find that replacing a cache-optimised B-Tree-Index with a deep learning model yields up to 70% speedups with a 10 \times reduction in memory on real workloads. Krishnan et al. [Kri+18] use deep reinforcement learning to optimise SQL join query implementations. When applying machine learning in a new domain, the challenge is often in finding a suitable program representation to use as the features.

3.3.2.1 Representing Programs with Features

The success of machine learning based code optimisation requires having high-quality features that capture the important characteristics of programs. There is an infinite number of potential features, and the suitability of features depends on the application

domain and the type of model used. Finding the right set of features to use for a particular case is a non-trivial, time-consuming task.

Various forms of features have been used to summarise programs. Dubach et al. [Dub+09] characterise programs using performance counters. Jiang et al. [Jia+10] extract program-level behaviours such as loop trip counts and the size of input files. Berral et al. [Ber+10] use additional runtime information such as system load.

In compiler research, the feature sets used for predictive models are often provided without explanation and rarely is the quality of those features evaluated. More commonly, an initially large, high dimensional candidate feature space is pruned through statistical analysis on training programs, or projected into a lower dimensional space. Stephenson and Amarasinghe [SA05] propose two approaches to select the most useful features from 38 candidates: the first using a Mutual Information Score to rank features, the second using a greedy feature selection. Collins et al. [Col+13] use Principal Component Analysis (PCA) to reduce a four-dimensional feature space to two dimensions, reducing the size of the space to 0.05%. Dubach et al. [Dub+07] also use PCA to reduce the dimensionality of their feature space, but determine the number of components to use such that the selected components account for some fraction of the total variance. In their case, 5 components account for 95% of the total variance. *FEAST* [Tin+17] combines a range of existing feature selection methods to select useful candidate features.

Prior works have sought to reduce the cost of feature design. Park, Cavazos, and Alvarez [PCA12] present a unique graph-based approach for feature representations. They use a Support Vector Machine (SVM) where the kernel is based on a graph similarity metric. Their technique still requires hand-coded features at the basic block level, but thereafter, graph similarity against each of the training programs takes the place of global features. Being a kernel method, it requires that training data graphs be shipped with the compiler, which may not scale as the size of the training data grows with the number of instances, and some training programs may be large. Additionally, their graph matching metric is expensive, requiring $O(n^3)$ time to compare against each training example. This thesis presents techniques to construct machine learning compiler heuristics without the need for program features. These techniques do not need any hand-built static code features, and the deployment memory footprint is constant and prediction time is linear in the length of the program, regardless of the size of the training set.

A few methods have been proposed to automatically generate features from the compiler’s intermediate representation (IR). These approaches closely tie the implementation of the predictive model to the compiler IR, which means changes to the IR will require modifications to the model. Leather, Bonilla, and O’Boyle [LBO14] use genetic programming to search for features, requiring a huge grammar to be written, some 160kB in length. Although much of this might be created from templates, selecting the right range of capabilities and search space bias is non-trivial and up to the expert. Namolaru et al. [Nam+10] express the space of features via logic programming over relations that represent information from the IRs. They greedily search for expressions that represent good features. However, this approach relies on expert selected relations, combinatorics and constraints to work. For both approaches, the search time may be significant.

Cavazos et al. [Cav+06] present a reaction-based predictive model for software-hardware co-design. Their approach profiles the target program using several carefully selected compiler options to see how a program runtime changes under these options for a given micro-architecture setting. They then use the program “reactions” to predict the best available application speedup. While their approach does not use static code features, developers must carefully select a few settings from a large number of candidate options for profiling, because poorly chosen options significantly affect the quality of the model. Moreover, the program must be run several times before optimisation, while the techniques presented in this thesis do not require the program to be profiled.

Compared to these approaches, the techniques presented in this thesis are entirely automatic and require no expert involvement. In the field of compiler optimisations, no work so far has developed deep learning methodologies for program feature generation and selection. This work is the first to do so.

3.3.2.2 Distributed Representations for Programs

This thesis presents deep learning methodologies for learning over programs, inspired by techniques developed in natural language processing. With these techniques, a program source code is tokenised into a sequence of vocabulary words, and each word in the vocabulary is mapped to a real-valued *embedding* space. There are many choices in how to construct vocabularies and embeddings. Chen and Monperrus [CM19] review some of the proposed techniques. Babii, Janes, and Robbes [BJR19] explore the impact that different permutations of vocabulary make on the convergence time of

language models for program code.

The techniques in this thesis use a hybrid character/token-level vocabulary to tokenise source code. This is to prevent the blow-up in vocabulary size that occurs from using a purely token-based vocabulary. Cvitkovic, Singh, and Anandkumar [CSA18] propose modelling vocabulary elements as nodes in a graph and then processing the graph using Graph Neural Networks; this enables learning over an unbounded vocabulary.

Mou et al. [Mou+16] derive an embedding space from the tokens in the source code of a program. Wang, Singh, and Su [WSS18] propose an embedding space extracted from program traces, rather than the syntactic structure of the program. Henkel et al. [Hen+18] use symbolic execution to abstract the program traces. Embeddings are then learned from these abstracted symbolic traces. Yin et al. [Yin+18] and Tufano et al. [Tuf+19] explore techniques for learning embedding representations of code edits.

Neural Code Comprehension [BJH18] builds on techniques proposed in Chapter 6 of this thesis to develop embeddings derived from a novel *Contextual Flow Graph* representation which contains the union of both data and control flow graphs. The embeddings are trained using a skip-gram model [Mik+13], using a vocabulary derived from LLVM bitcode. By deriving a vocabulary from a compiler intermediate representation, this enables the same embeddings to be used for any programming language for which there exists a front-end to LLVM.

3.4 Deep Learning over Programs

Deep learning is a nascent branch of machine learning in which deep or multi-level systems of processing layers are used to detect patterns in natural data [LBH15; WRX17]. Deep learning techniques for program generation and optimisation were reviewed in Section 3.2.2.3 and Section 3.3.2 respectively, but there are other applications of deep learning over programs related to this work.

The great advantage of deep learning over prior machine learning techniques is its ability to process natural data in its raw form. This overcomes the traditionally laborious and time-consuming practice of engineering feature extractors to process raw data into an internal representation or feature vector. Deep learning has successfully discovered structures in high-dimensional data and is responsible for many breakthrough achievements in machine learning such as achieving human parity in conversational speech recognition [Xio+16]; super-human level performance in video

games [Mni+15]; and autonomous vehicle control [LCW12]. The use of deep learning techniques for software engineering has long been a goal of research [Whi+15].

A 2017 survey by Allamanis et al. describes the fast-moving field of deep learning techniques for programming languages [All+17]. *AutoComment* [WYT13] mines the popular Q&A site StackOverflow to automatically generate code comments. *Naturalize* [All+14] employs techniques developed in the natural language processing domain to model coding conventions. *JSNice* [RVK15] leverages probabilistic graphical models to predict program properties such as identifier names for JavaScript. Allamanis, Peng, and Sutton [APS16] use attentional neural networks to generate summaries of source code. *Nero* [DAY19] uses an encoder-decoder architecture to predict method names in stripped binaries. The system takes as input a sequence of call sites from the execution of a binary and produces as output a predicted method name.

There is an increasing interest in mining source code repositories at a large scale [AS13; Kal+09; Whi+15]. Previous uses outside the field of machine learning have involved data mining of GitHub to analyse software engineering practices [Bai+14; GAL14; VFS15; Wu+14]. Allamanis [All18] raises concerns about code duplicates in corpora of open-source programs used for machine learning. They find that corpora often contain a high percentage of duplicate or near-duplicate code. This impacts cases where the corpus is divided into training and test sets. Duplicate code appearing both in the training and test sets leads to artificially high accuracies of models on the test set. The work in this thesis does not use open source corpora as test sets.

Machine learning has also been applied to other areas such as bug detection and static analysis. Heo, Oh, and Yi [HOY17] present a machine-learning technique to tune static analysis to be selectively unsound, based on anomaly detection techniques. Koc et al. [Koc+17] develop a classifier to predict whether the error report produced by a static analysis is a false positive based on the program structures of previous reports that produced false error reports. Reports. In ASE. Lam et al. [Lam+15] employ artificial neural networks to relate keywords in bug reports to code tokens and terms in source files and documentation to accelerate bug localisation. Wang, Liu, and Tan [WLT16] employ a deep belief network [HOT06] to automatically learn semantic features from token vectors extracted from the abstract syntax trees of programs. The features are then used for automatic defect detection. Chen et al. [Che+17] train two models on compiler test cases, one to predict whether a test case will trigger a compiler bug, the other to predict the execution of the test program. The outputs of these two models are used to schedule test cases so as to maximise the potential for exposing bugs

in the shortest amount of time. *DeepBugs* [PS18] combines a binary classification of correct and incorrect code with semantic processing to name bugs. *Code2Inv* [Si+18] uses reinforcement learning to learn loop invariants for program verification.

Machine learning has been applied to the task of automatic software repair. Monperrus surveys the literature [Mon18]. *DeepRepair* [Whi+19] uses an encoder-decoder architecture to sort code fragments according to their similarity to suspicious elements. Vasic et al. [Vas+19] train a model to jointly localise and repair variable-misuse bugs using multi-headed pointer networks. *SequenceR* [Che+18] uses sequence-to-sequence learning to generate patches. *Getafix* [Bad+19] uses a hierarchical clustering algorithm that summarises fix patterns into a hierarchy ranging from general to specific patterns. Brockschmidt et al. [Bro+18] present a novel methodology for program generation in which a graph is used as the intermediate representation.

CodeBuff [TV16] uses a hand-designed set of features to learn abstract code formatting rules from a representative corpus of programs. Raychev, Vechev, and Yahav [RVY14] use statistical models to provide contextual code completion. Gu et al. [Gu+16] use deep learning to generate example code for APIs as responses to natural language queries. Oda et al. [Oda+15] employ machine translation techniques to generate pseudo-code from source code.

3.5 Summary

This chapter has surveyed the relevant literature in the fields of program generation, program optimisation, and the rapidly evolving application of deep learning for programming languages. The next chapter presents a novel technique to improve the performance of machine learning for compiler heuristics by generating executable benchmarks using models trained on corpora of example programs.

Chapter 4

Improving the Performance of Predictive Models for Compiler Heuristics

4.1 Introduction

Predictive modelling using machine learning is an effective method for building compiler heuristics, but there is a shortage of benchmarks. Typical machine learning experiments outside of the compilation field train over thousands or millions of examples. In machine learning for compilers, however, there are typically only a few dozen common benchmarks available. This limits the quality of learned models, as they have very sparse training data for what are often high-dimensional feature spaces. What is needed is a way to generate an unbounded number of training programs that finely cover the feature space. At the same time, the generated programs must be similar to the types of programs that human developers actually write, otherwise, the learning will target the wrong parts of the feature space.

This chapter introduces *CLgen*, a generator for OpenCL benchmarks. Open source repositories are mined for program fragments which are used to automatically construct deep learning models for how humans write programs. The models are sampled to generate an unbounded number of runnable training programs. The quality of the programs is such that even human developers struggle to distinguish the generated programs from handwritten code. In this chapter, CLgen is used to automatically synthesise thousands of programs and show that learning over these improves the performance heterogeneous workloads using a state-of-the-art predictive model by

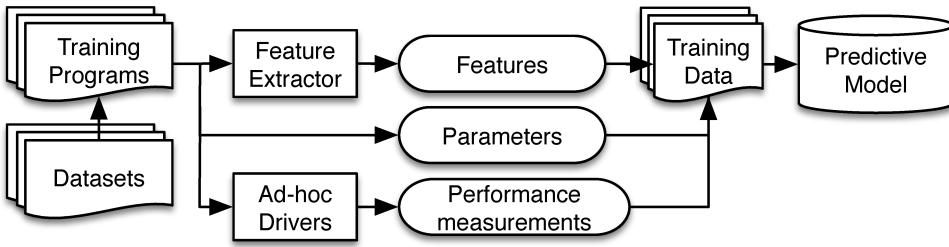


Figure 4.1: Training a predictive model for compiler optimisations. A model is constructed from training data, which comprises the features, performance measurements, and runtime parameters of training programs and their accompanying data sets.

$1.27\times$. In addition, the fine covering of the feature space automatically exposes weaknesses in the feature design which are invisible with the sparse training examples from existing benchmark suites. Correcting these weaknesses further increases workload performance by $4.30\times$.

Predictive modelling is a well-researched method for building optimisation heuristics that often exceed human experts and reduces development time [CFL12; Cum+16b; FE15; LBO14; MDO14; MSD16; Ogi+14; Wan+14; WO09; WO10; WWO14]. Figure 4.1 shows the process by which a predictive model is constructed. A set of training programs are identified that are expected to be representative of the application domain. The programs are compiled and executed with different parameter values for the target heuristic, to determine which are the best values for each training program. Each program is also summarised by a vector of features which describe the information that is expected to be important in predicting the best heuristic parameter values. These training examples of program features and desired heuristic values are used to create a machine learning model which, when given the features from a new, unseen program, can predict good heuristic values for it.

It is common for feature vectors to contain dozens of elements. This means that a large volume of training data is needed to have adequate sampling over the feature space. Without it, the machine-learned models only capture the coarse characteristics of the heuristic, and new programs which do not lie near to training points may be wrongly predicted. The accuracy of the machine-learned heuristic is thus limited by the sparsity of available training points.

There have been efforts to solve this problem using templates. The essence of the approach is to construct a probabilistic grammar with embedded semantic actions that define a language of possible programs. New programs may be created by sampling

the grammar and, through setting probabilities on the grammar productions, the sampling is biased towards producing programs from one part of the space or another. This technique is potentially completely general since a grammar might theoretically be constructed to match any desired program domain. However, despite being theoretically possible, it is not easy to construct grammars which are both suitably general and also produce programs that are in any way similar to human-written programs. It has been shown to be successful only over a highly restricted space of stencil benchmarks with little control flow or program variability [Cum+16a; FE15], or by using a very limited set of programming language features [KSX19]. But, it is not clear how much effort it will take, or even if it is possible for human experts to define grammars that are capable of producing human-like programs in more complex domains, without restrictions on programming language features.

The approach introduced in this chapter does not require an expert to define what human programs look like. Instead, the structure of programs is automatically inferred over a huge corpus of handwritten programs taken from open source projects. A probability distribution is constructed over sets of characters seen in human-written code. This distribution is sampled to generate new random programs which, because the distribution models human-written code, are indistinguishable from human code. These samples can be used to populate training data with an unbounded number of human-like programs, covering the space far more finely than either existing benchmark suites or even the corpus of open source projects. The approach is enabled by two recent developments:

The first is the breakthrough in effectiveness of deep learning for modelling complex structure in natural languages [Gra13; SVL14]. Deep learning is capable not just of learning the macro syntactical and semantic structure of programs, but also the nuances of how humans typically write code. It is truly remarkable when one considers that it is given no prior knowledge of the syntax or semantics of the language.

The second is the increasing popularity of public and open platforms for hosting software projects and source code. This popularity provides thousands of programming examples that are necessary to feed into the deep learning. These open source examples are not, sadly, as useful for directly learning the compiler heuristics since they are not presented in a uniform, runnable manner, nor do they typically have extractable test data. Preparing each of the thousands of open source projects to be directly applicable for learning compiler heuristics would be an insurmountable task. In addition to the program generator, CLgen, this chapter presents an accompanying host

driver which generates data sets for, then executes and profiles synthesised programs.

In the course of evaluating the technique against prior work, the generator is found to be useful also for evaluating the quality of features. Since the program space is covered so much more finely than in the prior work, which only used standard benchmark suites, CLgen is able to find multiple programs with identical feature values but different best heuristic values. This indicates that the features are not sufficiently discriminative and should be extended with more information to allow those programs to be separated. Doing so significantly increases the performance of the learned heuristic.

This chapter is organised as follows: first, Section 4.2 presents the motivation for the use of benchmark generators in predictive modelling. Then Section 4.3 introduces CLgen, a generator for human-like source code. Section 4.4 describes the driver for executing synthesised source code. CLgen is evaluated first through a qualitative evaluation comparing the output to handwritten code in Section 4.5 then quantitatively by extending the training set of a state-of-the-art machine learning optimisation heuristic. The setup of the quantitative experiments is described in Section 4.6 and the results in Section 4.7. Finally, Section 4.8 concludes this chapter.

4.2 The Case for Generating Benchmarks

This section makes the argument for synthetic benchmarks. Frequently used benchmark suites were identified in a survey of 25 GPGPU performance tuning research papers from four top tier conferences between 2013–2016: CGO, HiPC, PACT, and PPoPP. The average number of benchmarks used in each paper is 17, and a small pool of benchmark suites account for the majority of results, illustrated in Figure 4.2. The performance of the state-of-the-art Grewe, Wang, and O’Boyle [GWO13] predictive model was evaluated across each of the 7 most frequently used benchmark suites (accounting for 94% of results in the surveyed papers). The model predicts whether running an OpenCL kernel on the GPU provides better performance than on the CPU. The full experimental setup is described in Section 4.6.

Table 4.1 summarises the results. The performance of a model trained on one benchmark suite and used to predict the mapping for another suite is generally poor. The benchmark suite which provides the best results, NVIDIA SDK, achieves on average only 49% of the optimal performance. The worst case is when training with Parboil to predict the optimal mappings for Polybench, where the model achieves only 11.5% of the optimal performance. From this, it is clear that models trained on one

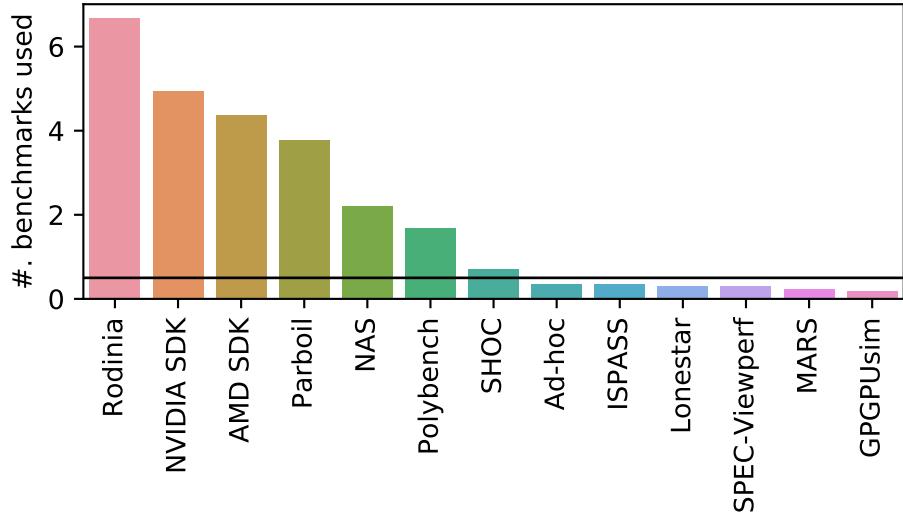


Figure 4.2: The average number of benchmarks used in GPGPU research papers published between 2013-2016 in CGO, HiPC, PACT, and PPoPP conferences. The average GPGPU research paper uses 17 benchmarks, with the seven most popular benchmark suites accounting for 94% of results.

	AMD	NPB	NVIDIA	Parboil	Polybench	Rodinia	SHOC
AMD	-	38.0%	74.5%	76.7%	21.7%	45.8%	35.9%
NPB	22.7%	-	45.3%	36.7%	13.4%	16.1%	23.7%
NVIDIA	29.9%	37.9%	-	21.8%	78.3%	18.1%	63.2%
Parboil	89.2%	28.2%	28.2%	-	41.3%	73.0%	33.8%
Polybench	58.6%	30.8%	45.3%	11.5%	-	43.9%	12.1%
Rodinia	39.8%	36.4%	29.7%	36.5%	46.1%	-	59.9%
SHOC	42.9%	71.5%	74.1%	41.4%	35.7%	81.0%	-

Table 4.1: Performance relative to the optimal of the Grewe, Wang, and O’Boyle predictive model across different benchmark suites on an AMD GPU. The columns show the suite used for training; the rows show the suite used for testing. On average, a predictive model trained on one benchmark suite and tested on another achieves only 49% of the optimal performance.

benchmark suite fail to generalise across other suites.

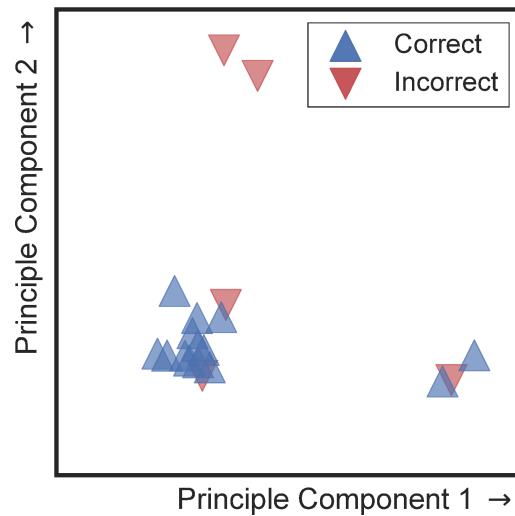
This problem is caused both by the limited number of benchmarks contained in each suite, and the distribution of benchmarks within the feature space. Figure 4.3 shows the feature space of the Parboil benchmark suite, showing whether, for each benchmark, the model was able to correctly predict the appropriate optimisation. Principal Component Analysis is used to reduce the multi-dimensional feature space to aid visualisation.

As seen in Figure 4.3a, there is a dense cluster of neighbouring benchmarks, a smaller cluster of three benchmarks, and two outliers. The lack of neighbouring observations means that the model is unable to learn a good heuristic for the two outliers, which leads to them being incorrectly optimised. In Figure 4.3b, additional benchmarks which are neighbouring in the feature space were hand-selected and the model retrained. The addition of these observations (and the information they provide about that part of the feature space) causes the two outliers to be correctly optimised. Such outliers can be found in all of the benchmark suites of Table 4.1.

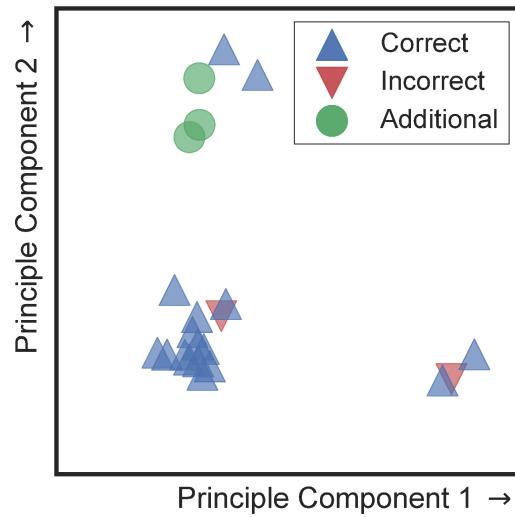
These results highlight the significant effect that the number and distribution of training programs have on the quality of predictive models. Without good coverage of the feature space, any machine learning methodology is unlikely to produce high-quality heuristics, suitable for general use on arbitrary real applications, or even applications from different benchmark suites. The novel approach described in the next section addresses this problem by generating an unbounded number of programs to cover the feature space with fine granularity.

4.3 CLgen: A System for Generating OpenCL Benchmarks

This section introduces CLgen, an undirected, general-purpose program synthesizer. It adopts and augments recent advanced techniques from deep learning to learn over massive code-bases. In contrast to existing grammar- and template-based approaches, CLgen is entirely probabilistic. The system *learns* to program using recurrent neural networks which model the semantics and usage of a huge corpus of code fragments in the target programming language.



(a)



(b)

Figure 4.3: A two dimensional projection of the Grewe, Wang, and O’Boyle feature space, showing predictive model results over Parboil benchmarks on an NVIDIA GPU. Two outliers in (a) are incorrectly predicted due to the lack of nearby observations. The addition of neighbouring observations in (b) corrects this.

4.3.1 Overview

Figure 4.4 provides an overview of the program synthesis and execution pipeline. CLgen learns the semantics and structure from over a million lines of handwritten code from GitHub, and synthesises programs through a process of iterative model sampling. A host driver, described in Section 4.4, executes the synthesised programs to gather performance data for use in predictive modelling. While the approach is demonstrated using OpenCL, it is language-agnostic. This approach extends the state-of-the-art by providing a general-purpose solution for benchmark synthesis, leading to better and more accurate predictive models.

Section 4.3.2 describes the assembly of a language corpus, Section 4.3.3 describes the application of deep learning over this corpus, and Section 4.3.4 describes the process of synthesising programs.

4.3.2 An OpenCL Language Corpus

Deep learning requires large data sets [LBH15]. For the purpose of modelling a programming language, this means assembling a large collection of real, handwritten source codes. OpenCL codes are assembled by mining public repositories on the popular code hosting site GitHub.

This is itself a challenging task since OpenCL is an embedded language, meaning device code is often difficult to untangle since GitHub does not presently recognise it as a searchable programming language. A search engine was developed which attempts to identify and download standalone OpenCL files through a process of file scraping and recursive header inlining. The result is a 2.8 million line data set of 8078 “content files” which potentially contain OpenCL code, originating from 793 GitHub repositories.

The raw data set extracted from GitHub is then pruned using a custom toolchain developed for rejection filtering and code rewriting, built on LLVM.

Rejection Filter The rejection filter accepts as input a content file and returns whether or not it contains compilable, executable OpenCL code. To achieve this, it attempts to compile the input to NVIDIA PTX byte code and performs static analysis to ensure a minimum static instruction count of three. Any inputs which do not compile or contain fewer than three instructions are discarded.

During initial development, it became apparent that isolating the OpenCL device code leads to a higher-than-expected discard rate (that is, seemingly valid OpenCL files

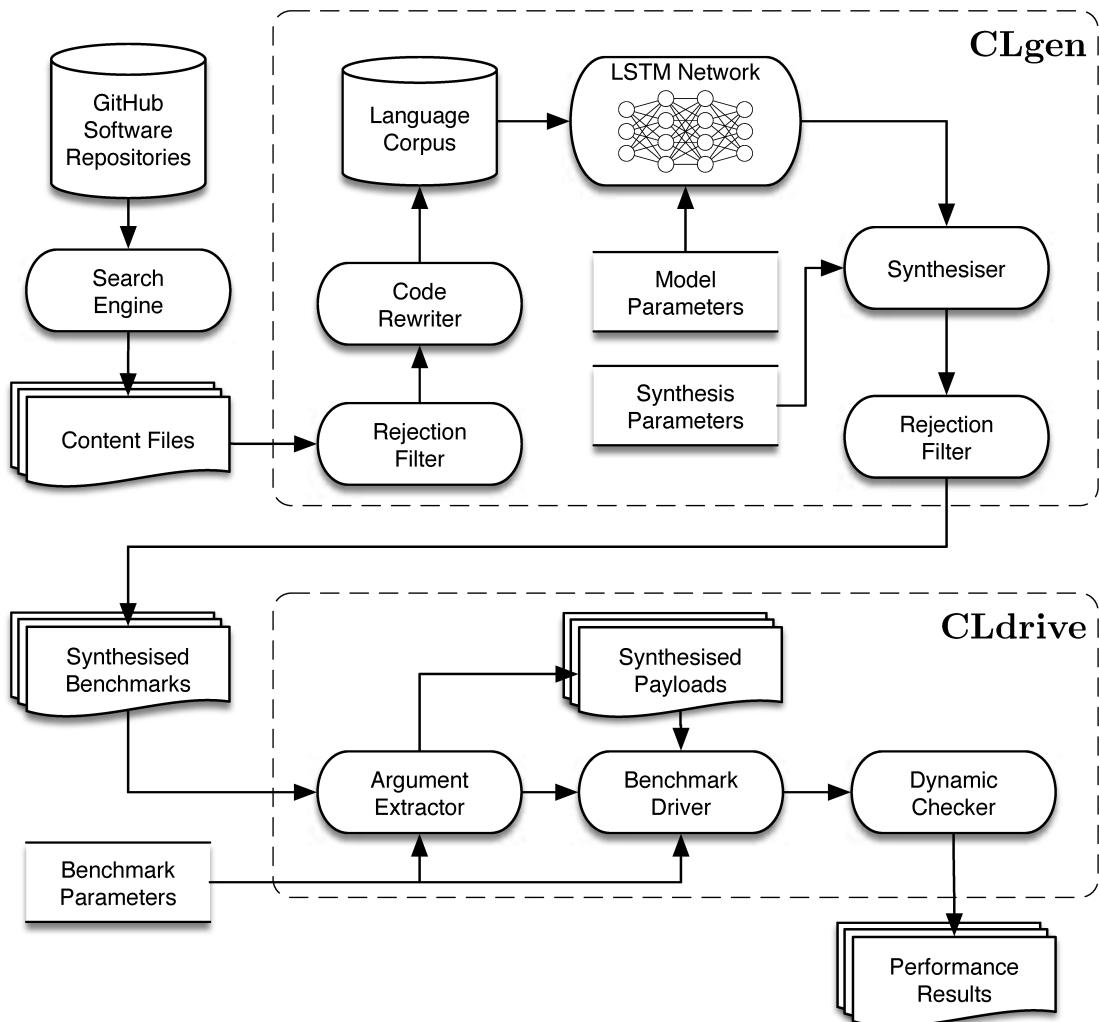


Figure 4.4: The benchmark synthesis and execution pipeline. Software is mined from GitHub; this is used to construct a language model from which new programs may be synthesised; a benchmark driver is used to produce performance results.

being rejected). Through analysing 148k lines of compilation errors, a large number of failures were discovered to be caused by undeclared identifiers, a result of isolating device code. 50% of undeclared identifier errors in the GitHub dataset were caused by only 60 unique identifiers. To address this, a *shim header* was developed which contains inferred values for common type definitions (e.g. `FLOAT_T`), and common constants (e.g. `WGSIZE`), shown in Listing 1.

Injecting the shim decreases the discard rate from 40% to 32%, responsible for an additional 88k lines of code in the final language corpus. The resulting data set is 2.0 million lines of compilable OpenCL source code.

