Program Generation and Optimisation through Recurrent Neural Networks

Chris Cummins

Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

University of Edinburgh

2019

Abstract

Building an optimising compiler is hard. LLVM, a leading industrial-grade compiler, is the product of XXX commits over a XX year period, costing an estimated XXX. There are more devices + heterogeneity. Each device requires a new compilers.

Demand outstrips supply. What is need is better tools to lower the cost of 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 *recurrent neural networks* to model the correlations between source code and program behaviour, enabling tasks which previously required enormous engineering effort to be automated. This is demonstrated in three domains:

First, a generative model for compiler benchmarks is developed. This model is inferred automatically from corpora of readily available open source programs, requiring no grammar or prior knowledge of the programming language. This greatly reduces the cost of development compared to prior approaches, yet the generator produces output of such quality that professional software developers cannot distinguish generated from handwritten code. The efficacy of the generator is demonstrated by supplementing the training data of state-of-the-art predictive models for compiler optimisations. The additional fine-grained exploration of the feature space yields both an automatic improvement in heuristic performance and exposes weaknesses in the prior art which, when corrected, yields further improvements in performance.

Second, this thesis presents techniques that extend the prior approach to the domain of compiler validation. A compiler fuzzer is developed which is far simpler than the state-of-the-art, yet is effective. By learning a generative model rather than engineering a generator from scratch using a grammar, it is implemented in 100$\times$ fewer lines of code than the state-of-the-art, and is capable of generating an expressive range of tests that expose bugs that prior techniques cannot. An extensive testing campaign of OpenCL compilers reveals 67 new bugs, many of which have now been fixed.

Finally, this thesis addresses the challenges of machine learning for compiler optimisations, developing methodologies for learning compiler heuristics without the need for code features. Contrasting prior approaches that require features to be expertly engineered and selected, the proposed approach learns directly over the raw textual representation of program code. Doing so outperforms state-of-the-art heuristics in two challenging optimisation domains. Additionally, the methodology permits the novel

transfer of information between optimisation problems, enabling a model trained for one task to be adapted to perform another, further improving performance.

TODO: Conclusions

Acknowledgements

Acknowledgements placeholder.

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

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

There has been an unprecedented increase in the scale and quantity of data-intensive workloads. Fundamental shifts in both hardware and software are required to meet the demands of the transition to *big data*.

In hardware, performance needs have long outstripped what can be provided by single processors, leading to a broad spectrum of parallel and increasingly heterogeneous architectures. These range from re-purposing existing hardware such as Graphics Processor Units (GPUs) to developing entirely novel application-specific chips [**Misra2010**; **Jouppi2017**]. While some compiler logic can be shared across architectures, the optimisation decisions to extract the best performance from specific hardware cannot. Each architecture requires extensive hand tuning by experts to extract good performance.

In software, the shift towards parallelism and heterogeneity has created a *programmability challenge*. Parallel programming poses far greater challenges than traditional single-threaded development; it is often not clear how best to partition work across the available parallel resources and doing so can easily introduce bugs. One of the most popular approaches to mitigating the programmability challenge is through the development and widespread adoption of high-level abstractions. High-level abstractions and libraries can greatly simplify parallel programming by providing complex parallel communication and coordination logic, allowing users to plug-in only the business logic required to solve a problem. Still, there is a wide range of approaches to implementing such libraries. For example, two of the most popular high-level libraries for parallelising deep learning workloads — TensorFlow [**Abadi**] and PyTorch [**Paszke2017**] — use opposing data-flow and imperative programming styles, respectively. Optimising such libraries provides new challenges to the compiler — the challenge of code analyses in the face of parallelised higher-order functions can defeat

many optimisations.

The combined burden of increased hardware and software diversity has resulted in compilers that are too complex to be fully understood by a single developer, and too cumbersome to keep up with the required pace of change. This results in slow performance, wasted energy, 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. The remainder of this chapter describes the application of machine learning to this issue, followed by the problems with this approach. Then the contributions of this thesis are detailed, followed by a description of the overall structure of the document.

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 compiler[1]. 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. Further to that, the outcome of the heuristic will depend not just on the program being compiled, but on properties of the target hardware, such as the number of registers 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 the developer will be able to craft a heuristic capable of extracting the best performance in all situations.

Instead, the developer may cast construction of a 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

[1]Loop 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 an increasing the size of the executable binary.

from empirical data of the performance of loops under different configurations of unrolling. To do this, the developer would run a suite of benchmark programs repeatedly using different unrolling decisions to determine the best decision for each case, then combine this with a numerical representation of each program. A machine learning system would then model the correlations between these numerical 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, and can be achieved with less effort by the developer.

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, enabling predictions to be made on unseen data; all without the need for expert domain knowledge. In essence, machine learning can negate the need for 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 field. In previous studies machine learning has been shown to simplify the construction of compiler optimisations, 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 speculates as to the cause by summarising some of the outstanding problems in applying machine learning to compilers.

1.2 Challenges in Machine Learning for Compilers

TODO: s/problem/challenge/

Machine learning techniques offer reduced costs and improved performance compared to expert approaches, yet there are challenges preventing widespread adoption. There are two significant problems that must be overcome:

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 should 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 very 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 the 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, 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 real programs.

Model and feature design Machine learning algorithms learn to correlate a set of *explanatory variables* with a target value. These explanatory variables, known as features, must be chosen so as to be discriminative for the target value.

Choosing the features to summarise a program so as to be discriminative for machine learning is a challenging task that depends on the thing being learned, 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, instruction counts can be extracted from the input source code. Knowing which attributes to extract to represent a program is not easy, there is no one-size-fits-all approach that works for all cases. Feature design is often an incremental process of trial and experimentation, and there are few clear signals when the iterative process is "done".

If machine learning is to be widely adopted in compilers, it must be made significantly easier and cheaper. The aim of this thesis is to 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 approach automatically enhances the predictive power of a state-of-the-art predictive model, improving the performance of heterogeneous workloads by 1.27×. Further, the additional benchmarks expose limitations in the feature design of the model which, after correcting, further increases performance by 4.30×.

- A novel, automatic, and fast approach for the generation of expressive random programs for compiler fuzzing. The system *infers* programming language syntax, structure, and use 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 and takes less than a day to train. 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 state-of-the-art, without any expensive reduction process.

- An extensive evaluation campaign of the compiler fuzzing approach using 10 OpenCL compilers and 1000 hours of automated testing. The campaign uncovers a similar number of bugs as the state-of-the-art, but also finds bugs which prior work cannot, covering more components of the compiler.

- 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 state-of-the-art predictive models by 14% and 12% in two challenging GPGPU compiler optimisation domains.

- The first application of *transfer learning* to compiler optimisations, improving heuristics by reusing training information across different optimisation problems, even if they are unrelated.

1.4 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. It describes an extensive testing campaign of OpenCL compilers, 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.

1.5 Summary

This introductory chapter has outlined the use of machine learning for reducing the cost of compiler construction and two significant issues preventing its widespread adoption: the scarcity of data and the challenge of designing features. Subsequent chapters describe novel techniques to address both issues. second

Chapter 2

Background

2.1 Introduction

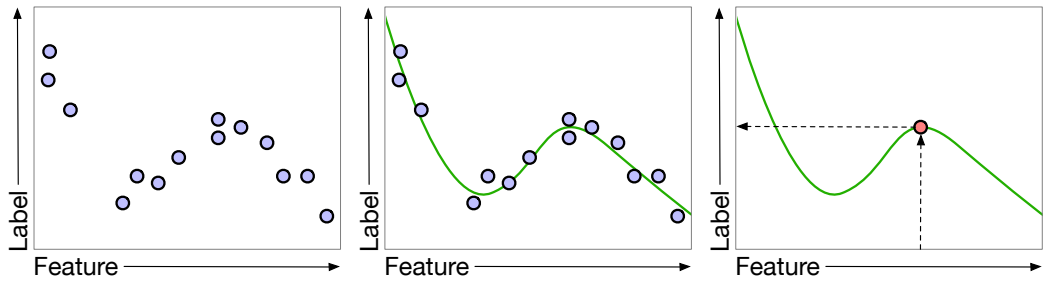
This chapter provides an overview of the techniques and theory used in this thesis. Sections 2.2 and 2.3 describe the machine learning and evaluation techniques used in this thesis, respectively. Section 2.4 concludes.

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 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 correct category, 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 briefly describes the classification and sequential modelling techniques used in this thesis.



(a) Training observations (b) Fitted model (c) Model inference

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 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 [**Hornik1991**; **Lu2017**; **Yarotsky2017**].

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

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

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

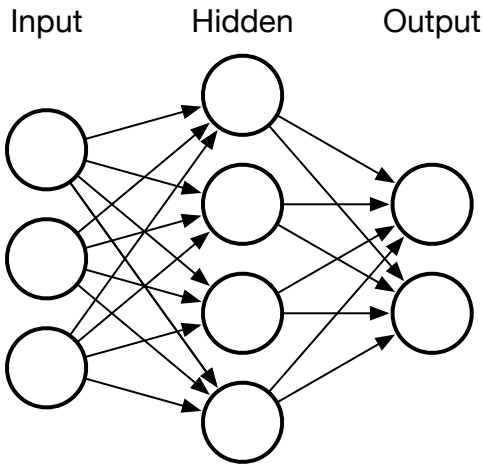


Figure 2.2: A feed-forward artificial neural network with a single hidden layer. Each node in the graph depicts an artificial neuron, and each edge represents the output of one neuron connected to the input of another.

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:

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

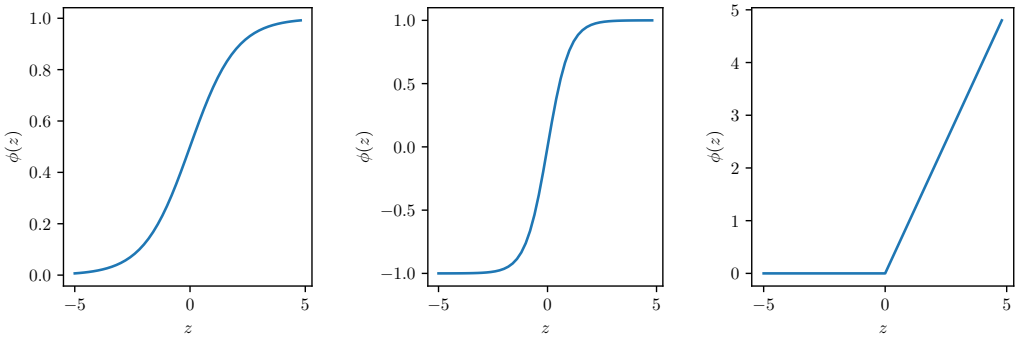
Sigmoid activations are bounded in the range $(0,1)$, depicted in Figure 2.3a. For hidden layers, a disadvantage of sigmoid activation is that the output is not zero-centred. To address this, the *Hyperbolic Tangent* (tanh) activation may be used, which is a scaled sigmoid function with values in the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

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

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 $[0,\infty)$:

$$\phi(z) = max(z,0) \tag{2.4}$$

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 in the network



(a) Sigmoid (b) Hyperbolic Tangent (c) Rectified Linear Unit

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

will not fire, so 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:

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

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:

$$\phi(\boldsymbol{z})^{(i)} = \frac{e^{\boldsymbol{z}^{(i)}}}{\sum_{j=1}^{K} e^{\boldsymbol{z}^{(i)}}}, i = 1, \ldots, K \tag{2.6}$$

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

Backpropagation and Gradient Descent The most widely used technique to train artificial neural networks is backpropagation [**Rumelhart1986**]. 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 $\hat{\boldsymbol{y}}$ are then compared against the true values $\boldsymbol{y}$

and used to compute an error $\mathcal{L}(\hat{\boldsymbol{y}}, \boldsymbol{y})$. The appropriate error function depends on the task. For a classification task with K classes, *categorical cross-entropy* may be used:

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

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

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

$$\boldsymbol{W}^{[\ell]} = \boldsymbol{W}^{[\ell]} - \alpha \frac{\partial J}{\partial \boldsymbol{W}^{[\ell]}} \tag{2.9}$$

$$\boldsymbol{b}^{[\ell]} = \boldsymbol{b}^{[\ell]} - \alpha \frac{\partial J}{\partial \boldsymbol{b}^{[\ell]}} \tag{2.10}$$

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 [**Goodfellow2016**].

2.2.2 Recurrent Neural Networks

A Recurrent Neural Network (RNN) [**Graves2012**] 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 networks[1]. Figure 2.4 depicts an RNN, where $\boldsymbol{x}^{(t)} \in \mathbb{R}^d$ represents point $t \in \tau$ in a sequence of inputs $\boldsymbol{x} = \left(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(\tau)}\right)$. The hidden states $\boldsymbol{h}^{(t)} \in \mathbb{R}^h$ and predicted output $\hat{\boldsymbol{y}}^{(t)} \in \mathbb{R}^y$ are updated at each step using:

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

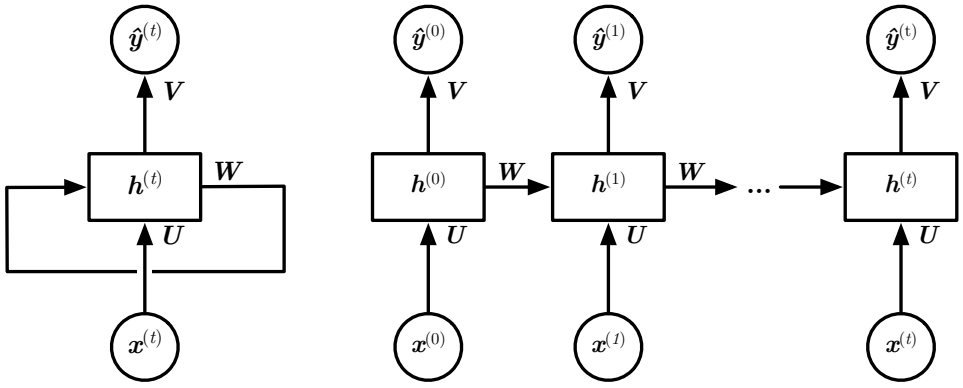


Figure 2.4: The computational graph of a Recurrent Neural Network, shown in (a) as a recurrence relation, and unfolded in (b). $\boldsymbol{x}^{(t)}$ is the input, $\boldsymbol{h}^{(t)}$ is the hidden state, and $\hat{\boldsymbol{y}}^{(t)}$ is the output. The network is parameterised through three weight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

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

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

Where $\phi(\boldsymbol{z})$ and $\sigma(\boldsymbol{z})$ are non-linear activation functions, $\boldsymbol{b}_h$ and $\boldsymbol{b}_{\hat{y}}$ are bias vectors, and $\boldsymbol{W} \in \mathbb{R}^{h\times d}$, $\boldsymbol{U} \in \mathbb{R}^{h\times h}$, and $\boldsymbol{V} \in \mathbb{R}^{y\times h}$ 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 $p(y|\boldsymbol{x})$, an RNN estimates a distribution conditioned on t prior observations $p\left(y^{(t)}|\{\boldsymbol{x}^{(1)},\ldots,\boldsymbol{x}^{(t)}\}\right)$.

