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

Compilers are a fundamental technology, yet constructing them is difficult and time consuming. A modern optimising compiler comprises thousands of components and millions of lines of code, with the demands of data intensive workloads requiring ever-more aggressive optimisations. In addition, the rapid transition to heterogeneous parallelism requires compilers to support a diverse range of hardware, which has left compiler developers struggling to keep up. The cost of this is software that has poor performance and more bugs. What is needed are techniques that radically reduce the cost of compiler construction.

This thesis presents deep learning methodologies to simplify compiler construction. First, a generative model for source code is developed, capable of producing executable programs derived from language models trained on open source corpora. Unlike prior approaches, the generative model presented in this thesis is inferred entirely from example code, greatly reducing the cost of development. It requires no grammar or prior knowledge of the language being modelled, yet is capable of producing code of such quality that professional software developers cannot distinguish generated from handwritten. Secondly, this thesis explores the use of recurrent neural networks for code comprehension through learning optimisation heuristics directly on raw source code.

The effectiveness of programs generated using this approach is investigated in two orthogonal domains. In the first, generated programs are used as benchmarks to supplement the training data of predictive models for compiler optimisations. The additional fine-grained exploration of the feature space that training on an additional 1000 generated programs provides yields a $1.27\times$ speedup of a state-of-the-art predictive model. In addition, the extra information automatically exposes weaknesses in the feature design which, when corrected, yields a further $4.30\times$ improvement in performance.

The second domain for which automatic program generation is applied is compiler validation. The generative model is extended and used to enable compiler fuzzing. Compared to a state-of-the-art fuzzer, the proposed approach presents an enormous reduction in developer effort, requiring $100\times$ fewer lines of code to implement, and is capable of generating an expressive range of tests that expose bugs that the state-of-the-art cannot. In a testing campaign of 10 OpenCL compilers, 67 new bugs are identified and reported, many of which are now fixed.

Finally, this thesis presents a new methodology for constructing compiler heuristics, which significantly reduces the cost of applying machine learning to compiler heuristics. Unlike state-of-the-art approaches in which program features have to be expertly engineered and selected, the proposed approach uses recurrent neural networks which learn directly over the textual representation of program code. Doing so yields 1.14× and 1.12× performance improvements in state-of-the-art predictive models. Additionally, by using the same neural network structure for different optimisation problems, this enables the novel transfer of information between optimisation problems.

Lay Summary

Crisis, solution, happiness

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*)

Dedication placeholder.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Listings

Chapter 1

Introduction

There has been an unprecedented increase in the scale and quantity of data intensive workloads. To meet the demands of the transition to *big data*, there have been fundamental shifts in both hardware and software. In hardware, the demand for performance has long outstripped what can be provided by single processors, leading to a broad spectrum of heterogeneous architectures being developed. These range from repurposing existing Graphics Processor Units (GPUs) to offload numeric computations, to adapting Field-programmable Gate Arrays (FPGAs), and even developing highly specialised ASICs to perform numeric tasks [Jou+17; MS10]). While much of the compiler logic can be reused across architectures, the optimisations that are required 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 is significantly more challenging than traditional single threaded development; there are many more opportunities to introduce bugs in software, and Amdahl's law can make extracting performance much more challenging. 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 the 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. One example is high level libraries for performing data intensive numeric workloads, two of the most popular examples of which — TensorFlow [Aba+16] and PyTorch [Pas+17] — use opposing dataflow and imperative programming styles, respectively. Optimising such libraries provides new challenges to the compiler — the challenge of code analyses in the face of highly parallel abstract code can defeat many optimisations.

The combined burden of increased hardware and software diversity has resulted in compilers that are too complex for the expert to fully reason about, and too large to keep up with the pace of change. This results in low performance, wasted energy, and buggy software. For the trend towards larger workloads and heterogeneous devices to continue, new techniques are required to reduce the cost of compiler construction.

1.1 Machine Learning for Compilers

Machine learning has been successfully applied across a broad range of fields and disciplines. In recent years this has been accelerated by the development of deep learning techniques. 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.

Within compilers, there are many tasks which require domain expertise that are eligible candidates for machine learning. Examples of which include learning models to control optimisation heuristics, and generating representative inputs to differential test the compiler. As such, the use of machine learning to aid in compiler construction is an established research pursuit. In many 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 to compilers has remained a largely academic pursuit, with little progress being made of adoption within industry. The following section speculates as to the cause by summarising some of the outstanding problems in applying machine learning to compilers. The remainder of this chapter then details the contributions of this thesis, followed by the overall structure of the document.

1.2 The Problem

Machine learning techniques can 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 techniques, a model is trained based on past observations to predict the values for unseen data points. In order to be able to generalise well to unseen points, plentiful training data should be provided, with a fine-grained overview of the feature space. Typical machine learning experiments outside of the compilation field train over thousands or millions of observations. In machine learning for compilers, however, there are typically only a few dozen common benchmarks available.