Code Rewriter Programming languages have few of the issues of semantic interpretation present in natural language, though there remain many sources of variance at the syntactic level. For example, the presence and content of comments in code, and the choice of identifying names given to variables. For the purposes of generative modelling, these ambiguities are considered to be *non-functional variance*. The *code rewriter* is a tool developed to normalise code of these variances so as to make code more amenable to machine learning. This is a three-step process:

1. The source file is pre-processed using the compiler front-end to remove macros, conditional compilation, and source comments.
2. Identifiers are rewritten to have a short but unique name based on their order of appearance, using the sequential series $\{a, b, c, \dots, aa, ab, ac, \dots\}$ for variables and $\{A, B, C, \dots, AA, AB, AC, \dots\}$ for functions. This process isolates the syntactic structure of the code, and unlike prior work [AS13], this rewrite method preserves program behaviour. Language built-ins (e.g. `get_global_id`, `asin`) are not rewritten.
3. A variant of the Google C++ code style [Wei+11] is enforced to ensure consistent use of braces, parentheses, and white space.

An example of the code rewriting process is shown in Listings 2 and 3. A side effect of this process is a reduction in code size, largely due to the removal of comments and excess white space. The final language corpus contains 1.3 million lines of transformed OpenCL, consisting of 9487 kernel functions. Identifier renaming reduces the *bag-of-words* vocabulary size — the number of unique entries in the tokenised corpus — by 84%.

```
1  /* Enable OpenCL features */
2  #define cl_clang_storage_class_specifiers
3  #define cl_khr_fp64
4  #include <clc/clc.h>
5
6  /* Inferred types */
7  typedef float FLOAT_T;
8  typedef float FLOAT_TYPE;
9  typedef float FPTYPE;
10 typedef unsigned int INDEX_TYPE;
11 ... (34 more)
12
13 /* Inferred constants */
14 #define M_PI 3.14025
15 #define WG_SIZE 128
16 #define IMAGEH 512
17 #define IMAGEW 1024
18 #define INPUT_HEIGHT 512
19 #define INPUT_WIDTH 1024
20 #define ZERO 0
21 #define ONE 1
22 ... (177 more)
```

Listing 1: An overview of the *shim* header file, providing commonly used type aliases and constants for compiling OpenCL files taken on GitHub.

```

1 // saxpy.cl - Compute kernel for SAXPY
2 #define DTTYPE float
3 #define ALPHA(a) 3.5f * a
4 inline DTTYPE ax(DTTYPE x) { return ALPHA(x); }
5
6 kernel void saxpy( /* SAXPY kernel */
7     global DTTYPE *input1,
8     global DTTYPE *input2,
9     const int nelem)
10 {
11     unsigned int idx = get_global_id(0);
12     if (idx < nelem) { // = ax + y
13         input2[idx] += ax(input1[idx]); } }
```

Listing 2: An example OpenCL content file prior to code rewriting.

```

1 inline float A(float a) {
2     return 3.5f * a;
3 }
4
5 kernel void B(global float* b, global float* c, const int
→ d) {
6     unsigned int e = get_global_id(0);
7
8     if (e < d) {
9         c[e] += A(b[e]);
10    }
11 }
```

Listing 3: The example OpenCL content file of Listing 2 after code rewriting. Conditional compilation has been removed, the variables and functions renamed, and a code style enforced.

4.3.3 Learning OpenCL

Generating valid, executable program code is an ambitious and challenging goal for unsupervised machine learning. CLgen employs state-of-the-art deep language modelling techniques to achieve this task.

The Long Short-Term Memory (LSTM) [HS97] architecture of Recurrent Neural Network [Mik10; SSN12] is used to learn a character-level language model over the corpus of OpenCL compute kernels. The LSTM network architecture comprises recurrent layers of *memory cells*, each consisting of input, output, and forget gates, and an output layer providing normalised probability values from a 1-of-K coded vocabulary [GS05].

A 3-layer LSTM network is used with 2048 nodes per layer, implemented in Torch. This 17-million parameter model is trained using *Stochastic Gradient Descent* for 50 epochs, using an initial learning rate of 0.002, decaying by a factor of one half every 5 epochs. Training took three weeks on a single machine using an NVIDIA GTX Titan, with a final model size of 648MB¹. Training the network is a one-off cost, and is parallelisable across devices. The trained network can be deployed to lower-compute machines for use.

4.3.4 Synthesising Source Code

OpenCL compute kernels are synthesised by iteratively sampling the learned language model. Two modes for model sampling are supported: the first involves providing an *argument specification*, stating the data types and modifiers of all kernel arguments. When an argument specification is provided, the model synthesises kernels matching this signature. In the second sampling mode, this argument specification is omitted, allowing the model to synthesise compute kernels of arbitrary signatures, dictated by the distribution of argument types within the language corpus.

In either mode, a *seed* text is generated and the model sampled, character by character, until the end of the compute kernel is reached, or until a predetermined maximum number of characters is reached. Algorithm 1 illustrates this process. The same rejection filter as is used during corpus assembly that either accepts or rejects the sample as a candidate synthetic benchmark. Listings 4, 5, and 6 show three examples of unique compute kernels generated in this manner from an argument specification of

¹No effort was made to minimise training time. Subsequent work using tuned parameters and a more efficient model implementation in TensorFlow has reduced this time considerably.

Algorithm 1 Using an LSTM model to sample a candidate OpenCL kernel.

Require: LSTM model M , maximum kernel length n .

Ensure: Completed sample string S .

```

1:  $S \leftarrow \text{"\_kernel void A(const int a) {”}$                                 ▷ Seed text
2:  $d \leftarrow 1$                                                                ▷ Initial code block depth
3: for  $i \leftarrow |S|$  to  $n$  do
4:    $c \leftarrow \text{PredictCharacter}(M, S)$                                      ▷ Generate new character
5:   if  $c = \{$  then
6:      $d \leftarrow d + 1$                                                  ▷ Entered code block, increase depth
7:   else if  $c = \}$  then
8:      $d \leftarrow d - 1$                                                  ▷ Exited code block, decrease depth
9:   end if
10:   $S \leftarrow S + c$                                               ▷ Append new character to sample
11:  if  $depth = 0$  then
12:    break                                                       ▷ Exited function block, stop sampling
13:  end if
14: end for

```

three single-precision floating-point arrays and a read-only signed integer. The quality of synthesised code is evaluated in Section 4.5.

4.4 CLdrive: A System for Driving Arbitrary OpenCL Kernels

This section presents *CLdrive*, a host driver developed to gather performance data from synthesised CLgen code. The driver accepts as input an OpenCL kernel, generates *payloads* of user-configurable sizes, then executes the kernel using the generated payloads so as to collect kernel runtimes, and to provide dynamic checking of kernel behaviour.

4.4.1 Generating Data Payloads

A *payload* encapsulates all of the arguments of an OpenCL compute kernel. After parsing the input kernel to derive argument types, a rule-based approach is used to generate synthetic payloads. For a given global size S_g : host buffers of S_g elements are allocated and populated with random values for global pointer arguments, device-only

```

1  kernel void A(global float* a,
2                  global float* b,
3                  global float* c,
4                  const int d) {
5
6      int e = get_global_id(0);
7      float f = 0.0;
8
9      for (int g = 0; g < d; g++) {
10         c[g] = 0.0f;
11     }
12
13     barrier(1);
14
15
16     a[get_global_id(0)] = 2*b[get_global_id(0)];
17 }
```

Listing 4: CLgen-synthesised vector operation with branching and synchronisation.

```

1  kernel void A(global float* a,
2                  global float* b,
3                  global float* c,
4                  const int d) {
5
6      int e = get_global_id(0);
7
8      if (e >= d) {
9          return;
10     }
11
12     c[e] = a[e] + b[e] + 2 * a[e] + b[e] + 4;
13 }
```

Listing 5: CLgen-synthesised zip operation which computes $c_i = 3a_i + 2b_i + 4$.

```

1  kernel void A(global float* a,
2               global float* b,
3               global float* c,
4               const int d) {
5
6     unsigned int e = get_global_id(0);
7     float16 f = (float16) (0.0);
8
9     for (unsigned int g = 0; g < d; g++) {
10
11        float16 h = a[g];
12
13        f.s0 += h.s0;
14
15        f.s1 += h.s1;
16
17        f.s2 += h.s2;
18
19        f.s3 += h.s3;
20
21        f.s4 += h.s4;
22
23        f.s5 += h.s5;
24
25        f.s6 += h.s6;
26
27        f.s7 += h.s7;
28
29        f.s8 += h.s8;
30
31        f.s9 += h.s9;
32
33        f.sA += h.sA;
34
35        f.sB += h.sB;
36
37        f.sC += h.sC;
38
39        f.sD += h.sD;
40
41        f.sE += h.sE;
42
43        f.sF += h.sF;
44
45    }
46
47    b[e] = f.s0 + f.s1 + f.s2 + f.s3 + f.s4 + f.s5 + f.s6 + f.s7 +
48          ↵ f.s8 + f.s9 + f.sA + f.sB + f.sC + f.sD + f.sE + f.sF;
49
50 }
```

Listing 6: CLgen-synthesised partial reduction over reinterpreted vector type.

buffers of S_g elements are allocated for local pointer arguments, integral arguments are given the value S_g , and all other scalar arguments are given random values. Host to device data transfers are enqueued for all non-write-only global buffers, and all non-read-only global buffers are transferred back to the host after kernel execution.

4.4.2 Dynamic Checker

A class of programs are defined as performing *useful work* if they predictably compute some result. For the purpose of performance benchmarking the values computed by a program are of little interest, so a low-overhead runtime behaviour check is used to validate that a synthesised program does useful work based on the outcome of four executions of a tested program:

1. Create 4 equal size payloads $A_{1in}, B_{1in}, A_{2in}, B_{2in}$, subject to restrictions: $A_{1in} = A_{2in}, B_{1in} = B_{2in}, A_{1in} \neq B_{1in}$.
2. Execute kernel k 4 times: $k(A_{1in}) \rightarrow A_{1out}, k(B_{1in}) \rightarrow B_{1out}, k(A_{2in}) \rightarrow A_{2out}, k(B_{2in}) \rightarrow B_{2out}$.
3. Assert that:
 - $A_{1out} \neq A_{1in}$ and $B_{1out} \neq B_{1in}$, else k has no output (for these inputs).
 - $A_{1out} \neq B_{1out}$ and $A_{2out} \neq B_{2out}$, else k is input insensitive t (for these inputs).
 - $A_{1out} = A_{2out}$ and $B_{1out} = B_{2out}$, else k is non-deterministic.

Equality checks for floating point values are performed with an appropriate epsilon to accommodate rounding errors, and a timeout threshold is also used to catch kernels which are non-terminating. The method is based on random differential testing [McK98], though I emphasise that this is not a general purpose approach and is tailored specifically for benchmarking for performance characterisation. For example, one would anticipate a false positive rate for kernels with subtle sources of non-determinism which more thorough methods may expose [BCD12; PM15; SD16], however for the purpose of performance modelling, such methods were deemed unnecessary.

4.5 Qualitative Evaluation of Generated Programs

This section evaluates the quality of programs synthesised by CLgen by their likeness to handwritten code.

4.5.1 Methodology

Judging whether a piece of code has been written by a human is a challenging task for a machine, so a methodology was adopted from artificial intelligence research based on the *Turing Test* [Gao+15; Vin+15; ZIE16]. If the output of CLgen is human-like code, it reasons that a human judge will be unable to distinguish it from handwritten code.

A double-blind test was devised in which 15 volunteer OpenCL developers from industry and academia were shown 10 OpenCL kernels each. Participants were tasked with judging whether, for each kernel, they believed it to have been written by hand or by machine. Kernels were randomly selected for each participant from two equal sized pools of synthetically generated and handwritten code from GitHub. The samples from GitHub were vetted to ensure that they were indeed handwritten and not generated by machine or template (such vetting is a manual process and was not applied during the assembly of the model training corpus). The code rewriting process was applied to all kernels to remove comments and ensure uniform identifier naming. The participants were divided into two groups of 10 and 5 members, with the larger group reviewing synthetic code generated CLgen. The smaller group acted as a control group, reviewing synthetic code generated by CLSmith [Lid+15], an OpenCL program generator for differential testing².

4.5.2 Experimental Results

Each participant’s answers were scored. The average score of the control group is 96% (stdev. 9%), an unsurprising outcome as programs generated using CLSmith for testing have multiple “tells”; for example, they make much heavier use of structs than is typical, they use unusual combinations of programming language features, and their only input is a single `ulong` pointer. There were no false positives (synthetic code labelled human) for CLSmith, only false negatives (human code labelled synthetic).

²An online version of this test is available at https://chriscummins.cc.uk/s/human_or_robot/.

With CLgen synthesised programs, the average score was 52% (stdev. 17%), and the ratio of errors was even. This suggests that CLgen code is indistinguishable from handwritten programs, with human judges scoring no better than random chance.

4.6 Experimental Methodology

The synthetic benchmark generator described in this chapter aims to improve the performance predictive models by augmenting their training data so as to provide a finer grained exploration of the feature space than would otherwise be possible. To test the hypothesis that CLgen-generated benchmarks are *useful for training*, an experiment was designed in which a state-of-the-art predictive model was trained and evaluated with and without the addition of synthetic benchmarks and the performance compared.

4.6.1 Experimental Setup

Predictive Model To evaluate the efficacy of synthetic benchmarks for training, the predictive model of Grewe, Wang, and O’Boyle is used [GWO13]. The predictive model is used to determine the optimal mapping of a given OpenCL kernel to either a GPU or CPU. It uses supervised learning to construct a decision tree with a combination of static and dynamic kernel features extracted from source code and the OpenCL runtime, detailed in Table 4.2.

Benchmarks As in [GWO13], the model is tested on the NAS Parallel Benchmarks (NPB) [Bai+91]. The hand-optimised OpenCL implementation of Seo, Jo, and Lee [SJL11] is used. In [GWO13] the authors augment the training set of the predictive model with 47 additional kernels taken from 4 GPGPU benchmark suites. To more fully sample the program space, a much larger collection of 142 kernels is used, summarised in Table 4.3. These additional programs are taken from all 7 of the most frequently used benchmark suites identified in Section 4.2. None of these programs were used to train CLgen. This brings the total number of OpenCL benchmark kernels used in the evaluation to 256. 1,000 kernels were synthesised with CLgen to use as additional benchmarks.

Experimental Platforms Two 64-bit CPU-GPU systems are used to evaluate the approach, detailed in Table 4.4. One system has an AMD GPU and uses OpenSUSE 12.3;

Name	Type	Description
comp	static	#. compute operations
mem	static	#. accesses to global memory
localmem	static	#. accesses to local memory
coalesced	static	#. coalesced memory accesses
transfer	dynamic	size of data transfers
wgsize	dynamic	#. work-items per kernel

(a) Individual code features

Name	Formulation	Description
F1	transfer / (comp+mem)	Communication-computation ratio
F2	coalesced/mem	% Coalesced memory accesses
F3	(localmem/mem) × wgsize	Memory access ratio × #. work-items
F4	comp/mem	Computation-memory ratio

(b) Combinations of raw features

Table 4.2: Features used by Grewe, Wang, and O’Boyle to predict CPU/GPU mapping of OpenCL kernels. The features are extracted using a custom analysis pass built using LLVM.

	Version	#. benchmarks	#. kernels
NPB (SNU [SJL11])	1.0.3	7	114
Rodinia [Che+09]	3.1	14	31
NVIDIA SDK	4.2	6	12
AMD SDK	3.0	12	16
Parboil [Str+12]	0.2	6	8
PolyBench [Gra+12]	1.0	14	27
SHOC [Dan+10]	1.1.5	12	48
Total	-	71	256

Table 4.3: List of benchmarks used to train and evaluate the Grewe, Wang, and O’Boyle predictive model.

	Intel CPU	AMD GPU	NVIDIA GPU
Model	Core i7-3820	Tahiti 7970	GTX 970
Frequency	3.6 GHz	1000 MHz	1050 MHz
#. Cores	4	2048	1664
Memory	8 GB	3 GB	4 GB
Throughput	105 GFLOPS	3.79 TFLOPS	3.90 TFLOPS
Driver	AMD 1526.3	AMD 1526.3	NVIDIA 361.42
Compiler	GCC 4.7.2	GCC 4.7.2	GCC 5.4.0

Table 4.4: Experimental platforms used to evaluate the Grewe, Wang, and O’Boyle predictive model.

the other is equipped with an NVIDIA GPU and uses Ubuntu 16.04. Both platforms were unloaded.

Data sets The NPB and Parboil benchmark suites are packaged with multiple data sets. All of the packaged data sets are used (5 per program in NPB, 1-4 per program in Parboil). For all other benchmarks, the default data sets are used. The CLgen host driver was configured to synthesise payloads between 128B-130MB, approximating that of the dataset sizes found in the benchmark programs.

4.6.2 Methodology

The same methodology is used as in [GWO13]. Each experiment is repeated five times and the average execution time is recorded. The execution time includes both device compute time and the data transfer overheads.

Models are evaluated using *leave-one-out cross-validation*. For each benchmark, a model is trained on data from all other benchmarks and used to predict the mapping for each kernel and dataset in the excluded program. The process is repeated with and without the addition of synthetic benchmarks in the training data. Only the real handwritten benchmarks are used to test model predictions, the synthetic benchmarks are not used.

4.7 Experimental Results

The effectiveness of synthetic benchmarks is evaluated on two heterogeneous systems. First, the performance of a state-of-the-art predictive model [GWO13] is compared with and without the addition of synthetic benchmarks, then synthetic benchmarks are shown to expose weaknesses in the feature design and how these can be addressed to develop a better model. Finally, the ability of CLgen to explore the program feature space is compared against a state-of-the-art program generator.

4.7.1 Performance Evaluation

Figure 4.5 shows speedups of the Grewe, Wang, and O’Boyle predictive model over the NAS Parallel Benchmark suite with and without the addition of synthesised benchmarks for training. Speedups are calculated relative to the best single-device mapping for each experimental platform, which is CPU-only for AMD and GPU-only for NVIDIA. The fine-grained coverage of the feature space which synthetic benchmarks provide improves performance dramatically for the NAS benchmarks. Across both systems, an average speedup of $2.42\times$ is achieved with the addition of synthetic benchmarks, with prediction improvements over the baseline for 62.5% of benchmarks on AMD and 53.1% on NVIDIA.

The strongest performance improvements are on NVIDIA with the FT benchmark which suffers greatly under a single-device mapping. However, the performance on AMD for the same benchmark slightly degrades after adding the synthetic benchmarks. This issue is addressed in the next section.

4.7.2 Extending the Predictive Model

Feature designers are bound to select as features only properties which are significant for the handful of benchmarks they test on, which limits a model’s ability to generalise over a wider range of programs. This is found to be the case with the Grewe, Wang, and O’Boyle model. The addition of automatically generated programs exposed two distinct cases where the model failed to generalise as a result of overspecialising to the NPB suite.

The first case is that the feature F3 of Table 4.2 is sparse on many programs. This is a result of the NPB implementation’s heavy exploitation of local memory buffers and the method by which they combined features (speculatively, this may have been

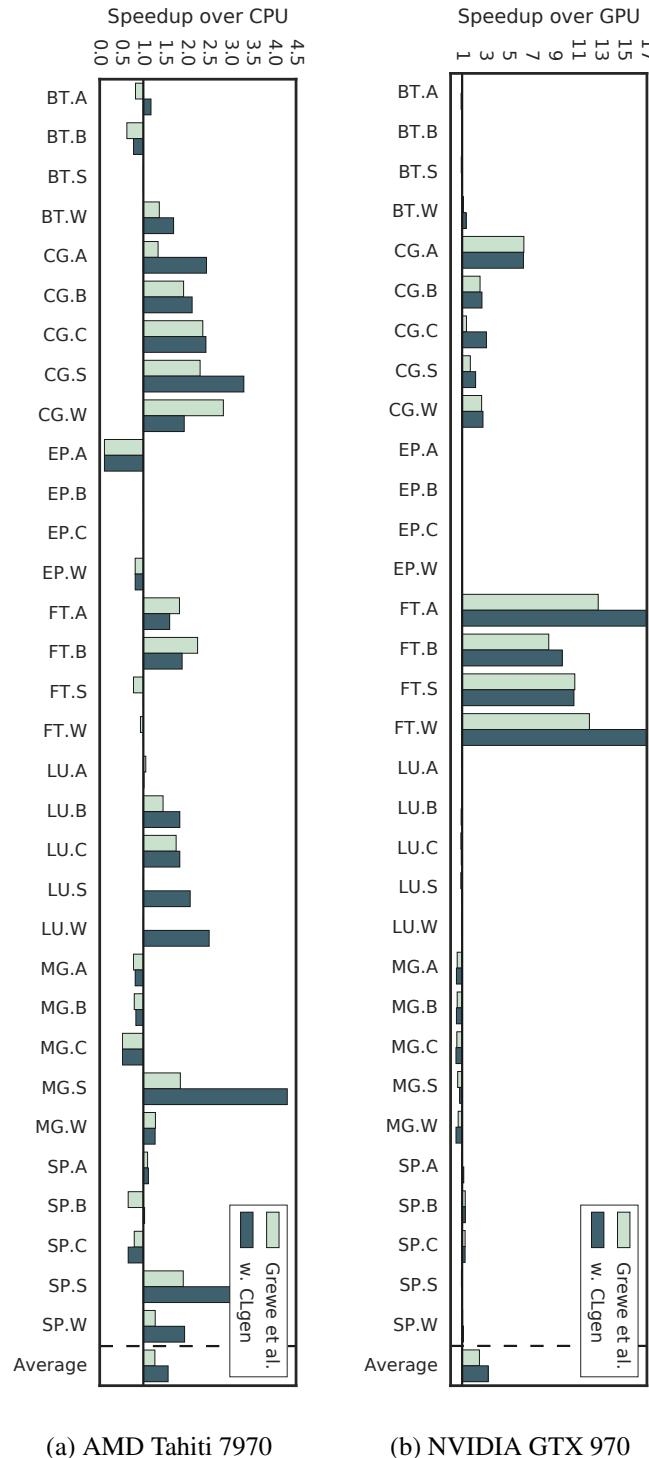


Figure 4.5: Speedup of programs using Grewe, Wang, and O'Boyle predictive model with and without synthetic benchmarks. The predictive model outperforms the best static device mapping by a factor of $1.26\times$ on AMD and $2.50\times$ on NVIDIA. The addition of synthetic benchmarks improves the performance to $1.57\times$ on AMD and $3.26\times$ on NVIDIA.

a necessary dimensionality reduction in the presence of sparse training programs). A simple countermeasure is taken to address this by extending the model to use the raw feature values in addition to the combined features.

The second case is that some CLgen-generated programs had identical feature values as in the benchmark set, but had different *behaviour* (i.e. optimal mappings). Listing 8 shows one example of a CLgen benchmark which is indistinguishable in the feature space to one of the existing benchmarks — AMD’s Fast Walsh-Hadamard transform, Listing 7 — but with different behaviour. Comparing the program fragments reveals two primary differences: Listing 7 contains indirect memory accesses that cannot be coalesced, and Listing 8 contains a branching operation whereas Listing 7 has linear control flow. Neither of these properties are captured by the features used in the Grewe, Wang, and O’Boyle model. In the case of the branching operation, we can speculate that the choice not to consider control flow was an intentional decision by the authors given that the NPB programs on which the model was initially evaluated on are implemented in a manner which aggressively minimises branching. However, as demonstrated through CLgen’s automatic exploration of the program space, such a decision hinders the model’s ability to generalize to a range of programs. To counter this the predictive model was extended with an additional feature containing a static count of branching operations in a kernel.

```

1 kernel
2 void fastWalshTransform(global float * tArray,
3                         const int    step
4                         )
5 {
6     unsigned int tid = get_global_id(0);
7
8     const unsigned int group = tid%step;
9     const unsigned int pair  = 2*step*(tid/step) + group;
10
11    const unsigned int match = pair + step;
12
13    float T1          = tArray[pair];
14    float T2          = tArray[match];
15
16    tArray[pair]         = T1 + T2;
17    tArray[match]        = T1 - T2;
18 }
```

Listing 7: AMD’s Fast Walsh Transform benchmark kernel. In the Grewe, Wang, and O’Boyle feature space this is indistinguishable from the CLgen program of Listing 8, but has very different runtime behaviour and optimal device mapping. The addition of a branching feature fixes this.

```

1  kernel void A(global float* a,
2               global float* b,
3               global float* c,
4               const int d) {
5
6     int e = get_global_id(0);
7
8     if (e < 4 && e < c) {
9
10       c[e] = a[e] + b[e];
11       a[e] = b[e] + 1;
12     }
13 }
```

Listing 8: In the Grewe, Wang, and O’Boyle feature space, this CLgen program is indistinguishable from the AMD Fast Walsh–Hadamard transform benchmark kernel of Listing 7, but has very different runtime behaviour and optimal device mapping. The addition of a branching feature fixes this.

Figure 4.6 shows speedups of the model with the extended feature set across all seven of the benchmark suites used in Section 4.2. Model performance, even on this tenfold increase of benchmarks, is good. There are three benchmarks on which the model performs poorly: MatrixMul, cutcp, and pathfinder. Each of those programs makes heavy use of loops, which changes the dynamic behaviour of the programs in ways that the static code features of the model are unlikely to capture. This could be addressed by extracting dynamic instruction counts using profiling, but this is beyond the scope of this work. It is not the aim of this work to perfect the predictive model but to show the performance improvements associated with training on synthetic programs. To this extent, the proposed approach is successful, achieving average speedups of $3.56\times$ on AMD and $5.04\times$ on NVIDIA across a very large test set.

4.7.3 Comparison of Source Features

As demonstrated in Section 4.2, the predictive quality of a model for a given point in the feature space is improved with the addition of observations from neighbouring points. By producing thousands of artificial programs modelled on the structure of real OpenCL programs, CLgen is able to consistently and automatically generate programs

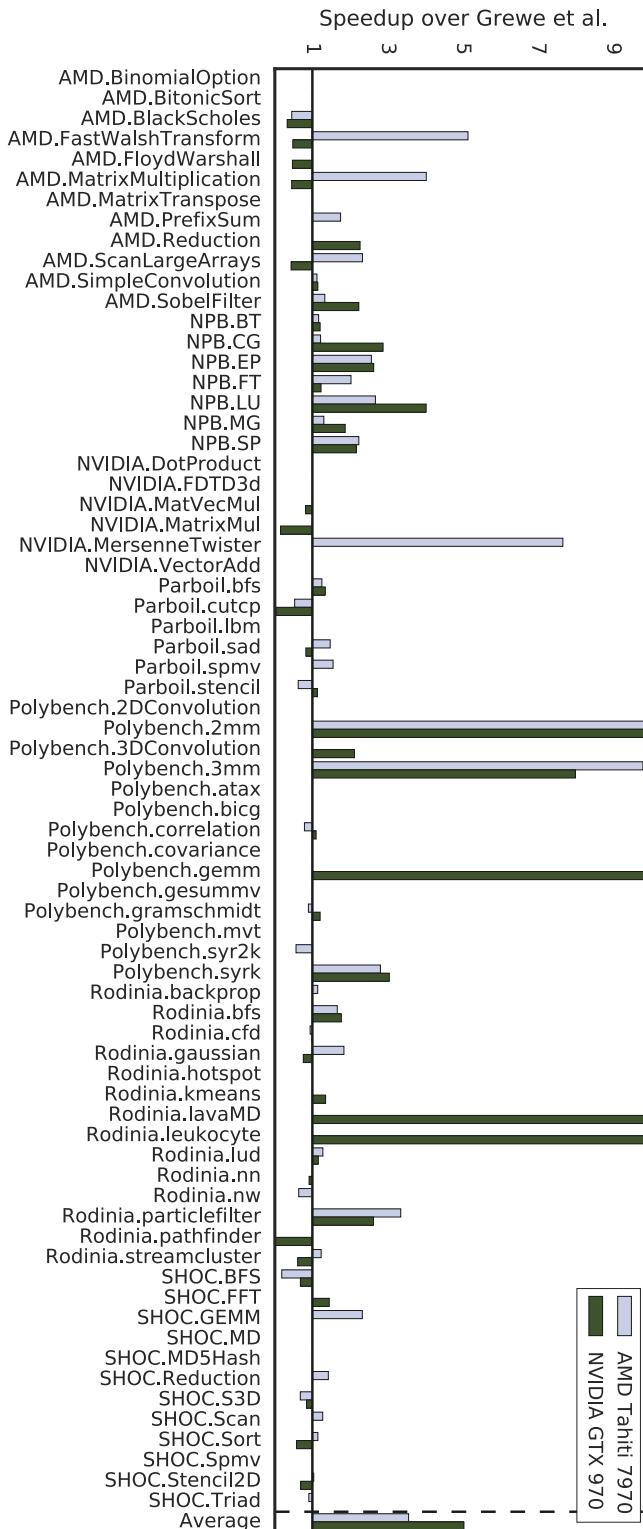


Figure 4.6: Speedups of predictions using an extended model over Grewe, Wang, and O'Boyle on both experimental platforms. Synthetic benchmarks and the additional program features outperform the original predictive model by a factor $3.56\times$ on AMD and $5.04\times$ on NVIDIA.

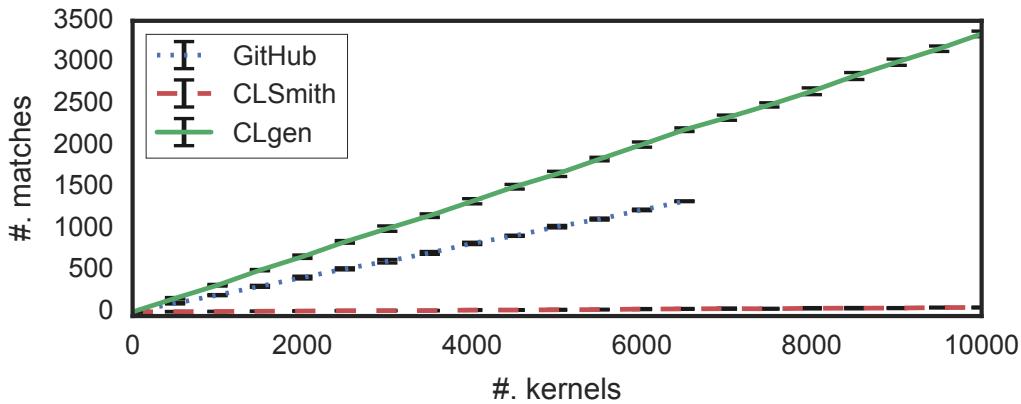


Figure 4.7: The number of kernels from GitHub, CLSmith, and CLGen with static code features matching the benchmarks. CLGen generates kernels that are closer in the feature space than CLSmith, and continues to do so long after the extent of the GitHub data set is exhausted. Error bars show the standard deviation from 10 random samplings.

which are close in the feature space to the unseen benchmarks that are in the test set.