Ordinary backpropagation may be used on an RNN by unfolding the computation graph over time, shown in Figure (b). This *Backpropagation Through Time* (BPTT) [**Werbos1990a**] 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 [**Sontag1991**]. In practice, a significant obstacle to the performance of RNNs is the diminishing ability to learn connections between

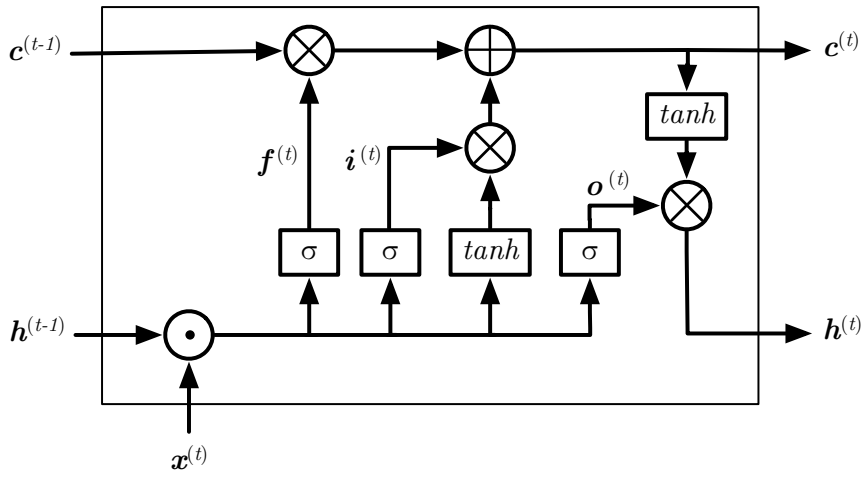


Figure 2.5: A Long Short-Term Memory cell block. Input $\boldsymbol{x}^{(t)}$ is concatenated with prior hidden state $\boldsymbol{h}^{(t-1)}$ and used along with prior cell state $\boldsymbol{c}^{(t-1)}$ to compute a new hidden state $\boldsymbol{h}^{(t)}$ and cell state $\boldsymbol{c}^{(t)}$. Symbols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

values over long sequences. This is caused by the exponential diminishing and enlarging of gradients as they are propagated through the recurrence relation. This issue is known as the *vanishing gradients* problem [**Bengio1994**].

Long Short-Term Memory The long short-term memory (LSTM) [**Hochreiter1997**] 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 $\boldsymbol{c}^{(t)} \in \mathbb{R}^h$ is propagated through time. The signal flow through the cell is controlled by three sigmoid activated gates: the forget gate $\boldsymbol{f}^{(t)} \in \mathbb{R}^h$, input gate $\boldsymbol{i}^{(t)} \in \mathbb{R}^h$, and output gate $\boldsymbol{o}^{(t)} \in \mathbb{R}^h$. At each time step, the values of the gate vectors are updated using:

$$\boldsymbol{f}^{(t)} = \sigma\left(\boldsymbol{W}_f \boldsymbol{x}^{(t)} + \boldsymbol{U}_f \boldsymbol{h}^{(t-1)} + \boldsymbol{b}_f\right) \tag{2.13}$$

$$\boldsymbol{i}^{(t)} = \sigma\left(\boldsymbol{W}_i \boldsymbol{x}^{(t)} + \boldsymbol{U}_i \boldsymbol{h}^{(t-1)} + \boldsymbol{b}_i\right) \tag{2.14}$$

$$\boldsymbol{o}^{(t)} = \sigma\left(\boldsymbol{W}_o \boldsymbol{x}^{(t)} + \boldsymbol{U}_o \boldsymbol{h}^{(t-1)} + \boldsymbol{b}_o\right) \tag{2.15}$$

$$\tag{2.16}$$

Where weight matrices $\boldsymbol{W} \in \mathbb{R}^{h \times d}$ and $\boldsymbol{U} \in \mathbb{R}^{h \times h}$, and bias vectors $\boldsymbol{b} \in \mathbb{R}^{h}$ 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:

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

$$\boldsymbol{h}^{(t)} = \boldsymbol{o}^{(t)} \otimes tanh(\boldsymbol{c}^{(t)}) \tag{2.18}$$

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 very long sequences in data. The LSTMs architecture (and its many variants [**Greff2015**]) have been responsible breakthrough results in a number of areas, for example machine translation [**Sutskever2014**], speech recognition [**Graves2005**], and weather prediction [**Shi2015a**].

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 a 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, and $p_{(+)}$ and $p_{(-)}$ are the number of positive and negative examples in D:

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

The gain of a feature x is found using:

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

Decision trees are a popular and low overhead form of classification. Implementations can be as simple and efficient as a set of nested conditional statements.

2.3 Evaluation Techniques

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 $\boldsymbol{y} = \{A, B, B, C\}$, a ZeroR model will produce output $\hat{\boldsymbol{y}} = B$ 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

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 $\boldsymbol{D} = (\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}), i = 1, \ldots, N$, let $k : 1, \ldots, N \rightarrow 1, \ldots, K$ 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 kth part of the data removed, yielding fitted function $\hat{f}^{-k}(\boldsymbol{x})$. 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:

$$\boldsymbol{y}_k = \{\boldsymbol{y}^{(i)} \in \boldsymbol{y} | i \in k(\boldsymbol{y})\} \tag{2.21}$$

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

Employing a second indexing function u to divide the data not set aside for testing 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:

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

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

2.3.3 K-Fold Cross-validation

Dividing the dataset into a fixed training set and a fixed test set can be 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-validations 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 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.

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

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 observation classes in all partitions is equal, this is called *stratified* K-fold cross-validation.

2.3.4 Principal Component Analysis

Principal Components Analysis PCA [**Jolliffe2011**] 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 $\boldsymbol{X} \in \mathbb{R}^{m \times n}$ 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 $\boldsymbol{v}^{(1)}, \ldots \boldsymbol{v}^{(m)}$, the matrix of k principal components $\hat{\boldsymbol{X}} \in \mathbb{R}^{m \times k}$ can be computed using:

$$\hat{\boldsymbol{X}} = [\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(k)}]^T \cdot \boldsymbol{X} \tag{2.26}$$

2.4 Summary

This chapter provides background on the machine learning techniques used in this thesis and the techniques used in evaluating them. The following chapter surveys research literature relevant to this work. second

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 benchmarking 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

The generation of artificial programs is a broad field with a wide range of applications. This section categorises the literature in 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 can hamper the other. As a result, there is no "one size fits all" benchmark suite.

Given its importance, benchmark suite characterisation has been the subject of much research. An evaluation of popular GPGPU benchmark suites [**Ryoo2015**] reveals there are important parts of the program space left untested. **Xiong2013** [**Xiong2013**] 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 [**Ferdman2012**] 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-Vall2008** [**Ould-Ahmed-Vall2008**] show that statistical models trained on 10% of SPEC CPU 2006 data are transferable to the remaining data. **Goswami2010** [**Goswami2010**] 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. **Phansalkar2007** [**Phansalkar2007**] show that a subset of 14 SPEC CPU 2006 programs can yield most of the information of the entire suite, and **Draft2018** [**Draft2018**] find that SPEC CPU 2017 contains workloads that can be safely removed without degrading 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 **Curnow1976** using a synthetic benchmark to compare the computing power of processors for scientific workloads [**Curnow1976**]. **Bell2005** [**Bell2005**] 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. **Joshi2008** [**Joshi2008**] 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* [**Bertran2012**] uses feedback-directed micro-benchmark generation to perform a systematic energy characterisation of a processor.

GENESIS [**Chiu2015**] is a language for generating synthetic training programs. 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 can 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 what humans write. Such grammar-based benchmark generators have been shown to be successful over a highly restricted space of stencil benchmarks with little control flow or program variability [**Garvey2015b**; **Falch2015**; **Cummins2016a**]. But, it remains unclear how much effort it will take to define grammars capable of producing human-like 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.

Dhok2016 [**Dhok2016**] apply mutation techniques to an initial set of coverage-driven inputs to expose inefficiencies in loops. *SlowFuzz* [**Petsios2017**] 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* [**Wei2018**] uses an evolutionary search over the space of program inputs but using input *patterns* to find the input with the worst case performance.

PerfSyn [**Toffola2018**] 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* [**Lemieux2018**] uses feedback-directed program mutation to generate programs which maximise execution counts at program locations. **Pedrosa2018** [**Pedrosa2018**] applies the 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.

Ding2019 [**Ding2019**] propose an alternate 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. 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 can introduce security vulnerabilities and catastrophic runtime failures. Therefore, testing that a compiler behaves as expected is of utmost importance.

Approaches to assuring compiler behaviour can be divided into *verification* and *validation*. The complexity of optimising compilers and programming languages renders formal verification of the entire compiler prohibitively expensive. Efforts have been made in this direction, for example, CompCert [**Leroy2013**], 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 lower performance compared to GCC. Still, even CompCert is not fully verified, and errors have been discovered in the unverified components [**Yang2011**].

Because of the difficulties of *verification*, compiler developers turn to *validation*, in which the behaviour of a compiler is validated using a set of handcrafted 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 verify that the compiler conforms to the specification, for those inputs. However, the absence of errors does not prove that the compiler is free from errors unless all possible inputs are tested exhaustively, and the input space for compilers is huge[1]. As such, 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.

[1] In theory, the input space of a compiler is infinite. In practice, however, constraints such as maximum input file size bound the space.

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 codes which trigger bugs. There are two main approaches: *program generation*, where inputs are synthesised from scratch; and *program mutation*, where existing codes are modified so as to identify anomalous behaviour.

3.2.2.1 Test Case Generation for Compilers

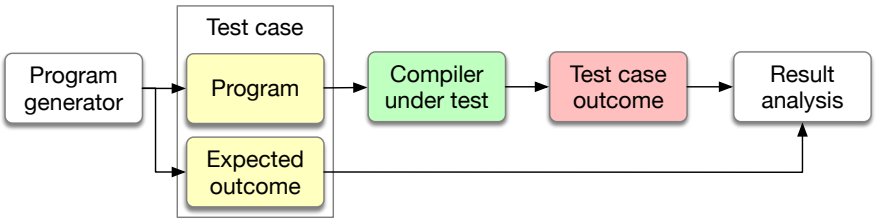
The idea of generating test cases for compilers is well-established. The majority of test case generation approaches are based on a formal specification of the programming language syntax and grammar. An early approach is presented by **Hanford1970a** [**Hanford1970a**], which randomly enumerates a context-free grammar to produce an inexhaustible supply of new programs. While the generated programs are syntactically valid, they are meaningless, and cannot be executed. This limits their value only to testing the compiler front end.

Deeper testing of compiler components is enabled by generating both a syntactically correct program and a *gold standard output* that would be produced by a conformant compiler. The compiled program can then be executed and its output compared against this gold standard. Figure 3.1a shows the process. Gold standard-based approaches are surveyed by **Boujarwah1997** [**Boujarwah1997**] and **Kossatchev2005** [**Kossatchev2005**]. The challenge of the approach is in generating the gold standard output.

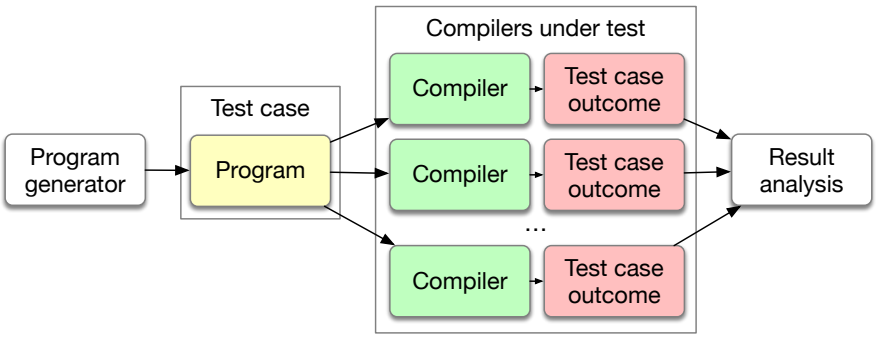
Differential testing, illustrated in Figure 3.1b, accelerates testing by enabling many compilers to be tested at once. The advantage of differential testing over prior approaches is that it does not require a gold standard for the expected behaviour of a conformant compiler. As such, any well-formed 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 no work in the literature has reported such issues.

Differential testing can be done across different compilers [**Chen2016b**; **Lidbury2015a**] or using the same compiler with different configurations [**Kyle2015b**; **Paka2011**] (or a combination of the two). **Chen2014a** [**Chen2014a**] empirically contrasts the two approaches, along with a comparison to Equivalence Module Inputs testing (described in the following subsection).

In the foundational work on differential testing for compilers, **McKeeman1998** [**McKeeman1998**]



(a) Expected outcome-based test case generation and evaluation



(b) Differential test case generation and evaluation

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.

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 [**Yang2011**] 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 very 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. **Lidbury2015a** [**Lidbury2015a**] 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 [**Nagai2013**] which targets arithmetic bugs.

Similarly to program generation for test compilers, prior work has focused on input generation for testing programs [**Godefroid2005**; **Claessen2015**; **Duregard2012**; **Fetscher1998**; **Runciman2008**]. Glade [**Bastani2017**] 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 very large. As such, once a bug has been identified, test case reduction [**Regehr2012a**] 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 [**Pflanzer2016**].

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

3.2.2.2 Mutation and Feedback-directed Testing

An alternate method for generating programs to use as compiler test cases is to mutate a seed input. Equivalence Modulo Inputs (EMI) testing, introduced by **Le2013a** [**Le2013a**], 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 [**Le2013a**],

code from dead regions is randomly deleted. *Athena* [**Le2015**] guides the approach using a Markov Chain Monte Carlo method, and supports dead code insertion as well as removal. *Proteus* [**Le2015a**] applies the EMI technique to test link-time optimizers. *Hermes* [**Sun2016a**] extends EMI to permit the mutation of *live code* regions, not just dead code. This greatly increases the expressiveness of the generated programs.

LangFuzz [**Holler2012**] 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* [**Peng2018**] uses dynamic tracing to detect input checks in programs and selectively removes them to expose defects. *Skeletal program enumeration* [**Zhang2017a**] 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* [**Mathis2019**] 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 [**Chen2016b**].

Machine learning has been used to guide test case mutation. **Cheng2019** [**Cheng2019**] 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* [**She2018**] learns a differentiable neural approximation of target program logic, then uses Stochastic Gradient Descent to guide program mutation. *Skyfire* [**Wang2017c**] 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 [**Zalewski**] 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 very long 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 at modelling sequential data [**Jozefowicz2016a**]. RNNs have been suc-

cessfully applied to a variety of generative tasks in other domains, including image captioning [**Vinyals**], colourising black and white photographs [**Zhang2016**], artistic style [**Gatys2015**], and image generation [**Gregor2014**].

The proficiency of RNNs for sequence modelling is well demonstrated [**Sutskever2014**]. **Sutskever2014** 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* [**Neelakantan2016**] is an early example of program generation through the latent representation of an artificial neural network.

Learn&Fuzz [**Godefroid2017**] and *IUST DeepFuzz* [**Nasrabadi2018**] 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.

Jitsunari2019 [**Jitsunari2019**] 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* [**Liu2019**], 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, it is trained on a wide range of handwritten programs with the intent of emulating *natural* programming styles.