The small number of common benchmarks limits the quality of learned models as they have very sparse training data. The problem is worsened exponentially with the dimensionality of the feature space. Complex heuristics often have high-dimensional feature spaces, and each additional dimension worsens the sparsity of training examples.

To address the issue, there must be a sizeable increase in the set of common benchmarks, or programs to perform machine learning over. Previously, researchers have sought to provide these through the use of random grammar based generation of programs, but this is a challenging task — the generated programs must be similar to that of real programs so as to be useful to the learning algorithm, and biasing the generation towards these types of programs is hard to do in the general case.

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 setup.

Many problems in compilers do not map directly to numeric attributes, requiring the extraction of a numeric representation from a non-numeric input. For example, extracting instruction counts from source code. Knowing which attributes to extract to represent a program is not easy, and developers are typically faced with the challenge of having to laborious select the right combination features from a large candidate set.

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 reduce the cost of compiler construction through developing *low cost* machine learning techniques to build and maintain compilers.

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 improve the performance of a state-of-the-art predictive model by $1.27\times$, and expose limitations in the feature design of the model which, after correcting, further increases performance by $4.30\times$.

- 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. 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 any need for 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 defines terminology and describes the methodologies and techniques used in this thesis.

Chapter 3 provides an overview of previous work on machine learning for compilers, with an emphasis on performance optimisation.

Chapter 4 describes a machine learning generator for source codes. The generator is evaluated for its ability to produce OpenCL benchmarks.

Chapter 4 introduces a novel machine learning generator for source codes. The generator is evaluated for its ability to produce OpenCL benchmarks.

Chapter 5 presents an application of the machine learning generator for synthesising compiler test cases. The chapter contains an extensive evaluation of OpenCL compilers using the synthesised test cases.

Chapter 6 introduces a novel methodology for constructing optimising compiler heuristics without the need for code features.

Chapter 7 summarises the findings and describes potential avenues for future research.

1.5 Summary

This introductory chapter has outlined the use of machine learning for compilers and two main issues preventing its widespread uptake: the scarcity of data, and the challenge of designing features. Subsequent chapters describe novel approaches to address both issues.

Chapter 2

Background

2.1 Introduction

This chapter provides a short and non-exhaustive overview of the theory and techniques used in this thesis.

First Section 2.2 defines the terminology. Then Section 2.3 describes the statistical tools and methodologies used in this thesis. This is followed by Section 2.4 providing an overview of the machine learning techniques used in this thesis. Section 2.5 provides an overview of GPGPU programming. Finally Section 2.6 concludes.

This entire chapter has to be writte
All current is placehold

2.2 Terminology

machine learning

deep learning

feature space

supervised learning

unsupervised learning

2.3 Evaluation Techniques

2.3.1 Principal Component Analysis

2.4 Machine Learning

2.4.1 Decision Trees

2.4.2 Deep Learning

2.4.3 Recurrent Neural Networks

A recurrent neural network (RNN) consists of hidden states **h** an an optional output **y**. It operates on a variable-length sequence, $\boldsymbol{x} = (x_1, x_2, \ldots, x_T)$. At each step $t \in T$, the hideen state $h_{<t>}$ of the RNN is updated by:

$$h_{<t>} = f\left(h_{<t-1>}, x_t\right)$$

where f is a non-linear activation function. An RNN can learn a probabilty distribution over a sequence of tokens to predict the next token. Therefore, at each timestep t, the output from the RNN is a conditional distribution $p\left(x_t | x_1, \ldots, x_{t-1}\right)$.

RNN evaluation [DDD18].

2.4.3.1 Long Short-Term Memory

LSTM variants review [Gre+15a].

$\boldsymbol{x}^t$ is the input vector at time t; $\boldsymbol{W}$ are input weight matrices; $\boldsymbol{R}$ are recurrent weight matrices; $\boldsymbol{p}$ are peephole weight vectors; $\boldsymbol{b}$ are bias vectors; functions g, h, and σ are point-wise nonlinear activation functions.