To quantify this effect, the static code features of Table 4.2a, plus the branching feature discussed in the previous subsection, are used to measure the number of CLGen kernels generated with the same feature values as those of the benchmarks examined in the previous sections. Only static code features are examined to allow comparison with the GitHub kernels for which there is no automated method to execute and extract runtime features, and to enable comparison against code generated by CLSmith.

Figure 4.7 plots the number of exact feature vector matches as a function of the number of kernels. Out of 10,000 unique CLGen kernels, more than a third have static feature values matching those of the benchmarks, providing on average 14 CLGen kernels for each benchmark. This supports the underlying intuition: CLGen kernels, by emulating the way real humans write OpenCL programs, are concentrated in the same area of the feature space as real programs. Moreover, since the number of CLGen kernels that can be generated is unbounded, the exploration of the feature space can be continually refined. This is not the case for GitHub, where the number of kernels is finite. CLSmith rarely produces code similar to real-world OpenCL programs, with only 0.53% of the generated kernels have matching feature values with benchmark kernels. The unique contribution of CLGen is its ability to generate many thousands of programs *that are appropriate for predictive modelling*.

4.8 Summary

The quality of predictive models is bounded by the quantity and quality of programs used for training, yet there are typically only a few dozen common benchmarks available for experiments. This *data scarcity challenge* (Section 1.2.1) limits the applicability of machine learning to compiler optimisations. This chapter presents a novel tool which is the first of its kind — an entirely probabilistic program generator capable of producing an unbounded number of human-like programs. The approach applies deep learning over a huge corpus of publicly available code from GitHub to automatically infer the syntactical properties and practical usage of a programming language.

The tool generates synthetic programs which to trained eyes are indistinguishable from handwritten code. The approach is tested using a state-of-the-art predictive model, improving its performance by a factor of $1.27\times$. Synthetic benchmarks automatically exposed weaknesses in the feature set which, when corrected, further improved the performance by $4.30\times$.

Given the ability of generative models for performance characterisation, it is natural to hypothesise that a generator for unbounded programs may prove useful for compiler validation. Compared to benchmarking, compiler validation places very different requirements for how the synthesised code fragments are used. The following chapter presents techniques to adapt and extend the generative model to the challenging domain of compiler test case generation.

Chapter 5

Lowering the Cost of Compiler Validation

5.1 Introduction

Compilers should produce correct code for valid inputs, and meaningful errors for invalid inputs. Failure to do so hinders software development and even causes catastrophic runtime errors. Still, properly testing compilers is hard. Modern optimising compilers are large and complex programs, and their input space is huge. Hand-designed suites of test programs, while important, are inadequate for covering such a large space and will not touch all parts of the compiler.

Random test case generation —*fuzzing*— is a well-established and effective method for identifying compiler bugs [Che+13; Che+16a; KP05]. When fuzzing, randomly generated valid or semi-valid inputs are fed to the compiler. Any kind of unexpected behaviour, including crashes, freezes, or wrong binaries, indicates a compiler bug. While crashes and freezes in the compiler are easy to detect, determining that binaries are correctly compiled is not generally possible without either developer provided validation for the particular program’s behaviour or a gold standard compiler from which to create reference outputs. In the absence of those, Differential Testing [McK98] can be used. The generated code and a set of inputs form a *test case* which is compiled and executed on multiple *testbeds*. If the test case should have deterministic behaviour, but the output differs between testbeds, then a bug has been discovered.

Compiler fuzzing requires efficiently generating test cases that trigger compiler bugs. The state-of-the-art approach, CSmith [Yan+11], generates large random programs by defining and sampling a probabilistic grammar which covers a subset of the

C programming language. Through this grammar, CSmith ensures that the generated code easily passes the compiler front-end and stresses the most complex part of the compiler, the middle-end. Complex static and dynamic analyses make sure that programs are free from undefined behaviour. The programs are then differentially tested.

While CSmith has been successfully used to identify hundreds of bugs in otherwise-robust compilers, it and similar approaches have a significant drawback. They represent a huge undertaking and require a thorough understanding of the target programming language. CSmith was developed over the course of years and consists of over 41k lines of handwritten C++ code. By tightly coupling the generation logic with the target programming language, each feature of the grammar must be painstakingly and expertly engineered for each new target language. For example, lifting CSmith from C to OpenCL [Lid+15] — a superficially simple task — took 9 months and an additional 8k lines of code. Given the difficulty of defining a new grammar, typically only a subset of the language is implemented.

This chapter introduces *DeepSmith*, a novel machine learning approach to accelerating compiler validation through the inference of generative models for compiler inputs. DeepSmith is a fast, effective, and low effort approach to the generation of random programs for compiler fuzzing. The methodology, extending the technique developed in Chapter 4, uses recent advances in deep learning to automatically *infer* probabilistic models of how humans write code, instead of painstakingly defining a grammar to the same end. By training a deep neural network on a corpus of handwritten code, it is able to infer both the syntax and semantics of the programming language and the common constructs and patterns. The approach essentially frames the generation of random programs as a language modelling problem. This greatly simplifies and accelerates the process. The expressiveness of the generated programs is limited only by what is contained in the corpus, not the developer’s expertise or available time. Such a corpus can readily be assembled from open source repositories. Once trained, the model is used to automatically generate tens of thousands of realistic programs. Finally, established differential testing methodologies are used on them to expose bugs in compilers.

In this chapter, the approach is applied to test compilers for the OpenCL programming language. In 48 hours of automated testing of commercial and open source compilers, bugs are discovered in all of them, and 67 bug reports are submitted. The generated test cases are on average two orders of magnitude smaller than the state-of-the-art, require $3.03 \times$ less time to generate and evaluate, and expose bugs which

the state-of-the-art cannot. The random program generator, comprising only 500 lines of code, took 12 hours to train for OpenCL versus the state-of-the-art taking 9 man-months to port from a generator for C and 50,000 lines of code. This work primarily targets OpenCL, an open standard for programming heterogeneous systems, though the approach is largely language-agnostic. OpenCL is chosen for three reasons: it is an emerging standard with the challenging promise of functional portability across a diverse range of heterogeneous hardware; OpenCL is compiled “online”, meaning that even compiler crashes and freezes may not be discovered until a product is deployed to customers; and there is already a hand written random program generator for the language to compare against. With 18 lines of code, the program generator is extended to a second language, uncovering crashes in Solidity compilers in 12 hours of automated testing.

This chapter is organised as follows: Section 5.2 presents DeepSmith, a novel approach to compiler validation. Section 5.3 describes the experimental setup of an extensive evaluation of OpenCL compilers using DeepSmith. Section 5.4 evaluates the results of the experiment, with Section 5.4.5 containing preliminary results supporting DeepSmith’s potential for multi-lingual compiler fuzzing. Section 5.5 provides concluding remarks for this chapter.

5.2 DeepSmith: Compiler Fuzzing Through Deep Learning

DeepSmith is an open source framework for compiler fuzzing. Figure 5.1 provides a high-level overview. This section describes the three key components: a generative model for random programs, a test harness, and voting heuristics for differential testing.

5.2.1 Generative Model

Generating test cases for compilers is hard because their inputs are highly structured. Producing text with the right structure requires expert knowledge and a significant engineering effort, which has to be repeated from scratch for each new language. Instead, the proposed approach frames the problem as an unsupervised machine learning task, extending prior work of Chapter 4 in employing state-of-the-art deep learning techniques to build models for how humans write programs. The approach is inspired by

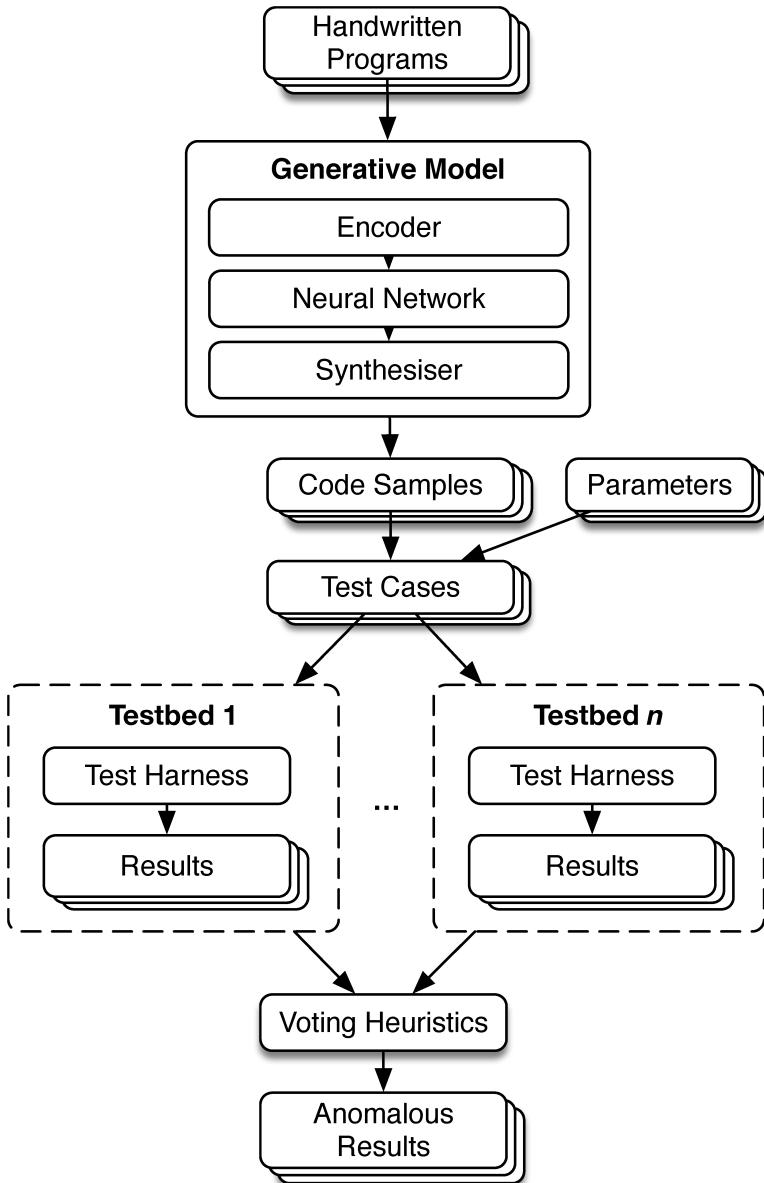


Figure 5.1: DeepSmith system overview. Handwritten programs are used to derive a generative model; from which code samples are produced and parameterised to make test cases. Test cases are broadcast to multiple testbeds, and voting heuristics used to determine the testbeds that deviate from the majority, exposing anomalous results.

breakthrough results in modelling challenging and high dimensional data sets through unsupervised learning [Bow+16; Rag+17; RJS17]. Contrary to existing tools, this approach does not require expert knowledge of the target language and is only a few hundred lines of code.

Handwritten Programs The generative model needs to be trained on a *seed corpus* of example programs. The assembly of this corpus is automated by mining 10k OpenCL kernels from open source repositories on GitHub. An *oracle compiler* (LLVM 3.9) is used to statically check each downloaded source file, discarding files that are not well-formed. The main purpose of this step is to remove the need to manually check that each file selected from GitHub does indeed contain OpenCL. A downside is that any training candidate which triggers a bug in the LLVM 3.9’s front end will not be included. However, this did not prevent the system from uncovering errors in that compiler (Section 5.4.4).

This corpus, exceeding one million lines of code, is used as a representative sample of OpenCL code from which a generative model can be derived.

As in Chapter 4, semantics-preserving transformations are employed to simplify the training programs. First, each source file is preprocessed to expand macros and remove conditional compilation and comments. Then, all user-declared identifiers are renamed using an arbitrary, but consistent pattern based on their order of declaration: $\{a, b, c, \dots, aa, ab, ac, \dots\}$ for variables and $\{A, B, C, \dots, AA, AB, AC, \dots\}$ for functions. This ensures a consistent naming convention, without modifying program behaviour. Finally, a uniform code style is enforced to ensure consistent use of braces, parentheses, and white space. These rewriting simplifications give more opportunities for the model to learn the structure and deeper aspects of the language and speed up the learning. On the other hand, some bugs in the preprocessor or front-end might no longer be discoverable. For the purpose of fuzzing OpenCL compilers, this was reasoned as an acceptable trade-off. For languages where the corpus may be many orders of magnitude larger, for example, C or Java, it may be possible to construct effective models without these modifications.

Encoder Source code is encoded as a sequence of integers for interpretation by artificial neural networks, where each integer is an index into a predetermined vocabulary. In [Joz+16], a character based vocabulary is used. This minimises the size of the vocabulary but leads to long sequences which are harder to extract structure

Algorithm 2 Deriving a hybrid token- and character-level vocabulary from a string.

Require: Candidate vocabulary V_c , string S .

Ensure: Vocabulary V .

```

1:  $V \leftarrow \emptyset$                                  $\triangleright$  Initialise empty derived vocabulary
2:  $i \leftarrow 1$ 
3: while  $S \neq \text{""}$  do                       $\triangleright$  While input not fully processed
4:    $i \leftarrow i + 1$                                  $\triangleright$  Advance to next character
5:    $c \leftarrow [S^{(1)}, \dots, S^{(i)}]$              $\triangleright$  Read token from input
6:   if  $\text{IsValidPrefix}(c, V_c)$  then         $\triangleright$  If token is not legal
7:      $c \leftarrow \text{FindLongestSubstring}(c, V_c)$      $\triangleright$  Revert to last legal token
8:      $S \leftarrow [S^{(|c|)}, \dots, S^{(|S|)}]$          $\triangleright$  Pop token from input
9:      $V \leftarrow V \cup \{c\}$                            $\triangleright$  Add token to vocabulary
10:     $i \leftarrow 1$ 
11:  end if
12: end while

```

from. In [AS13], a token based vocabulary is used. This leads to shorter sequences, but causes an explosion in the vocabulary size, as every identifier and literal must be represented uniquely.

A hybrid, partially tokenised vocabulary approach is developed. This allows common multi-character sequences such as `float` and `if` to be represented as unique vocabulary items, while literals and other infrequently used words are encoded at the character level.

First, a candidate vocabulary V_c is assembled for the OpenCL programming language containing the 208 data types, keywords, and language builtins of the OpenCL programming language. From this, the subset of the candidate vocabulary $V \in V_c$ which is required to encode a corpus of 45k lines of GPGPU benchmark suite kernels is derived. Beginning with the first character in the corpus, the algorithm consumes the longest matching sequence from the candidate vocabulary. This process continues until every character in the corpus has been consumed, illustrated in Algorithm 2. The resulting derived vocabulary consists of 128 symbols which are used to encode new program sources.

Neural Network The Long Short-Term Memory (LSTM) architecture of Recurrent Neural Network is used to model program code [HS97]. In the LSTM architecture ac-

tivations are learned with respect not just to their current inputs but to previous inputs in a sequence. In this case, this allows modelling the probability of a token appearing in the text given a history of previously seen tokens. Unlike previous recurrent networks, LSTMs employ a *forget gate* with a linear activation function, allowing them to avoid the *vanishing gradients* problem [PMB13]. This makes them effective at learning complex relationships over long sequences [LBE15] which is important for modelling program code. Extending the character-based model of Chapter 4, LSTM networks are employed to model the token-vocabulary distribution over the encoded corpus.

Compared to prior character-based models, the hybrid token vocabulary caused models to respond differently to model parameters. Initial experiments using different model parameters revealed that a two-layer LSTM network of 512 nodes per layer provided a good trade-off between the fidelity of the learned distribution and the size of the network, which limits the rate of training and inference. The network is trained using Stochastic Gradient Descent for 50 epochs, with an initial learning rate of 0.002 and decaying by 5% every epoch. Training the model on the OpenCL corpus took 12 hours using a single NVIDIA Tesla P40. The model is given no prior knowledge of the structure or syntax of a programming language.

Program Generation The trained network is sampled to generate new programs. The model is seeded with the start of a kernel (identified in OpenCL using the keywords `kernel void`) and sampled token-by-token. A “bracket depth” counter is incremented or decremented upon production of { or } tokens respectively so that the end of the kernel is detected and sampling halted. Unlike in Chapter 4, there is no support for forcing kernel specifications, and there is no upper bound on the length of a sample. The generated sequence of tokens is then decoded back to text and used for compiler testing.

5.2.2 Test Harness

OpenCL is an embedded compute kernel language, requiring host code to compile, execute, and transfer data between the host and device. For the purpose of compiler fuzzing, this requires a *test harness* to run the generated OpenCL programs. At first, the test harness of CLSmith was used. The harness assumes a kernel with no input and a `ulong` buffer as its single argument where the result is written. Only 0.2% of the GitHub kernels share this structure. A more flexible harness was desired so as to test a

more expressive range of programs, capable of supporting multi-argument kernels and generating data to use as inputs.

A new harness was developed, extending CLdrive, which first determines the expected arguments from the function prototype and generates host data for them. At the moment, scalars and arrays of all OpenCL primitive and vector types are supported. For a kernel execution across n threads, buffers of size n are allocated for pointer arguments and populated with values $[1 \dots n]$; scalar inputs are given value n since scalar integer arguments are frequently used in OpenCL for specifying buffer sizes.

The training programs from which the generative model is created are real programs, and as such do not share the argument type restrictions. The model, therefore, may generate correct programs for which the driver cannot create example inputs. In this case, a “compile-only” stub is used, which only compiles the kernel, without generating input data or executing the compiled kernel.

Unlike the generative model, this test harness is language-specific and the design stems from domain knowledge. Still, it is a relatively simple procedure, consisting of a few hundred lines of Python.

Test Harness Output Classes Executing a test case on a testbed leads to one of seven possible outcomes, illustrated in Figure 5.2. A *build failure* occurs when the online compilation of an OpenCL kernel fails, usually accompanied by an error diagnostic. A *build crash* or *build timeout* outcome occurs if the compiler crashes or fails to produce a binary within 60 seconds, respectively. For compile-only test cases, a *pass* is achieved if the compiler produces a binary. For test cases in which the kernel is executed, kernel execution leads to one of three potential outcomes: *runtime crash* if the program crashes, *timeout* if the kernel fails to terminate within 60 seconds, or *pass* if the kernel terminates gracefully and computes an output.

5.2.3 Voting Heuristics for Differential Testing

Established Differential Testing methodologies are employed to expose compiler defects. As in prior work, voting on the output of programs across compilers has been used to circumvent the *oracle problem* and detect miscompilations [McK98]. However, this approach is extended to describe not only miscompilations, but also anomalous build failures and crashes.

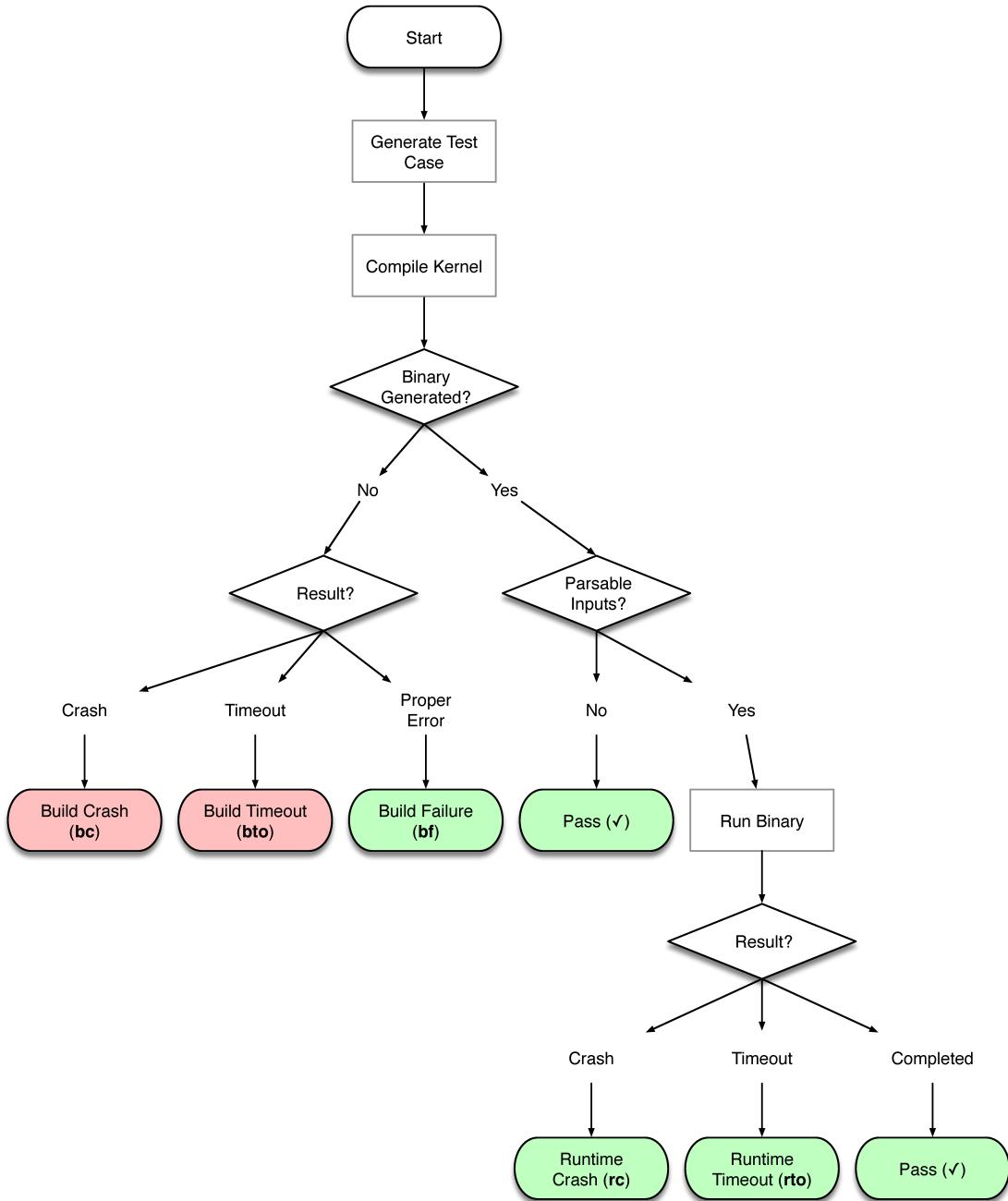


Figure 5.2: Test case execution, and possible results. Generating and executing a test case leads to one of six possible outcomes. Of these, build crashes and build timeouts are considered errors. In the remaining four cases, differential testing may be used to determine if the outcome is anomalous.

When evaluating the outcomes of test cases, build crash (**bc**) and build timeout (**bto**) outcomes are of immediate interest, indicative of erroneous compiler behaviour (examples may be found in Section 5.4.1). For all other outcomes, *differential tests* are required to confirm anomalous behaviour. The idea is to identify test cases where there is a majority outcome – i.e. for which some fraction of the testbeds behaves the same – but some testbed deviates. The presence of the majority increases the likelihood that there is a ‘correct’ behaviour for the test case. In this work, a majority fraction of $\lceil \frac{2}{3}n \rceil$ is used, where n is the number of testbeds.

An *anomalous build failure* (**abf**) or *anomalous runtime crash* (**arc**) occurs if, for a given test case, the majority of testbeds execute successfully, and a testbed yields a compilation error or runtime crash. An *anomalous wrong-output* (**awo**) occurs if, for a given test case, the majority of testbeds execute successfully, producing the same output values, and a testbed yields a result which differs from this majority output. Anomalous wrong-output results are indicative of *miscompilations*, a particularly hard to detect class of bug in which the compiler silently emits wrong code. CSmith is designed specifically to target this class of bug.

False Positives for Anomalous Runtime Behaviour Generated programs may contain undefined or non-deterministic behaviour which will incorrectly be labelled as anomalous. CSmith circumvents this problem by performing complex analyses during generation so as to minimise the chance of producing programs with undefined behaviour. Although similar analyses could be created as filters for DeepSmith, a simpler approach is taken, filtering only the few types of undefined and non-deterministic behaviour that have been actually observed to happen in practice.

Data races, out-of-bounds and uninitialized accesses are filtered using GPUVerify [BCD12] and Oclgrind [PM15]. Some compiler warnings provide a strong indication of undefined behaviour (e.g. comparison between pointer and integer) – these warnings are checked for and filtered accordingly.

Floating point operations in OpenCL are imprecise, so kernels can produce different output on different testbeds. For this reason, CSmith and CLSmith do not support floating point operations. DeepSmith permits floating point operations but since it cannot apply differential testing on the outputs, it can detect all results except for the *anomalous wrong-output* results.

The last type of undefined behaviour observed comes from division by zero and related mathematical functions which require non-zero values. A simple detection and

filtering heuristic was applied – the input values are changed and the output is checked to see if it remains anomalous. While theoretically insufficient, in practice no false positives have been found to remain.

5.3 Experimental Setup

This section describes the experimental parameters used for a testing campaign of OpenCL compilers.

5.3.1 OpenCL Systems

Testing was conducted on 10 OpenCL systems, summarised in Table 5.1. A broad range of hardware was covered: 3 GPUs, 4 CPUs, a co-processor, and an emulator. 7 of the compilers tested are commercial products, 3 of them are open source. This suite of systems includes both combinations of different drivers for the same device, and different devices using the same driver.

5.3.2 Testbeds

For each OpenCL system, two testbeds are created. In the first, the compiler is run with optimisations disabled. In the second, optimisations are enabled. Each testbed is then a triple, consisting of $\langle device, driver, is_optimised \rangle$ settings. This mechanism gives 20 testbeds to evaluate.

5.3.3 Test Cases

Inputs are created for each generated program as described in Section 5.2.2. The test harness is parameterised by a number of threads to use. Two test cases are generated, one using one thread, the other using 2048 threads. A test case is then a triple, consisting of $\langle program, inputs, threads \rangle$ settings.

5.3.4 Bug Search Time Allowance

DeepSmith and CLSmith are compared by allowing both to run for 48 hours on each of the 20 testbeds. CLSmith used its default configuration. The total runtime for a test case consists of the generation and execution time.

#.	Platform	Device	Driver	OpenCL
1	NVIDIA CUDA	GeForce GTX 1080	375.39	1.2
2	NVIDIA CUDA	GeForce GTX 780	361.42	1.2
3	Beignet	Intel HD Haswell GT2	1.3	1.2
4	Intel OpenCL	Intel E5-2620 v4	1.2.0.25	2.0
5	Intel OpenCL	Intel E5-2650 v2	1.2.0.44	1.2
6	Intel OpenCL	Intel i5-4570	1.2.0.25	1.2
7	Intel OpenCL	Intel Xeon Phi	1.2	1.2
8	POCL	POCL (Intel E5-2620)	0.14	1.2
9	Codeplay	ComputeAorta (Intel E5-2620)	1.14	1.2
10	Oclgrind	Oclgrind Simulator	16.10	1.2

(a)

#.	Operating system	Device Type	Open Source?	Bug Reports Submitted
1	Ubuntu 16.04 64bit	GPU		8
2	openSUSE 13.1 64bit	GPU		1
3	Ubuntu 16.04 64bit	GPU	Yes	13
4	Ubuntu 16.04 64bit	CPU		6
5	CentOS 7.1 64bit	CPU		1
6	Ubuntu 16.04 64bit	CPU		5
7	CentOS 7.1 64bit	Accelerator		3
8	Ubuntu 16.04 64bit	CPU	Yes	22
9	Ubuntu 16.04 64bit	CPU		1
10	Ubuntu 16.04 64bit	Emulator	Yes	7

(b)

Table 5.1: OpenCL systems and the number of bug reports submitted to date (22% of which have been fixed, the remainder are pending). For each system, two testbeds are created, one with compiler optimisations, the other without.

5.4 Evaluation

This section reports on the results of DeepSmith testing of the 10 OpenCL systems from Table 5.1, in which each ran for 48 hours. Bugs were found in all the compilers tested — every compiler crashed, and every compiler generated programs which either crash or silently compute the wrong result. To date, 67 unique bug reports have been submitted to compiler vendors. This section first contains a qualitative analysis of compile-time and runtime defects found, followed by a quantitative comparison of the approach against the state-of-the-art in OpenCL compiler fuzzing — CL-Smith [Lid+15]. DeepSmith is able to identify a broad range of defects, many of which CLSmith cannot, for only a fraction of the engineering effort. Finally, this section contains a quantitative analysis of compiler robustness over time, using the compiler crash rate of every LLVM release in the past two years as a metric of compiler robustness. The findings show that progress is good, compilers are becoming more robust, yet the introduction of new features and regressions ensures that compiler validation remains a moving target.

Unless stated otherwise, DeepSmith code listings are presented verbatim, with only minor formatting changes applied to preserve space. No test case reduction, either manual or automatic, was needed.

For the remainder of this chapter, testbeds are identified using the OpenCL system number from Table 5.1, suffixed with $+$, $-$, or \pm to denote optimisations on, off, or either, respectively.

5.4.1 Compile-time Defects

OpenCL is typically compiled online, which amplifies the significance of detecting compile-time defects, as they may not be discovered until code has been shipped to customers. Numerous cases were found where DeepSmith kernels trigger a crash in the compiler (and as a result, the host process), or cause the compiler to loop indefinitely. In the testing time allotted, DeepSmith uncovered 199 unique unreachable code failures, 31 different compiler assertions, and 114 distinct stack traces caused by other compiler crashes.

```
1 void A() { (global a*) ()
```

(a) Reduced from 48 line kernel.

```
1 void A() {void* a; uint4 b=0; b=(b>b)?a:a
```

(b) Reduced from 52 line kernel.

```
1 void A() {double2 k; return (float4)(k,k,k,k)
```

(c) Reduced from 68 line kernel.