Kosta2019 [**Kosta2019**] 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 this approach is the explosion in vocabulary size arising from the contextual grammar. To mitigate this, the authors must limit the expressiveness of the generator by using only a small subset of the grammar. They 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 program generation, 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-practise is for compilers to use a fixed ordering of optimisations, and for each optimisation to contain a heuristic to determine when to use it and with what parameters. Such heuristics require expert design at the expense of great effort and compiler expertise. 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 **Georgiou2018** [**Georgiou2018**] 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 selec-

tively *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, **Ryoo2008a** [**Ryoo2008a**] find speedups of up to 432× 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 the method of performance tuning in which a program is compiled and profiled using multiple optimisation configurations to find the configuration which maximises 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 runtime) is selected. Pioneered by **Bodin1998**, the technique was initially demonstrated to find good configurations in the non-linear three-dimensional optimisation space of a matrix multiplication benchmark [**Bodin1998**]. 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 can 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, **Chen2010** [**Chen2010**] found iterative compilation to yield speedups in GCC over the highest optimisation level (`-O3`) of up 2.23×.

The greatest challenge of iterative compilation is the exponential blowup 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

Triantafyllis2003 [**Triantafyllis2003**] 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. **Pan2006** [**Pan2006**] 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 [**Kulkarni2012**; **Purini2013**].

Ryoo2008 [**Ryoo2008**] 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%. There is no guarantee that for a new program, the reduced search space will include the optimal configuration. Similarly, **Purini2013** [**Purini2013**] 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* [**Ansel2013**] which provides ensemble search techniques and *CLTune* [**Nugteren2015**] for tuning OpenCL kernels.

A complementary approach to search space pruning is knowledge sharing. The idea is that, since most software has many users, share knowledge of the optimisation space 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* [**Saclay2010**], *Crowdtuning* [**Memon2013**], and *Collective Mind* [**Fursin2014**]. **Fursin2014** argue

that the challenges facing widespread adoption of iterative compilation techniques can be attributed to: a lack of common, diverse benchmarks and data sets; a lack of common experimental methodology; problems with continuously changing hardware and software; and the difficulty to validate 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 [**Fursin2014**]. In past work [**Cummins2016**], 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. In a real-world system, evaluating many sub-optimal configurations can 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.

Tartara2013 [**Tartara2013**] 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.

Ansel2012 [**Ansel2012**] 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.

Mpeis2015 [**Mpeis2015**] 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* [**Bala2000**] 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 & 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 selects compiler optimisations to produce different configurations, algorithmic choice selects between permutations of semantically equivalent algorithms, typically explicitly provided by the user.

PetaBricks [**Ansel2009a**] 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 [**Chan2009**], with the wider ambition that these auto-tuning techniques may be applied to all algorithmic choice programs [**Ansel2014**]. 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 optimisation space fruitful.

Halide [**Ragan-Kelley2013**] alleviates the burden of algorithmic rewriting by providing a high-level domain-specific language that allows users to express pipelines of stencil computations succinctly. The *Lift* framework [**Steuwer2017**] 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 **Ashouri2018** [**Ashouri2018**] and **Zhang2018** [**Zhang2018**] provide further detail.

Pioneered by **Agakov**, the idea is to use 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 can then be used to make predictions on unseen programs by extracting the features describing the program. **Agakov** [**Agakov**] use machine learning to guide the iterative compilation search. **Stephenson2003** [**Stephenson2003**] used "meta optimisation" to tune compiler heuristics through an evolutionary algorithm to automate the search of the optimisation space. **Kulkarni2012** [**Kulkarni2012**] 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. **Ashouri2017** [**Ashouri2017**] approach the phase-ordering problem by clustering optimisations and using machine learning to predict the speedup of sequences of optimisation clusters.

Lutz2013 [**Lutz2013**] and past work [**Cummins2016a**] 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. **Ganapathi2009** [**Ganapathi2009**] present an auto-tuner for stencil codes which can achieve 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 [**Dastgeer2011b**]. 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 heteroge-

neous device mapping, though the system is capable of making predictions for arbitrary GPU programs, it is not bound to a single template library. **Moren2018** [**Moren2018**] also tackle the task of mapping arbitrary OpenCL kernels to CPU/GPU using dynamic features extracted from the kernel at runtime.

Milepost GCC [**Fursin2011**] 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.

Ogilvie2017 [**Ogilvie2017**] 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. **Kraska2017** [**Kraska2017**] find that replacing a cache-optimised B-Tree-Index implementation with a deep learning model yields up to 70% speedups with a 10× reduction in memory on real workloads. **Krishnan2018** [**Krishnan2018**] 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. Given that there is an infinite number of potential features, finding the right set of features is a non-trivial, time-consuming task.

Various forms of features have been used to summarise programs. **Dubach2009** [**Dubach2009**] characterise programs using performance counters. **Jiang2010** [**Jiang2010**] extract program-level behaviours such as loop trip counts and the size of input files. **Berral2010a** [**Berral2010a**] 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 via feature

selection, or projected into a lower dimensional space. **Stephenson2005** [**Stephenson2005**] 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. **Collins2013** [**Collins2013**] use Principal Component Analysis (PCA) to reduce a four-dimensional feature space to two dimensions, reducing the size of the space to 0.05%. **Dubach2007** [**Dubach2007**] 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* [**Ting2016**] employs a range of existing feature selection methods to select useful candidate features.

Prior works have sought to reduce the cost of feature design. **Park2012** [**Park2012**] 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 very large. Finally, their graph matching metric is expensive, requiring $O(n^3)$ 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. **Leather2014** [**Leather2014**] uses genetic programming to search for features, requiring a huge grammar to be written, some 160kB in length. Although much of this can be created from templates, selecting the right range of capabilities and search space bias is non-trivial and up to the expert. **Namolaru2010a** [**Namolaru2010a**] 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, combinators and constraints to work. For both approaches, the search time may be significant.

Cavazos2006 [**Cavazos2006**] 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 can 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 natural language processing. With these techniques, a program source code is tokenised into a vocabulary of words, and the words mapped into a real-valued *embedding* space. There are many choices in how to construct the vocabulary and embedding. **Chen2019** [**Chen2019**] review some of the proposed techniques. **Babii** [**Babii**] explore the impact that choices in vocabulary have on time to convergence of software language models.

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. **Cvitkovic2018a** [**Cvitkovic2018a**] 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.

Mou2016 [**Mou2016**] derive an embedding space from the tokens in the source code of a program. **Wang2017d** [**Wang2017d**] propose an embedding space extracted from program traces, rather than the syntactic structure of the program. **Henkel2018** [**Henkel2018**] use symbolic execution to abstract the program traces. Embeddings are then learned from these abstracted symbolic traces. **Yin2018** [**Yin2018**] and **Tufano2019** [**Tufano2019**] present techniques for learning representations of code edits.

Neural Code Comprehension [**Ben-nun2018**] builds on techniques proposed in

Chapter 6 of this thesis to develop embeddings derived from a novel *Contextual Flow Graph* (XFG) representation which contains the union of both data and control flow graphs. The embeddings are trained using a skip-gram model [**Mikolov2013a**], using a vocabulary derived from LLVM bitcode. This enables the same embeddings to be re-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 [**LeCun2015**; **Wang2017**]. 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 traditional techniques is its ability to process natural data in its raw form. This overcomes the traditionally laborious and time-consuming practise 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 [**Xiong2016**]; super-human level performance in video games [**Mnih2015**]; and autonomous vehicle control [**Lozano-Perez2012**]. The use of deep learning techniques for software engineering has long been a goal of research [**White2015a**].

A **Allamanis2017a** survey by **Allamanis2017a** describes the fast-moving field of deep learning techniques for programming languages [**Allamanis2017a**]. *Auto-Comment* [**Wong2013**] mines the popular Q&A site StackOverflow to automatically generate code comments. *Naturalize* [**Allamanis2014a**] employs techniques developed in the natural language processing domain to model coding conventions. *JSNice* [**Raychev2015**] leverages probabilistic graphical models to predict program properties such as identifier names for JavaScript. **Allamanis2016** [**Allamanis2016**] use attentional neural networks to generate summaries of source code. *Nero* [**David2019**] 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 large scale [**Allamanis2013a**; **White2015a**; **Bird2009**]. Previous uses outside the field of machine learning have in-

volved data mining of GitHub to analyse software engineering practices [**Wu2014**; **Guzman2014**; **Baishakhi2014a**; **Vasilescu2015**]. **Allamanis2018** [**Allamanis2018**] 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. **Heo2017** [**Heo2017**] present a machine-learning technique to tune static analysis to be selectively unsound, based on anomaly detection techniques. **Koc2017** [**Koc2017**] present a classifier that attempts to predict whether a static analysis tool's error report is a false positive based on the program structures of previous reports that produced false error reports. **Lam2016** [**Lam2016**] employ artificial neural networks to relate keywords in bug reports to code tokens and terms in source files and documentation to accelerate bug localisation. **Wang2016c** [**Wang2016c**] employ a deep belief network [**Hinton2006a**] to automatically learn semantic features from token vectors extracted from programs' abstract syntax trees. The features are then used for automatic defect detection. **Chen2017** [**Chen2017**] 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* [**Pradel2018**] combines a binary classification of correct and incorrect code with semantic processing to name bugs. *Code2Inv* [**Si2018**] uses reinforcement learning to learn loop invariants for program verification.

Machine learning has been applied to the task of automatic software repair. **Monperrus2018** surveys the literature [**Monperrus2018**]. *DeepRepair* [**White2019**] using an encoder-decoder architecture to sort code fragments according to their similarity to suspicious elements. **Vasic2019** [**Vasic2019**] train a model to jointly localise and repair variable-misuse bugs using multi-headed pointer networks. *SequenceR* [**Chen2018**] uses sequence-to-sequence learning to generate patches. *Getafix* [**Bader2019**] uses a hierarchical clustering algorithm that summarises fix patterns into a hierarchy ranging from general to specific patterns. **Brockschmidt2018** [**Brockschmidt2018**] present a novel methodology for program generation in which a graph is used as the intermediate representation.

CodeBuff [**Terence2016**] uses a hand-designed set of features to learn abstract code formatting rules from a representative corpus of programs. **Raychev2014** [**Raychev2014**] use statistical models to provide contextual code completion. **Zhang2015a** [**Zhang2015a**] use deep learning to generate example code for APIs as responses to natural language queries. **Oda2015** [**Oda2015**] 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

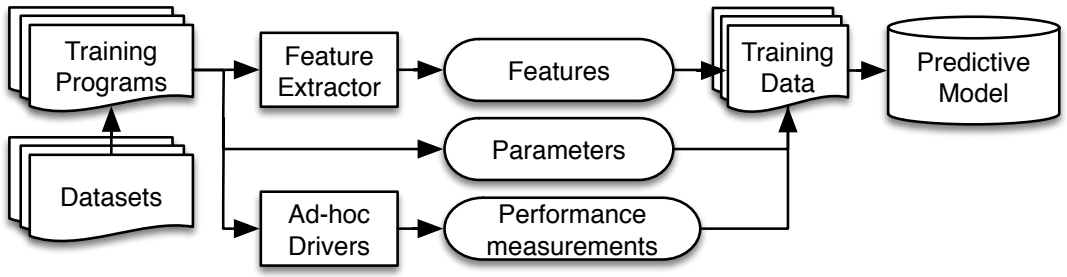


Figure 4.1: Training a predictive model for compiler optimisations. A model is constructed from training data, which comprises the features, performance measurements, and runtime parameters of training programs and their accompanying datasets.

1.27×. 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×.

Predictive modelling is a well-researched method for building optimisation heuristics that often exceed human experts and reduces development time [**Micolet2016**; **Wang2014c**; **Magni2014**; **Cummins2016**; **Wang2009**; **Wen2015**; **Wang2010**; **Falch2015**; **Collins2012**; **Leather2014**; **Ogilvie2014a**]. 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 can 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 can 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 over a highly restricted space of stencil benchmarks with little control flow or program variability [**Falch2015**; **Cummins2016a**]. But, it is not clear how much effort it will take, or even if it is possible for human experts to define grammars capable of producing human-like programs in more complex domains.

The approach introduced in this chapter does not require an expert to define what human programs look like. Instead, the structure and likelihood of programs are automatically inferred over a huge corpus of 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 effectiveness of deep learning for modelling complex structure in natural languages [**Graves2013**; **Sutskever2014**]. 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, it is discovered 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 this significantly increases the performance of the learned heuristic. This indicates potential value of this technique for feature designers.

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 then 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 research papers in the field of GPGPU performance tuning 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 benchmarks suites account for the majority of results, illustrated in Figure 4.2. The performance of the state-of-the-art **Grewe2013** [**Grewe2013**] predictive model was evaluated across each of the 7 most frequently used benchmark suites (accounting for 92% of results in the surveyed papers). The **Grewe2013** model predicts whether running a given OpenCL kernel on the GPU gives 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 very 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 heuristics learned on one

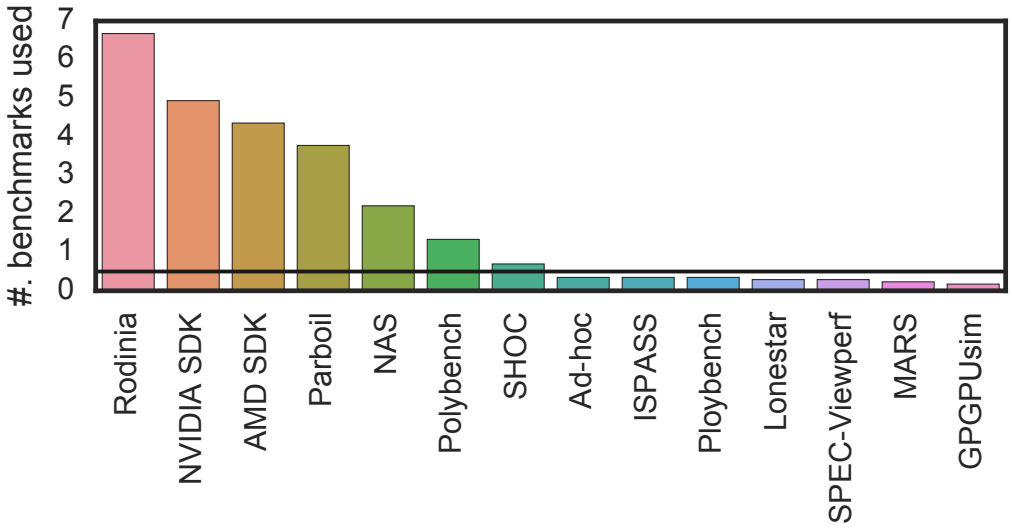


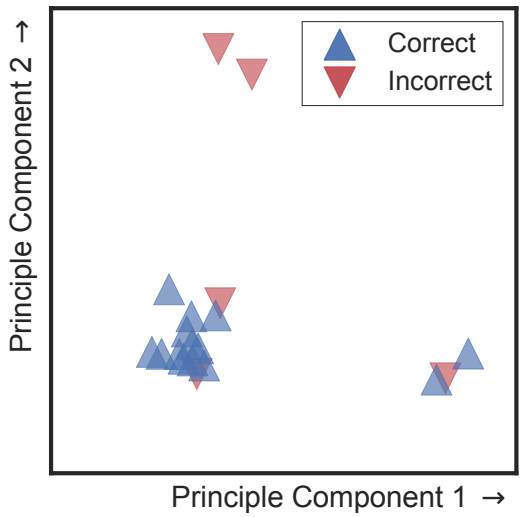
Figure 4.2: The average number of benchmarks used in GPGPU research papers published between 2013-2016 in CGO, HiPC, PACT, and PPoPP conferences. The average GPGPU research paper uses 17 benchmarks, with the seven most popular benchmark suites accounting for 92% of results.

benchmark suite fail to generalise across other suites.