block input:

$$\boldsymbol{z}^t = g\left(\boldsymbol{W}_z \boldsymbol{x}^t + \boldsymbol{R}_z \boldsymbol{y}^{t-1} + \boldsymbol{b}_z\right)$$

input gate:

$$\boldsymbol{i}^t = \sigma\left(\boldsymbol{W}_i \boldsymbol{x}^t + \boldsymbol{R}_i \boldsymbol{y}^{t-1} + \boldsymbol{p}_i \odot c^{t-1} + \boldsymbol{b}_i\right)$$

forget gate:

$$\boldsymbol{f}^t = \sigma\left(\boldsymbol{W}_f \boldsymbol{x}^t + \boldsymbol{R}_f \boldsymbol{y}^{t-1} + \boldsymbol{p}_f \odot c^{t-1} + \boldsymbol{b}_f\right)$$

cell state:

$$\boldsymbol{c}^t = \boldsymbol{i}^t \odot \boldsymbol{z}^t + \boldsymbol{f}^t \odot \boldsymbol{c}^{t-1}$$

output gate:

$$\boldsymbol{o}^t = \sigma\left(\boldsymbol{W}_i \boldsymbol{x}^t + \boldsymbol{R}_o \boldsymbol{y}^{t-1} + \boldsymbol{p}_o \odot c^{t-1} + \boldsymbol{b}_o\right)$$

block output:

$$\boldsymbol{y}^t = \boldsymbol{o}^t \odot h(\boldsymbol{c}^t)$$

Number of params =

...

The Generative Adversarial Network (GAN) is a means to estimate a generative model [Goo+14]. It uses an adversarial process in which two models are simultaneously trained: a generator model G that captures the data distribution, and a discriminative model D that estimates the probability that a sample came from the training data rather than G. The training procedure for G is to maximise the probability of D making a mistake.

If both models are neural networks: learn a generator's distribution p_g over data $\boldsymbol{x}$. Define a prior on input noise variables $p_z(\boldsymbol{z})$. Generator $G(\boldsymbol{z};\Theta_g)$, using parameters Θ_g. Discriminator $D(\boldsymbol{x};\Theta_d)$ outputs a scalar, the probability that $\boldsymbol{x}$ came from the data rather than p_g.

Simultaneously train D to maximise the probability of assigning the correct label to both training examples and samples from G, and train G to minimise $\log(1-D(G(\boldsymbol{z})))$. D and G play the two-player minimax game with value function $V(G,D)$:

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{data}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{z\sim p_z(\boldsymbol{z})}[\log(1-D(G(\boldsymbol{z})))]$$

Challenge: there may not be sufficient gradient for G to learn well. Early in learning, when G is poor, D can reject samples with high confidence because they are clearly different from the training data.

2.5 GPGPU Programming

General purpose programming with graphics hardware is a nascent field, but has shown to enable massive data parallel throughput by re-purposing the hardware traditionally dedicated to the rendering of 3D graphics for generic computation. This was enabled by hardware support for programmable shaders replacing the fixed function graphics pipeline, and support for floating point operations in 2001. Owens et al. provide a review of the first five years of general purpose computation on graphics hardware in [Owe+06].

In the ensuing progress towards increasingly programmable graphics hardware, two dominant programming models have emerged: CUDA and OpenCL, which both abstract the graphics primitives of GPU hardware and provide a platform for GPGPU programming. CUDA is a language developed by NVIDIA for programming their GPUs using a proprietary SDK and API [Nvi07], while OpenCL is a vendor-independent open standard based on a subset of the ISO C99 programming language, with implementations for devices from most major GPU manufactures [SGS10]. Quantitative evaluations of the performance of CUDA and OpenCL programs suggest that performance is comparable between the two systems, although the wider range of target architectures for OpenCL means that appropriate optimisations must be made by hand or by the compiler [FVS11; Kom+10].

2.5.1 The OpenCL Programming Model

OpenCL is a parallel programming framework which targets CPUs, GPUs, and other parallel processors such as Field-Programmable Gate Arrays. It provides a set of APIs and a language (based on an extended subset of C) for controlling heterogeneous *compute devices* from a central host. Programs written for these devices are called *kernels*, and are compiled by platform-specific tool chains. At runtime, an OpenCL *platform* is selected and a *context* object is created which exposes access to each supported compute device through *command queues*. When a kernel is executed, each unit of computation is referred to as a *work-item*, and these work-items are grouped into *work-groups*. The sum of all work-group dimensions defines the *global size*. For GPUs, work-groups execute on the Single Instruction Multiple Data (SIMD) processing units in lockstep. This is very different from the behaviour of traditional CPUs, and can cause severe performance penalties in the presence of flow control, as work-items must be stalled across divering flow paths.

2.5.1.1 Memory Model

Unlike the flat model of CPUs, OpenCL uses a hierarchical memory model, shown in Figure 2.1. The host and each OpenCL device has a single global memory address space. Each work-group has a local memory space, and each work-item has a region of private memory.