Figure 5.3: Example codes which crash OpenCL compilers during parsing.

5.4.1.1 Semantic Analysis Failures

Compilers should produce meaningful diagnostics when inputs are invalid, yet dozens of compiler defects were discovered attributable to improper or missing error handling. Many generation and mutation based approaches to compiler validation have focused solely on testing under *valid inputs*. As such, this class of bugs may go undiscovered. Compared to these approaches, DeepSmith may contribute a significant improvement for generating plausibly-erroneous code over prior random-enumeration approaches.

The use of undeclared identifiers is a core error diagnostic which one would expect to be robust in a mature compiler. DeepSmith discovered cases in which the presence of undeclared identifiers causes the Testbeds 10± compiler to crash. For example, the undeclared identifier `c` in Figure 5.4a raises an assertion during semantic analysis of the AST when used as an array index.

Type errors were an occasional cause of compile-time defects. Figure 5.4b induces a crash in NVIDIA compilers due to an implicit conversion between global to constant address qualifiers. Worse, Testbeds 3± were found to loop indefinitely on some kernels containing implicit conversions from a pointer to an integer, as shown in Figure 5.5a. While spinning, the compiler would utilise 100% of the CPU and consume an increasing amount of host memory until the entire system memory is depleted and the process crashes.

Occasionally, incorrect program semantics will remain undetected until late in the compilation process. Both Figures 5.4c and 5.4d pass the type checker and semantic analysis, but trigger compiler assertions during code generation.

```

1 kernel void A(global float* a, global float* b) {
2     a[0] = max(a[c], b[2]);
3 }
```

(a) Testbeds 1± assertion “*Uncorrected typos!*” during semantic analysis.

```

1 kernel void A(float4 a, global float4* b,
2             global float4* c, unsigned int d,
3             global double* e, global int2* f,
4             global int4* g, constant int* h,
5             constant int* i) {
6     A(a, b, c, d, e, f, g, h);
7 }
```

(b) Testbeds 1±, 2± segmentation fault due to implicit address space conversion.

```

1 kernel void A(read_only image2d_t a,
2             global double2* b) {
3     b[0] = get_global_id(0);
4 }
```

(c) Testbeds 3± assertion *sel.hasDoubleType()* during code generation.

```

1 kernel void A(global float4* a) {
2     a[get_local_id(0) / 8][get_local_id(0)] =
3         get_local_id(0);
4 }
```

(d) Testbeds 3± assertion *scalarizeInsert* during code generation.

```

1 kernel void A() {
2     __builtin_astype(d, uint4);
3 }
```

(e) Of the 10 compilers tested, 6 crash with segfault when compiling this kernel.

Figure 5.4: Example OpenCL kernels which crash compilers.

An interesting yet unintended by-product of having trained DeepSmith on thousands of real-world examples is that the model learned to occasionally generate compiler-specific code, such as invoking compiler intrinsics. The quality of error handling on these builtins was found to vary wildly. For example, Figure 5.4e silently crashes 6 of the 10 compilers, which, to the best of my knowledge, makes DeepSmith the first random program generator to induce a defect through exploiting compiler-specific functionality.

5.4.1.2 Parser Failures

Parser development is a mature and well-understood practice. Still, parser errors were discovered in several compilers. Each of the code samples in Figure 5.3 induce crash errors during parsing of compound statements in both Testbeds 5 \pm and 7 \pm . For space, the listings have been hand-reduced to minimal code samples, which have been reported to Intel. Each reduction took around 6 edit-compile steps, taking less than 10 minutes. In total, 100 distinct programs have been generated which crash compilers during parsing.

5.4.1.3 Compiler Hangs

As expected, some compile-time defects are optimisation sensitive. Testbed 1+ hangs on large loop bounds, shown in Figure 5.5b. All of the Intel compilers tested hang during optimisation of non-terminating loops (Figure 5.5d).

Testbeds 7 \pm loop indefinitely during compilation of the simple OpenCL kernel in Figure 5.5c.

5.4.1.4 Other errors

Some compilers are more permissive than others. Testbeds 4 \pm , 6 \pm , 9 \pm reject out-of-range literal values e.g. `int i = 0xFFFFFFFFFFFFFFFFFFFFFFF;`, whilst Testbeds 3 \pm , 5 \pm , 7 \pm , 8 \pm , and 10 \pm interpret the literal as an `unsigned long long` and implicitly cast to an integer value of -1. Testbeds 1 \pm , 2 \pm emit no warning.

Testbeds 1 \pm , 2 \pm , 3 \pm rejected address space qualifiers on automatic variables, where all other testbeds successfully compiled and executed.

On Testbeds 3 \pm , the statement `int n = mad24(a, (32), get_global_size(0));` (a call to a maths builtin with mixed types) is rejected as ambiguous.

```

1 kernel void A(global int* a) {
2     int b = get_global_id(0);
3     a[b] = (6 * 32) + 4 * (32 / 32) + a;
4 }
```

(a) Testbeds 3± loop indefinitely, leaking memory until the entire system memory is depleted and the process crashes.

```

1 kernel void A(global float* a, global float* b,
2                               global float* c) {
3     int d, e, f;
4     d = get_local_id(0);
5     for (int g = 0; g < 100000; g++)
6         barrier(1);
7 }
```

(b) Testbed 1+ hangs during optimisation of kernels with large loop bounds. Testbeds 1– and 2± compile in under 1 second.

```

1 kernel void A(global unsigned char* a,
2                               unsigned b) {
3     a[get_global_id(0)] %= 42;
4     barrier(1);
5 }
```

(c) Testbeds 7± loops indefinitely, consuming 100% CPU usage.

```

1 kernel void A(global int* a) {
2     int b = get_global_id(0);
3     while (b < 512) { }
4 }
```

(d) Testbeds 4+, 5+, 6+, 7+ hang during optimisation of kernels with non-terminating loops.

Figure 5.5: Example OpenCL kernels which hang compilers.

```

1 kernel void A(global double* a, global double* b,
2           global double* c, int d, int e) {
3     double f;
4     int g = get_global_id(0);
5     if (g < e - d - 1)
6         c[g] = (((e) / d) % 5) % (e + d);
7 }
```

(a) Testbeds 4+, 6+ incorrectly optimise the `if` statement, causing the conditional branch to execute (when it shouldn't). This pattern of integer comparison to thread ID is widely used.

```

1 kernel void A(global int* a, global int* b) {
2     switch (get_global_id(0)) {
3         case 0:
4             a[get_global_id(0)] = b[get_global_id(0)+13];
5             break;
6         case 2:
7             a[get_global_id(0)] = b[get_global_id(0)+11];
8             break;
9         case 6:
10            a[get_global_id(0)] = b[get_global_id(0)+128];
11        }
12        barrier(2);
13    }
```

(b) A race condition in `switch` statement evaluation causes $10 \pm$ to sporadically crash when executed with a number of threads > 1 .

Figure 5.6: Example OpenCL kernels which are miscompiled.

```

1 kernel void A(global int* a, global int* b,
2           global int* c) {
3     c[0] = (a[0] > b[0]) ? a[0] : 0;
4     c[2] = (a[3] <= b[3]) ? a[4] : b[5];
5     c[4] = (a[4] <= b[5]) ? a[7] : b[7];
6     c[7] = (a[7] < b[0]) ? a[0] : (a[0] > b[1]);
7 }
```

- (a) Testbeds 3± silently miscompile ternary assignments in which the operands are different global buffers.

```

1 kernel void A(local int* a) {
2   for (int b = 0; b < 100; b++)
3     B(a);
4 }
```

- (b) Compilation should fail due to call to undefined function B(); Testbeds 8± silently succeed then crash upon kernel execution.

Figure 5.7: Further example OpenCL kernels which are miscompiled.

5.4.2 Runtime Defects

Prior work on compiler test case generation has focused on extensive stress-testing of compiler middle-ends to uncover miscompilations [Che+16a]. CSmith, and by extension, CLSmith, specifically targets this class of bugs. Grammar-based enumeration is highly effective at this task, yet is bounded by the expressiveness of the grammar. Here, examples are provided of bugs found by DeepSmith which cannot currently be discovered by CLSmith.

5.4.2.1 Thread-dependent Flow Control

A common pattern in OpenCL is to obtain the thread identity, often as an `int`, and to compare this against some fixed value to determine whether or not to complete a unit of work (46% of OpenCL kernels on GitHub use this ($tid \rightarrow \text{int}$, `if (tid < ...)` { \dots }) pattern). DeepSmith, having modelled the frequency with which this pattern occurs in real handwritten code, generates many permutations of this pattern. And in doing so, exposed a bug in the optimiser of Testbeds 4+ and 6+ which causes the `if` branch in Figure 5.6a to be erroneously executed when the kernel is compiled with optimisations enabled. This issue has been reported to Intel. CLSmith does not permit the thread identity to modify control flow, rendering such productions impossible.

Figure 5.6b shows a simple program in which thread identity determines the program output. This test case was found to sporadically crash Testbeds 10±, an OpenCL device simulator and debugger. Upon reporting to the developers, the underlying cause was quickly diagnosed as a race condition in `switch` statement evaluation and fixed within a week.

5.4.2.2 Kernel Inputs

CLSmith kernels accept a single buffer parameter into which each thread computes its result. This fixed prototype limits the ability to detect bugs which depend on input arguments. Figure 5.7a exposes a bug of this type. Testbeds 3± will silently miscompile ternary operators when the ternary operands consist of values stored in multiple different global buffers. CLSmith, with its fixed single input prototype, is unable to discover this bug.

5.4.2.3 Latent Compile-time Defects

Sometimes, invalid compiler inputs may go undetected, leading to runtime defects only upon program execution. Since CLSmith enumerates only well-formed programs, this class of bugs cannot be discovered.

Figure 5.7b exposes a bug in which a kernel containing an undefined symbol will successfully compile without warning on Testbeds 8±, then crash the program when attempting to run the kernel. This issue has been reported to the developers and fixed.

5.4.3 Comparison to State-of-the-art

This section provides a quantitative comparison of the bug-finding capabilities of DeepSmith and CLSmith.

5.4.3.1 Results Overview

Tables 5.2 and 5.3 shows the results of 48 hours of consecutive testing for all Testbeds using CLSmith and DeepSmith, respectively. An average of 15k CLSmith and 91k DeepSmith test cases were evaluated on each Testbed, taking 12.1s and 1.90s per test case respectively. There are three significant factors providing the sixfold increase in testing throughput achieved by DeepSmith over CLSmith: test cases are faster to generate, test cases are less likely to timeout (execute for 60 seconds without termination), and the test cases which do not timeout execute faster.

Figure 5.8a shows the generation and execution times of DeepSmith and CLSmith test cases, excluding timeouts¹. DeepSmith generation time grows linearly with program length and is on average $2.45\times$ faster than CLSmith. Test case execution is on average $4.46\times$ faster than CLSmith.

The optimisation level generally does not affect testing throughput significantly, with the exception of Testbed 7+. Optimisation of large structs is expensive on Testbed 7+, and CLSmith test cases use global structs extensively. This is a known issue — in [Lid+15] the authors omit large-scale testing on this device for this reason. The use of structs in handwritten OpenCL is comparatively rare — only 7.1% of kernels on GitHub use them.

¹If timeouts are included then the performance improvement of DeepSmith is $6.5\times$ with the execution times being $11\times$ faster. However, this number grows as the arbitrary timeout threshold is changed, so for fairness timeouts have been excluded.

#.	Device	±	bc	bto	abf	arc	awo	✓	total
1	GeForce GTX 1080	−	0	0	0	2	2	15628	15632
		+	0	71	0	6	9	14007	14093
2	GeForce GTX 780	−	0	0	0	28	5	18220	18253
		+	26	14	0	0	3	17654	17697
3	Intel HD Haswell GT2	−	2714	2480	0	0	3	1121	6318
		+	2646	2475	0	0	3	1075	6199
4	Intel E5-2620 v4	−	0	27	1183	0	0	16313	17523
		+	487	87	1130	0	0	17350	19054
5	Intel E5-2650 v2	−	0	11	0	0	0	17887	17898
		+	112	175	0	0	0	14626	14913
6	Intel i5-4570	−	0	14	1226	0	0	17118	18358
		+	526	63	1180	0	0	19185	20954
7	Intel Xeon Phi	−	4	84	0	0	8	13265	13361
		+	42	1474	0	0	2	3258	4776
8	POCL (Intel E5-2620)	−	0	0	0	675	0	17250	17925
		+	0	3	0	99	5	13980	14087
9	ComputeAorta	−	0	0	0	0	0	18479	18479
		+	0	0	0	300	11	18625	18936
10	Oclgrind Simulator	−	0	0	0	0	0	5287	5287
		+	0	0	0	0	0	5334	5334

Table 5.2: Results from 48 hours of testing using CLSmith. System #. as per Table 5.1. ± denotes optimisations off (−) vs on (+). The remaining columns denote the number of build crash (**bc**), build timeout (**bto**), anomalous build failure (**abf**), anomalous runtime crash (**arc**), anomalous wrong-output (**awo**), and pass (✓) results.

#.	Device	\pm	bc	bto	abf	arc	awo	\checkmark	total
1	GeForce GTX 1080	-	27	0	3	0	5	62105	62140
		+	20	1	1	0	7	57361	57390
2	GeForce GTX 780	-	27	0	3	0	9	87129	87168
		+	32	1	1	0	9	82666	82709
3	Intel HD Haswell GT2	-	574	200	2	0	12	136977	137765
		+	569	200	5	0	10	135430	136214
4	Intel E5-2620 v4	-	57	0	9	1	0	107982	108049
		+	320	147	7	3	0	113616	114093
5	Intel E5-2650 v2	-	152	2	0	0	0	90882	91036
		+	170	117	0	0	1	90478	90766
6	Intel i5-4570	-	73	0	9	2	1	111240	111325
		+	318	140	7	2	1	117049	117517
7	Intel Xeon Phi	-	68	4	0	0	1	37171	37244
		+	77	47	0	0	0	37501	37625
8	POCL (Intel E5-2620)	-	54	1	2	89	3	85318	85467
		+	46	0	1	104	4	81267	81422
9	ComputeAorta	-	51	0	1	3	1	112324	112380
		+	59	0	0	48	4	115323	115434
10	Oclgrind Simulator	-	2081	0	0	0	1	73261	75343
		+	2265	0	0	0	0	77959	80224

Table 5.3: Results from 48 hours of testing using DeepSmith. System #. as per Table 5.1. \pm denotes optimisations off (-) vs on (+). The remaining columns denote the number of build crash (**bc**), build timeout (**bto**), anomalous build failure (**abf**), anomalous runtime crash (**arc**), anomalous wrong-output (**awo**), and pass (**✓**) results.

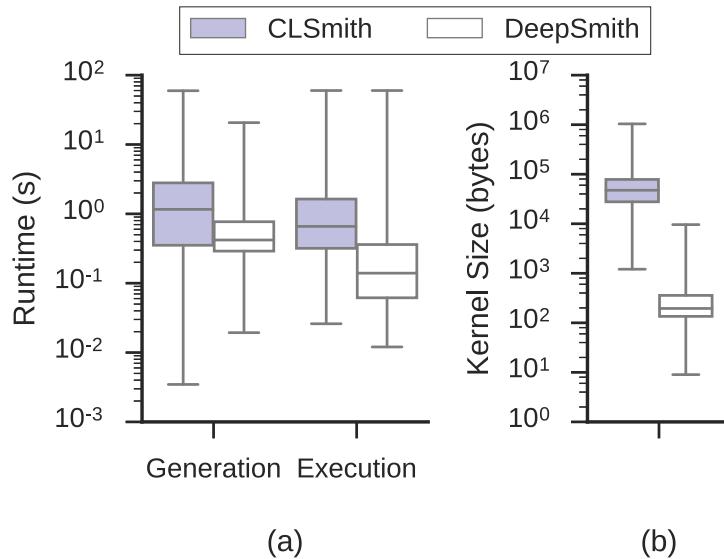


Figure 5.8: Comparison of runtimes (a) and test case sizes (b). DeepSmith test cases are on average evaluated $3.03 \times$ faster than CLSmith ($2.45 \times$, and $4.46 \times$ for generation and execution, respectively), and are two orders of magnitude smaller. Timings do not include the cost of timeouts which would increase the performance gains of DeepSmith by nearly a factor of two.

5.4.3.2 Comparison of Test Cases

The average CLSmith program is 1189 lines long (excluding headers). CLSmith test cases require reduction in order to expose the underlying bug. An automated approach to OpenCL test case reduction is presented in [PDL16], though it requires on average 100 minutes for each test case using a parallelised implementation (and over 6 hours if this parallelisation is not available); still, the authors suggest a final manual pass after automated reduction. In contrast, DeepSmith learned to program from humans, and humans do not typically write such large kernel functions. The average DeepSmith kernel is 20 lines long, which is interpretable without reduction, either manual or automatic.

5.4.3.3 Comparison of Results

Both testing systems found anomalous results of all types. In 48 hours of testing, CLSmith discovered compile-time crashes (**bc**) in 8 of the 20 testbeds, DeepSmith crashed all of them. DeepSmith triggered 31 distinct compiler assertions, CLSmith 2. Both of the assertions triggered by CLSmith were also triggered by DeepSmith. DeepSmith

also triggered 3 distinct *unreachable!* compile-time crashes, CLSmith triggered 0. The ratio of build failures is higher in the token-level generation of DeepSmith (51%) than the grammar-based generation of CLSmith (26%).

The Intel CPU Testbeds ($4\pm$, $5\pm$, $6\pm$, and $7\pm$) would occasionally emit a stack trace upon crashing, identifying the failure point in a specific compiler pass. CLSmith triggered such crashes in 4 distinct passes. DeepSmith triggered crashes in 10 distinct passes, including 3 of the 4 in which CLSmith did. Figures 5.9 and 5.10 provide examples. Many of these crashes are optimisation sensitive and are more likely to occur when optimisations are enabled. CLSmith was able to induce a crash in only one of the Intel testbeds with optimisations disabled. DeepSmith crashed all of the compilers with both optimisations enabled and disabled.

CLSmith produced many **bto** results across 13 Testbeds. Given the large kernel size, it is unclear how many of those are infinite loops or simply a result of the slow compilation of large kernels. The average size of CLSmith **bto** kernels is 1558 lines. Automated test case reduction — in which thousands of permutations of a program are executed — may be prohibitively expensive for test cases with long runtimes. DeepSmith produced **bto** results across 11 Testbeds and with an average kernel size of 9 lines, allowing for rapid identification of the underlying problem.

The integrated GPU Testbeds ($3\pm$) frequently failed to compile CLSmith kernels, resulting in over 10k **bc** and **bto** results. Of the build crashes, 68% failed silently, and the remainder were caused by the same two compiler assertions for which DeepSmith generated 4 line test cases, shown in Figure 5.11. DeepSmith also triggered silent build crashes in Testbeds $3\pm$, and a further 8 distinct compiler assertions.

The 4719 **abf** results for CLSmith on Testbeds $4\pm$ and $6\pm$ are all a result of compilers rejecting empty declarations, (e.g. `int;`) which CLSmith occasionally emits. DeepSmith also generated these statements, but with a much lower probability, given that it is an unusual construct (0.6% of test cases, versus 7.0% of CLSmith test cases).

ComputeAorta (Testbeds $9\pm$) defers kernel compilation to enable optimisations dependent on runtime parameters. This may contribute to the relatively large number of **arc** results and few **bc** results of Testbeds $9\pm$. Only DeepSmith was able to expose compile-time defects in this compiler.

Over the course of testing, a combined 3.4×10^8 lines of CLSmith code was evaluated, compared to 3.8×10^6 lines of DeepSmith code. This provides CLSmith with a greater potential to trigger miscompilations. CLSmith generated 33 programs with anomalous wrong-outputs. DeepSmith generated 30.

```

1 kernel void A() {
2     while (true)
3         barrier(1);
4 }
```

(a) *Post-Dominance Frontier Construction* pass.

```

1 kernel void A(global float* a, global float* b,
2             const int c) {
3     for (int d = 0; d < c; d++)
4         for (d = 0; d < a; d += 32)
5             b[d] = 0;
6 }
```

(b) *Simplify the CFG* pass.

```

1 kernel void A(global int* a) {
2     int b = get_global_id(0);
3     while (b < *a)
4         if (a[0] < 0)
5             a[1] = b / b * get_local_id(0);
6 }
```

(c) *Predicator* pass.

```

1 kernel void A(global float* a, global float* b,
2               global float* c, const int d) {
3     for (unsigned int e = get_global_id(0);
4           e < d; e += get_global_size(0))
5         for (unsigned f = 0; f < d; ++f)
6             e += a[f];
7 }
```

(d) *Combine redundant instructions* pass.

Figure 5.9: Example OpenCL kernels which crash Intel compiler passes.

```

1 kernel void A(int a, global int* b) {
2     int c = get_global_id(0);
3     int d = work_group_scan_inclusive_max(c);
4     b[c] = c;
5 }
```

(a) *PrepareKernelArgs* pass.

```

1 kernel void A() {
2     local float a; A(a);
3 }
```

(b) *Add SPIR related module scope metadata* pass.

```

1 kernel void A() {
2     local int a[10];
3     local int b[16][16];
4     a[1024 + (2 * get_local_id(1) +
5         get_local_id(0)) + get_local_id(0)] = 6;
6     barrier(b);
7 }
```

(c) *Intel OpenCL RemoveDuplicationBarrier* pass.

```

1 kernel void A(global half* a) {
2     int b = get_global_id(0);
3     a[b] = b * b;
4 }
```

(d) *X86 DAG-DAG Instruction Selection* pass.

Figure 5.10: Further example OpenCL kernels which crash Intel compiler passes.

```

1 kernel void A(global int* a, global int* b,
2           global int* c) {
3     a[get_global_id(0)] = a[get_global_id(0)] > b;
4 }
```

(a) Assertion *storing/loading pointers only support private array.*

```

1 kernel void A(global int* a) {
2   global int* b = ((void*)0);
3   b[0] = a;
4 }
```

(b) Assertion *iter != pointerOrigMap.end()*.

Figure 5.11: Example kernels which trigger compiler assertions which both CLSmith and DeepSmith exposed.

5.4.4 Compiler Stability Over Time

The Clang front-end to LLVM supports OpenCL, and is commonly used in OpenCL drivers. This, in turn, causes Clang-related defects to potentially affect multiple compilers, for example, the one in Figure 5.4e. To evaluate the impact of Clang, debug+assert builds of every LLVM release in the past 24 months were used to process 75,000 DeepSmith kernels through the Clang front-end (this includes the lexer, parser, and type checker, but not code generation).

Figure 5.12 shows that the number of crashes observed in the Clang front-end is, for the most part, steadily decreasing over time. The number of failing compiler crashes decreased tenfold between 3.6.2 and 5.0.0. Table 5.4 shows the 7 distinct assertions triggered during this experiment. Assertion 1 (*Uncorrected typos!*) is raised on all compiler versions — see Figure 5.4a for an example. The overall rate at which the assertion is triggered has decreased markedly, although there are slight increases between some releases. Notably, the current development trunk has the second lowest crash rate but is joint first in terms of the number of unique assertions. Assertions 3 (*Addr == 0* — *hasTargetSpecificAddressSpace()*) and 4 (*isScalarType()*) were triggered by some kernels in the development trunk but not under any prior release.

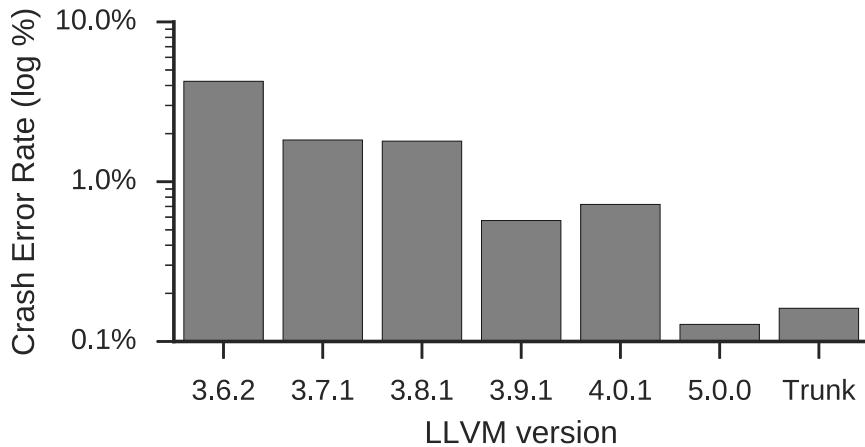


Figure 5.12: The crash rate of the Clang front-end of every LLVM release in the past 24 months compiling 75k DeepSmith kernels, plotted with a logarithmic scale.

Bug reports have been submitted for each of the three assertions triggered in the development trunk, as well as for two distinct unreachables.

The results emphasise that compiler validation is a moving target. Every change and feature addition has the potential to introduce regressions or new failure cases. Since LLVM will not release unless their compiler passes their own extensive test suites, this also reinforces the case for compiler fuzzing. DeepSmith provides an effective means for the generation of such fuzzers, at a fraction of the cost of existing techniques.

5.4.5 Extensibility of Language Model

A large portion of the DeepSmith architecture is language-agnostic, requiring only a corpus, encoder, and harness for each new language. This potentially significantly lowers the barrier-to-entry compared with prior grammar-based fuzzers. This section reports on initial results in extending DeepSmith to the Solidity programming language. Solidity is the smart contract programming language of the Ethereum blockchain. At less than four years old, it lacks much of the tooling of more established programming languages. Yet, it is an important candidate for rigorous testing, as exploitable bugs may undermine the integrity of the blockchain and lead to fraudulent transactions.

	3.6.2	3.7.1	3.8.1	3.9.1	4.0.1	5.0.0	Trunk
Assertion 1	2962	1327	1332	414	523	83	97
Assertion 2		1	1				
Assertion 3						1	
Assertion 4						2	
Assertion 5	147						
Assertion 6	1						
Assertion 7				1	1		
Unreachable	86	42	14	14	18	13	21

Table 5.4: The number of DeepSmith programs which trigger distinct Clang front-end assertions, and the number of programs which trigger unreachables.

Compiler	\pm	Silent Crashes	Assertion 1	Assertion 2
solc	-	204	1	
	+	204	1	
solc-js	-	3628	1	1
	+	908	1	1

Table 5.5: The number of DeepSmith programs that trigger Solidity compiler crashes in 12 hours of testing.

5.4.5.1 Testing Methodology

The same methodology was applied to train the program generator as for OpenCL. A corpus of 1.74M Solidity contracts was assembled from GitHub, recursively inlining imported modules where possible. The same tokeniser was used as for OpenCL, only changing the list of language keywords and builtins. Code style was enforced using clang-format. The model is trained in the same manner as OpenCL. No modification to either the language model or generator code was required. A simple compile-only test harness is used to drive the generated Solidity contracts.

5.4.5.2 Initial Results

The generator and harness loop was run for 12 hours on four testbeds: the Solidity reference compiler `solc` with optimisations on or off, and `solc-js`, which is an Emscripten compiled version of the `solc` compiler. Table 5.5 summarises the results. Numerous cases were found where the compiler silently crashes, and two distinct compiler assertions. The first is caused by missing error handling of language features (this issue is known to the developers). The source of the second assertion is the JavaScript runtime and is triggered only in the Emscripten version, suggesting an error in the automatic translation from LLVM to JavaScript.

Extending DeepSmith to a second programming required an additional 150 lines of code (18 lines for the generator and encoder, the remainder for the test harness) and took about a day. Given the re-usability of the core DeepSmith components, there is a diminishing cost with the addition of each new language. For example, the OpenCL encoder and re-writer, implemented using LLVM, could be adapted to C with minimal changes. Given the low-cost of extensibility, these preliminary results indicate the utility of the approach for simplifying test case generation.

5.5 Summary

This chapter presents a novel framework for compiler fuzzing. By posing the generation of random programs as an unsupervised machine learning problem, the cost and human effort required to engineer a compiler fuzzer are drastically lowered. This aims to address the *adoption challenge* of machine learning (Section 1.2.3). Large parts of the stack are programming language-agnostic, requiring only a corpus of example programs, an encoder, and a test harness to target a new language.

The approach is demonstrated by targeting the challenging many-core domain of OpenCL. The implementation, DeepSmith, has uncovered dozens of bugs in both commercial and open-source OpenCL compilers. DeepSmith exposed bugs in parts of the compiler where current approaches have not, for example in missing error handling. A preliminary exploration of the extensibility of this approach to other languages has been performed. DeepSmith test cases are small, two orders of magnitude shorter than the state-of-the-art, and easily interpretable.

Chapter 6

Simplifying the Construction of Optimisation Heuristics

6.1 Introduction

There are countless scenarios during the compilation and execution of a program where decisions must be made as to how, or if, a particular optimisation should be applied. Modern compilers and runtimes are rife with hand-coded *heuristics* which perform this decision making. The performance of programs is thus dependent on the quality of these heuristics.

Handwritten heuristics require expert knowledge, take a lot of time to construct, and in many cases lead to sub-optimal decisions. Researchers have focused on machine learning as a means of constructing high-quality heuristics that often outperform their handcrafted equivalents [Aga+06; Cum+16a; FE15; MSD16; SA05]. A *predictive model* is trained, using supervised machine learning, on empirical performance data and important quantifiable properties, or *features*, of representative programs. The model learns the correlation between these features and the optimisation decision that maximises performance. The learned correlations are used to predict the best optimisation decisions for new programs. Previous works in this area were able to build machine learning based heuristics with less effort, that outperform ones created manually by experts [GWO13; MDO14].