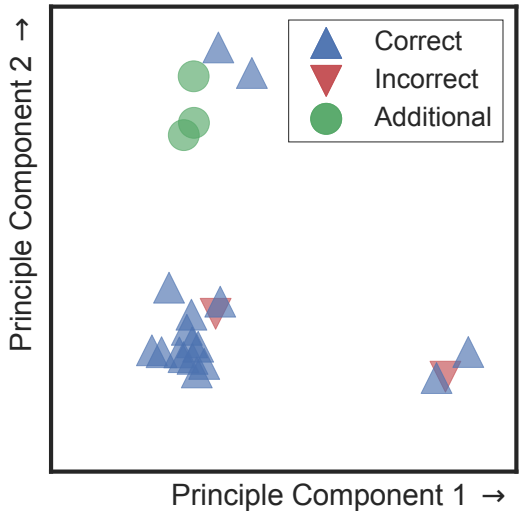
This problem is caused both by the limited number of benchmarks contained in each suite, and the distribution of benchmarks within the feature space. Figure 4.3 shows the feature space of the Parboil benchmark suite, showing whether, for each benchmark, the model was able to correctly predict the appropriate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

As can be 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



(a)



(b)

Figure 4.3: A two dimensional projection of the **Grewe2013** 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.

	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 **Grewe2013** 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.

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.

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 [**LeCun2015**]. For the purpose of modelling a programming language, this means assembling a very 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

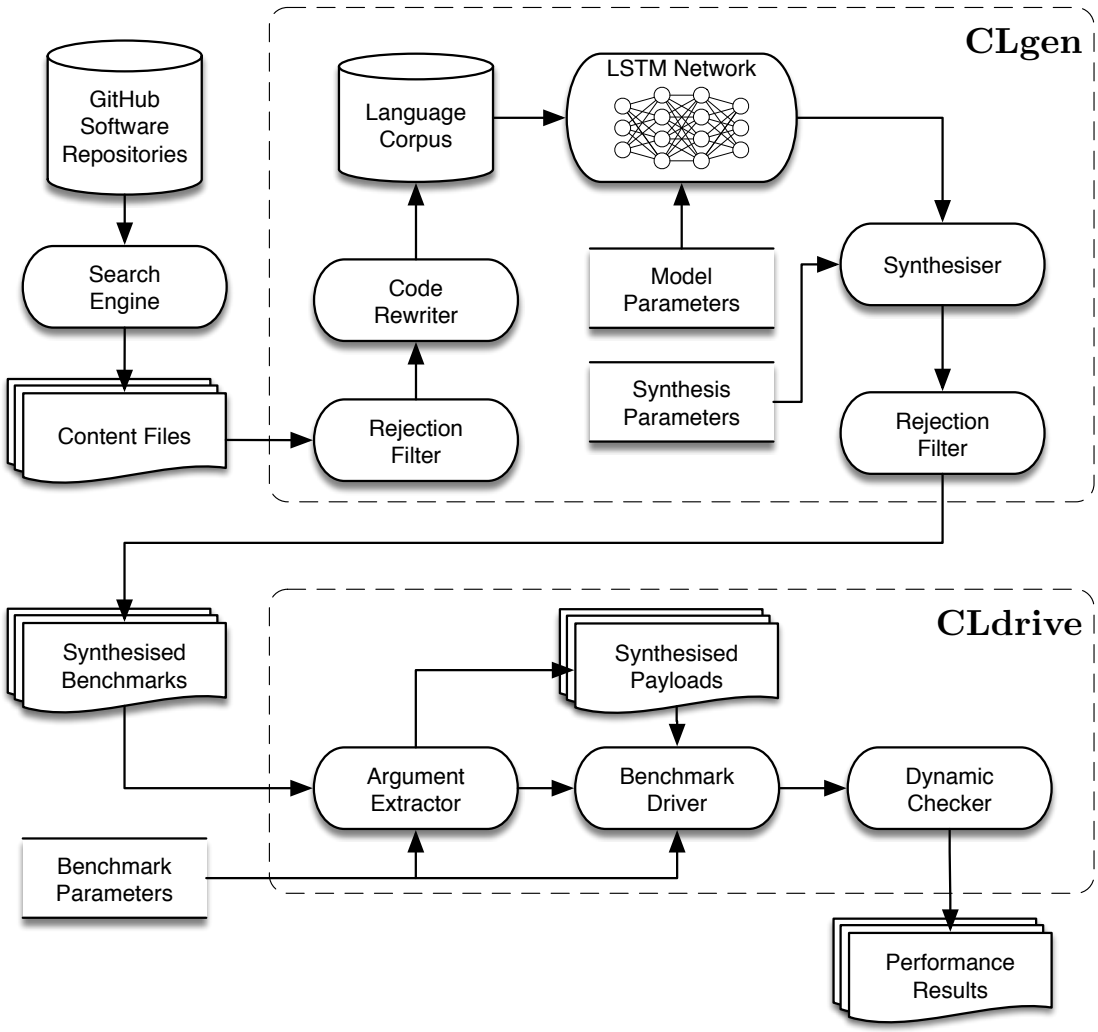


Figure 4.4: The benchmark synthesis and execution pipeline. Software is mined from GitHub; this is used to construct a language model from which new programs may be synthesised; a benchmark driver is used to produce performance results.

```
/* Enable OpenCL features */
#define cl_clang_storage_class_specifiers
#define cl_khr_fp64
#include <clc/clc.h>

/* Inferred types */
typedef float FLOAT_T;
typedef unsigned int INDEX_TYPE;
... (36 more)

/* Inferred constants */
#define M_PI 3.14025
#define WG_SIZE 128
... (185 more)
```

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

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 $\{a, b, c, \ldots, aa, ab, ac, \ldots\}$ for variables and $\{A, B, C, \ldots, AA, AB, AC, \ldots\}$ for functions. This process isolates the syntactic structure of the code, and unlike prior work [**Allamanis2013a**], 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 [**Weinberger2011**] 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%.

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) [**Hochreiter1997**] architecture of Recurrent Neural Network [**Sundermeyer2012**; **Mikolov2015**] 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 [**Graves2005**].

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,

```
1  // saxpy.cl - Compute kernel for SAXPY
2  #define DTYPE float
3  #define ALPHA(a) 3.5f * a
4  inline DTYPE ax(DTYPE x) { return ALPHA(x); }
5
6  kernel void saxpy( /* SAXPY kernel */
7      global DTYPE *input1,
8      global DTYPE *input2,
9      const int nelem)
10 {
11      unsigned int idx = get_global_id(0);
12      // = ax + y
13      if (idx < nelem) {
14          input2[idx] += ax(input1[idx]); }
15 }
```

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

```
inline float A(float a) {
  return 3.5f * a;
}

kernel void B(global float* b, global float* c, const int d)
 ↪ {
  unsigned int e = get_global_id(0);

  if (e < d) {
    c[e] += A(b[e]);
  }
}
```

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.

with a final model size of 648MB[1]. Training the network is a one-off cost, and can be parallelised 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

[1]No effort was made to minimise training train. 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 ← "__kernel void A(const int a) {"          ▷ Seed text
2:  d ← 1                                          ▷ Initial code block depth
3:  for i ← |S| to n do
4:      c ← PredictCharacter(M, S)                 ▷ Generate new character
5:      if c = "{" then
6:          d ← d + 1                              ▷ Entered code block, increase depth
7:      else if c = "}" then
8:          d ← d − 1                              ▷ Exited code block, decrease depth
9:      end if
10:     S ← 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
```

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 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.

```
kernel void A(global float* a,
              global float* b,
              global float* c,
              const int d) {
  int e = get_global_id(0);
  float f = 0.0;
  for (int g = 0; g < d; g++) {
    c[g] = 0.0f;
  }
  barrier(1);

  a[get_global_id(0)] = 2*b[get_global_id(0)];
}
```

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

```
kernel void A(global float* a,
              global float* b,
              global float* c,
              const int d) {
  int e = get_global_id(0);
  if (e >= d) {
    return;
  }
  c[e] = a[e] + b[e] + 2 * a[e] + b[e] + 4;
}
```

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

```
kernel void A(global float* a,
              global float* b,
              global float* c,
              const int d) {
  unsigned int e = get_global_id(0);
  float16 f = (float16)(0.0);
  for (unsigned int g = 0; g < d; g++) {
    float16 h = a[g];
    f.s0 += h.s0;
    f.s1 += h.s1;
    f.s2 += h.s2;
    f.s3 += h.s3;
    f.s4 += h.s4;
    f.s5 += h.s5;
    f.s6 += h.s6;
    f.s7 += h.s7;
    f.s8 += h.s8;
    f.s9 += h.s9;
    f.sA += h.sA;
    f.sB += h.sB;
    f.sC += h.sC;
    f.sD += h.sD;
    f.sE += h.sE;
    f.sF += h.sF;
  }
  b[e] = f.s0 + f.s1 + f.s2 + f.s3 + f.s4 + f.s5 + f.s6 + f.s7 +
   ↪  f.s8 + f.s9 + f.sA + f.sB + f.sC + f.sD + f.sE + f.sF;
}
```

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

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 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 [**McKeeman1998**], 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 [**Betts2012**; **Price2015**; **Sorensen2016**], 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* [**Gao2015a**; **Zhang2016**; **Vinyals**]. 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 [**Lidbury2015a**], an OpenCL program generator for differential testing[2].

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`

[2] An online version of this test is available at *http://humanorrobot.uk/*.

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).

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

This section describes the methodology for a quantitative evaluation of CLgen-generated benchmarks.

4.6.1 Experimental Setup

Predictive Model To evaluate the efficacy of synthetic benchmarks for training, the predictive model of **Grewe2013** is used [**Grewe2013**]. 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 [**Grewe2013**], the model is tested on the NAS Parallel Benchmarks (NPB) [**Bailey1991a**]. The hand-optimised OpenCL implementation of **Seo2011** [**Seo2011**] is used. In [**Grewe2013**] 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; the other is equipped with an NVIDIA GPU and uses Ubuntu 16.04. Both platforms were unloaded.

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 **Grewe2013** 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 [Seo2011])	1.0.3	7	114
Rodinia [Che2009]	3.1	14	31
NVIDIA SDK	4.2	6	12
AMD SDK	3.0	12	16
Parboil [Stratton2012]	0.2	6	8
PolyBench [Grauer-Gray2012]	1.0	14	27
SHOC [Danalis2010]	1.1.5	12	48
Total	-	71	256

Table 4.3: List of benchmarks used to train and evaluate the **Grewe2013** 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 **Grewe2013** predictive model.

Data sets The NPB and Parboil benchmark suites are packaged with multiple data sets. We use all of the packaged data sets (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 [**Grewe2013**]. 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 [**Grewe2013**] 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 **Grewe2013** 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× 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.

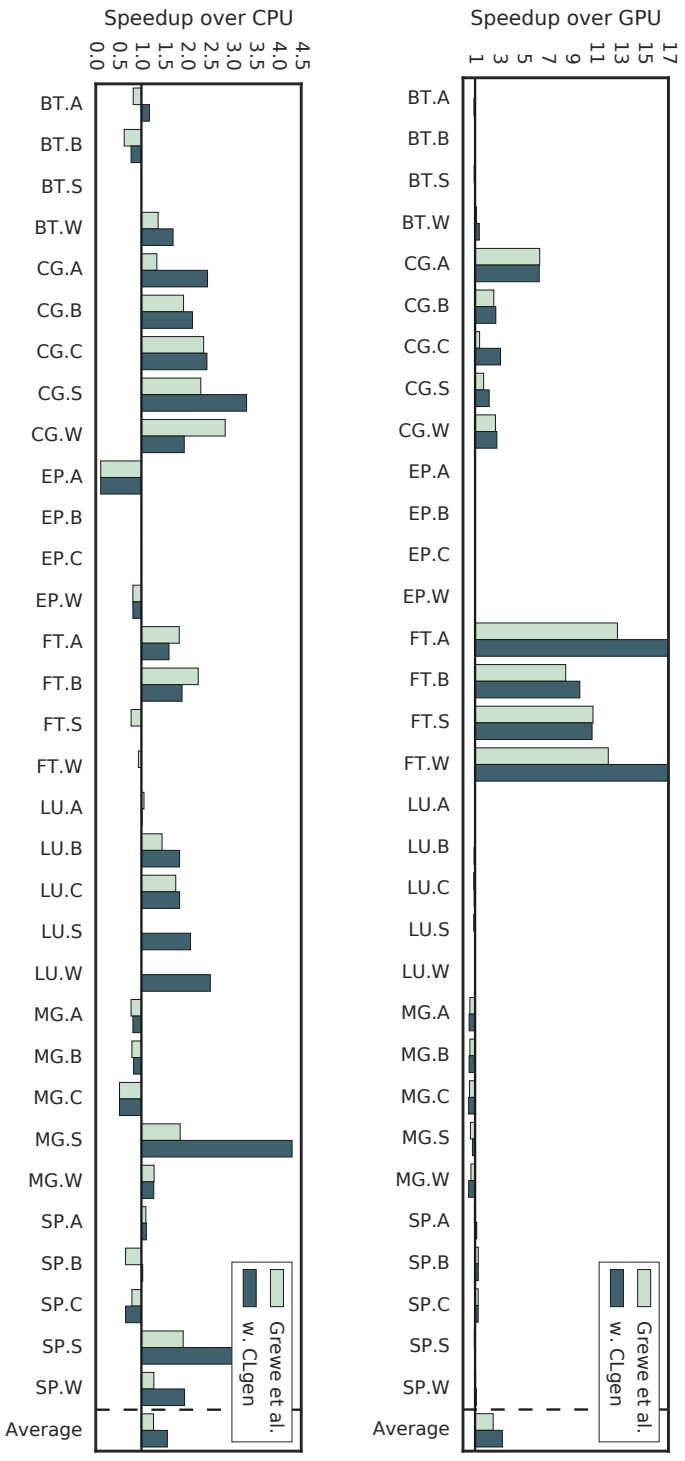
The strongest performance improvements are on NVIDIA with the `FT` benchmark which suffers greatly under a single-device mapping. However, the performance on AMD for the same benchmark slightly degrades after adding the synthetic benchmarks. This issue is addressed in the next section.

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 can limit a model's ability to generalise over a wider range of programs. This is found to be the case with the **Grewe2013** 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 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 the of existing benchmarks — AMD's Fast Walsh-Hadamard



(a) AMD Tahiti 7970 (b) NVIDIA GTX 970

Figure 4.5: Speedup of programs using **Grewe2013** 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.