Work-groups cannot access the memory of neighbouring work-groups, nor can work-items access the private memory of other work-items. OpenCL provides syn-

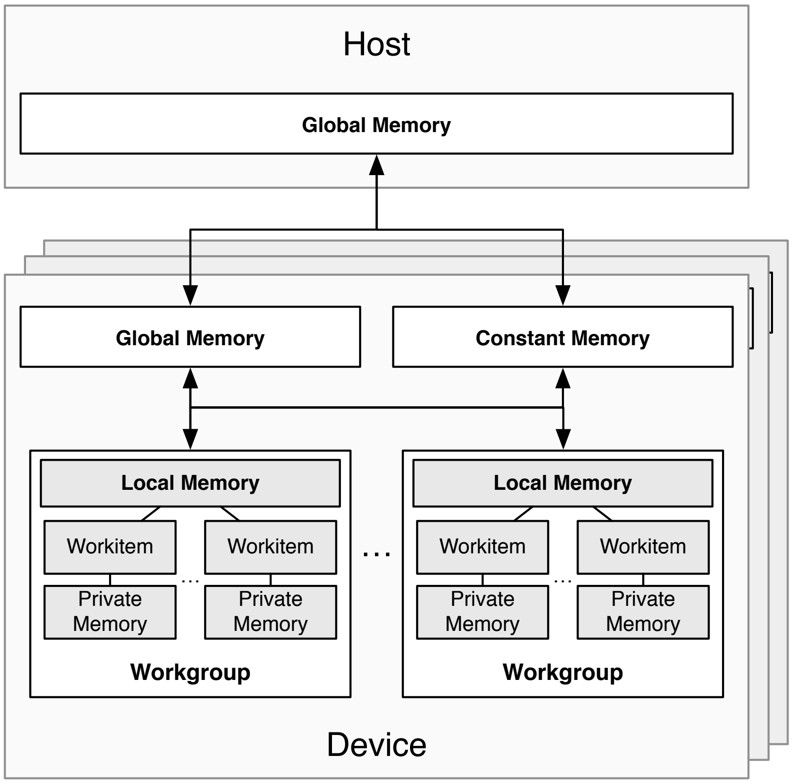


Figure 2.1: The OpenCL memory model. The host communicates with each device through transfers between global memory spaces. The capacity of each type of memory is dependent on the device hardware. In general, private memory is the fastest and smallest, and global memory is the largest and slowest.

chronisation barriers to allow for communication between work-items within a single work-group via the local memory, but not global barriers. Memory transfers between the host and devices occurs between global memory regions. In the case of programming heterogeneous devices, these transfers must occur over the connection bus between the CPU and device (e.g. PCIe for discrete GPUs), which typically creates a performance bottleneck by introducing a performance overhead to transfer data to the device for processing, then back to the device afterwards. Direct transfers of data between devices is not supported, requiring an intermediate transfer to the host memory.

2.5.1.2 Performance Optimisations

The wide range of supported execution devices and differing standards-compliant implementations makes portable performance tuning of OpenCL programs a difficult task [Rul+10], and the interactions between optimisations and the hardware are complex and sometimes counter-intuitive [Ryo+08b].

The overhead introduced by memory transfers between host and compute devices further complicates comparisons of OpenCL performance on different devices. The conclusion of [GH11] is that this overhead can account for a $2\times$ to $50\times$ difference of GPU program runtime. In [Lee+10], Lee et al. present a performance analysis of optimised throughput computing applications for GPUs and CPUs. Of the 14 applications tested, they found GPU performance to be $0.7\times$ to $14.9\times$ that of multi-threaded CPU code, with an average of only $2.5\times$. This is much lower than the $100\times$ to $1000\times$ values reported by other studies, a fact that they attribute to uneven comparison of optimised GPU code to unoptimised CPU code, or vice versa. Lee et al. found that multi-threading, cache blocking, reordering of memory accesses and use of SIMD instructions to contribute most to CPU performance. For GPUs, the most effective optimisations are reducing synchronisation costs, and exploiting local shared memory. In all cases, the programs were optimised and hand-tuned by programmers with expert knowledge of the target architectures. It is unclear whether their performance results still hold for subsequent generations of devices.

Despite the concerns of over-represented speedups, the potential for high performance coupled with the complexity and low levels of abstraction provided by OpenCL make it an ideal target for skeletal abstractions. SkelCL and SkePU are two such examples which add a layer of abstraction above OpenCL and CUDA respectively in order to simplify GPGPU programming [EK10].

2.6 Summary

Chapter 3

Related Work

3.1 Introduction

This chapter reviews research in areas relevant to this thesis.

The chapter is divided into two sections, covering the two themes of this thesis. Section 3.2 reviews the literature of program generation, focusing on compiler testing and benchmarking. Then Section 3.3 reviews the literature of program optimisation, starting with empirical techniques, iterative compilation, and machine learning. Finally Section 3.4 discusses related works that do not fall under either category, and 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 of two use cases that are relevant to this thesis: program generation for performance characterisation, and program generation for compiler validation.

3.2.1 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 create a diverse set of workloads that is both representative of the target real world use while providing an 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.