Still, experts are not completely removed from the design process, which is shown in Figure 6.1a. Selecting the appropriate features is a manual undertaking which requires a deep understanding of the system. The designer essentially decides which compile or runtime characteristics affect optimisation decisions and expresses them in

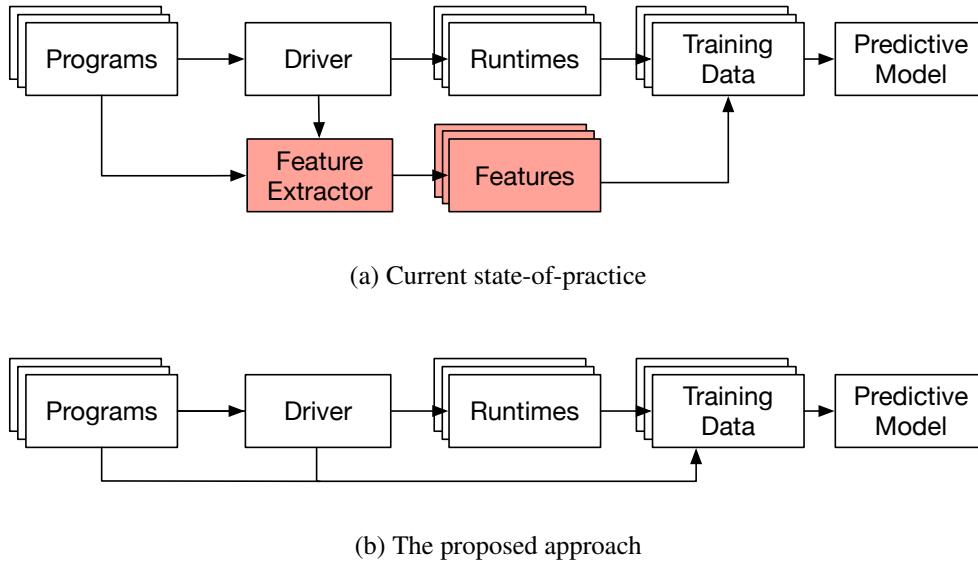


Figure 6.1: Building a predictive model. The model is originally trained on performance data and features extracted from the source code and the runtime behaviour. The proposed approach bypasses feature extraction, instead learning directly over raw program source code.

ways that make it easy to model their relationship to performance. Failing to identify an important feature has a negative effect on the resulting heuristic. For example, in Section 4.7.2 the Grewe, Wang, and O’Boyle model is shown to be missing one such feature, causing performance to be 40% lower on average.

To make heuristic construction fast and cheap, humans must be taken out of the loop. While techniques for automatic feature generation from the compiler IR have been proposed in the past [LBO14; Nam+10], they do not solve the problem in a practical way. They are deeply embedded into the compiler, require expert knowledge to guide the generation, have to be repeated from scratch for every new heuristic, and their search time can be prohibitive. Inspired by the success of recurrent neural networks at generating program code for benchmarking and compiler testing, I hypothesised that a learning system should be able to automatically extract feature representations from source code. These learned feature representations could then be used as inputs to other learning systems for task-specific purposes such as learning optimisation heuristics. This would mitigate the *feature design* challenge (Section 1.2.2).

The experiments showed that this was a conservative target: with deep neural networks, one can entirely bypass static feature extraction and learn optimisation heuristics directly on raw code, without the need for an intermediate representation.

Figure 6.1b shows the proposed methodology. Instead of manually extracting features from input programs to generate training data, program code is used directly as training data. Programs are fed through a series of artificial neural networks which learn how code correlates with performance. Internally and without prior knowledge, the networks construct complex abstractions of the input program characteristics and correlations between those abstractions and performance. This chapter proposes replacing the need for compile-time or static code features, merging feature and heuristic construction into a single process of joint learning. The system admits auxiliary features to describe information unavailable at compile time, such as the sizes of runtime input parameters. Beyond these optional inclusions, the system is able to learn optimisation heuristics without human guidance.

By employing *transfer learning* [Yos+14], the proposed approach is able to produce high-quality heuristics even when learning on a small number of programs. The properties of the raw code that are abstracted by the beginning layers of the artificial neural networks are mostly independent of the optimisation problem. Parts of the network may be reused across heuristics, and, in the process, speed up learning considerably.

The approach is evaluated on two parallel compilation problems: heterogeneous device mapping and GPU thread coarsening. Good heuristics for these two problems are important for extracting performance from heterogeneous systems, and the fact that machine learning has already been applied to constructing heuristics for these problems allows direct comparison. Prior machine learning approaches resulted in good heuristics which extracted 73% and 79% of the available performance respectively but required extensive human effort to select the appropriate features. Nevertheless, the approach presented in this chapter was able to outperform them by 14% and 12%, which indicates a better identification of important program characteristics, without any expert help.

This chapter is organised as follows: Section 6.2 describes DeepTune, a novel system for building optimisation heuristics. In Sections 6.3 and 6.4, case studies are presented that evaluate DeepTune’s capabilities first at predicting heterogeneous device mapping, then thread coarsening. Section 6.5 explores the novel transfer of information between the two problems. Section 6.6 illuminates the inner workings of DeepTune by analysing the internal state within the artificial neural networks. Finally, Section 6.7 contains concluding remarks.

6.2 DeepTune: Learning On Raw Program Code

DeepTune is an end-to-end machine learning pipeline for optimisation heuristics. Its primary input is the source code of a program to be optimised, and through a series of artificial neural networks, it directly predicts the optimisation which should be applied. By learning on source code, the approach is not tied to a specific compiler, platform, or optimisation problem. The same design can be reused to build multiple heuristics. The most important innovation of DeepTune is that it obviates the need for human experts to select and tune appropriate features.

6.2.1 Overview

Figure 6.2 provides an overview of the system. A source re-writer removes semantically irrelevant information (such as comments) from the source code of the target program and passes it to a language model. The language model converts the arbitrary length stream of code into a fixed length vector of real values which fully capture the properties and structure of the source, replacing the role of hand-designed features. This vector can then optionally be concatenated with auxiliary inputs, which allow passing additional data about runtime or architectural parameters to the model for heuristics which need more than just compile-time information. Finally, a standard feed-forward network is used to predict the best heuristic parameters to optimise the program.

DeepTune is open source. The model is implemented in Keras, with TensorFlow [Aba+16] and Theano [Ber+11] back-ends.

6.2.2 Language Model

Learning effective representations of source code is a difficult task. A successful model must be able to:

- derive semantic and syntactic patterns of a programming language entirely from sample codes;
- identify the patterns and representation in source codes which are relevant to the task at hand; and
- discriminate performance characteristics arising from potentially subtle differences in similar codes.

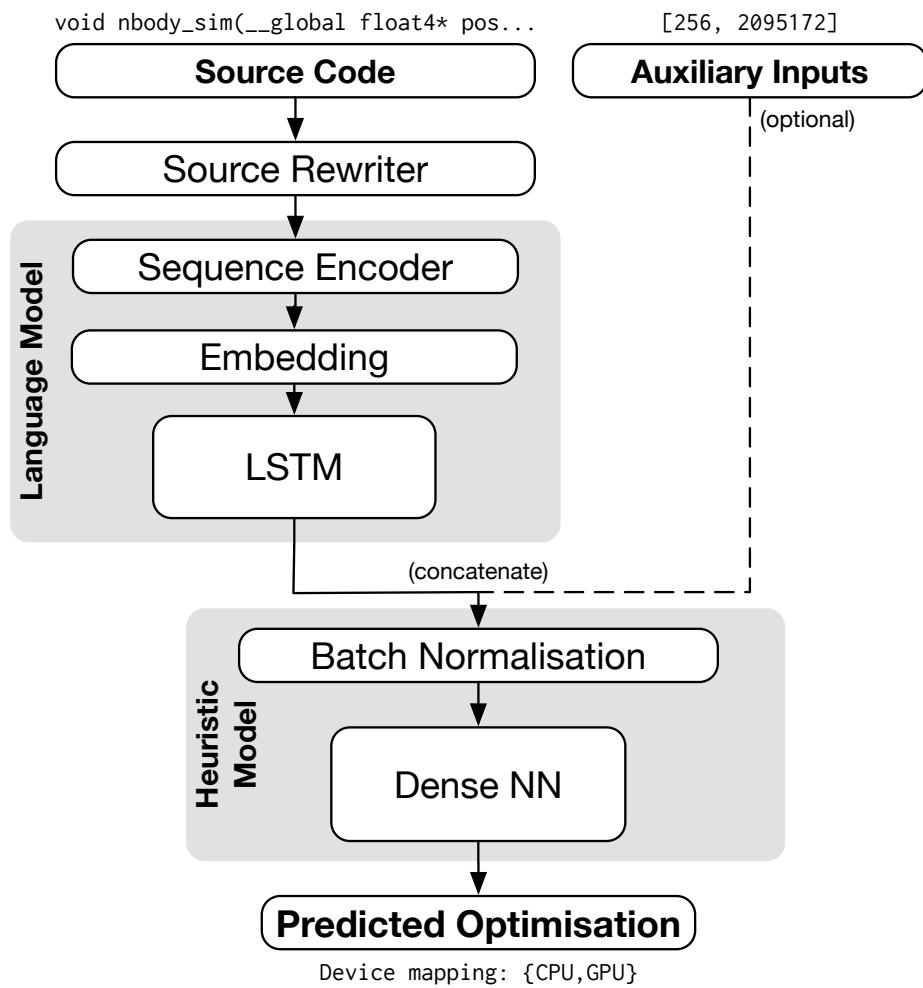


Figure 6.2: DeepTune system overview. Code properties are extracted from source code by the language model. They are fed, together with optional auxiliary inputs, to the heuristic model to produce the final prediction.

To achieve this task, state-of-the-art language modelling techniques are employed, coupled with a series of generic, language-agnostic code transformations.

6.2.2.1 Source Re-writer

To begin with, a series of *source normalising* transformations are applied, extending the system described in Chapter 5. These transformations, implemented as an LLVM pass, parse the AST, removing conditional compilation, then rebuild the input source code using a consistent code style and identifier naming scheme. The role of source normalisation is to simplify the task of modelling source code by ensuring that trivial semantic differences in programs such as the choice of variable names or the insertion of comments do not affect the learned model. Figures 6.3a and 6.3b show the source rewriting process applied to a simple program.

6.2.2.2 Sequence Encoder

The textual representation of program codes must be encoded as numeric sequences for feeding as input to the machine learning model. This is achieved by extending the encoder described in Section 5.2, in which a programming language’s keywords and common names are treated as individual tokens while the rest of the text is encoded on a character-level basis. This approach hits a balance between compressing the input text and keeping the number of tokens in the vocabulary low. Figure 6.3c shows the vocabulary derived from the single input source code in Figure 6.3b.

6.2.2.3 Embedding

During encoding, tokens in the vocabulary are mapped to unique integer values, e.g. `float` → 0, `int` → 1. The chosen integer values are arbitrary, and offer a *sparse* data representation, meaning that a language model cannot infer the relationships between tokens based on their mappings. This is in contrast to the *dense* representations of other domains, such as pixels in images, which can be interpolated between to derive the differences in colours.

To mitigate this, an *embedding* is used, which translates tokens in a sparse, integer-encoded vocabulary into a lower dimensional vector space, allowing semantically related tokens like `float` and `int` to be mapped to nearby points [BDK14; Mik+13]. An embedding layer maps each token in the integer-encoded vocabulary to a vector of real values. Given a vocabulary size $|V|$ and embedding dimensionality D , an embedding

matrix $\mathbf{W}_E \in \mathbb{R}^{|V| \times D}$ is learned during training such that an integer-encoded sequence of tokens $\mathbf{t} \in \mathbb{N}^L$ is mapped to the matrix $\mathbf{T} \in \mathbb{R}^{L \times D}$. In this work, a embedding dimensionality $D = 64$ is used, selected somewhat arbitrarily, with best-practices suggesting an empirical approach to determine good values. Research into analytical models for selecting embedding dimensionality is ongoing [Nau19; YS18].

6.2.2.4 Sequence Characterisation

Once source codes have been encoded into sequences of embedding vectors, artificial neural networks are used to extract a fixed size vector which characterises the entire sequence. This is comparable to the hand engineered feature extractors used in prior works, but is a *learned* process that occurs entirely — and automatically — within the hidden layers of the network.

The Long Short-Term Memory (LSTM) architecture is used for sequence characterisation [HS97]. LSTMs implement a Recurrent Neural Network in which the activations of neurons are learned with respect not just to their current inputs, but to previous inputs in a sequence. Unlike regular recurrent networks in which the strength of learning decreases over time (a symptom of the *vanishing gradients* problem [PMB13]), LSTMs employ a *forget gate* with a linear activation function, allowing them to retain activations for arbitrary durations. This makes them effective at learning complex relationships over long sequences [LBE15], an especially important capability for modelling program code, as dependencies in sequences frequently occur over long ranges (for example, a variable may be declared as an argument to a function and used throughout).

The LSTM network has two layers of cells. The network receives a sequence of embedding vectors, and returns a single output vector, characterising the entire sequence.

6.2.3 Auxiliary Inputs

An arbitrary number of additional real-valued *auxiliary inputs* may be optionally used to augment the source code input. These inputs are provided as a means of increasing the flexibility of the system, for example, to support applications in which the optimisation heuristic depends on dynamic values which cannot be statically determined from the program code [Din+15; SA05]. When present, the values of auxiliary inputs are concatenated with the output of the language model and fed into a heuristic model.

```

1 kernel void memset_kernel(global char * mem_d, short val, int
→ number_bytes) {
2   const int thread_id = get_global_id(0);
3   mem_d[thread_id] = val; }
```

(a) An example, short OpenCL kernel, taken from Nvidia's *streamcluster*.

```

1 kernel void A(global char* a, short b, int c) {
2   const int d = get_global_id(0);
3   a[d] = b;
4 }
```

(b) The *streamcluster* kernel after source rewriting. Variable and function names are normalised, comments removed, and code style enforced.

idx	token	idx	token	idx	token
1	'_kernel'	10	','	19	'const'
2	' '	11	'short'	20	'd'
3	'void'	12	'b'	21	'='
4	'A'	13	'int'	22	'get_global_id'
5	'('	14	'c'	23	'0'
6	'_global'	15	')'	24	';'
7	'char'	16	'{'	25	'['
8	'**'	17	'\\n'	26	']'
9	'a'	18	' '	27	'}'

(c) Derived vocabulary, ordered by their appearance in the input (b). The vocabulary maps tokens to integer indices.

01	02	03	02	04	05	06	02	07	08	02
09	10	02	11	02	12	10	02	13	02	14
15	02	16	17	18	19	02	13	02	20	02
21	02	22	05	23	15	24	17	18	09	25
20	26	02	21	02	12	24	17	27	<pad...>	

(d) Indices encoded kernel sequence. Sequences may be padded to a fixed length by repeating an out-of-vocabulary integer (e.g. -1).

Figure 6.3: Deriving a tokenised 1-of- k vocabulary encoding from source code.

6.2.4 Heuristic Model

The heuristic model takes the learned representations of the source code and auxiliary inputs (if present) and uses these values to make the final optimisation prediction.

First, the values are normalised. Normalisation is necessary because the auxiliary inputs can have any values, whereas the language model activations are in the range [0,1]. If the heuristic model inputs were not normalised, then scaling the auxiliary inputs could affect the training of the heuristic model. Normalisation occurs in batches. The batch normalisation method of [IS15] is used, in which each scalar of the heuristic model's n inputs $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(n)}$ is independently normalised to a mean 0 and standard deviation of 1:

$$\hat{\mathbf{x}}^{(i)} = \gamma^{(i)} \frac{\mathbf{x}^{(i)} - E(\mathbf{x}^{(i)})}{\sqrt{Var(\mathbf{x}^{(i)})}} + \beta^{(i)} \quad (6.1)$$

Where γ and β are scale and shift parameters, learned during training.

The final component of DeepTune is comprised of two fully connected artificial neural network layers. The first layer consists of 32 neurons. The second layer consists of a single neuron for each possible heuristic decision. Each neuron applies an activation function $\phi(z)$ over its inputs. Rectified linear units $\phi(z) = \max(z, 0)$ are used in the first layer due to their improved performance during the training of deep networks [NH10]. For the output layer, sigmoid activation functions $\phi(z) = \frac{1}{1+e^{-z}}$ are used which provide activations in the range [0, 1].

The activation of each neuron in the output layer represents the model's confidence that the corresponding decision is the correct one. Taking the arg max of the output layer produces the decision with the largest activation. For example, for a binary optimisation heuristic the final layer will consist of two neurons, and the predicted optimisation is the neuron with the largest activation.

6.2.5 Training the Network

DeepTune is trained in the same manner as prior supervised machine learning works, the key difference being that instead of having to manually create and extract features from programs, the raw program codes themselves are used.

The model is trained with Stochastic Gradient Descent (SGD) using the Adam optimiser [KB15]. For training data $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(n)}$, SGD attempts to find the model parameters $\hat{\Theta}$ that minimise the output of a loss function:

$$\hat{\Theta} = \arg \min_{\Theta} \frac{1}{n} \sum_{i=1}^n \mathcal{L}(\mathbf{x}^{(i)}, \Theta) \quad (6.2)$$

where loss function $\mathcal{L}(\mathbf{x}, \Theta)$ computes the logarithmic difference between the predicted and expected values given a model constructed using parameters Θ .

To reduce training time, multiple inputs are *batched* together and fed into the artificial neural network simultaneously, reducing the frequency of costly weight updates during back-propagation. This requires that the inputs to the language model be the same length. Sequences are padded up to a fixed length of 1024 tokens using a special out-of-vocabulary padding token $\tau \notin V$, allowing matrices of `batch_size × max_seq_len` tokens to be processed simultaneously. Batching and padding sequences to a maximum length is only to improve training time. In production use, sequences do not need to be padded, allowing classification of arbitrary length codes in linear time.

6.3 Case Study A: OpenCL Heterogeneous Mapping

OpenCL provides a platform-agnostic framework for heterogeneous parallelism. This allows a program written in OpenCL to execute transparently across a range of different devices, from CPUs to GPUs and FPGAs. Given a program and a choice of execution devices, the question then is on which device should one execute the program to maximise performance?

6.3.1 State-of-the-art

I return to the Grewe, Wang, and O’Boyle [GWO13] predictive model of Chapter 4 for mapping OpenCL kernels to the optimal device in CPU/GPU heterogeneous systems. The authors use supervised learning to construct decision trees, using a combination of static and dynamic kernel features. The static program features are extracted using a custom LLVM pass; the dynamic features are taken from the OpenCL runtime.

Expert Chosen Features Table 6.1a shows the features used in their work. Each feature is an expression built upon the code and runtime metrics given in Table 6.1b.

Name	Description
F1: data size / (comp+mem)	commun.-computation ratio
F2: coalesced/mem	% coalesced memory accesses
F3: (localmem/mem) × wgsiz	ratio local to global mem accesses × #. work-items
F4: comp/mem	computation-mem ratio

(a) Feature values

Name	Type	Description
comp	static	#. compute operations
mem	static	#. accesses to global memory
localmem	static	#. accesses to local memory
coalesced	static	#. coalesced memory accesses
data size	dynamic	size of data transfers
work-group size	dynamic	#. work-items per kernel

(b) Values used in feature computation

Table 6.1: Features used by Grewe, Wang, and O’Boyle to predict heterogeneous device mappings for OpenCL kernels.

	Version	#. benchmarks	#. kernels
NPB (SNU [SJL11])	1.0.3	7	114
Rodinia [Che+09]	3.1	14	31
NVIDIA SDK	4.2	6	12
AMD SDK	3.0	12	16
Parboil [Str+12]	0.2	6	8
PolyBench [Gra+12]	1.0	14	27
SHOC [Dan+10]	1.1.5	12	48
Total	-	71	256

Table 6.2: Benchmarks used in Case Study A.

	Frequency	Memory	Driver
Intel Core i7-3820	3.6 GHz	8GB	AMD 1526.3
AMD Tahiti 7970	1000 MHz	3GB	AMD 1526.3
NVIDIA GTX 970	1050 MHz	4GB	NVIDIA 361.42

Table 6.3: Experimental platforms used in Case Study A.

6.3.2 Experimental Setup

The predictive model of Grewe, Wang, and O’Boyle [GWO13] is replicated. The same experimental setup is used as in Chapter 4 in which the experiments are extended to a larger set of 71 programs, summarised in Table 6.2. The programs are evaluated on two CPU-GPU platforms, detailed in Table 6.3.

DeepTune Configuration Figure 6.4a shows the artificial neural network configuration of DeepTune for the task of predicting optimal device mapping. The OpenCL kernel source code is used as input, along with the two dynamic values *work-group size* and *data size* available to the OpenCL runtime.

Model Evaluation *Stratified 10-fold cross-validation* is used to evaluate the quality of the predictive models [HKP11]. Each program is randomly allocated into one of 10 equally-sized sets; the sets are balanced to maintain a distribution of instances from each class consistent with the full set. A model is trained on the programs from all but one of the sets, then tested on the programs of the unseen set. This process is repeated for each of the 10 sets to construct a complete prediction over the whole data set.

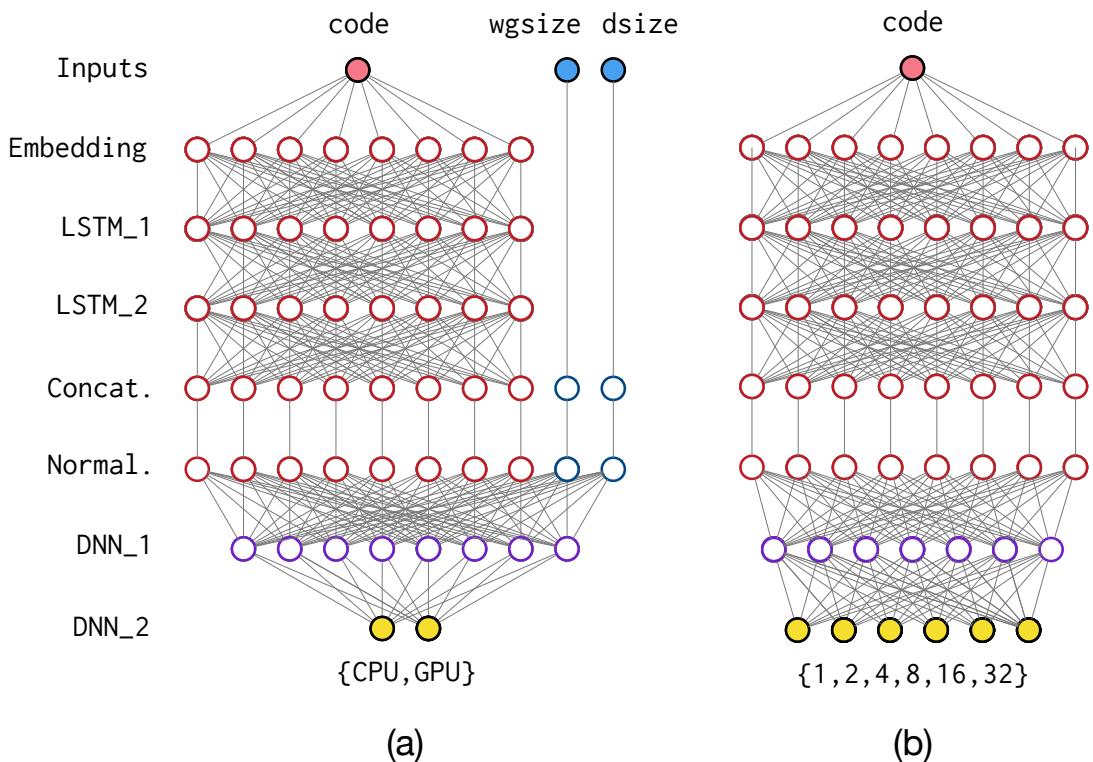


Figure 6.4: DeepTune artificial neural networks, configured for (a) heterogeneous mapping, and (b) thread coarsening factor. The design stays almost the same regardless of the optimisation problem. The only changes are the extra input for (a) and size of the output layers. To aid in visualisation, the number of neurons in each layer is reduced. For the true neuron counts, see Table 6.7

6.3.3 Experimental Results

Selecting the optimal execution device for OpenCL kernels is essential for maximising performance. For a CPU/GPU heterogeneous system, this presents a binary choice. In this experiment, the approach is compared to a static single-device approach and the Grawe, Wang, and O’Boyle predictive model. The *static mapping* selects the device which gave the best average case performance over all the programs. On the AMD platform, the best-performing device is the CPU; on the NVIDIA platform, it is the GPU.

Figure 6.5 shows the accuracy of both predictive models and the static mapping approach for each of the benchmark suites. The static approach is accurate for only 58.8% of cases on AMD and 56.9% on NVIDIA. This suggests the need for choosing the execution device on a per program basis. The Grawe, Wang, and O’Boyle model

achieves an average accuracy of 73%, a significant improvement over the static mapping. By automatically extracting useful feature representations from the source code, DeepTune gives an average accuracy of 82%, an improvement over both schemes.

Using the static mapping as a baseline, the relative performance of each program is computed using the device selected by the Grewe, Wang, and O’Boyle and DeepTune models. Figure 6.6 shows these speedups. Both predictive models significantly outperform the static mapping; the Grewe *et al.* model achieves an average speedup of $2.91\times$ on AMD and $1.26\times$ on NVIDIA (geometric mean $1.18\times$). In 90% of cases, DeepTune matches or outperforms the predictions of the Grewe *et al.* model, achieving an average speedup of $3.34\times$ on AMD and $1.41\times$ on NVIDIA (geometric mean $1.31\times$). This 14% improvement in performance comes at a greatly reduced cost, requiring no intervention by humans.

6.4 Case Study B: OpenCL Thread Coarsening Factor

Thread coarsening is an optimisation for parallel programs in which the operations of two or more threads are fused together. The number of threads which must be executed is then reduced by this thread coarsening factor. This optimisation might prove beneficial on certain combinations of programs and architectures, for example, programs with a large potential for Instruction-level Parallelism on Very Long Instruction Word architectures.

6.4.1 State-of-the-art

Magni, Dubach, and O’Boyle present a predictive model for OpenCL thread coarsening in [MDO14]. They implement an iterative heuristic which determines whether a given program would benefit from coarsening. If yes, then the program is coarsened, producing a new program from which features can be computed. This process is repeated, allowing further coarsening. In this manner, the problem is reduced from a multi-label classification problem into a series of binary decisions, shown in Figure 6.7a. They select from one of six possible coarsening factors: (1, 2, 4, 8, 16, 32), divided into 5 binary choices.

Expert Chosen Features Magni, Dubach, and O’Boyle followed a comprehensive feature engineering process. 17 candidate features were assembled from previous stud-

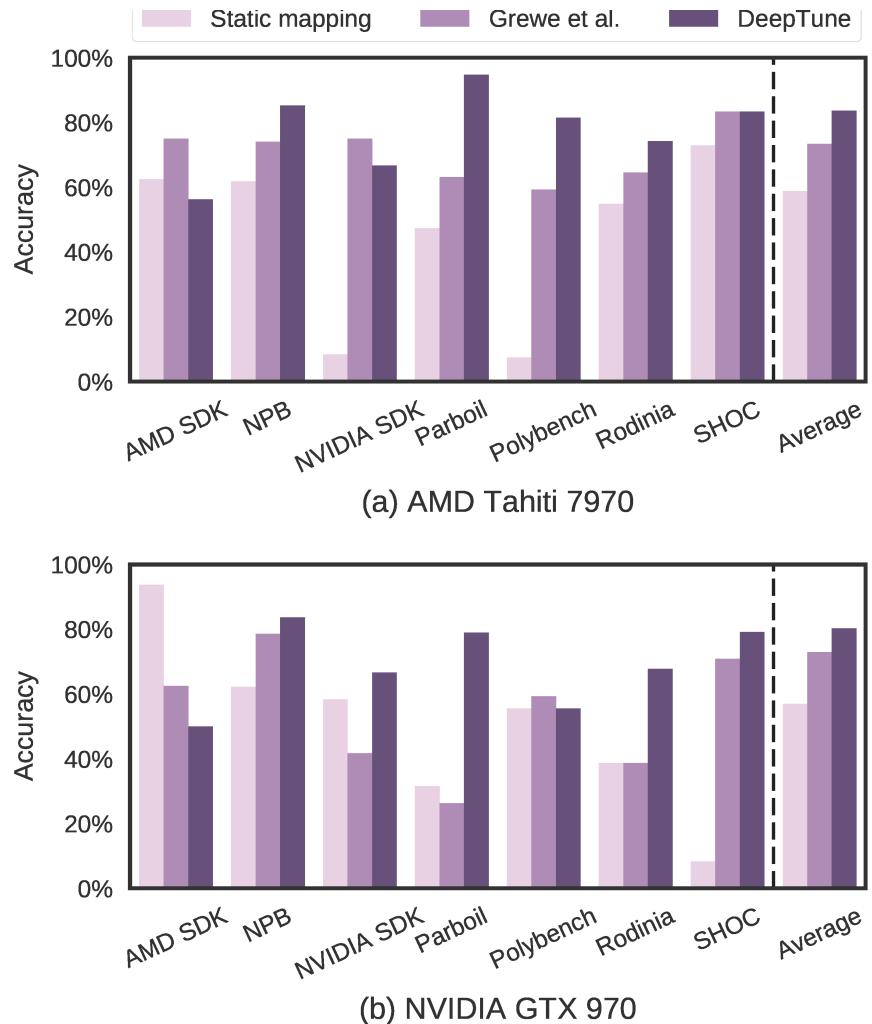


Figure 6.5: Accuracy of optimisation heuristics for heterogeneous device mapping, aggregated by benchmark suite. The optimal static mapping achieves 58% accuracy. The Grewe, Wang, and O’Boyle and DeepTune predictive models achieve accuracies of 73% and 84%, respectively.