```
kernel
void fastWalshTransform(global float * tArray,
                        const  int   step
                       )
{
    unsigned int tid = get_global_id(0);

    const unsigned int group = tid%step;
    const unsigned int pair  = 2*step*(tid/step) + group;

    const unsigned int match = pair + step;

    float T1          = tArray[pair];
    float T2          = tArray[match];

    tArray[pair]              = T1 + T2;
    tArray[match]             = T1 - T2;
}
```

Listing 7: AMD's Fast Walsh Transform benchmark kernel. In the **Grewe2013** 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.

transform, Listing 7 — but with different behaviour. We found this to be caused by the lack of discriminatory features for branching, since the NPB programs are implemented in a manner which aggressively minimised branching. To counter this the predictive model was extended with an additional feature containing a static count of branching operations in a kernel.

Figure 4.6 shows speedups of the extended model 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

```
kernel void A(global float* a,
              global float* b,
              global float* c,
              const int d) {
  int e = get_global_id(0);
  if (e < 4 && e < c) {
    c[e] = a[e] + b[e];
    a[e] = b[e] + 1;
  }
}
```

Listing 8: In the **Grewe2013** 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.

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 succeeds, achieving average speedups of 3.56× on AMD and 5.04× 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 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

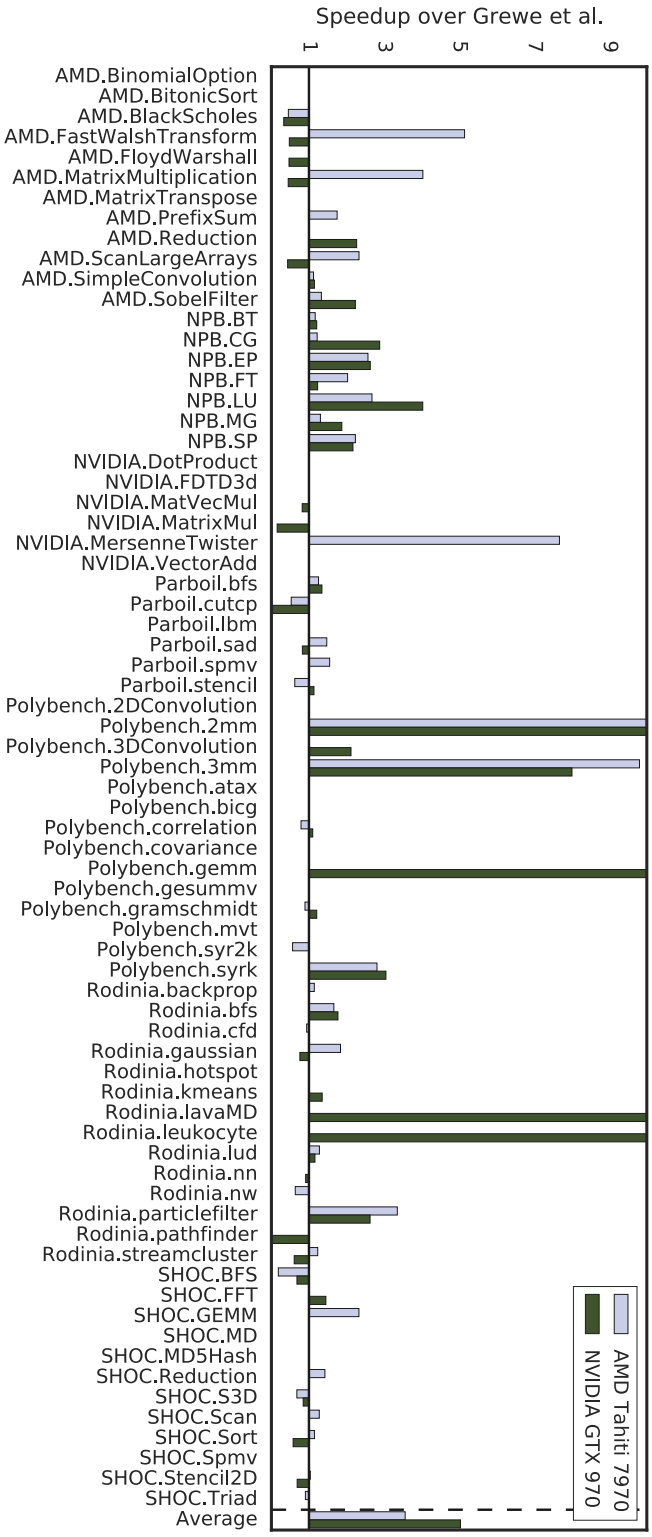


Figure 4.6: Speedups of predictions using an extended model over **Grewe2013** on both experimental platforms. Synthetic benchmarks and the additional program features outperform the original predictive model by a factor $3.56\times$ on AMD and $5.04\times$ on NVIDIA.

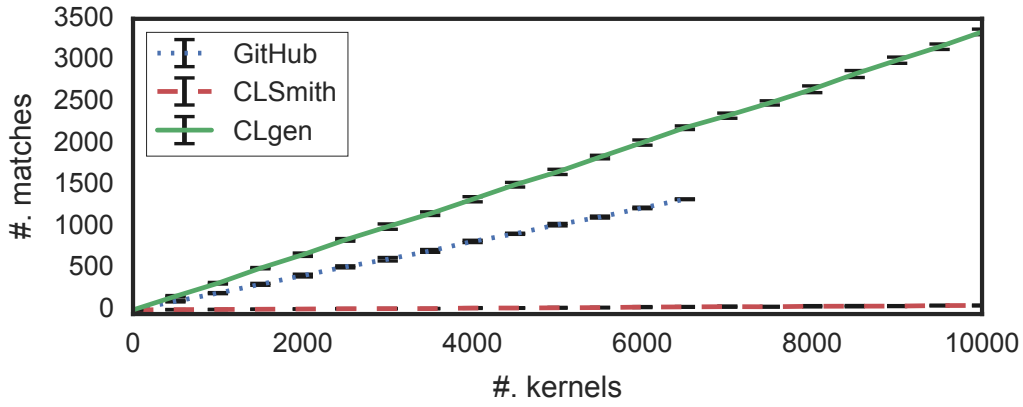


Figure 4.7: The number of kernels from GitHub, CLSmith, and CLgen with static code features matching the benchmarks. CLgen generates kernels that are closer in the feature space than CLSmith, and can continue to do so long after we have exhausted the extent of the GitHub data set. Error bars show the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

runtime features, and programs 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 bound by the quantity and quality of programs used for training, yet there are typically only a few dozen common benchmarks available for experiments. 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 semantics 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×. Synthetic benchmarks automatically exposed weaknesses in the feature set which, when corrected, further improved the performance by 4.30×.

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 synthesized 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 can hinder software development or even cause 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 [**Chen2014a**; **Chen2013**; **Kossatchev2005**]. 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 [**McKeeman1998**] 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 [**Yang2011**], generates large random pro-

grams 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 [**Lidbury2015a**] — 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 the test of OpenCL programming language compilers. 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× 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 tech-

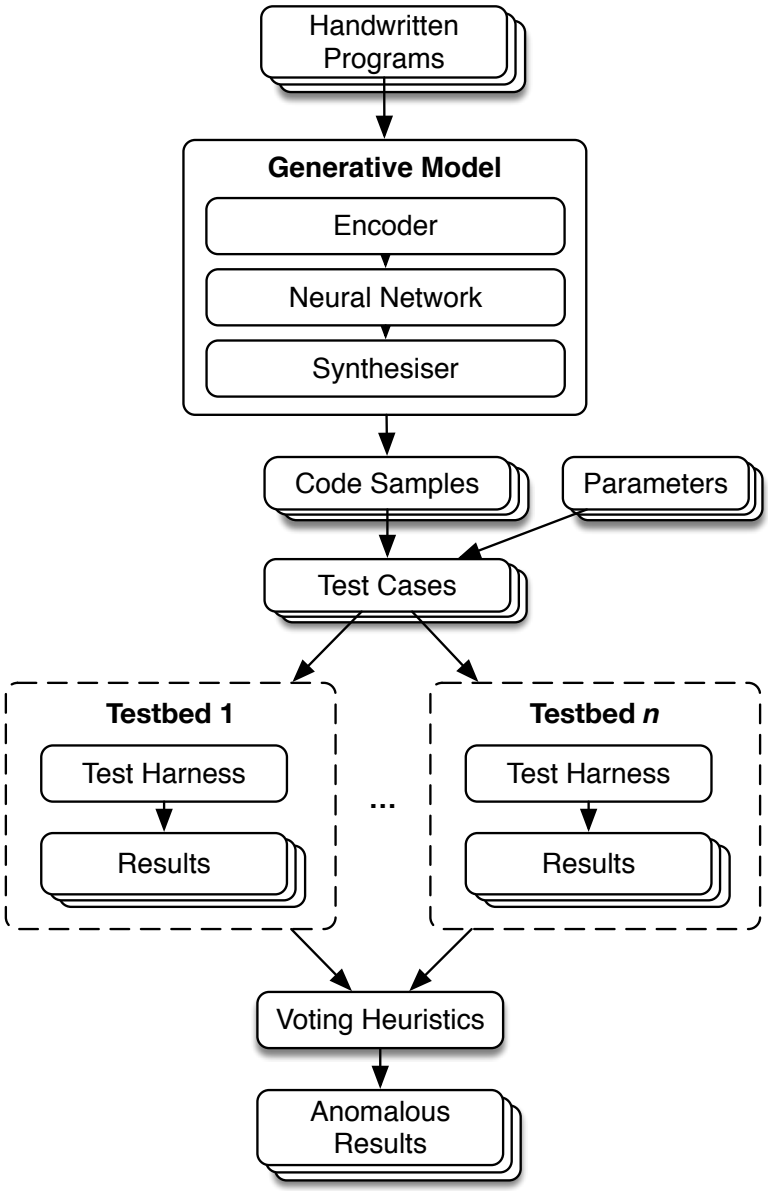


Figure 5.1: DeepSmith system overview. Handwritten programs are used to derive a generative model; from which code samples are produced and parameterised to make test cases. Test cases are broadcast to multiple testbeds, and voting heuristics used to determine the testbeds that deviate from the majority,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

niques to build models for how humans write programs. The approach is inspired by breakthrough results in modelling challenging and high dimensional data sets through unsupervised learning [**Raghu2016**; **Radford2016b**; **Bowman2015**]. 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 our 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, semantic-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: $\{a, b, c, \ldots, aa, ab, ac, \ldots\}$ for variables and $\{A, B, C, \ldots, AA, AB, AC, \ldots\}$ 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 can 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 [**Jozefowicz2016a**], a character based vocabulary is used. This minimises the size

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

Require: Candidate vocabulary V_c, string S.

Ensure: Vocabulary V.

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

of the vocabulary but leads to long sequences which are harder to extract structure from. In [**Allamanis2013a**], 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 used. 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 we use to encode new program sources.

Neural Network The Long Short-Term Memory (LSTM) architecture of Recurrent Neural Network is used to model program code [**Hochreiter1997**]. In the LSTM architecture activations are learned with respect not just to their current inputs but to previous inputs in a sequence. In our 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 [**Pacanu2013**]. This makes them effective at learning complex relationships over long sequences [**Lipton2015**] 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 can be 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 [**McKeeman1998**].

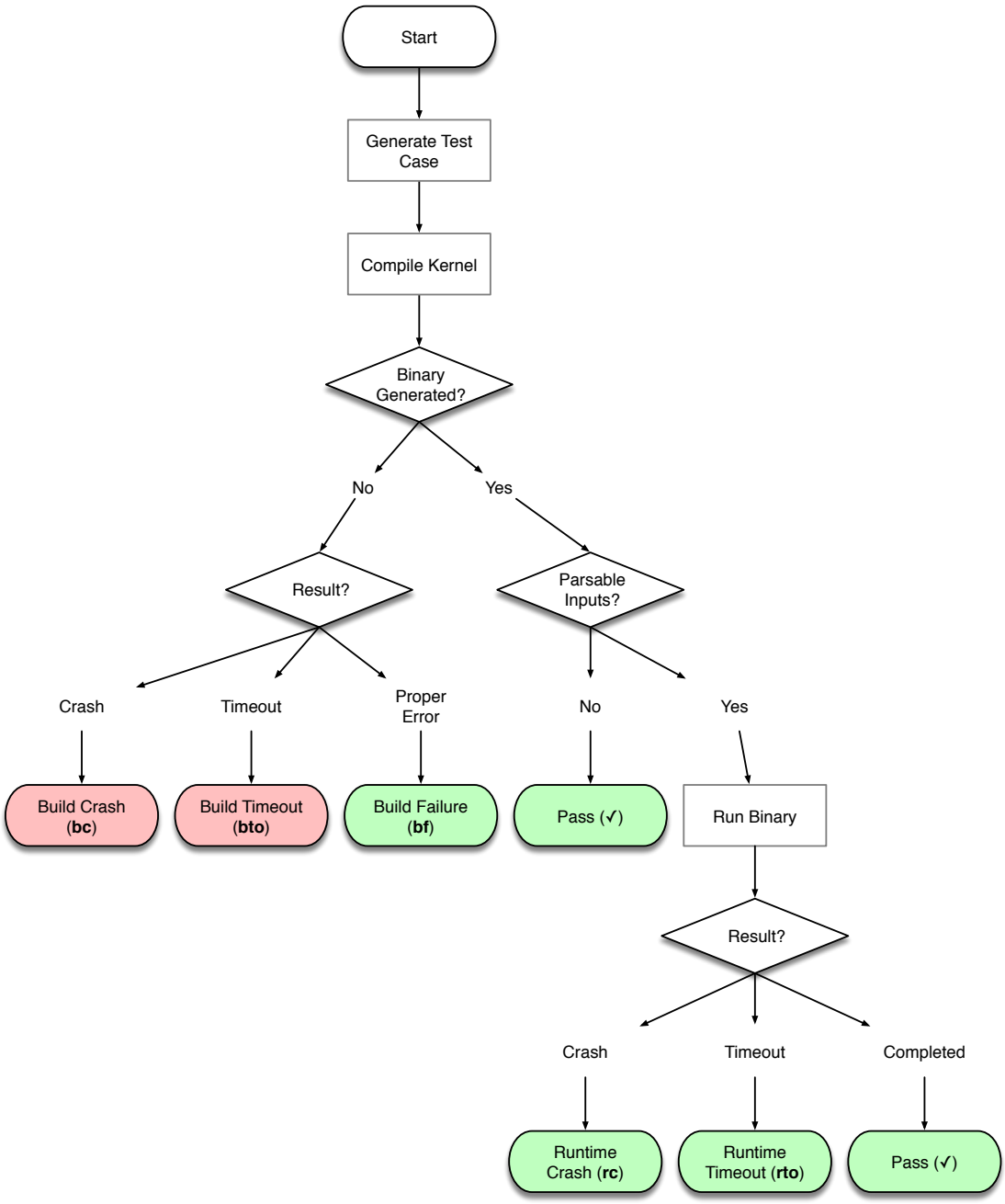


Figure 5.2: Test case execution, and possible results. Generating and executing a test case leads to one of six possible outcomes. Of these, build crashes and build timeouts are considered errors. In the remaining four cases, differential testing may be used to determine if the outcome is anomalous.

However, this approach is extended to describe not only miscompilations, but also anomalous build failures and crashes.

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 non-deterministic behaviour that have been actually observed to happen in practice.

Data races, out-of-bounds and uninitialised accesses are filtered using GPUverify [**Betts2012**] and Oclgrind [**Price2015**]. Some compiler warnings provide a strong indication of non-deterministic behaviour (e.g. comparison between pointer and integer) – these warnings are checked for and filtered accordingly.

Floating point operations in OpenCL can be 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 $<$*device, driver, is_optimised*$>$ 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. Test cases are generated, one using one thread, the other using 2048 threads. A test case is then a triple, consisting of $<$*program, inputs, threads*$>$ 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

#.	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.

case consists of the generation and execution time.

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 — CLSmith [**Lidbury2015a**]. 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 199 test cases were identified which trigger unreachable code failures, triggered 31 different compiler assertions, and produced 114 distinct stack traces from 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.

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-

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

(a) Testbeds 10± assertion *Uncorrected typos!* during semantic analysis.

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

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

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

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

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

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

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

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

Figure 5.4: Example OpenCL kernels which crash compilers.

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± and 7±. 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 commercial Intel compilers tested hang during optimisation of non-terminating loops (Figure 5.5d).

Testbeds 7± 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±, 6±, 9± reject out-of-range literal values e.g. `int i = 0xFFFFFFFFFFFFFFFFFFFFFFFF`, whilst Testbeds 3±, 5±, 7±, 8±, and 10± interpret the literal as an `unsigned long long` and implicitly cast to an integer value of `-1`. Testbeds 1±, 2± emit no warning.

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

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

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

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

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

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

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

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

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

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

Figure 5.5: Example OpenCL kernels which hang compilers.

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 [**Chen2014a**]. 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 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 ($tid \rightarrow$ int, `if (tid < ...)` $\{\ldots\}$) 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 5.6c 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.

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

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