An evaluation of GPGPU benchmark suites reveals there are important parts of the program space were left untested [Ryo+15]. Goswami et al. evaluate 38 benchmark workloads, finding that *Similarity Score*, *Scan of Large Arrays*, and *Parallel Reduction* workloads show significantly different behaviour due to their large number of diverse kernels, but the remaining 35 workloads provide similar characteristics [Gos+10]. Xiong et al. demonstrate that workload behaviour is highly input dependent, and argue that benchmarks created for academic research cannot represent the cases of real world applications [Xio+13]. A review of big data benchmarks found many to be unrepresentative, and that current hardware designs, while optimized for existing benchmark suites, are inefficient for true workloads [Fer+12].

As well as covering the program space, benchmarks within suites should be diverse. Ould-Ahmed-Vall et al. show that statistical models trained on 10% of SPEC CPU 2006 data is transferable to the remaining data [Oul+08]. Phansalkar, Joshi, and John show that a subset of 14 SPEC CPU 2006 programs can yield most of the information of the entire suite [PJJ07], and Panda et al. find that SPEC CPU 2017 contains workloads that can be safely removed without degrading coverage of the program space [Pan+18].

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

A popular use of synthetic benchmark generation techniques is to aid microprocessor design. Joshi et al. use micro-architecture-independent characteristics such as basic block sizes and data footprint to summarize 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 [Jos+08]. *MicroProbe* uses feedback-directed micro-benchmark generation to perform a systematic energy characterisation of an processor [**Bertran2012**].

GENESIS [CGA15] is a language for generating synthetic training programs. The essence of the approach is to construct a probabilistic grammar with embedded semantic actions that defines 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 [Cum15; FE15; GA15]. 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.

Interesting recent developments in synthetic benchmarking have combined elements from feedback-directed test case synthesis (reviewed in the next section) with synthetic benchmarking for the purpose of generating *adversarial* benchmarks that expose performance issues in systems. Dhok and Ramanathan apply mutation techniques to an initial set of coverage-driven inputs to expose inefficiencies in loops [DR16]. *SlowFuzz* uses a resource-usage-guided evolutionary search to find inputs that expose poor algorithmic complexity that could be exploited by attackers to produce Denial-of-Service attacks [Pet+17]. 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 maximizes slowdown. *Singularity* uses an evolutionary search over the space of input *patterns* to find the input with worst case performance [Wei+18]. *PerfSyn* 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 [TPG18]. *PerfFuzz* uses feedback-directed program mutation to generate programs which maximise execution counts at program locations [Lem+18]. Pedrosa et al. 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 [Ped+18].

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, as presented in this thesis.

3.2.2 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, verifying the behaviour of a compiler is of utmost importance.

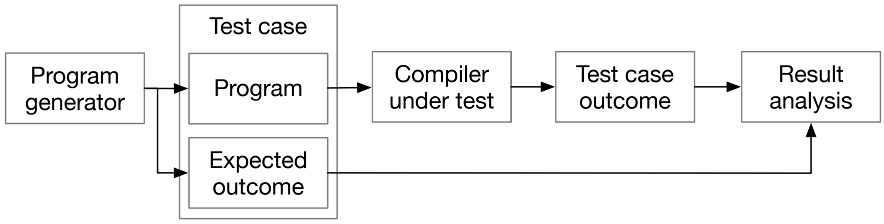
The complexity of optimizing compilers and programming languages renders formal verification of the entire compiler prohibitively expensive. Efforts have been made in this direction, for example CompCert [Ler17], a formally verified compiler for the C programming language, but this comes at the cost of supporting only a subset of the language features and with lower performance compared to GCC. Still, even CompCert is not fully verified, and errors have been discovered in the unverified components of it [Yan+11].

Because of the difficulties of *verification*, compilers developers turn to *validation*, in which the behaviour of a compiler is validated using a set of 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 for 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. 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.

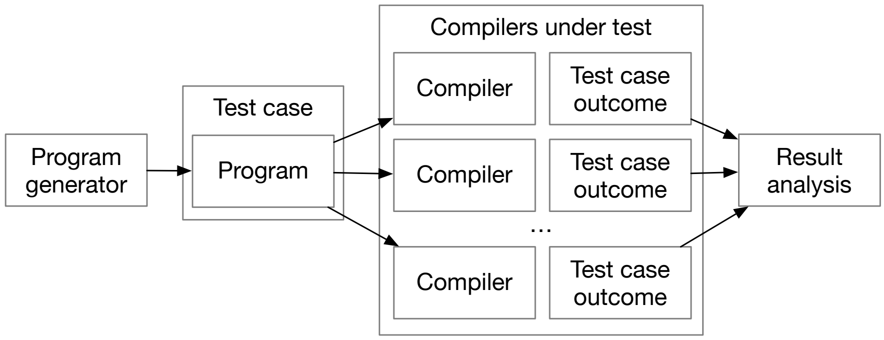
The random generation of programs to test compilers is a well established approach to the compiler validation problem. The main question of interest is in how to efficiently generate codes which trigger bugs. There are two main approaches: *program generation*, where inputs are synthesised from scratch; 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**, which randomly enumerates a 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 [**Hanford1970a**].