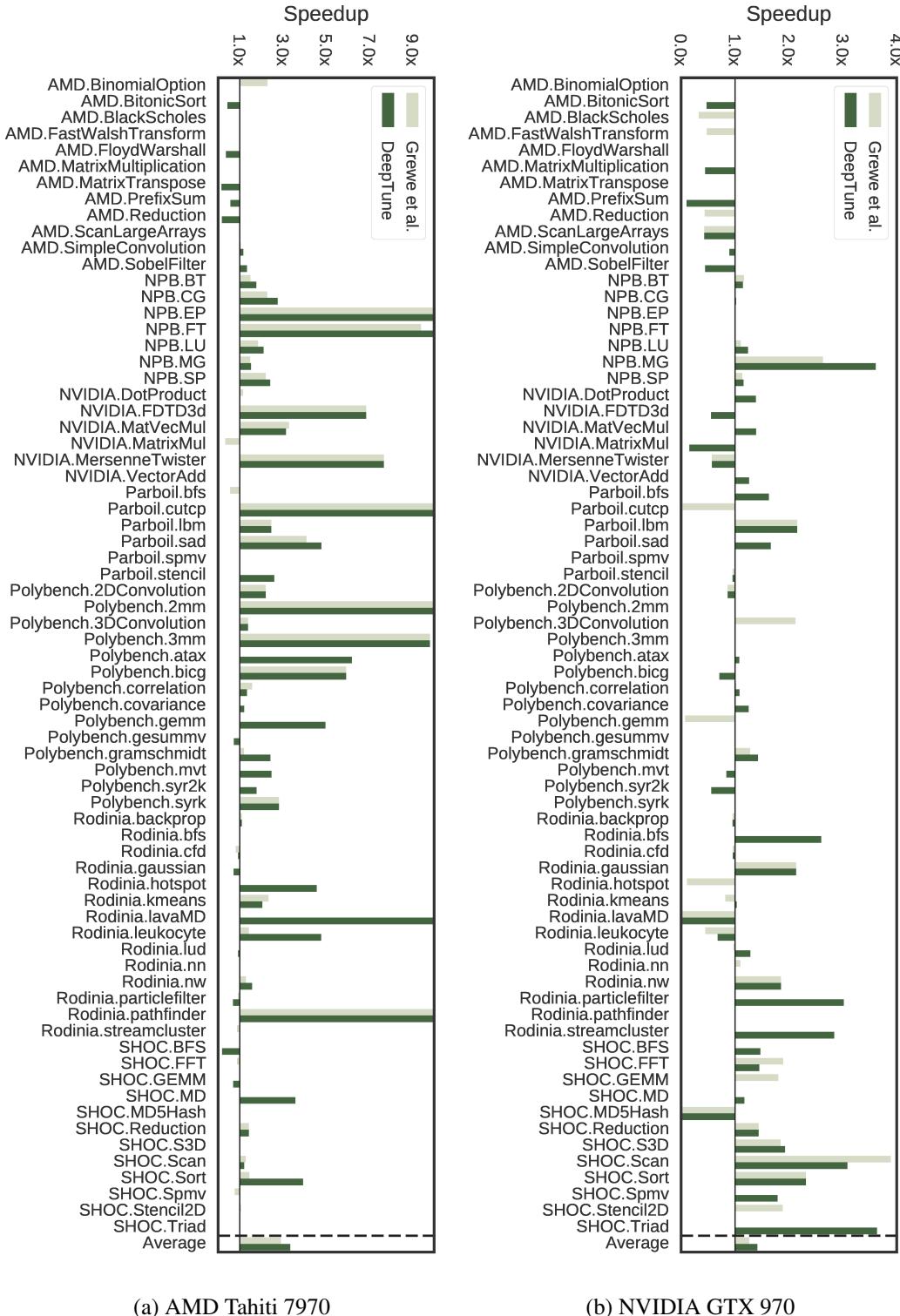
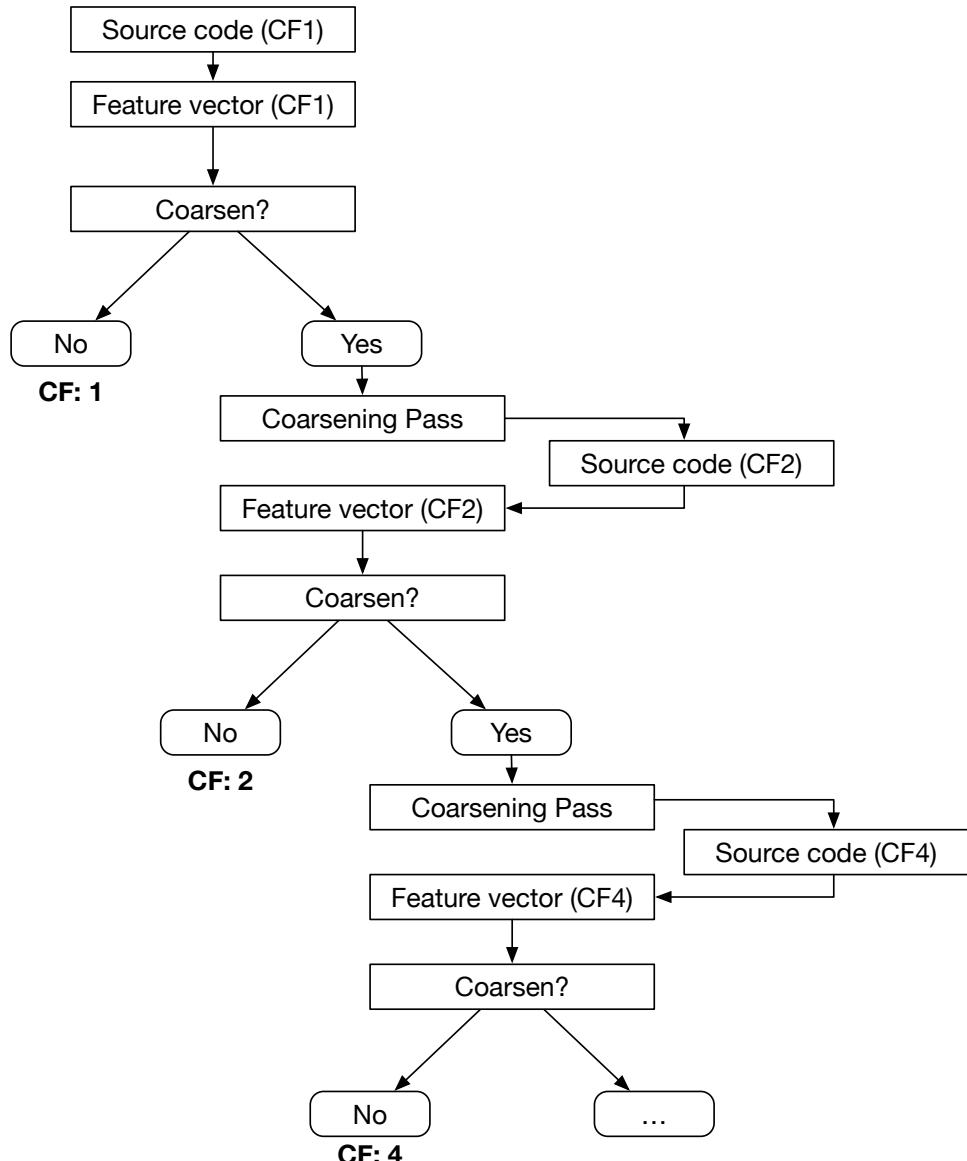
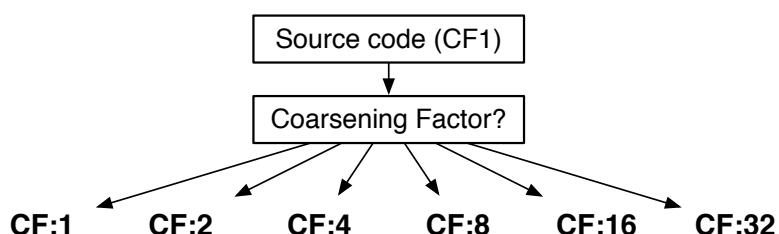


Figure 6.6: Speedup of predicted heterogeneous mappings over the best static mapping for both platforms. In (a), DeepTune achieves an average speedup of 3.43x over static mapping and 18% over Grewe, Wang, and O’Boyle. In (b), the speedup is 1.42x and 13% respectively.



(a) Magni, Dubach, and O'Boyle cascading binary model.



(b) Proposed approach.

Figure 6.7: Two approaches for predicting coarsening factor (CF) of OpenCL kernels. Magni, Dubach, and O'Boyle reduce the multi-label classification problem to a series of binary decisions by iteratively applying the optimisation and computing new feature vectors. The proposed approach simply predicts the coarsening factor directly from the source code.

Name	Description
BasicBlocks	#. basic blocks
Branches	#. branches
DivInsts	#. divergent instructions
DivRegionInsts	#. instructions in divergent regions
DivRegionInstsRatio	#. instr. in divergent regions / total instructions
DivRegions	#. divergent regions
TotInsts	#. instructions
FPIsts	#. floating point instructions
ILP	average ILP / basic block
Int/FP Inst Ratio	#. branches
IntInsts	#. integer instructions
MathFunctions	#. match builtin functions
MLP	average MLP / basic block
Loads	#. loads
Stores	#. stores
UniformLoads	#. loads unaffected by coarsening direction
Barriers	#. barriers

Table 6.4: Candidate features used by Magni, Dubach, and O’Boyle for predicting thread coarsening. From these values, they compute relative deltas for each iteration of coarsening, then use PCA for selection.

	Version	#. benchmarks	#. kernels
NVIDIA SDK	4.2	3	3
AMD SDK	3.0	10	10
Parboil [Str+12]	0.2	4	4
Total	-	17	17

Table 6.5: Benchmarks used in Case Study B.

	Frequency	Memory	Driver
AMD HD 5900	725 MHz	2GB	AMD 1124.2
AMD Tahiti 7970	1000 MHz	3GB	AMD 1084.4
NVIDIA GTX 480	700 MHz	1536 MB	NVIDIA 304.54
NVIDIA K20c	706 MHz	5GB	NVIDIA 331.20

Table 6.6: Experimental platforms used in Case Study B.

ies of performance counters and computed theoretical values [MDO13; Sim+12]. For each candidate feature, they compute its coarsening *delta*, reflecting the change in each feature value caused by coarsening: $f_\Delta = (f_{\text{after}} - f_{\text{before}})/f_{\text{before}}$, adding it to the feature set. Then they use Principal Component Analysis (PCA) on the 34 candidates and selected the first 7 principal components, accounting for 95% of variance in the space.

6.4.2 Experimental Setup

The experimental setup of Magni, Dubach, and O’Boyle [MDO14] is replicated for this case study. The thread coarsening optimisation is evaluated on 17 programs, listed in Table 6.5. Four different GPU architectures are used, listed in Table 6.6.

DeepTune Configuration Figure 6.4b shows the artificial neural network configuration. The OpenCL kernel is the sole input and coarsening factor is the predicted output.

Model Evaluation Compared to Case Study A, the size of the evaluation is small. As such, *leave-one-out cross-validation* is used to evaluate the models. For each program, a model is trained on data from all other programs and used to predict the coarsening factor of the excluded program.

	#. neurons		#. parameters	
	HM	CF	HM	CF
Embedding	64	64	256	256
LSTM_1	64	64	33,024	33,024
LSTM_2	64	64	33,024	33,024
Concatenate	64 + 2	-	-	-
Batch Normalisation	66	64	264	256
DNN_1	32	32	2,144	2,080
DNN_2	2	6	66	198
Total			68,778	68,838

Table 6.7: The size and number of parameters of the DeepTune components of Figure 6.4, configured for heterogeneous mapping (HM) and coarsening factor (CF).

The parameters of the artificial neural network are not described in [MDO14], so an additional, *nested* cross-validation process is used to find the optimal model parameters. For every program in the training set, a grid search of 48 combinations of network parameters is performed. The best performing parameter configuration is selected from these 768 results to train a model for prediction on the excluded program. This nested cross-validation is repeated for each of the training sets. No such tuning of hyper-parameters is performed for DeepTune.

6.4.3 Comparison to Case Study A

For the two different optimisation heuristics, the authors arrived at very different predictive model designs, with very different features. By contrast, the DeepSmith approach is exactly the same for both problems. None of DeepTune’s parameters were tuned for the case studies presented above. Their settings represent conservative choices expected to work reasonably well for most scenarios.

Table 6.7 shows the similarity of the models. The only difference between the network designs is the auxiliary inputs for Case Study A and the different number of optimisation decisions. The differences between DeepTune configurations is only two lines of code: the first, adding the two auxiliary inputs; the second, increasing the size of the output layer for Case Study B from two neurons to six. The description of these differences is larger than the differences themselves.

6.4.4 Experimental Results

Exploiting thread coarsening for OpenCL kernels is a difficult task. On average, coarsening slows programs down. The speedup attainable by a perfect heuristic is only $1.36\times$.

Figures 6.8 and 6.9 show speedups achieved by the Magni, Dubach, and O’Boyle and DeepTune models for all programs and platforms. The performance of programs without coarsening is used as a baseline. On the four experimental platforms (AMD HD 5900, Tahiti 7970, NVIDIA GTX 480, and Tesla K20c), the Magni, Dubach, and O’Boyle model achieves average speedups of $1.21\times$, $1.01\times$, $0.86\times$, and $0.94\times$, respectively. DeepTune outperforms this, achieving speedups of $1.10\times$, $1.05\times$, $1.10\times$, and $0.99\times$.

Some programs — especially those with large divergent regions or indirect memory accesses — respond poorly to coarsening. No performance improvement is possible on the `mvCoal` and `spmv` programs. Both models fail to achieve positive average speedups on the NVIDIA Tesla K20c, because thread coarsening does not give performance gains for the majority of the programs on this platform.

The disappointing results for both predictive models may be attributed to the small training program set (only 17 programs in total). As a result, the models suffer from sparse training data. In Chapter 4 of this thesis, a methodology for overcoming data sparsity using additional programs is presented. In this instance, the shared structure of the DeepTune models between the two case studies enables an alternative strategy to overcome data scarcity. The following subsection describes and tests a novel strategy for training optimisation heuristics on a small number of programs by exploiting knowledge learned from other optimisation domains.

6.5 Transfer Learning Across Problem Domains

There are inherent differences between the tasks of building heuristics for heterogeneous mapping and thread coarsening, evidenced by the contrasting choices of features and models used by the state-of-the-art approaches in Case Studies A and B. However, in both cases, the first role of DeepTune is to extract meaningful abstractions and representations of OpenCL code. Prior research in deep learning has shown that models trained on similar inputs for different tasks often share useful commonalities. The idea is that in classification using artificial neural networks, information learned at the early

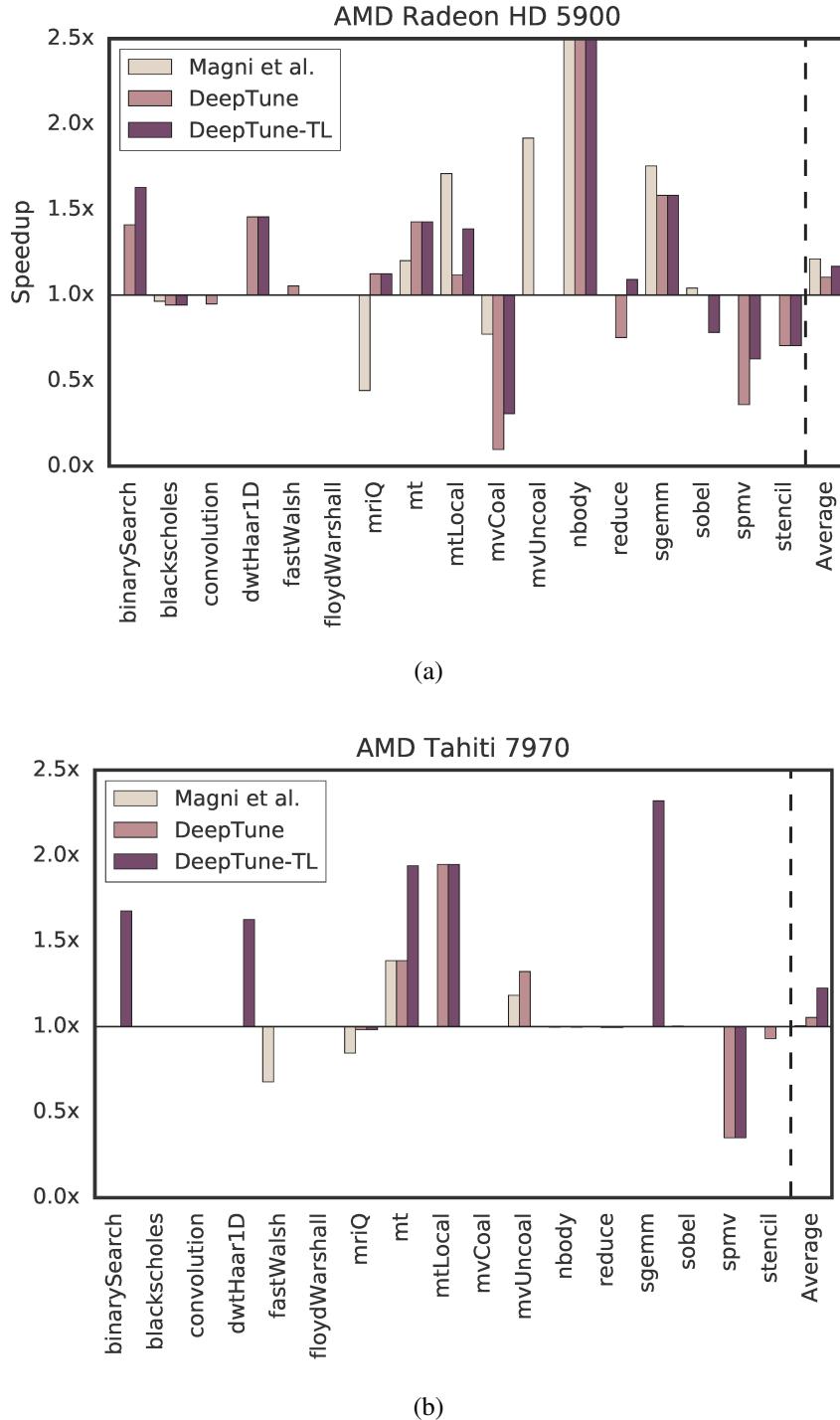


Figure 6.8: Speedups of predicted coarsening factors on AMD platforms. DeepTune outperforms Magni, Dubach, and O’Boyle on three of the four platforms. Transfer learning improves DeepTune speedups further, by 16% on average.

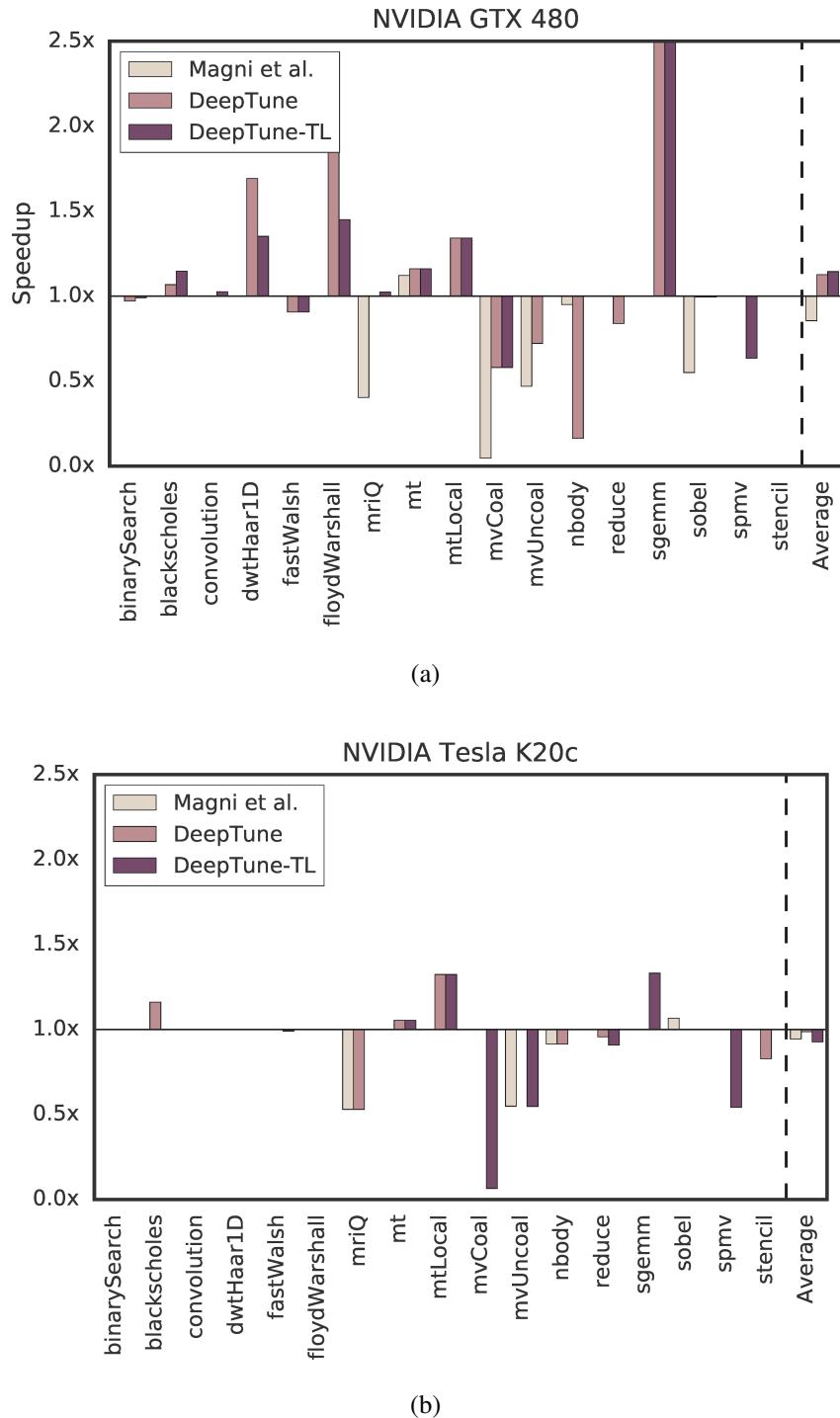


Figure 6.9: Speedups of predicted coarsening factors on NVIDIA platforms. DeepTune outperforms Magni, Dubach, and O’Boyle on three of the four platforms. Transfer learning improves DeepTune speedups further, by 16% on average.

layers of artificial neural networks (i.e. closer to the input layer) will be useful for multiple tasks. The later the network layers are (i.e. closer to the output layer), the more specialised the layers become [ZF14].

Hypothesising that this would be the case for DeepTune would enable the novel transfer of information *across different optimisation domains*. To test this, the language model — the Embedding, and LSTM_{1, 2} layers — trained for the heterogeneous mapping task was extracted and *transferred* over to the new task of thread coarsening. Since DeepTune keeps the same design for both optimisation problems, this is as simple as copying the learned parameters of the three layers. The model is then trained as normal.

As shown in Figures 6.8 and 6.9, the newly trained model, DeepTune-TL has improved performance for 3 of the 4 platforms: 1.17×, 1.23×, 1.14×, 0.93×, providing an average 12% performance improvement over Magni, Dubach, and O’Boyle. In 81% of cases, the use of transfer learning matched or improved the optimisation decisions of DeepTune, providing up to a 16% improvement in per platform performance.

On the NVIDIA Tesla K20c, the platform for which no predictive model achieves positive average speedups, DeepTune-TL matches or improve performance in the majority of cases, but over-coarsening on three of the programs causes a modest reduction in average performance. For this platform, further performance results are suspected necessary due to its unusual optimisation profile.

6.6 DeepTune Internal Activation States

In previous sections, DeepTune is shown to automatically outperform state-of-the-art predictive models for which experts have invested a great amount of time in engineering features. This section attempts to illuminate the inner workings, using a single example from Case Study B: predicting the thread coarsening factor for Parboil’s `mriQ` benchmark on four different platforms.

Figure 6.10 shows the DeepTune configuration, with visual overlays showing the internal state. From top to bottom, the input to the model is the 267 lines of OpenCL code for the `mriQ` kernel. This source code is preprocessed, formatted, and rewritten using variable and function renaming, shown in Figure 6.10b. The rewritten source code is tokenised and encoded in a 1-of- k vocabulary. Figure 6.10c shows the first 80 elements of this encoded sequence as a heat map in which each cell’s colour reflects its encoded value. The input, rewriting, and encoding is the same for each of the four

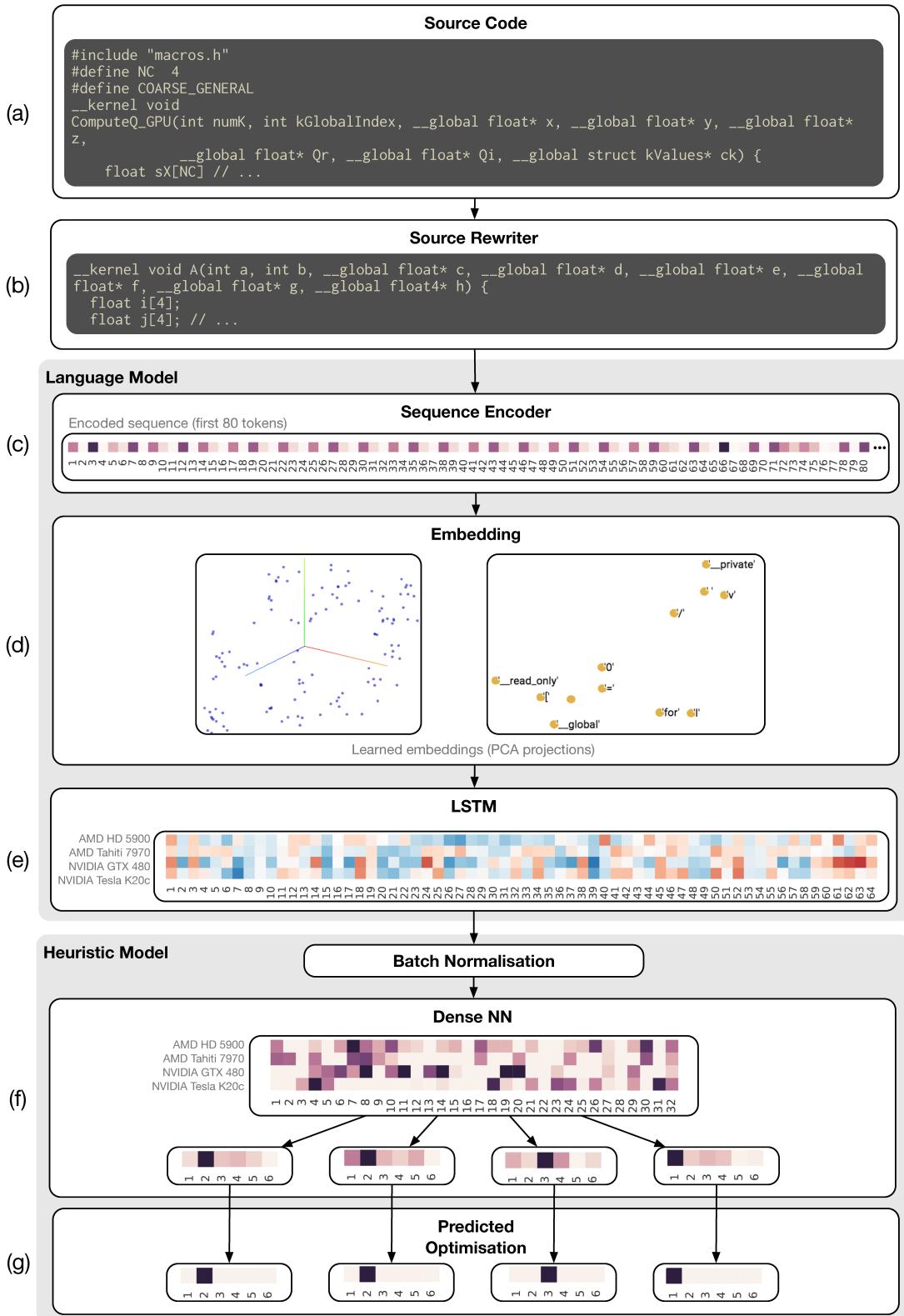


Figure 6.10: Visualising the internal state of DeepTune when predicting coarsening factor for Parboil’s `mriQ` benchmark on four different architectures. The activations in each layer of the four models increasingly diverge the lower down the network.

platforms.

The encoded sequences are then passed into the Embedding layer. This maps each token of the vocabulary to a point in a 64 dimension vector space. Embeddings are learned during training so as to cluster semantically related tokens together. As such, they may differ between the four platforms. Figure 6.10d shows a 3-dimensional PCA projection of the embedding space for one of the platforms, showing multiple clusters of tokens. Although achieving good separation of embedding clusters typically requires a much larger training set [BJH18], by honing in on one of the clusters and annotating each point with its corresponding token, it can be observed that the cluster contains the semantically related OpenCL address space modifiers `_private`, `_global`, and `_read_only`.

Two layers of 64 LSTM neurons model the sequence of embeddings, with the neuron activations of the second layer being used to characterise the entire sequence. Figure 6.10e shows the neurons in this layer for each of the four platforms, using a red-blue heat map to visualise the intensity of each activation. Comparing the activations between the four platforms reveals a number of neurons in the layer with different responses across platforms. This indicates that the language model is partly specialised to the target platform. Subsequent work [BJH18] supports this reasoning, in which performance is slightly degraded by training parts of the language model in a platform-agnostic manner.

As information flows through the network, the layers become progressively more specialised to the specific platform. This can be seen in Figure 6.10f, which shows the two layers of the heuristic model. The activations within these increasingly diverge. The mean variance of activations across platforms increases threefold compared to the language model, from 0.039 to 0.107. Even the activations of the AMD HD 5900 and AMD Tahiti 7970 platforms are dissimilar, despite the final predicted coarsening factor for both platforms being the same. The largest activation of the output layer is taken in Figure 6.10g as the final predicted coarsening factor. For this particular program, a state-of-the-art model achieves 54% of the maximum performance. DeepTune achieves 99%.

6.7 Summary

Applying machine learning to compiler and runtime optimisations requires generating features first. This is a time-consuming process, it needs supervision by an expert,

and even then one cannot be sure that the selected features are optimal. This *feature design* challenge (Section 1.2.2) places a significant burden on developers looking to adopt machine learning for constructing compiler optimisation heuristics. This chapter presents a novel tool for building optimisation heuristics, DeepTune, which forgoes feature extraction entirely, relying on powerful language modelling techniques to automatically build effective representations of programs directly from raw source code. The result translates into a huge reduction in development effort, improved heuristic performance, and simpler model designs.