```
kernel void A(global int* a, global int* b) {
  switch (get_global_id(0)) {
  case 0:
    a[get_global_id(0)]=b[get_global_id(0)+13];
    break;
  case 2:
    a[get_global_id(0)]=b[get_global_id(0)+11];
    break;
  case 6:
    a[get_global_id(0)]=b[get_global_id(0)+128];
  }
  barrier(2);
}
```

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

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

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

Figure 5.6: Example OpenCL kernels which are miscompiled.

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

(a) 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 kernel which is miscompiled.

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 5.7a 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 timeouts[1]. DeepSmith generation time grows linearly with pro-

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

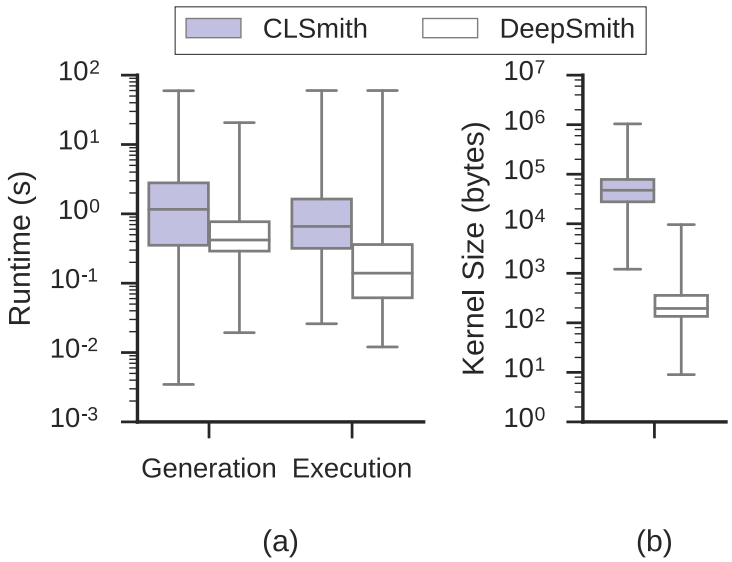


Figure 5.8: Comparison of runtimes (a) and test case sizes (b). DeepSmith test cases are on average evaluated $3.03\times$ faster than CLSmith ($2.45\times$, and $4.46\times$ for generation and execution, respectively), and are two orders of magnitude smaller. Timings do not include the cost of timeouts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

gram 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 [**Lidbury2015a**] 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.

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 [**Pflanzer2016**], 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

#.	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. $\pm$ 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.

#.	Device	±	bc	bto	abf	arc	awo	✓	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.

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±, 5±, 6±, and 7±) 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 very 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±) 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±, and a further 8 distinct compiler assertions.

The 4719 **abf** results for CLSmith on Testbeds 4± and 6± 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

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

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

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

(b) *Simplify the CFG* pass.

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

(c) *Predicator* pass.

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

(d) *Combine redundant instructions* pass.

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

(e) *PrepareKernelArgs* pass.

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

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

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

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

(b) *Intel OpenCL RemoveDuplicationBarrier* pass.

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

(c) *X86 DAG-¿DAG Instruction Selection* pass.

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

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

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

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

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

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

that it is an unusual construct (0.6% of test cases, versus 7.0% of CLSmith test cases).

ComputeAorta (Testbeds 9±) defers kernel compilation so that it can perform optimisations dependent on runtime parameters. This may contribute to the relatively large number of **arc** results and few **bc** results of Testbeds 9±. 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.

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 crash rate of 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. 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

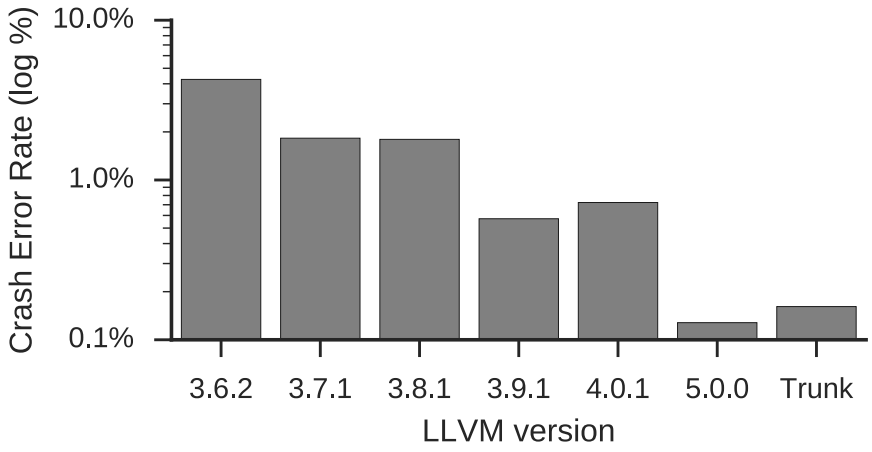


Figure 5.12: Crash rate of the Clang front-end of every LLVM release in the past 24 months compiling 75k DeepSmith kernels.

	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	±	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.

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.

5.4.5.1 Testing Methodology

The same methodology was applied to train the program generator as for OpenCL. A corpus of 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. 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 our 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

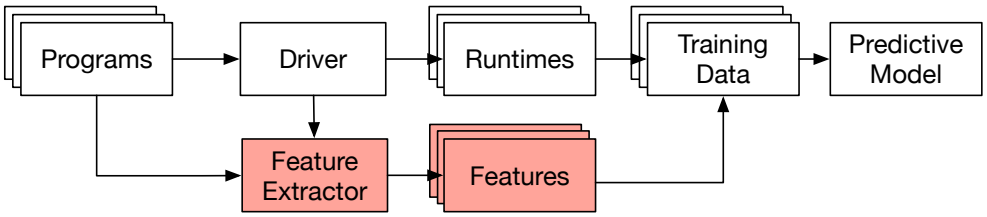
Simplifying the Construction of Optimisation Heuristics

6.1 Introduction

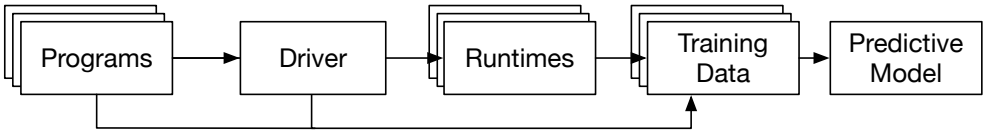
There are countless scenarios during the compilation and execution of a program where decisions must be made as to how, or if, a particular optimisation should be applied. Modern compilers and runtimes are rife with hand-coded *heuristics* which perform this decision making. The performance of programs 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 [**Micolet2016**; **Falch2015**; **Stephenson2005**; **Agakov**; **Cummins2016a**]. 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 [**Grewe2013**; **Magni2014**].

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



(a) Current state-of-practice



(b) The proposed approach

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. One such feature was identified in Section 4.7.2, 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 [**Namolaru2010a**; **Leather2014**], 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 astounding 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 remove the challenge of feature design, leaving only model selection.

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 in the 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 run-time input parameters. Beyond these optional inclusions, the system is able to learn optimisation heuristics without human guidance.

By employing *transfer learning* [**Yosinski2014**], 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 our artificial neural networks are mostly independent of the optimisation problem. Parts of the network may be reused across heuristics, and, in the process, can 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 been used before for heuristic construction 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, evaluating 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. Then Section 6.6 illuminates the inner workings of DeepTune. 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 forgoes 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 [**Abadi**] and Theano [**Bergstra2011**] 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.

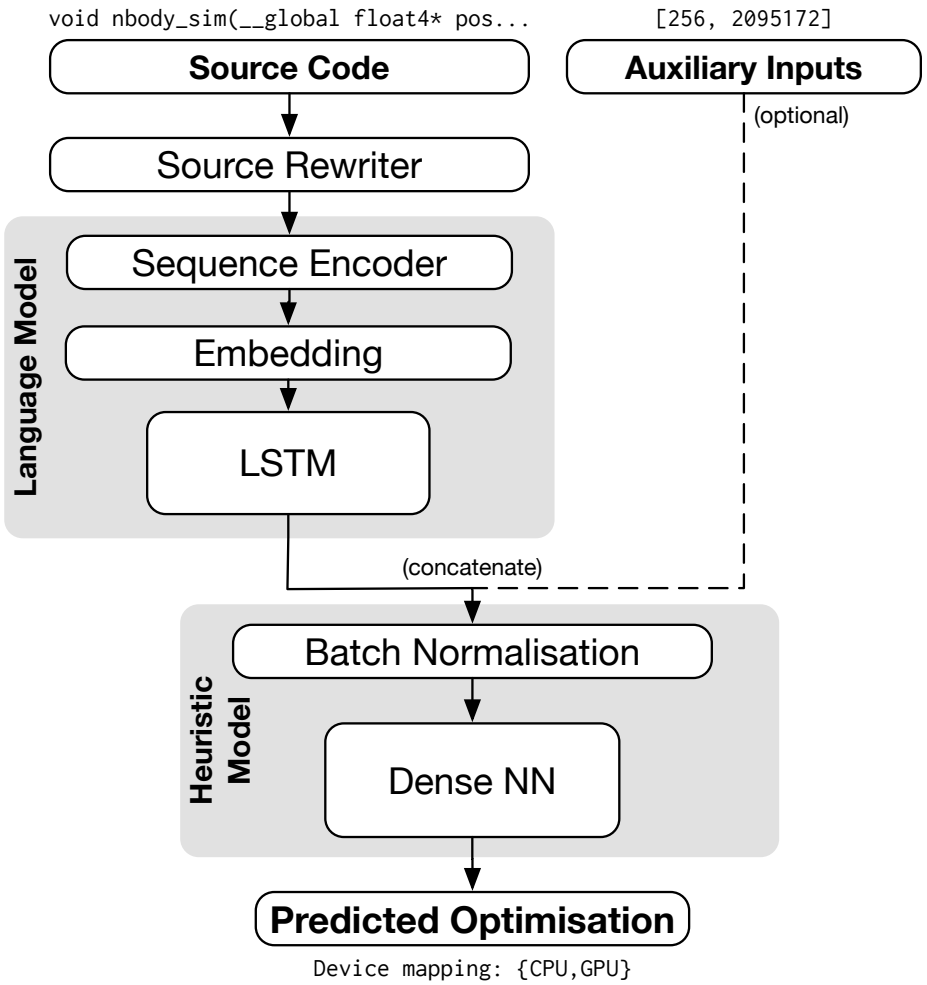


Figure 6.2: DeepTune system overview. Code properties are extracted from source code by the language model. They are fed, together with optional auxiliary inputs, to the heuristic model to produce the final prediction.

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 4. 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 for a single input source code Figure 6.3b.

6.2.2.3 Embedding

During encoding, tokens in the vocabulary are mapped to unique integer values, e.g. `float` $\rightarrow 0$, `int` $\rightarrow 1$. The integer values chosen 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 [**Mikolov2013a**; **Baroni2014**]. 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 $\boldsymbol{W}_E \in \mathbb{R}^{|V| \times D}$ is learned during training, so that an integer-encoded sequences of tokens $\boldsymbol{t} \in \mathbb{N}^L$ is mapped to the matrix $\boldsymbol{T} \in \mathbb{R}^{L \times D}$. In this work, a embedding dimensionality $D = 64$ is selected somewhat arbitrarily, with best-practises suggesting empirical an approach to determine good values. Research into analytical models for selecting embedding dimensionality is ongoing [**Yin2018a**; **Naumov2019**].

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 [**Hochreiter1997**]. LSTMs implements 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 [**Pacanu2013**]), 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 [**Lipton2015**], 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 [**Ding2015**; **Stephenson2005**]. When present, the values of