(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 comprises a program and its expected outcome. In (b), only a program is required, and the expected outcome is determined by majority voting on the observed outcomes across multiple compilers.

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 conforment compiler. The compiled program can then be executed and its output compared against this gold standard. Figure 3.1a shows the process. The challenge of this approach is in generating the gold standard output. These early approaches are surveyed by Boujarwah and Saleh [BS97] and Kossatchev and Posypkin [KP05].

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. Lacking a gold standard for behaviour makes differential testing less sound than an oracle approach, though in practise 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 compilers, or using the same compiler with optimizations on or off (or a combination of the two). Chen et al. empirically contrasts the two methodologies in [Che+16], along with a comparison to Equivalence Module Inputs (described in the next subsection).

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

CSmith [Yan+11] is a widely known and effective generator which enumerates programs by pairing infrequently combined language features. In doing so, it produces correct programs with clearly defined behaviour but 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. Lidbury et al. extend CSmith to the OpenCL programming language, a superficially simple task, yet this required 8 man-months of development and 8000 lines of code [Lid+15]. Subsequent generators influenced by CSmith, like Orange3 [NHI13], focus on features and bug types beyond the scope of CSmith, arithmetic bugs in the case of Orange3.

Glade [Bas+17] derives a grammar from a corpus of example programs. The derived grammar is enumerated to produce new programs, though no distribution is learned over the grammar; program 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 [Reg+12] is required to minimise the size of the program and isolate the code of interest. Automated test case reduction does not scale to the rate at which effective compilers find bug; sometimes taking hours for a single test case. An alternate method to generating test cases is to instead mutate a seed input.

3.2.2.2 Mutation and Feedback-directed Testing

Equivalence Modulo Inputs (EMI) testing, introduced by Le, Afshari, and Su [LAS14] follows a different approach to test case generation. Starting with existing code, it inserts or deletes statements that will not be executed so that the functionality of the code is unchanged. If it is affected, it is due to a compiler bug. *Hermes* extends the initial EMI approach to permit the mutation of *live code* regions, not just dead code [SLS16]. This greatly increases the expressiveness of the generated programs.

LangFuzz 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 [HHZ12]. Starting with a coverage-guided set of inputs, *T-Fuzz* uses dynamic tracing to detect input checks in programs and selectively removes them to expose defects [PSP18]. *Skeletal program enumeration* again works by transforming existing code. It identifies algorithmic patterns in short pieces of code and enumerates all the possible permutations of variable usage [ZSS17]. *pFuzzer* targets input parsers, using dynamic tainting to produce a set of legal inputs that cover all conditions during parsing [Mat+19]. Coverage-directed mutation techniques have been for differential testing the Java Virtual Machine [**Chen2016b**].

Machine learning has been used to guide test case mutation. Cheng et al. construct neural networks to discover correlation between PDF test case and execution in the target program. The correlations are then leveraged to generate new paths in the target program [Che+19]. *NEUZZ* learns a differentiable neural approximation of target program logic, then uses Stochastic Gradient Descent to guide program mutation [She+18]. *Skyfire* learns a probabilistic context-sensitive grammar over a corpus of programs to generate input seeds for mutation testing [Wan+17]. The generated seeds are shown to improve the code coverage of AFL [Zal] 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, it often still requires an external code generator. Similarly to CSmith, these methods 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. Many methods based are based on the astonishing success of Recurrent Neural Networks (RNNs) at modelling sequential data [Joz+16]. RNNs have been successfully applied to a variety of other generative tasks, including image captioning [Vin+15], colourising black and white photographs [ZIE16], artistic style [GEB15], and image generation [Gre+15b].

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

Although nascent, the use of neural networks to generate programs is evolving rapidly. *Neural Programmer* was an early example at program generation through the latent representation of a neural network [NLS16]. Most similar to the work presented in this thesis is *Learn&fuzz*, in which an LSTM network is trained over a corpus of PDF files to generate test inputs for the Microsoft Edge renderer, yielding one bug [GPS17]. Unlike compiler testing, PDF test cases require no inputs and no pre-processing of the training corpus. *DeepFuzz* uses a sequence-to-sequence model to generate C programs, uncovering 8 bugs in GCC. Unlike the technique presented in this work, sampling occurs only occurs on white-space so as to maximise the probability of a generated program being syntactically correct, though this does not address the issue of semantic correctness [Liu+19]. *IUST DeepFuzz* is a neural file generator for file format fuzzing, trained and evaluated on a corpus of PDF files [**Nasrabadi2018**]. Brockschmidt et al. present a novel methodology for program generation in which a graph is used as the intermediate representation [Bro+18].