The approach is fully automated. Using DeepTune, developers no longer need to spend months using statistical methods and profile counters to select program features via trial and error. It is worth mentioning that the model design or parameters are not tailored for the optimisation task at hand, yet DeepTune achieves performance on par with and in most cases *exceeding* state-of-the-art predictive models.

In this chapter, DeepTune is used to automatically construct heuristics for two challenging compiler and runtime optimisation problems. In both cases, DeepTune is found to outperform state-of-the-art predictive models by 14% and 12%. The DeepTune architecture is shown also to allow the exploitation of information learned from another optimisation problem to give the learning a boost. Doing so provides up to a 16% performance improvement when training using a handful of programs. This approach may prove useful in other domains for which training data are a scarce resource.

Chapter 7

Conclusions

This thesis presents new techniques for the generation and optimisation of programs using deep learning to address the challenges of compiler construction outlined in Section 1.2. Chapter 4 addresses the data scarcity issue (Section 1.2.1) by developing a methodology for the unguided generation of realistic benchmarks. Chapter 5 addresses the adoption challenge (Section 1.2.3) by developing an effective compiler fuzzer that is significantly simpler than state-of-the-art approaches, extending the generative technique to the domain of compiler validation. Finally, Chapter 6 explores techniques that address the feature design challenge (Section 1.2.2) by addressing the difficult and time-consuming task of manually constructing features for programs.

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

7.1 Contributions

The challenges identified in Section 1.2 prevent the adoption of machine learning in compilers. This section summarises the main contributions of this thesis which make a strong argument for machine learning as a valuable tool in compiler construction.

7.1.1 A Solution for Benchmark Scarcity

There is a shortage of benchmarks, forcing compiler developers to work with sparse samplings of the program space. This data scarcity challenge limits the quality of

learned models. Chapter 4 develops a novel generator for compiler benchmarks, capable of generating an unbounded number of training programs. The usefulness of the generated benchmarks is evaluated on a state-of-the-art learned optimisation heuristic, finding that the additional exploration of the program space provided by the generated benchmarks improves performance by $1.27\times$.

This is the first use of machine learning over handwritten code to generate benchmarks. Compared to previous works [CGA15], this approach is entirely automatic, requiring no expert tuning or direction. Only a corpus of example programs is needed to guide the distributions of generated programs. Despite no a priori knowledge of the programming language, the generator is capable of producing executable benchmarks of such quality that professional software developers cannot distinguish code generated by it from handwritten code.

The approach, in generating an unbounded number of runnable programs, enables a finer-grained exploration of the compiler optimisation feature space than was previously possible, without the development costs previously associated with benchmark generation. This simplifies the construction of compilers by enabling performance models to be learned from automatically-generated data, and addresses the data scarcity challenge described in Section 1.2.1. The technique may also prove valuable to compiler feature designers (Section 1.2.2), as the granular exploration of the feature space exposes deficiencies in the choice of features.

7.1.2 Low-cost and Effective Compiler Fuzzing

For machine learning techniques to be widely adopted in compilers, they must be present significant advantages of existing formal methods, without introducing errors. Chapter 5 extends the application of recurrent neural networks to the domain of compiler testing to address this adoption challenge. Development of the state-of-the-art compiler test case generator CSmith was a significant undertaking, comprising over 50,000 lines of handwritten code [Lid+15; Yan+11]. The technique presented in this thesis presents an enormous reduction in developer effort compared to the state-of-the-art grammar-based approach. It is implemented in as few as 500 lines of code. This $100\times$ reduction in code size is complemented by improved portability of the implementation, with only parts of the stack being specific to the input language of the compiler being tested. The remainder being language-agnostic. This lowers the barrier to entry for developing new fuzzers for addressing the compiler validation problem.

The portability of the approach is demonstrated by extending the generator from its initial implementation targeting OpenCL to the Solidity programming language. This extension required only 18 lines of code. By contrast, extending a state-of-the-art generator from C to OpenCL required over 8000 lines of code [Lid+15].

Despite its simplicity, the proposed technique is effective. To date, 67 new bugs in commercial and open source OpenCL compilers have been identified and reported. Many of the bugs identified could not be exposed by state-of-the-art approaches due to the limitations in the expressiveness of prior grammar-based approaches. The expressiveness of the generated test cases is limited only by the code that has been uploaded to GitHub; this led to unintentional outcomes such as exploiting compiler-specific features to expose bugs in the error handling of compilers’ intrinsics. By enabling probabilistic machine learning systems to be developed at such low cost and without compromising the correctness of the differential testing approach, this work aims to address the challenge of encouraging adoption of machine learning practices by the compiler community, described in Section 1.2.3.

7.1.3 Automatic Compiler Optimisation Tuning

A significant challenge facing machine learning techniques is the design of features. Constructing program features is time-consuming and error-prone. Additionally, the choice of features typically couples the learning system tightly with the compiler implementation. This means that new features must be computed and the model retrained with every change to the compiler. Chapter 6 proposes a technique to address both issues. Instead of extracting numerical representations of programs, a learning system is developed that permits the entire source code of the program to be fed directly as input. This simpler approach decouples the learning systems from the compilers’ internal representations.

The technique is evaluated for two distinct optimisation problems, finding that in both cases, the approach is able to match or outperform the state-of-the-art approach using hand-crafted features, achieving speedups of $1.14\times$ and $1.05\times$. This is in spite of using the same model parameters for both problems, without any specialising of the structure of the learning system to the task being learned. In abstracting the structure of the solution from the problem, the approach enables the novel transfer of information learned for one task to the other. By enabling transfer learning, the performance of a predictive model improves by a further $1.06\times$, despite only being provided with

information learned for a different optimisation task.

In bypassing the need to engineer features, the proposed technique simplifies the construction of optimisation heuristics through machine learning, while leading to higher performance in the heuristics themselves. This addresses the feature design challenge described in Section 1.2.2. Since compilers typically contain hundreds or even thousands of distinct optimisation heuristics, techniques that enable the sharing of information between tasks, like the one proposed in this work, are prudent to the practical development of machine learning in optimising compilers.

7.2 Critical Analysis

This section contains a critical analysis of the techniques presented in this work.

7.2.1 Generative Models for Source Code

Chapters 4 and 5 develop generative models that enable the synthesis of more human-like programs than current state-of-the-art program generators, and without the expert guidance required by template-based generators, but they have limitations. The technique of seeding the language models with the start of a function means that user-defined types or calls to user-defined functions are not supported. In turn, this restricts the inputs that can be fed to generated programs. Currently, only scalars and arrays may be used as inputs, whereas 6 (2.3%) of the OpenCL benchmark kernels listed in Table 4.3 use irregular data types as inputs. This may be addressed through recursive program synthesis, whereby a call to a user-defined function or unrecognised type will trigger candidate functions and type definitions to be synthesised.

This work evaluates the use of recurrent neural networks for generating programs in the OpenCL and Solidity programming languages. Although the languages are dissimilar (one extends the C programming language, the other is derived from JavaScript), it is unclear whether the generative modelling approach will prove effective for all possible grammars. Unlike approaches which generate programs by enumerating randomly from a specification of the programming language grammar, the ability to generate programs of arbitrary syntaxes cannot be guaranteed.

By learning from a corpus of programs assembled from GitHub, the model induces the biases of programs on GitHub. This makes the implicit assumption that code uploaded to GitHub is representative of the real-world usage of a programming language.

The contents of the GitHub corpus used in this work were only lightly vetted to ensure that it did not contain programs that would later be used to evaluate the model. This did not preclude the model training on programs that may not be considered *representative* of true handwritten code. For example, inspecting the corpus revealed a small number of large, automatically generated programs which may bias the generator. Additionally, test cases for an OpenCL static analysis tool were found that deliberately contain runtime defects. While the corpus was filtered to ensure that training programs were syntactically valid, no checks were made to ensure that programs used for training had correct semantics.

7.2.2 Rejection Sampling for Program Generation

The techniques presented in this work sample recurrent neural networks on a per-token basis to generate programs. Once an entire sample has been generated, the sample is checked to see if it is a valid program. If not, the entire sample is discarded. Although automatic, this *rejection sampling* approach is wasteful. Grammar-based sampling approaches have been proposed that could increase the likelihood of generating a valid program through masked sampling [Dye+16]. Of course, this would make the generator more complicated. Ultimately there is a trade-off between implementation complexity and sampling efficiency. This work emphasises simplicity.

Moreover, rejection sampling results in a bias towards shorter programs. This is because, on average, the probability that a sample is a valid program decreases with each additional token. This skews the distribution of generated programs away from the training programs. This issue, arising from rejection sampling, can coincidentally be alleviated through further rejection sampling. To correct the bias towards shorter programs, an additional filter could be placed on the output of the generative model that discards samples with a random probability inversely proportional to their length. By removing more short samples than long, the bias in the distribution is corrected, albeit at the cost of fewer accepted samples.

7.2.3 Characterisation of OpenCL Compiler Bugs

Chapter 5 presents *DeepSmith*, a tool for generating compiler test cases, and compares it against the state-of-the-art *CLSmith*. For each approach, the number of bug-exposing test cases is reported. However, it is not possible to determine which generator identified more *unique* bugs. To determine this, one would need to de-duplicate the counts

by locating the exact bug-exposing property of each test case and correlating it with a compiler defect. There are two challenges preventing this: the first is the amount of compute required to perform automated test case reduction in many thousands of CLSmith programs; the second is that in the general case it is not possible to identify the root cause of a compiler bug without access to its source code.

While it is not possible to compare the rate at which DeepSmith and CLSmith identify unique bugs, the properties of each approach allow partial characterisation of the bugs that can be found. DeepSmith is capable of exposing bugs that CLSmith cannot; for example, by generating plausible but malformed inputs to expose bugs in compiler error handling, or by generating programs with thread-dependent control-flow which CLSmith’s static analyses prevent.

Where de-duplication of underlying bugs is possible (such as by comparing compiler stack traces), DeepSmith matches or exceeds CLSmith’s findings; however, CLSmith is also capable of exposing bugs that DeepSmith cannot. For example, CLSmith programs make heavy use of structs, which DeepSmith does not support. As such I believe the approach presented in this work to be complementary to the prior art. The functionality of DeepSmith and CLSmith overlap, but neither is a superset of the other.

7.2.4 Driving arbitrary OpenCL kernels

This thesis presents a technique for driving arbitrary OpenCL kernels, provided they have regular scalar or array inputs. This host driver accepts as input an OpenCL kernel, which it then compiles, produces input data sets, and runs the compiled kernel using the data sets. The host driver generates data sets from uniform random distributions, as do many OpenCL benchmark suites. For cases where non-uniform inputs are required (e.g. profile-directed feedback), an alternative methodology for generating inputs must be adopted.

7.2.5 Modelling Program Semantics from Syntactic Sequences

Chapter 6 feeds a sequence of program tokens into a recurrent neural network to predict an optimisation decision that should be made on it. By treating the serialised representation of a program (its source code) as the sequence of syntactic tokens, the technique is vulnerable to changes in code order, since $p(y|[\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \mathbf{x}^{(3)}]) \neq p(y|[\mathbf{x}^{(3)}, \mathbf{x}^{(2)}, \mathbf{x}^{(1)}])$. The text inputs used to evaluate the approach are single kernels. It is not clear how the approach will respond to multi-procedure inputs, where the or-

der that procedures are declared may have a large impact on the pattern of activations produced in the recurrent neural network.

A common criticism of machine learning systems is that they are *black boxes*. When the system fails to produce the desired result, there is no obvious method to correct the system so as to prevent similar errors. Still, in traditional machine learning, it may be possible to correct problems by adjusting the features. In an absence of features, there are fewer meaningful ways to improve a model based on an analysis of failure cases.

7.3 Future Work

This section outlines four promising avenues for future research enabled by this thesis.

7.3.1 Guided Program Synthesis to Minimise Benchmarking Cost

This thesis presents a technique for the *unguided* synthesis of compiler benchmarks. Using the technique may provide a fine-grained exploration of the space of representative programs. For some use cases, more efficient use of data will be achieved through *directed* program generation.

One approach to direct program generation could be to employ a rejection filter that tests for the presence of a property of interest and rejects programs that do not satisfy this property. Another approach would be to train the generative model simultaneously for both the structure of programs (as is done in this work), along with a representation of the properties of interest (such as a feature vector). At sample time, the feature values of the desired program could be used as input to steer the program generation. A third approach would use the learned language model not to generate programs, but to guide an existing program generator by biasing the weights of grammar productions.

If successful, such a technique would enable the exploration of larger feature spaces than is currently possible by efficiently navigating the generation of benchmarks to map out the decision surface. Additional insight into the behaviour of a model and its failure cases would be enabled by steering synthesis towards the parts of the space where the model has the lowest confidence, or parts of the space where the model frequently makes wrong predictions.

7.3.2 Neural Model Selection through Adversarial Games

Section 4.5 employs *Turing tests* to evaluate the quality of synthetic code. The task presented to human participants was to identify whether a series of code snippets were written by hand or machine. This was used to evaluate whether or not the model produced human-like output. In future work, this approach could be extended to aid in the challenging task of model selection by instead presenting the participants with pairs of samples side by side, and asking the participant to select the sample which is *more* human-like. If the two samples were both generated by different configurations of a generative model, this would provide a means to compare generative models on the otherwise hard-to-assess quality of “humanness”. The selection of the best model from a pool of candidates is thus turned into a series of zero-sum games, where each game pits a single sample from a pair of generative models head-to-head with a human selecting the winner, and an Elo rating can be used to assign scores and pair matches. The limiting factor of this approach would likely be the availability of human participants.

Generative Adversarial Networks (GANs) [Goo+14] employ a similar adversarial approach, but use a second artificial neural network as a discriminative adversary. The generator and discriminator networks are trained concurrently; the generator is trained to maximise the probability of failure in the discriminator, the discriminator is trained to minimise this probability. This approach is a good fit for the domain of program generation, though filtering of the generator outputs to check for program correctness may be required to prevent the generator training to a local maxima where the output is hard to distinguish from the test set, but rarely contains meaningful program semantics.

7.3.3 Learning Representations for Dynamic Program Inputs

Chapter 6 presents an approach for learning optimisation heuristics from the raw representation of a program, but in the presence of dynamic properties, traditional feature extraction was used. For example, feeding in the size of input data sets. In future work, this approach could be extended to also account for dynamic properties. Unlike with program source code, it is not clear what the raw representation of program inputs may be as there is no equivalent human-interpretable representation of program data. One approach could be to model the sequence of bytes that the program reads and writes, though this may introduce a high overhead for runtime instrumentation [GSB14].

7.3.4 Towards General-Purpose Program Comprehension

The techniques presented in this thesis apply deep learning techniques to the task of modelling the behaviour of programs. For each task, be it program generation or optimisation, artificial neural networks are trained from scratch. Chapter 6 explores the use of transfer learning to seed an artificial neural network with information from another task. Future work could explore this idea further by iteratively training and retraining a single network across a wide range of tasks, with the goal of finding a common set of model parameters to be used as an effective base for each task.

An ambitious goal would be the development and distribution of a model architecture for *general-purpose* program comprehension. Such a system would enable, with little effort, a single model to be re-purposed for a variety of compiler tasks. This is analogous to the widespread distribution of pre-trained state-of-the-art models in the field of image recognition. If developing an image classifier, a user may start by re-training an existing model such as ResNet [He+16], rather than constructing a model from scratch. This drastically simplifies the adoption of machine learning for image classification as the model architecture has been pre-selected and tuned, and reduces the amount of training data required.

A prerequisite for developing this system will be applying techniques such as those proposed in this thesis to a wide range of different compiler tasks. My hope in publicly releasing all of the software developed during the course of my research is to enable and expedite this discovery in other domains.

7.4 Summary

An optimising compiler’s critical role in translating software to machine code must be performed without error, while maximising the performance and efficiency of the generated code. This is a challenging task that is only getting harder as the complexity and number of architectures that must be supported by compilers grows. Addressing this challenge requires developing new tools that lower the cost of compiler construction.

This thesis leverages deep learning to develop new methodologies for program generation and optimisation that are dramatically simpler than established approaches. The methodologies are applied across two domains: compiler optimisations and compiler validation, and in both cases the proposed approaches outperform state-of-the-art techniques that are the culmination of years of development.

The outcomes demonstrated in this thesis open new lines of research into the modelling, optimisation, and generation of programs through deep learning. The preliminary results are promising and there is much work to be done. The development of automated learning systems capable of reasoning about complex program semantics would enable a dramatic rethinking of compiler construction methodologies. Promising future research directions include adapting these techniques to new problem domains, further reducing development costs through end-to-end machine learning, and developing improved program representations for artificial neural networks that overcome limitations in current approaches. My hope in publicly releasing¹ all of the code and data that has made up the work in this thesis is to expedite discovery in other domains.

¹All of the code and data for this thesis is available at: <https://chriscummins.cc/phd>.

Bibliography

- [Aba+16] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. “TensorFlow: A System for Large-scale Machine Learning”. In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2016.
- [Aga+06] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. O’Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. “Using Machine Learning to Focus Iterative Optimization”. In: *International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2006.
- [All+14] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. “Learning Natural Coding Conventions”. In: *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2014.
- [All+17] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton. “A Survey of Machine Learning for Big Code and Naturalness”. In: *arXiv:1709.06182* (2017).
- [All18] M. Allamanis. “The Adverse Effects of Code Duplication in Machine Learning Models of Code”. In: *arXiv:1812.06469* (2018).
- [And19] A. S. G. Andrae. “Prediction Studies of Electricity Use of Global Computing in 2030”. In: *International Journal of Science and Engineering Investigations (IJSEI) 8.86* (2019).
- [Ans+09] A. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. “PetaBricks: A Language and Compiler for Algorithmic Choice”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2009.

- [Ans+13] J. Ansel, S. Kamil, K. Veeramachaneni, U. O. Reilly, and S. Amarasinghe. “OpenTuner: An Extensible Framework for Program Autotuning”. In: *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM, 2013.
- [Ans14] J. Ansel. “Autotuning Programs with Algorithmic Choice”. PhD thesis. Massachusetts Institute of Technology, 2014.
- [APS16] M. Allamanis, H. Peng, and C. Sutton. “A Convolutional Attention Network for Extreme Summarization of Source Code”. In: *International Conference on Machine Learning (ICML)*. PMLR, 2016.
- [AR12] J. Ansel and U. O. Reilly. “SiblingRivalry: Online Autotuning Through Local Competitions”. In: *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*. ACM, 2012.
- [AS13] M. Allamanis and C. Sutton. “Mining Source Code Repositories at Massive Scale using Language Modeling”. In: *Working Conference on Mining Software Repositories (MSR)*. 2013.
- [Ash+17] A. H. Ashouri, A. Bignoli, G. Palermo, C. Silvano, S. Kulkarni, and J. Cavazos. “MiCOMP: Mitigating the Compiler Phase-ordering Problem Using Optimization Sub-sequences and Machine Learning”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* (2017).
- [Ash+18] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano. “A Survey on Compiler Autotuning using Machine Learning”. In: *ACM Computing Surveys (CSUR)* 51.5 (2018).
- [Bad+19] J. Bader, A. Scott, M. Pradel, and S. Chandra. “Getafix: Learning to Fix Bugs Automatically”. In: *arXiv:1902.06111* (2019).
- [Bai+14] R. Baishakhi, D. Posnett, V. Filkov, and P. Devanbu. “A Large Scale Study of Programming Languages and Code Quality in Github”. In: *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2014.
- [Bai+91] D. H. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. “The NAS Parallel Benchmarks”. In: *International Journal of High Performance Computing Applications (IJHPCA)* 5.3 (1991).

- [Bas+17] O. Bastani, R. Sharma, A. Aiken, and P. Liang. “Synthesizing Program Input Grammars”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2017.
- [BCD12] A. Betts, N. Chong, and A. Donaldson. “GPUVerify: A Verifier for GPU Kernels”. In: *Object-oriented Programming Systems, Languages and Applications (OOPSLA)*. ACM, 2012.
- [BDB00] V. Bala, E. Duesterwald, and S. Banerjia. “Dynamo: A Transparent Dynamic Optimization System”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2000.
- [BDK14] M. Baroni, G. Dinu, and G. Kruszewski. “Don’t Count, Predict! A Systematic Comparison of Context-Counting vs . Context-Predicting Semantic Vectors”. In: *Annual Meeting of the Association for Computational Linguistics (ACL)*. 2014.
- [Ber+10] J. L. Bernal, Í. Goiri, R. Nou, F. Julià, J. Guitart, R. Gavaldà, and J. Torres. “Towards Energy-aware Scheduling in Data Centers using Machine Learning”. In: *International Conference on energy-Efficient Computing and Networking (e-Energy)*. ACM, 2010.
- [Ber+11] J. Bergstra, F. Bastien, O. Breuleux, P. Lamblin, R. Pascanu, O. Delalleau, G. Desjardins, D. Warde-Farley, I. Goodfellow, A. Bergeron, and Y. Bengio. “Theano: Deep Learning on GPUs with Python”. In: *BigLearning Workshop*. 2011.
- [Ber+12] R. Bertran, A. Buyuktosunoglu, M. S. Gupta, M. Gonzalez, and P. Bose. “Systematic Energy Characterization of CMP/SMT Processor Systems via Automated Micro-Benchmarks”. In: *International Symposium on Microarchitecture (MICRO)*. IEEE, 2012.
- [BJ05] R. H. Bell and Lizy K. John. “Improved Automatic Testcase Synthesis for Performance Model Validation”. In: *ACM/IEEE Supercomputing Conference (SC)*. 2005.
- [BJH18] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler. “Neural Code Comprehension: A Learnable Representation of Code Semantics”. In: *Conference on Neural Information Processing Systems (NeurIPS)*. 2018.
- [BJR19] H. Babii, A. Janes, and R. Robbes. “Modeling Vocabulary for Big Code Machine Learning”. In: *arXiv:1904.01873* (2019).

- [Bod+98] F. Bodin, T. Kisuki, P. M. W. Knijnenburg, M. O’Boyle, and E. Rohou. “Iterative Compilation in a Non-linear Optimisation Space”. In: *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM, 1998.
- [Bow+16] S. R. Bowman, L. Vilnis, O. Vinyals, A. M. Dai, R. Jozefowicz, and S. Bengio. “Generating Sentences from a Continuous Space”. In: *SIGNLL Conference on Computational Natural Language Learning (CONLL)*. 2016.
- [Bro+18] M. Brockschmidt, M. Allamanis, A. L. Gaunt, and O. Polozov. “Generative Code Modeling with Graphs”. In: *arXiv:1805.08490* (2018).
- [BS97] A. S. Boujarwah and K. Saleh. “Compiler Test Case Generation Methods: A Survey and Assessment”. In: *Information and Software Technology* 39.9 (1997).
- [BSF94] Y. Bengio, P. Simard, and P. Frasconi. “Learning Long-Term Dependencies with Gradient Descent is Difficult”. In: *IEEE Transactions on Neural Networks* 5.2 (1994).
- [Cav+06] J. Cavazos, C. Dubach, F. Agakov, E. Bonilla, M. O’Boyle, G. Fursin, and O. Temam. “Automatic Performance Model Construction for the Fast Software Exploration of New Hardware Designs”. In: *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*. 2006.
- [CDP15] K. Claessen, J. Duregård, and M. H. Pałka. “Generating Constrained Random Data with Uniform Distribution”. In: *Journal of Functional Programming* 25 (2015).
- [CFL12] A. Collins, C. Fensch, and H. Leather. “Auto-Tuning Parallel Skeletons”. In: *Parallel Processing Letters* 22.02 (2012).
- [CGA15] A. Chiu, J. Garvey, and T. S. Abdelrahman. “Genesis: A Language for Generating Synthetic Training Programs for Machine Learning”. In: *ACM International Conference on Computing Frontiers (CF)*. ACM, 2015.
- [Cha+09] C. Chan, H. Ansel, Y. L. Wong, S. Amarasinghe, and A. Edelman. “Autotuning Multigrid with PetaBricks”. In: *ACM/IEEE Supercomputing Conference (SC)*. 2009.

- [Che+09] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. “Rodinia: A Benchmark Suite for Heterogeneous Computing”. In: *International Symposium on Workload Characterization (IISWC)*. IEEE, 2009.
- [Che+10] Y. Chen, Y. Huang, L. Eeckhout, G. Fursin, L. Peng, O. Temam, and C. Wu. “Evaluating Iterative Optimization Across 1000 Data Sets”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2010.
- [Che+13] Y. Chen, A. Groce, C. Zhang, W. Wong, X. Fern, E. Eide, and J. Regehr. “Taming Compiler Fuzzers”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2013.
- [Che+16a] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie. “An Empirical Comparison of Compiler Testing Techniques”. In: *International Conference on Software Engineering (ICSE)*. 2016.
- [Che+16b] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao. “Coverage-Directed Differential Testing of JVM Implementations”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2016.
- [Che+17] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, and B. Xie. “Learning to Prioritize Test Programs for Compiler Testing”. In: *International Conference on Software Engineering (ICSE)*. 2017.
- [Che+18] Z. Chen, S. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, and M. Monperrus. “SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair”. In: *arXiv:1901.01808* (2018).
- [Che+19] L. Cheng, Y. Zhang, Y. Zhang, C. Wu, Z. Li, Y. Fu, and H. Li. “Optimizing Seed Inputs in Fuzzing with Machine Learning”. In: *arXiv:1902.02538* (2019).
- [CM19] Z. Chen and M. Monperrus. “A Literature Study of Embeddings on Source Code”. In: *arXiv:1904.03061* (2019).
- [Col+13] A. Collins, C. Fensch, H. Leather, and M. Cole. “MaSiF: Machine Learning Guided Auto-tuning of Parallel Skeletons”. In: *International Conference on High Performance Computing (HiPC)*. IEEE, 2013.

- [Con+16] J. Cong, M. Huang, D. Wu, and C. H. Yu. “Heterogeneous Datacenters: Options and Opportunities”. In: *Design Automation Conference (DAC)*. ACM, 2016.
- [CSA18] Milan Cvitkovic, Badal Singh, and Anima Anandkumar. “Deep Learning On Code with an Unbounded Vocabulary”. In: *Machine Learning* 4 (2018).
- [Cum+16a] C. Cummins, P. Petoumenos, M. Steuwer, and H. Leather. “Autotuning OpenCL Workgroup Size for Stencil Patterns”. In: *International Workshop on Adaptive Self-tuning Computing Systems (ADAPT)*. 2016.
- [Cum+16b] C. Cummins, P. Petoumenos, M. Steuwer, and H. Leather. “Towards Collaborative Performance Tuning of Algorithmic Skeletons”. In: *High-Level Programming for Heterogeneous and Hierarchical Parallel Systems (HLPGPU)*. 2016.
- [Cum+17a] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather. “End-to-end Deep Learning of Optimization Heuristics”. In: *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2017.
- [Cum+17b] C. Cummins, P. Petoumenos, W. Zang, and H. Leather. “Synthesizing Benchmarks for Predictive Modeling”. In: *International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2017.
- [Cum+18] C. Cummins, P. Petoumenos, A. Murray, and H. Leather. “DeepSmith: Compiler Fuzzing through Deep Learning”. In: *ACACES*. 2018.
- [CW76] H. J. Curnow and B. A. Wichmann. “A Synthetic Benchmark”. In: *Computer* 19.1 (1976).
- [Dan+10] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. “The Scalable Heterogeneous Computing (SHOC) Benchmark Suite”. In: *Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU)*. ACM, 2010.
- [Dav01] A. David. *SLOCCount*. 2001.
- [DAY19] Y. David, U. Alon, and E. Yahav. “Neural Reverse Engineering of Stripped Binaries”. In: *arXiv:1902.09122* (2019).

- [DEK11] U. Dastgeer, J. Enmyren, and C. W. Kessler. “Auto-tuning SkePU: a Multi-Backend Skeleton Programming Framework for Multi-GPU Systems”. In: *International Workshop on Multicore Software Engineering (IWMSE)*. ACM, 2011.
- [Din+15] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U. O’Reilly, and S. Amarasinghe. “Autotuning Algorithmic Choice for Input Sensitivity”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2015.
- [DJW12] J. Duregard, P. Jansson, and M. Wang. “Feat: Functional Enumeration of Algebraic Types”. In: *Haskell*. ACM, 2012.
- [DMH19] Y. Ding, N. Mishra, and H. Hofmann. “Generative and Multi-phase Learning for Computer Systems Optimization”. In: *International Symposium on Computer Architecture (ISCA)*. 2019.
- [DR16] M. Dhok and M. K. Ramanathan. “Directed Test Generation to Detect Loop Inefficiencies”. In: *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2016.
- [Dub+07] C. Dubach, J. Cavazos, B. Franke, G. Fursin, M. O’Boyle, and O. Temam. “Fast Compiler Optimisation Evaluation Using Code-Feature Based Performance Prediction”. In: *ACM International Conference on Computing Frontiers (CF)*. ACM, 2007.
- [Dub+09] C. Dubach, T. M. Jones, E. V. Bonilla, G. Fursin, and M. O’Boyle. “Portable Compiler Optimisation Across Embedded Programs and Microarchitectures using Machine Learning”. In: *International Symposium on Microarchitecture (MICRO)*. ACM, 2009.
- [Dye+16] C. Dyer, A. Kuncoro, M. Ballesteros, and N. A. Smith. “Recurrent Neural Network Grammars”. In: *Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL-HLT)*. 2016.
- [FE15] T. L. Falch and A. C. Elster. “Machine Learning Based Auto-tuning for Enhanced OpenCL Performance Portability”. In: *International Parallel & Distributed Processing Symposium (IPDPSW)*. IEEE, 2015.