```
//#define Elements
kernel void memset_kernel(global char * mem_d, short val, int
↪ number_bytes){
  const int thread_id = get_global_id(0);
  mem_d[thread_id] = val;
}
```

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

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

(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 an OpenCL source code.

auxiliary inputs are concatenated with the output of the language model and fed into a heuristic model.

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 [**Ioffe2015a**] is used, in which each scalar of the heuristic model's n inputs $\boldsymbol{x}^{(1)},\ldots,\boldsymbol{x}^{(n)}$ is independently normalised to a mean 0 and standard deviation of 1:

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

Where $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ 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 $f(x)$ over its inputs. Rectifier activation functions $f(x) = \max(0,x)$ are used in the first layer due to their improved performance during the training of deep networks [**Nair2010**]. For the output layer, sigmoid activation functions $f(x) = \frac{1}{1+e^{-x}}$ 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 [**Kingma2015**]. For training data $X^{(1)}, \ldots, X^{(n)}$, SGD attempts to find the model parameters Θ that minimise the output of a loss function:

$$\Theta = \arg\min_{\Theta} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\left(\boldsymbol{X}^{(i)}, \Theta\right) \tag{6.2}$$

where loss function $\mathcal{L}(x, \Theta)$ 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` $\times$ `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

We return to the **Grewe2013** [**Grewe2013**] predictive model of Chapter 4 for mapping OpenCL kernels to the optimal device in CPU/GPU heterogeneous systems. They 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)×wgsize`	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 **Grewe2013** to predict heterogeneous device mappings for OpenCL kernels.

	Version	#. benchmarks	#. kernels
NPB (SNU [Seo2011])	1.0.3	7	114
Rodinia [Che2009]	3.1	14	31
NVIDIA SDK	4.2	6	12
AMD SDK	3.0	12	16
Parboil [Stratton2012]	0.2	6	8
PolyBench [Grauer-Gray2012]	1.0	14	27
SHOC [Danalis2010]	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 **Grewe2013** [**Grewe2013**] 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 were 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 [**Han2011**]. 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.

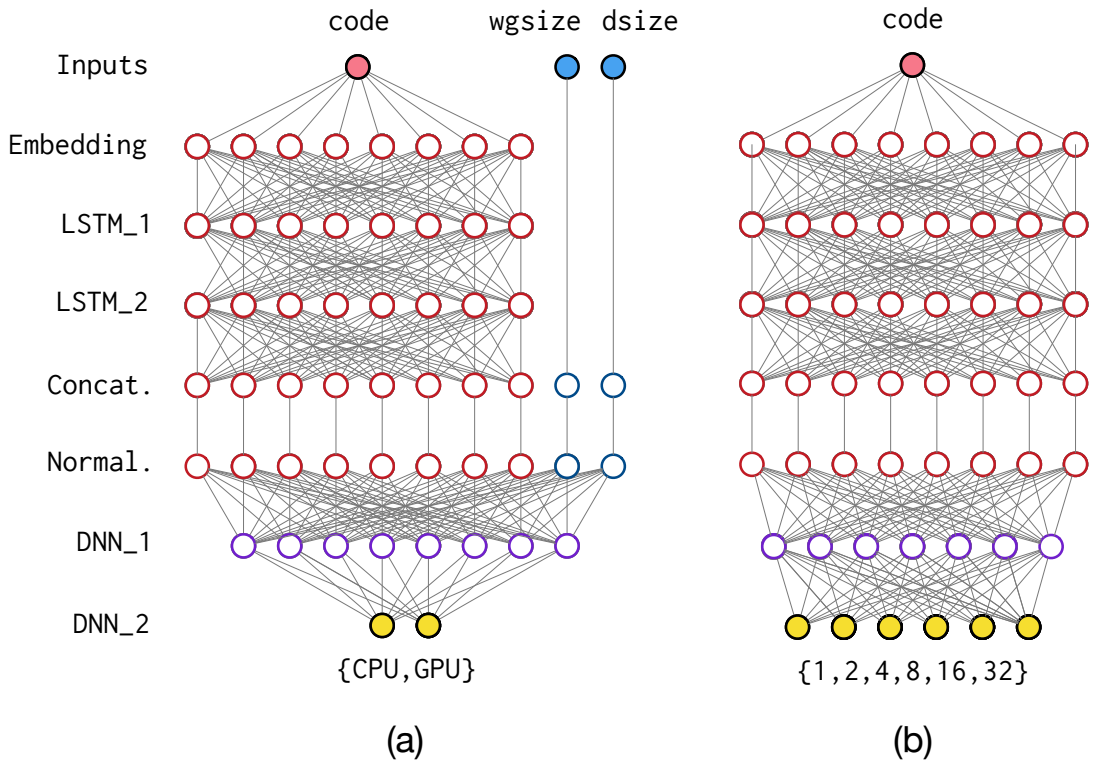


Figure 6.4: DeepTune artificial neural networks, configured for (a) heterogeneous mapping, and (b) thread coarsening factor. The design stays almost the same regardless of the optimisation problem. The only changes are the extra input for (a) and size of the output layers. To aid in visualisation, the 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 **Grewe2013** 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 **Grewe2013** model achieves an average accuracy of 73%, a significant improvement over the static mapping. By au-

tomatically 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 **Grewe2013** 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× on AMD and 1.26× on NVIDIA (geometric mean 1.18×). In 90% of cases, DeepTune matches or outperforms the predictions of the Grewe *et al.* model, achieving an average speedup of 3.34× on AMD and 1.41× on NVIDIA (geometric mean 1.31×). 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. This optimisation can 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

Magni2014 present a predictive model for OpenCL thread coarsening in [**Magni2014**]. They implement an iterative heuristic which determines whether a given program would benefit from coarsening. If yes, then the program is coarsened, and the process repeats, 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 **Magni2014** followed a very comprehensive feature engineering process. 17 candidate features were assembled from previous studies of performance counters and computed theoretical values [**Magni2**; **Sim2012**]. For each candidate feature, they compute its coarsening *delta*, reflecting the change in each feature value caused by coarsening: $f_\Delta = (f_{after} - f_{before})/f_{before}$, adding it to the feature

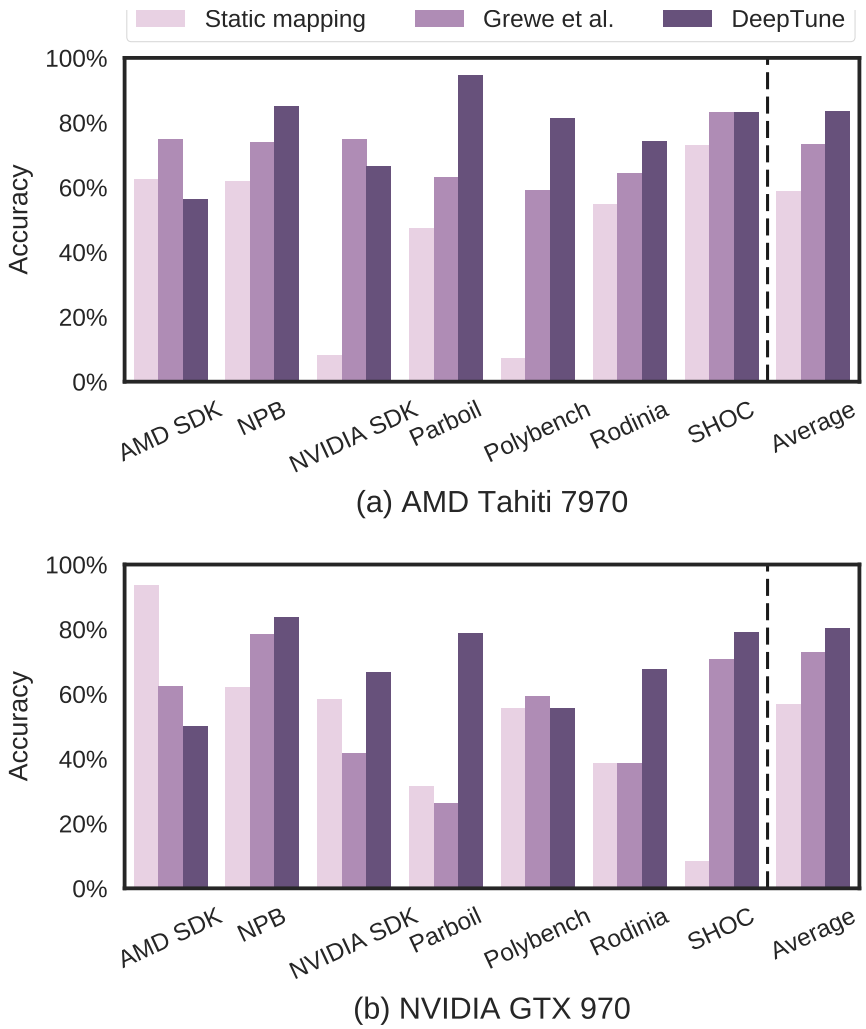
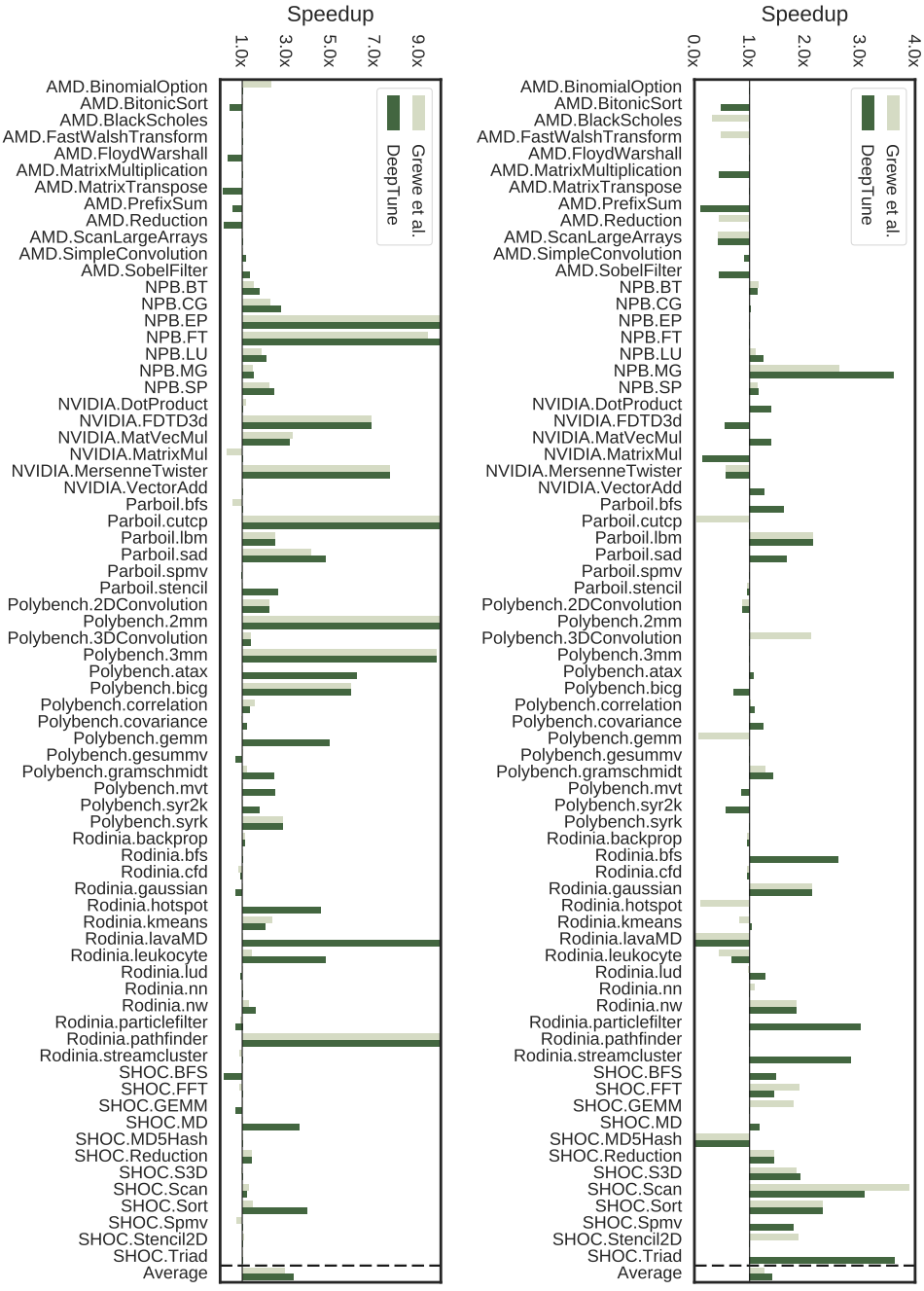
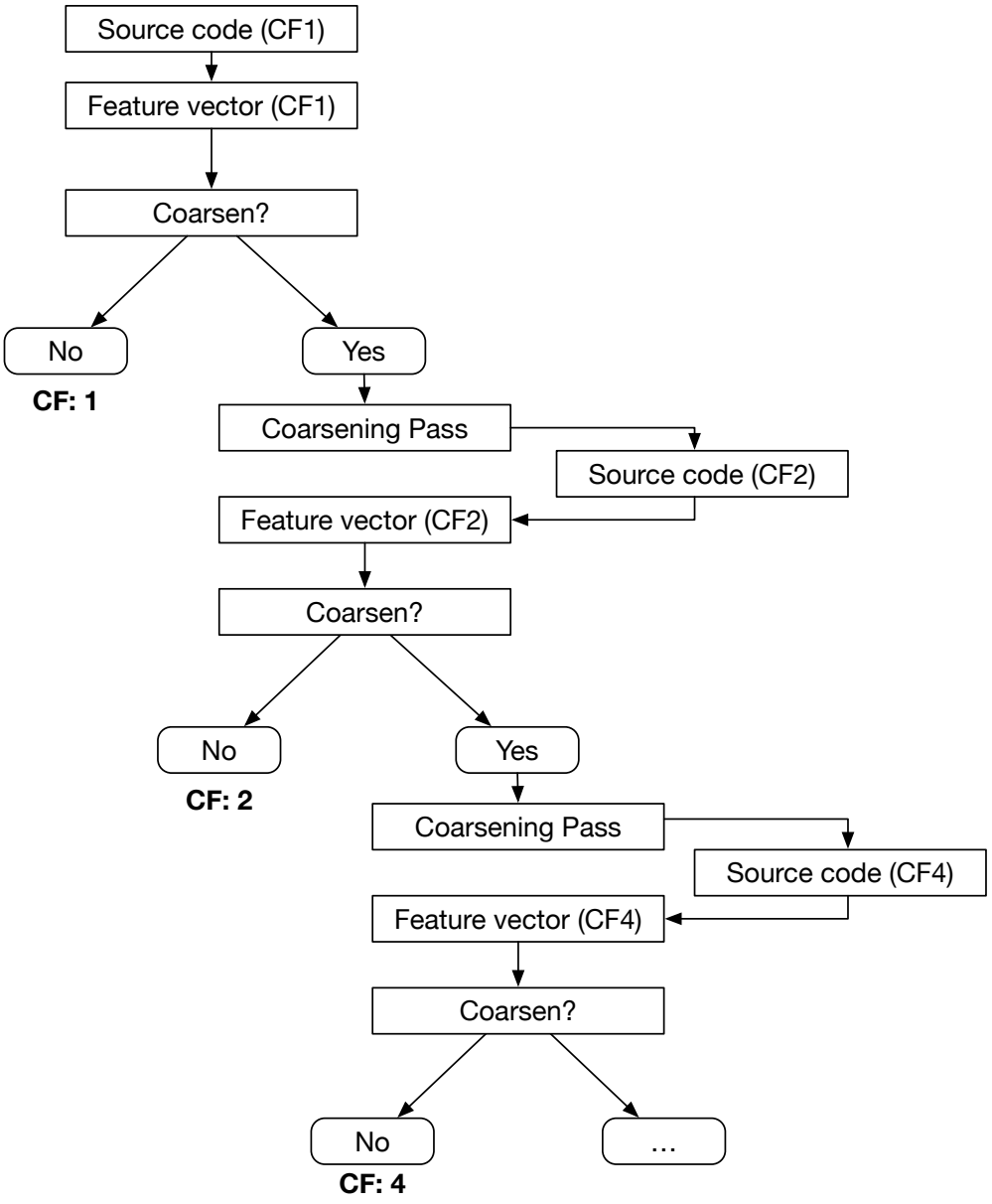


Figure 6.5: Accuracy of optimisation heuristics for heterogeneous device mapping, aggregated by benchmark suite. The optimal static mapping achieves 58% accuracy. The **Grewe2013** and DeepTune predictive models achieve accuracies of 73% and 84%, respectively.

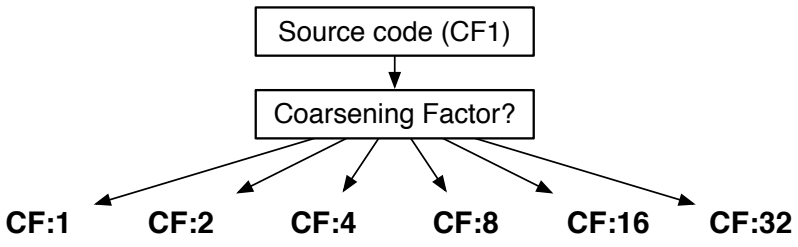


(a) AMD Tahiti 7970 (b) NVIDIA GTX 970

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 **Grewe2013**. In (b), the speedup is 1.42x and 13% respectively.



(a) **Magni2014** cascading binary model.



(b) Proposed approach.

Figure 6.7: Two approaches for predicting coarsening factor (CF) of OpenCL kernels. **Magni2014** reduce the multi-label classification problem to a series of binary decisions, by iteratively applying the optimisation and computing new feature vectors. Our 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
FPInsts	#. 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 **Magni2014** 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 [Stratton2012]	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.

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 **Magni2014** [**Magni2014**] 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 the coarsening factor is the predicted output.

Model Evaluation Compared to Case Study A, the size of the evaluation is small. We use *leave-one-out cross-validation* 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.

The parameters of the artificial neural network are not described in [**Magni2014**], 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.

	#. neurons		#. parameters	
	HM	**CF**	**HM**	**CF**
Embedding	64	64	,256	8,256
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			76,778	76,838

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).

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 **Magni2014** 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 **Magni2014** model achieves average speedups of 1.21×, 1.01×, 0.86×, and 0.94×, respectively. DeepTune outperforms this, achieving speedups of 1.10×, 1.05×, 1.10×, and 0.99×.

Some programs — especially those with large divergent regions or indirect memory accesses — respond very 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 alternate 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 in **Grewe2013** and **Magni2014**. 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 artificial neural network classification, information learned at the early 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 [**Zeiler2014**].

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 hetero-

geneous 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 weights 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 **Magni2014**. 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.

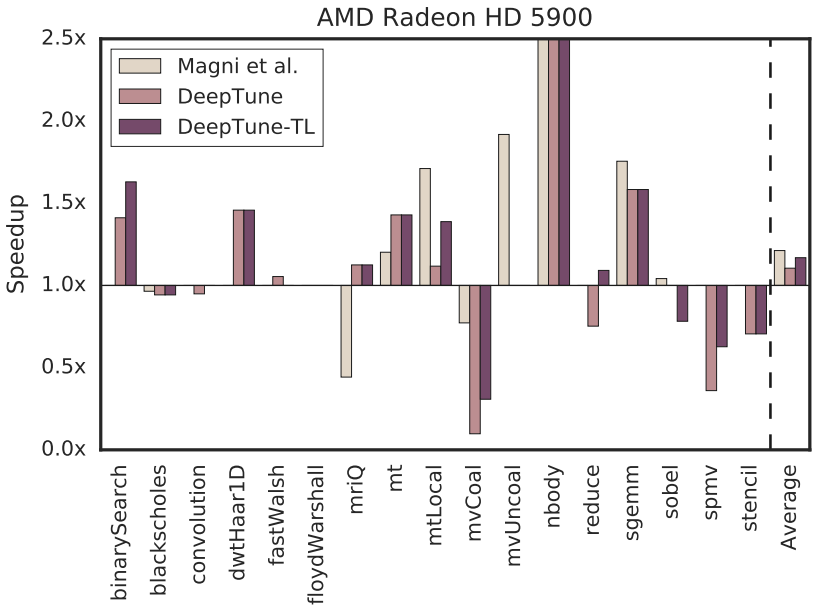