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

3.3 Program Optimisation

Modern compilers are complex, typically containing dozens or hundreds of 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 achieving ideal performance.

Extracting maximum performance in a compiler is not as simple as enabling more optimisations, but in identifying which, out of a set of candidate optimisations, will provide the best performance for the current case. A recent study by Georgiou et al. illustrates the scale of the challenge. Taking two modern releases of LLVM, an industry-standard compiler, they obtained an average 3.9% performance improvement across 71 benchmarks on embedded processors by selectively *disabling* optimisations enabled at the standard `-O2` optimisation level [Geo+18]. Selecting the right optimisations is critical. In some domains, the margin of performance to be gained is significant. For example, Ryoo et al. find speedups of up to 432× through the appropriate selection and use of tiling and loop unrolling optimisations on a GPU matrix multiplication implementation [Ryo+08a].

Given the challenges of heuristics and analytical methods to extract performance, researchers have turned to empirical methods. This section reviews proposed approaches for two popular empirical approaches, iterative compilation and auto-tuning.

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 different configurations of optimisations in order 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 Bodin et al., the technique was initially demonstrated to find good configurations in the non-linear three-dimensional optimisation space of a matrix multiplication benchmark [Bod+98]. By exhaustively enumerating the optimisation space they were able to find the global minima of the cost function; however, the authors state that in practise 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** found iterative compilation to yield speedups in GCC over the highest optimisation level (`-O3`) of up 2.23× [**Chen2010**].

Frameworks for iterative compilation offer mechanisms to abstract the iterative compilation process from the optimisation space. This can lower the cost to adopting iterative compilation techniques by reusing the logic to search optimisation spaces. Examples include *OpenTuner* [Ans+13] which provides ensemble search techniques and *CLTune* [NC15] for tuning OpenCL kernels.

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

3.3.1.1 Pruning the Iterative Compilation Search Space

Triantafyllis and August proposes using feedback during the evaluation of configurations to prune the optimisation space [TA03]. This is combined with a fast static performance estimator to obviate the need to run each configuration of a program. Pan and Eigenmann formalise the iterative compilation problem as: given a set of compiler optimization options $\{F_1, F_2, \ldots, F_n\}$, find the combination that minimizes 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 [PE06]. Posing the problem as a subset search negates the challenge of optimisation *ordering*, though this challenge has been the focus of other work [KC12; PJ13].

Ryoo et al. 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% [Ryo+08b]. There is no guarantee that for a new program, the reduced search space will include the optimal configuration.

Purini and Jain prune the *solution* space for optimisation orderings. Rather than attempting to find a universally optimal sequence of optimisations, they identify a *set* of good optimisation sequences that is small enough that each new program can 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 [PJ13]. 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 set.

A complementary approach to search space pruning is knowledge sharing. The idea is that, since most software is shared across many users, leverage this by sharing knowledge of the optimisation space between users, rather than each user redundantly performing their own exploration of the optimisation space from scratch. Such “big data” approaches to auto-tuning has been variously proposed as *Collective Optimization* [FT10], *Crowdtuning* [MF13], and *Collective Mind* [Fur+14].Fursin et al. argue that the challenges facing widespread adoption of iterative compilation techniques 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 stacks; and the difficulty to validate techniques due to a lack of sharing in publications. They propose a system for addressing these concerns which provides a modular infrastructure for sharing iterative compilation performance data and related artefacts across the internet [Fur+14]. In past work, 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 [Cum+16b].

Another challenge facing iterative compilation is that results are not portable. Any change to the combination of program, input data, and hardware may impact the optimisation space, requiring a new iterative compilation process to start form scratch. This challenge can be mitigated using online compilation.

3.3.1.2 Online Iterative Compilation

The expensive optimisation space exploration required by iterative compilation has spurred development of dynamic optimisers that attempt to negate this "training" phase by interleaving 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, and that *exploration* and *exploitation* are balanced so as to maintain an acceptable quality of service for the user.

Tartara and Crespi Reghizzi 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 goal of finding the best set of compiler heuristics for a given binary [TC13].

Ansel and Reilly present an adversarial approach to online evolutionary performance tuning. At runtime, the available parallel resources of a device are divided in to two partitions. A different configuration of the application is then executed simultaneously on each partition. The configuration used for one of the configuration is chosen to be "safe", the other, experimental. The configuration which yields the best performance is retained as the "safe" choice for future iterations, and the process repeats.