- [Fer+12] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. “Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware”. In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2012.
- [Fet+15] B. Fettscher, K. Claessen, M. Pałka, J. Hughes, and R. B. Findler. “Making Random Judgments: Automatically Generating Well-Typed Terms from the Definition of a Type-System”. In: *European Symposium on Programming (ESOP)*. 2015.
- [FT10] G. Fursin and O. Temam. “Collective Optimization: A Practical Collaborative Approach”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 7.4 (2010).
- [Fur+11] G. Fursin, Y. Kashnikov, A. W. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois, F. Bodin, P. Barnard, E. Ashton, E. Bonilla, J. Thomson, C. K. I. Williams, and M. O’Boyle. “Milepost GCC: Machine Learning Enabled Self-tuning Compiler”. In: *International Journal of Parallel Programming (IJPP)* 39.3 (2011).
- [Fur+14] G. Fursin, R. Miceli, A. Lokhmotov, M. Gerndt, M. Baboulin, A. D. Malony, Z. Chamski, D. Novillo, and D. Del Vento. “Collective Mind: Towards practical and collaborative auto-tuning”. In: *Scientific Programming* 22.4 (2014).
- [GA15] J. D. Garvey and T. S. Abdelrahman. “Automatic Performance Tuning of Stencil Computations on GPUs”. In: *International Conference on Parallel Processing (ICPP)*. ACM, 2015.
- [GAL14] E. Guzman, D. Azócar, and Y. Li. “Sentiment Analysis of Commit Comments in GitHub: an Empirical Study”. In: *Working Conference on Mining Software Repositories (MSR)*. 2014.
- [Gan+09] A. Ganapathi, K. Datta, A. Fox, and D. Patterson. “A Case for Machine Learning to Optimize Multicore Performance”. In: *USENIX conference on Hot topics in parallelism (HotPar)*. 2009.

- [Gao+15] H. Gao, J. Mao, J. Zhou, Z. Huang, L. Wang, and W. Xu. “Are You Talking to a Machine? Dataset and Methods for Multilingual Image Question Answering”. In: *Conference on Neural Information Processing Systems (NIPS)*. 2015.
- [GBC16] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.
- [GEB15] L. A. Gatys, A. S. Ecker, and M. Bethge. “A Neural Algorithm of Artistic Style”. In: *arXiv:1508.06576* (2015).
- [Geo+18] K. Georgiou, C. Blackmore, S. Xavier-de-Souza, and K. Eder. “Less is More: Exploiting the Standard Compiler Optimization Levels for Better Performance and Energy Consumption”. In: *International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. 2018.
- [GH15] A. Gandomi and M. Haider. “Beyond the Hype: Big data concepts, methods, and analytics”. In: *International Journal of Information Management (IJIM)* (2015).
- [GKS05] P. Godefroid, N. Klarlund, and K. Sen. “DART: Directed Automated Random Testing Patrice”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2005.
- [Gla19] Glassdoor. *Software Engineer Salaries in United States*. 2019. URL: <https://www.glassdoor.com> (visited on 08/15/2019).
- [Goo+14] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. “Generative Adversarial Networks”. In: *Conference on Neural Information Processing Systems (NIPS)*. 2014.
- [Gos+10] N. Goswami, R. Shankar, M. Joshi, and T. Li. “Exploring GPGPU workloads: Characterization methodology, analysis and microarchitecture evaluation implications”. In: *International Symposium on Workload Characterization (IISWC)*. 2010.
- [GPS17] P. Godefroid, H. Peleg, and R. Singh. “Learn&Fuzz: Machine Learning for Input Fuzzing”. In: *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017.
- [Gra+12] S. Grauer-Gray, L. Xu, R. Searles, S. Ayala-Somayajula, and J. Cavazos. “Auto-tuning a High-Level Language Targeted to GPU Codes”. In: *Innovative Parallel Computing (InPar)*. 2012.

- [Gra12] A. Graves. “Supervised Sequence Labelling with Recurrent Neural Networks”. PhD thesis. Technical University of Munich, 2012.
- [Gra13] A. Graves. “Generating Sequences with Recurrent Neural Networks”. In: *arXiv:1308.0850* (2013).
- [Gre+15a] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber. “LSTM: A Search Space Odyssey”. In: *arXiv:1503.04069* (2015).
- [Gre+15b] K. Gregor, I. Danihelka, A. Graves, D. J. Rezende, and D. Wierstra. “DRAW: A Recurrent Neural Network For Image Generation”. In: *arXiv:1502.04623* (2015).
- [GS05] A. Graves and J. Schmidhuber. “Framewise Phoneme Classification with Bidirectional LSTM and Other Neural Network Architectures”. In: *Neural Networks* 18.5 (2005).
- [GSB14] R. Gad, T. Suss, and A. Brinkmann. “Compiler Driven Automatic Kernel Context Migration for Heterogeneous Computing”. In: *International Conference on Distributed Computing Systems (IDCS)*. IEEE, 2014.
- [Gu+16] X. Gu, H. Zhang, D. Zhang, and S. Kim. “Deep API Learning”. In: *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2016.
- [GWO13] D. Grawe, Z. Wang, and M. O’Boyle. “Portable Mapping of Data Parallel Programs to OpenCL for Heterogeneous Systems”. In: *International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2013.
- [Han70] K. V. Hanford. “Automatic Generation of Test Cases”. In: *IBM Systems Journal* 9.4 (1970).
- [Has+15] I. A. T. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, and S. Ullah Khan. “The Rise of ”Big Data” on Cloud Computing: Review and Open Research Issues”. In: *Information Systems* 47 (2015).
- [He+16] K. He, X. Zhang, S. Ren, and J. Sun. “Deep Residual Learning for Image Recognition”. In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2016.
- [Hen+18] J. Henkel, S. K. Lahiri, B. Liblit, and T. Reps. “Code Vectors: Understanding Programs Through Embedded Abstracted Symbolic Traces”. In: *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*. 2018.

- [HHZ12] C. Holler, K. Herzig, and A. Zeller. “Fuzzing with Code Fragments”. In: *{USENIX} Security Symposium*. 2012.
- [HKP11] J. Han, M. Kamber, and J. Pei. *Data Mining: Concepts and Techniques*. Elsevier, 2011.
- [Hor91] K. Hornik. “Approximation Capabilities of Multilayer Feedforward Networks”. In: *Neural Networks* 4.2 (1991).
- [HOT06] G. E Hinton, S. Osindero, and Y. Teh. “A Fast Learning Algorithm for Deep Belief Nets”. In: *Neural Computation* 18.7 (2006).
- [HOY17] K. Heo, H. Oh, and K. Yi. “Machine-Learning-Guided Selectively Unsound Static Analysis”. In: *International Conference on Software Engineering (ICSE)*. 2017.
- [HS97] S. Hochreiter and J. Schmidhuber. “Long Short-Term Memory”. In: *Neural Computation* 9.8 (1997).
- [IS15] S. Ioffe and C. Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *International Conference on Machine Learning (ICML)*. PMLR, 2015.
- [JA19] Y. Jitsunari and Y. Arahorri. “Coverage-Guided Learning-Assisted Grammar-Based Fuzzing”. In: *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2019.
- [Jia+10] Y. Jiang, E. Z. Zhang, K. Tian, F. Mao, M. Gethers, X. Shen, and Y. Gao. “Exploiting Statistical Correlations for Proactive Prediction of Program Behaviors”. In: *International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2010.
- [Jol11] I. Jolliffe. *Principal Components Analysis*. Springer, 2011.
- [Jos+08] A. M. Joshi, L. Eeckhout, L. K. Johnz, and C. Isen. “Automated Microprocessor Stressmark Generation”. In: *International Symposium on High-Performance Computer Architecture (HPCA)*. 2008.
- [Jou+17] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A.

- Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. Mackean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and G. H. Yoon. “In-Datacenter Performance Analysis of a Tensor Processing Unit”. In: *International Symposium on Computer Architecture (ISCA)*. 2017.
- [Joz+16] R. Jozefowicz, O. Vinyals, M. Schuster, N. Shazeer, and Y. Wu. “Exploring the Limits of Language Modeling”. In: *arXiv:1602.02410* (2016).
- [Kal+09] E. Kalliamvakou, L. Singer, G. Gousios, D. M. German, K. Blincoe, and D. Damian. “The Promises and Perils of Mining GitHub”. In: *Working Conference on Mining Software Repositories (MSR)*. 2009.
- [KB15] D. P. Kingma and J. L. Ba. “Adam: a Method for Stochastic Optimization”. In: *International Conference on Learning Representations (ICLR)*. 2015.
- [KC12] S. Kulkarni and J. Cavazos. “Mitigating the Compiler Optimization Phase-Ordering Problem using Machine Learning”. In: *Object-oriented Programming Systems, Languages and Applications (OOPSLA)*. ACM, 2012.
- [Koc+17] U. Koc, P. Saadatpanah, J. S. Foster, and A. A. Porter. “Learning a Classifier for False Positive Error Reports Emitted by Static Code Analysis Tools”. In: *Machine Learning and Programming Languages (MAPL)*. 2017.
- [KP05] A. S. Kossatchev and M. A. Posypkin. “Survey of Compiler Testing Methods”. In: *Programming and Computer Software* 31.1 (2005).
- [Kra+18] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. “The Case for Learned Index Structures”. In: *International Conference on Management of Data (SIGMOD)*. ACM, 2018.
- [Kri+18] S. Krishnan, Z. Yang, K. Goldberg, J. M. Hellerstein, and I. Stoica. “Learning to Optimize Join Queries With Deep Reinforcement Learning”. In: *arXiv:1808.03196* (2018).

- [KSX19] L. Kosta, L. Seaman, and H. Xi. “Program Synthesis and Vulnerability Injection Using a Grammar VAE”. In: *Journal of Software* 14.6 (2019).
- [Kyl+15] S. Kyle, D. Butcher, S. Monteith, H. Leather, B. Franke, D. Butcher, and S. Monteith. “Application of Domain-aware Binary Fuzzing to Aid Android Virtual Machine Testing”. In: *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*. 2015.
- [Lam+15] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. “Combining Deep Learning with Information Retrieval to Localize Buggy Files for Bug Reports”. In: *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2015.
- [LAS14] V. Le, M. Afshari, and Z. Su. “Compiler Validation via Equivalence Modulo Inputs”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2014.
- [LBE15] Z. C. Lipton, J. Berkowitz, and C. Elkan. “A Critical Review of Recurrent Neural Networks for Sequence Learning”. In: *arXiv:1506.00019* (2015).
- [LBH15] Y. LeCun, Y. Bengio, and G. Hinton. “Deep learning”. In: *Nature* 521.7553 (2015).
- [LBO14] H. Leather, E. Bonilla, and M. O’Boyle. “Automatic Feature Generation for Machine Learning Based Optimizing Compilation”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 11.1 (2014).
- [LCW12] T. Lozano-Perez, I. J. Cox, and G. T. Wilfong. *Autonomous Robot Vehicles*. Springer, 2012.
- [Lem+18] C. Lemieux, R. Padhye, K. Sen, and D. Song. “PerfFuzz: Automatically Generating Pathological inputs”. In: *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2018.
- [Ler17] X. Leroy. *The CompCert C Verified Compiler*. 2017.
- [LFC13] T. Lutz, C. Fensch, and M. Cole. “PARTANS: An Autotuning Framework for Stencil Computation on Multi-GPU Systems”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 9.4 (2013).
- [Lid+15] C. Lidbury, A. Lascu, N. Chong, and A. Donaldson. “Many-Core Compiler Fuzzing”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2015.

- [Liu+19] X. Liu, X. Li, R. Prajapati, and D. Wu. “DeepFuzz: Automatic Generation of Syntax Valid C Programs for Fuzz Testing”. In: *Conference on Artificial Intelligence (AAAI)*. 2019.
- [LSS15a] V. Le, C. Sun, and Z. Su. “Finding Deep Compiler Bugs via Guided Stochastic Program Mutation”. In: *Object-oriented Programming Systems, Languages and Applications (OOPSLA)*. 2015.
- [LSS15b] V. Le, C. Sun, and Z. Su. “Randomized Stress-Testing of Link-Time Optimizers”. In: *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 2015.
- [Lu+17] Z. Lu, H. Pu, F. Wang, Z. Hu, and L. Wang. “The Expressive Power of Neural Networks: A View from the Width”. In: *Conference on Neural Information Processing Systems (NeurIPS)*. 2017.
- [Mat+19] B. Mathis, A. Kampmann, R. Gopinath, M. Höschele, M. Mera, and A. Zeller. “Parser-Directed Fuzzing”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2019.
- [McK98] W. M. McKeeman. “Differential Testing for Software”. In: *Digital Technical Journal (DTJ)* 10.1 (1998).
- [MDO13] A. Magni, C. Dubach, and M. O’Boyle. “A Large-Scale Cross-Architecture Evaluation of Thread-Coarsening”. In: *ACM/IEEE Supercomputing Conference (SC)*. 2013.
- [MDO14] A. Magni, C. Dubach, and M. O’Boyle. “Automatic Optimization of Thread-Coarsening for Graphics Processors”. In: *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM, 2014.
- [MF13] A. W. Memon and G. Fursin. “Crowdtuning: Systematizing Auto-tuning using Predictive Modeling and Crowdsourcing”. In: *International Conference on Parallel Computing (PARCO)*. 2013.
- [MG18] K. Moren and D. Gohringer. “Automatic Mapping for OpenCL-Programs on CPU/GPU Heterogeneous Platforms”. In: *International Conference on Computational Science (ICCS)*. 2018.

- [Mik+13] T. Mikolov, K. Chen, G. Corrado, and J. Dean. “Distributed Representations of Words and Phrases and their Compositionality”. In: *Conference on Neural Information Processing Systems (NIPS)*. 2013.
- [Mik10] T. Mikolov. “Recurrent Neural Network based Language Model”. In: *Annual Conference of the International Speech Communication Association (Interspeech)*. 2010.
- [Mni+15] V. Mnih, K. Kavukcuoglu, D. Silver, A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. “Human-level Control Through Deep Reinforcement Learning”. In: *Nature* 518.7540 (2015).
- [Mon18] M. Monperrus. “Automatic Software Repair: a Bibliography”. In: *ACM Computing Surveys (CSUR)* 51.1 (2018).
- [Mou+16] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. “Convolutional Neural Networks over Tree Structures for Programming Language Processing”. In: *Conference on Artificial Intelligence (AAAI)*. 2016.
- [MPL16] P. Mpeis, P. Petoumenos, and H. Leather. “Iterative Compilation on Mobile Devices”. In: *International Workshop on Adaptive Self-tuning Computing Systems (ADAPT)*. 2016.
- [MS10] J. Misra and I. Saha. “Artificial Neural Networks in Hardware: A Survey of Two Decades of Progress”. In: *Neurocomputing* 74.1-3 (2010).
- [MSD16] P. Micolet, A. Smith, and C. Dubach. “A Machine Learning Approach to Mapping Streaming Workloads to Dynamic Multicore Processors”. In: *Languages, Compilers, Tools and Theory of Embedded Systems (LCTES)*. ACM, 2016.
- [Nam+10] M. Namolaru, A. Cohen, G. Fursin, A. Zaks, and A. Freund. “Practical Aggregation of Semantical Program Properties for Machine Learning Based Optimization”. In: *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*. 2010.
- [Nau19] M. Naumov. “On the Dimensionality of Embeddings for Sparse Features and Data”. In: *arXiv:1901.02103* (2019).

- [NC15] C. Nugteren and V. Codreanu. “CLTune: A Generic Auto-Tuner for OpenCL Kernels”. In: *International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. 2015.
- [NH10] V. Nair and G. E. Hinton. “Rectified Linear Units Improve Restricted Boltzmann Machines”. In: *International Conference on Machine Learning (ICML)*. PMLR, 2010.
- [NHI13] E. Nagai, A. Hashimoto, and N. Ishiura. “Scaling up Size and Number of Expressions in Random Testing of Arithmetic Optimization of C Compilers”. In: *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI)*. 2013.
- [NLS16] A. Neelakantan, Q. V. Le, and I. Sutskever. “Neural Programmer: Inducing Latent Programs with Gradient Descent”. In: *International Conference on Learning Representations (ICLR)*. 2016.
- [NPK18] M. Z. Nasrabadi, S. Parsa, and A. Kalaei. “Neural Fuzzing: A Neural Approach to Generate Test Data for File Format Fuzzing”. In: *arXiv:1812.09961* (2018).
- [Oda+15] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura. “Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation”. In: *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015.
- [Ogi+14] W. F. Ogilvie, P. Petoumenos, Z. Wang, and H. Leather. “Fast Automatic Heuristic Construction Using Active Learning”. In: *International Workshop on Languages and Compilers for Parallel Computing (LCPC)*. 2014.
- [Ogi+17] W. F. Ogilvie, P. Petoumenos, Z. Wang, and H. Leather. “Minimizing the Cost of Iterative Compilation with Active Learning”. In: *International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2017.
- [Oul+08] E. Ould-Ahmed-Vall, K. A. Doshi, C. Yount, and J. Woodlee. “Characterization of SPEC CPU2006 and SPEC OMP2001: Regression models and their transferability”. In: *International Symposium on Performance Analysis of Systems and software (ISPASS)*. IEEE, 2008.

- [Pał+11] M. H. Pałka, K. Claessen, A. Russo, and J. Hughes. “Testing an Optimising Compiler by Generating Random Lambda Terms”. In: *International Conference on Software Engineering (ICSE)*. 2011.
- [Pan+18] R. Panda, S. Song, J. Dean, and L. K. John. “Wait of a Decade: Did SPEC CPU 2017 Broaden the Performance Horizon?” In: *International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2018.
- [PCA12] E. Park, J. Cavazos, and M. A. Alvarez. “Using Graph-Based Program Characterization for Predictive Modeling”. In: *International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2012.
- [PDL16] M. Pflanzer, A. Donaldson, and A. Lascu. “Automatic Test Case Reduction for OpenCL”. In: *International Workshop on OpenCL (IWOCCL)*. 2016.
- [PE06] Z. Pan and R. Eigenmann. “Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning”. In: *International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2006.
- [Ped+18] L. Pedrosa, R. Iyer, A. Zaostrovnykh, J. Fietz, and K. Argyraki. “Automated Synthesis of Adversarial Workloads for Network Functions”. In: *Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2018.
- [Pet+17] T. Petsios, J. Zhao, A. D. Keromytis, and Suman Jana. “SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities”. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2017.
- [PJ13] S. Purini and L. Jain. “Finding Good Optimization Sequences Covering Program Space”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* (2013).
- [PJ07] Aashish Phansalkar, Ajay Joshi, and Lizy K. John. “Analysis of Redundancy and Application Balance in the SPEC CPU2006 Benchmark Suite”. In: *ACM SIGARCH Computer Architecture News* 35.2 (2007).

- [PM15] J. Price and S. McIntosh-Smith. “Oclgrind: An Extensible OpenCL Device Simulator”. In: *International Workshop on OpenCL (IWOC)*. ACM, 2015.
- [PMB13] R. Pacanu, T. Mikolov, and Y. Bengio. “On the Difficulties of Training Recurrent Neural Networks”. In: *International Conference on Machine Learning (ICML)*. PMLR, 2013.
- [PS18] M. Pradel and K. Sen. “DeepBugs: A Learning Approach to Name-based Bug Detection”. In: *Object-oriented Programming Systems, Languages and Applications (OOPSLA)*. ACM, 2018.
- [PSP18] H. Peng, Y. Shoshitaishvili, and M. Payer. “T-Fuzz: Fuzzing by Program Transformation”. In: *Symposium on Security and Privacy (SP)*. 2018.
- [Rag+13] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. “Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2013.
- [Rag+17] M. Raghu, B. Poole, J. Kleinberg, S. Ganguli, and J. Sohl-Dickstein. “On the Expressive Power of Deep Neural Networks”. In: *International Conference on Machine Learning (ICML)*. PMLR, 2017.
- [Reg+12] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. “Test-Case Reduction for C Compiler Bugs”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2012.
- [RHW86] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. “Learning Representations by Back-propagating Errors”. In: *Nature* 323 (1986).
- [RJS17] A. Radford, R. Jozefowicz, and I. Sutskever. “Learning to Generate Reviews and Discovering Sentiment”. In: *arXiv:1704.01444* (2017).
- [RNL08] C. Runciman, M. Naylor, and F. Lindblad. “SmallCheck and Lazy Small-Check Automatic Exhaustive Testing for Small Values”. In: *Haskell*. ACM, 2008.
- [RVK15] V. Raychev, M. Vechev, and A. Krause. “Predicting Program Properties from ”Big Code””. In: *Symposium on Principles of Programming Languages (POPL)*. 2015.

- [RVY14] V. Raychev, M. Vechev, and E. Yahav. “Code Completion with Statistical Language Models”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2014.
- [Ryo+08a] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu. “Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA”. In: *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 2008.
- [Ryo+08b] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S. Ueng, J. A. Stratton, and W. W. Hwu. “Program Optimization Space Pruning for a Multithreaded GPU”. In: *International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2008.
- [Ryo+15] J. H. Ryoo, S. J. Quirem, M. Lebeane, R. Panda, S. Song, and L. K. John. “GPGPU Benchmark Suites: How Well Do They Sample the Performance Spectrum?” In: *International Conference on Parallel Processing (ICPP)*. ACM, 2015.
- [SA05] M. Stephenson and S. Amarasinghe. “Predicting Unroll Factors Using Supervised Classification”. In: *International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2005.
- [SD16] T. Sorensen and A. Donaldson. “Exposing Errors Related to Weak Memory in GPU Applications”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2016.
- [SD17] M. Steuwer and C. Dubach. “Lift: A Functional Data-Parallel IR for High-Performance GPU Code Generation”. In: *International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2017.
- [She+18] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana. “NEUZZ: Efficient Fuzzing with Neural Program Learning”. In: *arXiv:1807.05620* (2018).
- [Shi+15] X. Shi, J. Liang, S. Di, B. He, H. Jin, L. Lu, Z. Wang, X. Luo, and J. Zhong. “Optimization of Asynchronous Graph Processing on GPU with Hybrid Coloring Model”. In: *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 2015.

- [Si+18] X. Si, H. Dai, M. Raghothaman, M. Naik, and L. Song. “Learning Loop Invariants for Program Verification”. In: *Conference on Neural Information Processing Systems (NeurIPS)*. 2018.
- [Sim+12] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc. “A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications”. In: *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, 2012.
- [SJL11] S. Seo, G. Jo, and J. Lee. “Performance Characterization of the NAS Parallel Benchmarks in OpenCL”. In: *International Symposium on Workload Characterization (IISWC)*. IEEE, 2011.
- [SLS16] C. Sun, V. Le, and Z. Su. “Finding Compiler Bugs via Live Code Mutation”. In: *Object-oriented Programming Systems, Languages and Applications (OOPSLA)*. 2016.
- [SMR03] M. Stephenson, M. Martin, and U. O. Reilly. “Meta Optimization: Improving Compiler Heuristics with Machine Learning”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2003.
- [SS91] H. T. Siegelmann and E. D. Sontag. “Turing Computability with Neural Networks”. In: *Applied Mathematics Letters* 4.6 (1991).
- [SSN12] M. Sundermeyer, R. Schl, and H. Ney. “LSTM Neural Networks for Language Modeling”. In: *Annual Conference of the International Speech Communication Association (Interspeech)*. 2012.
- [Str+12] J. A. Stratton, C. Rodrigues, I. Sung, N. Obeid, L. Chang, N. Anssari, G. D. Liu, and W. W. Hwu. “Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing”. In: *Center for Reliable and High-Performance Computing* (2012).
- [SVL14] I. Sutskever, O. Vinyals, and Q. V. Le. “Sequence to Sequence Learning with Neural Networks”. In: *Conference on Neural Information Processing Systems (NIPS)*. 2014.
- [TA03] S. Triantafyllis and D. I. August. “Compiler Optimization-Space Exploration”. In: *International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2003.

- [TC13] M. Tartara and S. Crespi Reghizzi. “Continuous Learning of Compiler Heuristics”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 9.4 (2013).
- [Tin+17] P. Ting, C. Tu, P. Chen, Y. Lo, and S. Cheng. “FEAST: An Automated Feature Selection Framework for Compilation Tasks”. In: *International Conference on Advanced Information Networking and Applications (AINA)*. 2017.
- [TPG18] L. D. Toffola, M. Pradel, and T. R. Gross. “Synthesizing Programs That Expose Performance Bottlenecks”. In: *International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2018.
- [Tuf+19] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk. “On Learning Meaningful Code Changes via Neural Machine Translation”. In: *arXiv:1901.09102* (2019).
- [TV16] P. Terence and J. Vinju. “Towards a Universal Code Formatter through Machine Learning”. In: *ACM SIGPLAN International Conference on Software Language Engineering (SLE)*. 2016.
- [Vas+19] M. Vasic, A. Kanade, P. Maniatis, D. Bieber, and R. Singh. “Neural Program Repair by Jointly Learning to Localize and Repair”. In: *International Conference on Learning Representations (ICLR)*. 2019.
- [VFS15] B. Vasilescu, V. Filkov, and A. Serebrenik. “Perceptions of Diversity on GitHub: A User Survey”. In: *International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. 2015.
- [Vin+15] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. “Show and Tell: A Neural Image Caption Generator”. In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2015.
- [Wan+14] Z. Wang, G. Tournavitis, B. Franke, and M. O’Boyle. “Integrating Profile-driven Parallelism Detection and Machine-learning-based Mapping”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* (2014).
- [Wan+17] J. Wang, B. Chen, L. Wei, and Y. Liu. “Skyfire: Data-Driven Seed Generation for Fuzzing”. In: *Symposium on Security and Privacy (SP)*. 2017.
- [Wei+11] B. Weinberger, C. Silverstein, G. Eitzmann, M. Mentovai, and T. Landry. *Google C++ Style Guide*. 2011.

- [Wei+18] J. Wei, J. Chen, Y. Feng, K. Ferles, and Isil Dillig. “Singularity: Pattern Fuzzing for Worst Case Complexity”. In: *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2018.
- [Wer90] P. J. Werbos. “Backpropagation Through Time: What It Does and How to Do It”. In: *Proceedings of the IEEE* 78.10 (1990).
- [Whi+15] M. White, C. Vendome, M. Linares-Vasquez, and D. Poshyvanyk. “Toward Deep Learning Software Repositories”. In: *Working Conference on Mining Software Repositories (MSR)*. 2015.
- [Whi+19] M. White, M. Tufano, M. Martínez, M. Monperrus, and D. Poshyvanyk. “Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities”. In: *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2019.
- [WLT16] S. Wang, T. Liu, and L. Tan. “Automatically Learning Semantic Features for Defect Prediction”. In: *International Conference on Software Engineering (ICSE)*. ACM, 2016.
- [WO09] Z. Wang and M. O’Boyle. “Mapping Parallelism to Multi-cores: A Machine Learning Based Approach”. In: *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 15. ACM, 2009.
- [WO10] Z. Wang and M. O’Boyle. “Partitioning Streaming Parallelism for Multi-cores: A Machine Learning Based Approach”. In: *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM, 2010.
- [WO18] Z. Wang and M. O’Boyle. “Machine learning in Compiler Optimization”. In: *Proceedings of the IEEE* 106.23 (2018).
- [WRX17] H. Wang, B. Raj, and E. P. Xing. “On the Origin of Deep Learning”. In: *arXiv:1702.07800* (2017).
- [WSS18] K. Wang, R. Singh, and Z. Su. “Dynamic Neural Program Embeddings for Program Repair”. In: *International Conference on Learning Representations (ICLR)*. 2018.

- [Wu+14] Y. Wu, J. Kropczynski, P. C. Shih, and J. M. Carroll. “Exploring the Ecosystem of Software Developers on GitHub and Other Platforms”. In: *ACM Conference on Computer Supported Cooperative Work & Social Computing (CSCW)*. 2014.
- [WWO14] Y. Wen, Z. Wang, and M. O’Boyle. “Smart Multi-Task Scheduling for OpenCL Programs on CPU/GPU Heterogeneous Platforms”. In: *International Conference on High Performance Computing (HiPC)*. IEEE, 2014.
- [WYT13] E. Wong, J. Yang, and L. Tan. “AutoComment: Mining Question and Answer Sites for Automatic Comment Generation”. In: *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013.
- [Xio+13] W. Xiong, Z. Yu, Z. Bei, J. Zhao, F. Zhang, Y. Zou, X. Bai, Y. Li, and C. Xu. “A Characterization of Big Data Benchmarks”. In: *International Conference on Big Data*. IEEE, 2013.
- [Xio+16] W. Xiong, J. Droppo, X. Huang, F. Seide, M. Seltzer, A. Stolcke, D. Yu, and G. Zweig. “Achieving Human Parity in Conversational Speech Recognition”. In: *arXiv:1610.05256* (2016).
- [Yan+11] X. Yang, Y. Chen, E. Eide, and J. Regehr. “Finding and Understanding Bugs in C Compilers”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2011.
- [Yar17] D. Yarotsky. “Error Bounds for Approximations with Deep ReLU Networks”. In: *Neural Networks* 94 (2017).
- [Yin+18] P. Yin, G. Neubig, M. Allamanis, M. Brockschmidt, and A. L. Gaunt. “Learning to Represent Edits”. In: *arXiv:1810.13337* (2018).
- [Yos+14] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson. “How Transferable are Features in Deep Neural Networks?” In: *Conference on Neural Information Processing Systems (NIPS)*. 2014.
- [YS18] Z. Yin and Y. Shen. “On the Dimensionality of Word Embedding”. In: *Conference on Neural Information Processing Systems (NeurIPS)*. 2018.
- [Zal14] M. Zalewski. *American Fuzzy Lop*. 2014.

- [ZF14] M. D. Zeiler and R. Fergus. “Visualizing and Understanding Convolutional Networks”. In: *European Conference on Computer Vision (ECCV)*. 2014.
- [Zha+18] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe. “GraphIt - A High-Performance DSL for Graph Analytics”. In: *arXiv:1805.00923* (2018).
- [ZIE16] R. Zhang, P. Isola, and A. A. Efros. “Colorful Image Colorization”. In: *European Conference on Computer Vision (ECCV)*. 2016.
- [ZSS17] Q. Zhang, C. Sun, and Z. Su. “Skeletal Program Enumeration for Rigorous Compiler Testing”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2017.