On the NVIDIA Tesla K20c, the platform for which no predictive model achieves positive average speedups, DeepTune-TL matches or improve performance in the majority of cases, but over-coarsening on three of the programs causes a modest reduction in average performance. For this platform, further performance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

6.6 DeepTune Internal Activation States

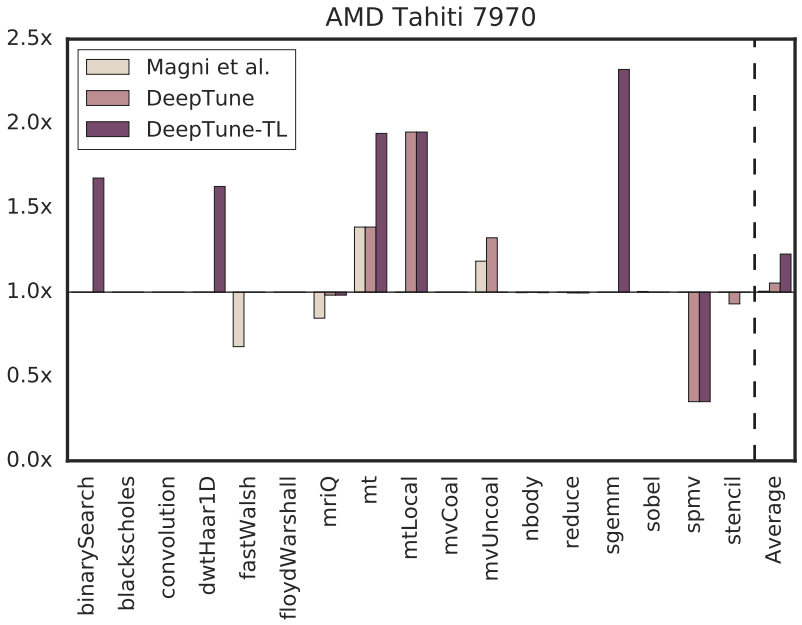
In previous sections, DeepTune is shown to automatically outperform state-of-the-art predictive models for which experts have invested a great amount of time in engineering features. This section attempts to illuminate the inner workings, using a single example from Case Study B: predicting the thread 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 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 clus-

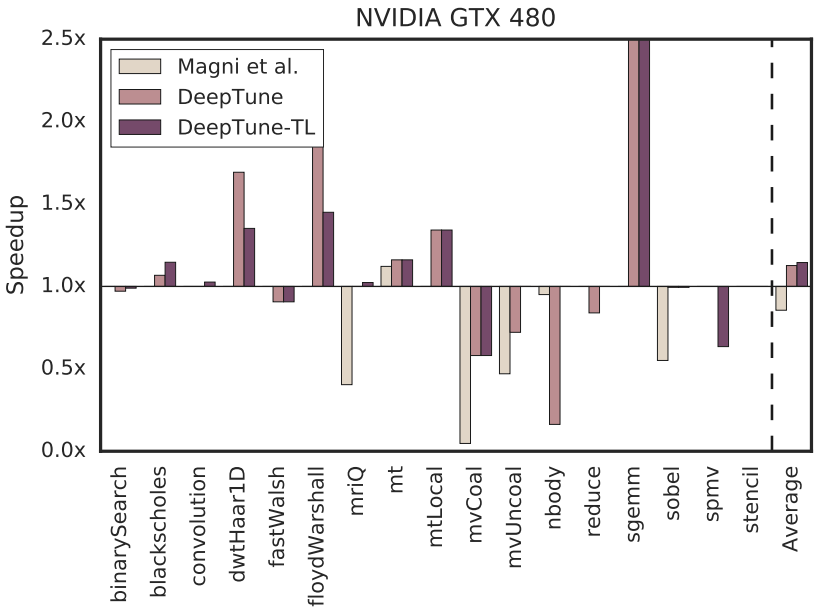


(a)

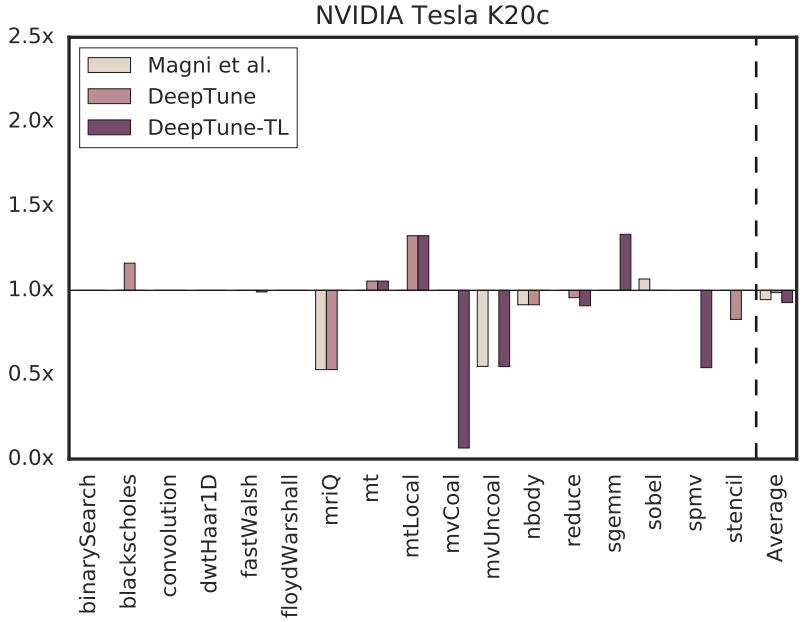


(b)

Figure 6.8: Speedups of predicted coarsening factors on AMD platforms. DeepTune outperforms **Magni2014** on three of the four platforms. Transfer learning improves DeepTune speedups further, by 16% on average.



(a)



(b)

Figure 6.9: Speedups of predicted coarsening factors on NVIDIA platforms. DeepTune outperforms **Magni2014** on three of the four platforms. Transfer learning improves DeepTune speedups further, by 16% on average.

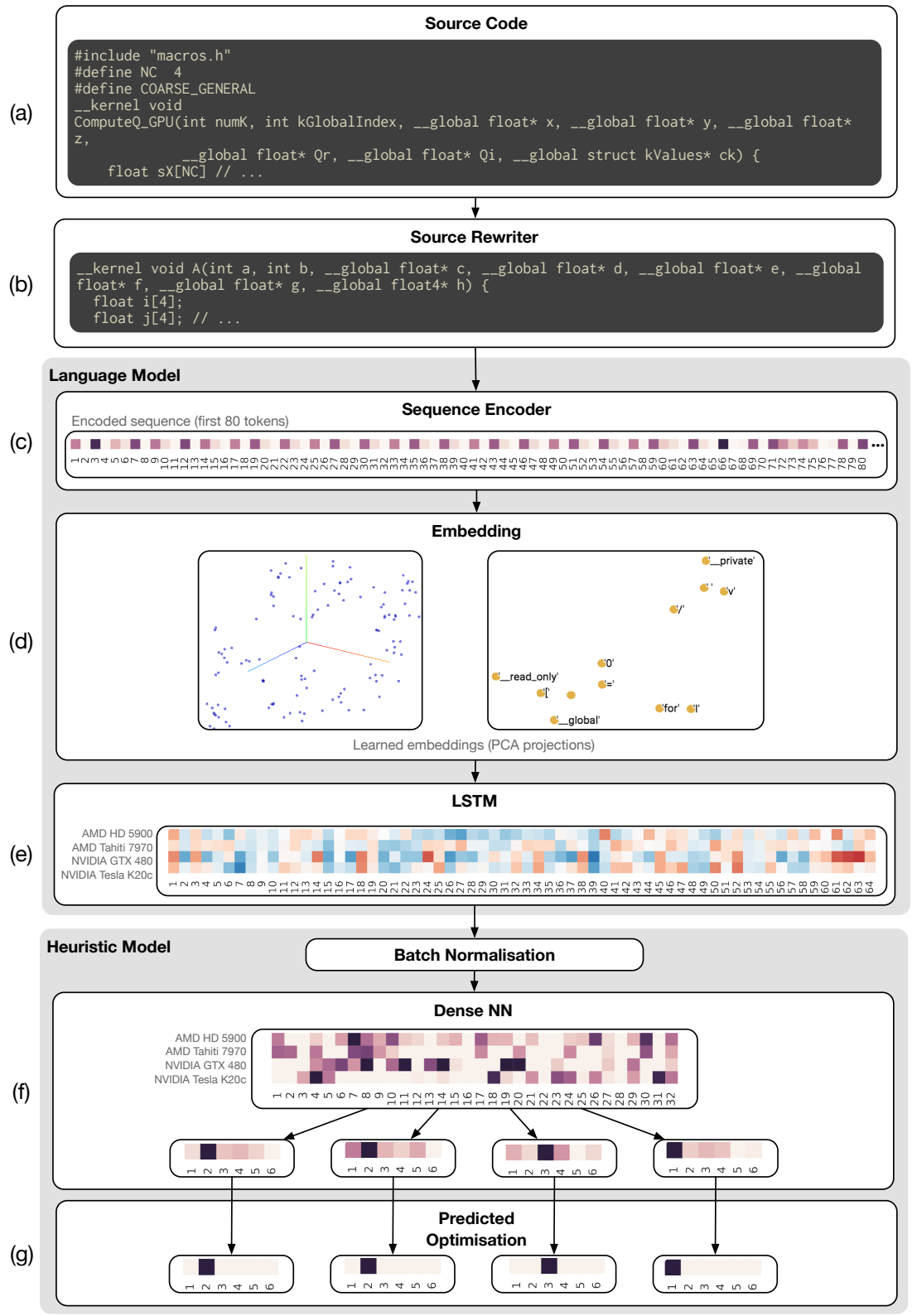


Figure 6.10: Visualising the internal state of DeepTune when predicting coarsening factor for Parboil's `mriQ` benchmark on four different architectures. The activations in each layer of the four models increasingly diverge the lower down the network.

ters of tokens. Although achieving good separation of embedding clusters typically requires a much larger training set [**Ben-nun2018**], 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, we note 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 [**Ben-nun2018**] 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 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 more simple 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 machine learning, to address practical issues in compiler construction. In particular, Chapter 4 addresses the benchmark scarcity problem by developing a methodology for the unguided generation of realistic benchmarks. Chapter 5 extends this generative technique to the domain of compiler validation, developing an effective compiler fuzzer that is significantly simpler than state-of-the-art approaches. Finally, Chapter 6 explores a technique that removes the need to manually construct features for programs.

Section 7.1 summarises the main contributions of this thesis, Section 7.2 presents a critical analysis of this work, and future work is described in Section 7.3.

7.1 Contributions

The problems addressed in this thesis are well-established. The main contributions of this thesis with respect to these problems are as follows.

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. 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 [**Chiu2015**], 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 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. The technique may also prove valuable to compiler feature designers, as the granular exploration of the feature space may expose deficiencies in the choice of features.

7.1.2 Low-cost and Effective Compiler Fuzzing

Chapter 5 extends the application of recurrent neural networks to the domain of compiler testing. The state-of-the-art in compiler test case generation is a significant undertaking, comprising over 50,000 lines of handwritten code [**Yang2011**; **Lidbury2015a**]. The technique presented in this thesis presents an enormous reduction in developer effort compared to the state-of-the-art grammar-based approach. It can be 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.

The portability of the approach is demonstrated by extending the generator from OpenCL to Solidity in only 12 lines of code. By contrast, extending a state-of-the-art generator from C to OpenCL required over 8000 lines of code [**Lidbury2015a**].

Despite its simplicity, the proposed technique is effective. To date, 67 new bugs have been identified and reported in OpenCL compilers. Many of the bugs identified could not be exposed by state-of-the-art approaches due to the limitations in the expressiveness of 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.

7.1.3 Automatic Compiler Optimisation Tuning

Constructing program features for machine learning 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, the source code of the entire program is fed directly into the learning system. This is a simpler approach that decouples the learning system from the compiler's internal representation.

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× and 1.05×. 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×, 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. 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 identified 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 can be 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 [**Dyer2016**]. 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 the 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 can be used to partially characterise 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 alternate methodology for generating inputs must be adopted.

7.2.5 Modelling Program Semantics as 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 $p(y|[\boldsymbol{x}^{(1)},\boldsymbol{x}^{(2)},\boldsymbol{x}^{(3)}]) \neq p(y|[\boldsymbol{x}^{(3)},\boldsymbol{x}^{(2)},\boldsymbol{x}^{(1)}])$. 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 order 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 analysis of failure cases.

7.3 Future Work

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

7.3.1 Guided Program Synthesis to Minimise Benchmarking Costs

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 can 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 can thus be 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.

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 must be 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 [**Gad2014**].

A Generative Adversarial Network (GAN) [**Goodfellow2014**] uses a similar adversarial approach, but using a second artificial neural network as a discriminative adversary. The generator and discriminator networks are trained concurrently; the generator is trained to maximize the probability of failure in the discriminator, the discriminator is trained to minimize this probability. This approach is a good fit for the domain of program generation, but filtering of the generator outputs to check for program correctness may be required to prevent the generator training to a local maxima which is hard to distinguish from the test set, but rarely contains meaningful program semantics.

7.3.4 Towards General-Purpose Program Comprehension

The techniques presented in this thesis apply recurrent neural networks to the task of modelling the syntax and semantics of programming languages. 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 which can 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 to no 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 can start by re-training an existing model such as ResNet [**He2016**], 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.