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

Related to online iterative compilation is dynamic optimisation. *Dynamo* is a dynamic optimiser which performs binary level transformations of programs using information gathered from runtime profiling and tracing [BDB00]. 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 performances, however, whereas in iterative compilation selects the compiler transformations that produce the best configuration, in algorithmic choice the different configurations are explicitly provided by the user.

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

Halide alleviates the burden of algorithmic rewriting by providing a high level domain-specific language that allows users to express pipelines of stencil computations succinctly [**Ragan-Kelley2013**]. The *Lift* framework then uses a set of semantic-preserving rewrite rules to transform the high-level expressions to candidate low-level implementations, creating a space of possible implementations [SD17].

3.3.1.4 Super-optimisation

A logical conclusion of iterative compilation is *super-optimisation*. Like with algorithmic choice, the process performs higher-level algorithmic rewrites than compiler transformations. However, the process begins with no description of the algorithm, not even a high-level representation. Super-optimisation strives to find the *globally* optimal implementation of an algorithm, typically only from a handful of input output examples. The term super-optimisation is a reference to the misnaming of compiler *optimisation*, where finding the true *optimal* is considered an unrealistic goal given the time and resource constraints of a compiler.

Pioneered by Massalin, the smallest possible program which performs a specific function is found through a brute force enumeration of the entire instruction set. Starting with a program of a single instruction, the super-optimiser tests to see if any possible instruction passes a set of conformity tests. If not, the program length is increased by a single instruction and the process repeats. The optimiser is limited to register to register memory transfers, with no support for pointers, a limitation which is addressed in [JNR02]. *Denali* is a super-optimiser which uses constraint satisfaction and rewrite rules to generate programs which are *provably* optimal, instead of searching for the optimal configuration through empirical testing. Denali first translates a low level machine code into guarded multi-assignment form, then uses a matching algorithm to build a graph of all of a set of logical axioms which match parts of the graph before using boolean satisfiability to disprove the conjecture that a program cannot be written in n instructions. If the conjecture cannot be disproved, the size of n is increased and the process repeats.

In practise, Massalin found their system to scale only to subroutines typically less than 13 instructions [Mas87]. As with iterative compilation, the main problem is in pruning and efficiently navigating the search space. Researchers have turned to machine learning techniques as a means to alleviate the cost of empirical evaluation.

3.3.2 Machine Learning for Compiler Optimisations

Machine learning has emerged as a viable means in automatically constructing heuristics for code optimisation. Its great advantage is that it can adapt to changes in the compiler and hardware environment as it has no a priori assumptions about their behaviour. The machine learning for compiler literature has been recently reviewed by Ashouri et al. [Ash+18] and Wang and O'Boyle [WO18].

Pioneered by Agakov et al., the idea is to iteratively evaluate a collection of training programs offline and gather explanatory variables describing the features of the programs. The program features and the optimisation decisions which yielded 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. In [Aga+06] machine learning is used to guide the iterative compilation search. In [SMR03], "meta optimisation" is used to tune compiler heuristics through an evolutionary algorithm to automate the search of the optimisation space. The phase-ordering problem is formulated as a Markov process in [KC12] and tackled using neuro-evolution to construct a neural network that predicts beneficial optimization orderings given a characterization of the state of code being optimized. A later approach to the phase-ordering problem clusters optimisations and uses machine learning to predict the speedup of a sequence of all optimisation clusters [Ash+17].

Ganapathi et al. tackle multi-core stencil code optimisation in [Gan+09], drawing upon the successes of statistical machine learning techniques in the compiler community. They present an auto-tuner 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.

In past work [Cum+15] and in [LFC13], domain-specific machine learning systems are used to optimise stencil computations on GPUs. Restricting the domain of optimisations to a single class of algorithm can simplify the problem by limiting the variance in the function being estimated. A domain-specific machine learning based auto-tuner is presented for the SkePU library in [DEK11]. SkePU is a C++ template library for data-parallel computations on GPUs. The auto-tuner predicts optimal device mapping (i.e. CPU, GPU) for a given program by predicting execution time and memory copy overhead based on problem size. Similarly, in this thesis a machine learning auto-tuner is used to predict optimal device mapping, though the auto-tuner is capable of making predictions for arbitrary GPU programs, it is not bound to a single template library. **Moren2018** also tackle the task of mapping arbitrary OpenCL kernels to CPU/GPU using dynamic features extracted from the kernel at runtime [**Moren2018**].

These three papers: [**Marco2017**]
[**Zhang2018d**]
[**Taylor2017**]

Milepost GCC 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 [Fur+11]. The embedded interface exposes candidate features which may be used to apply machine learning to an optimisation in GCC, however it does not address the issue of feature selection.

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

Besides compilers, there are a broad range of applications for machine learning in improving software performance. Even purposes as conventional as hash key functions have been the subject of machine learning. Kraska et al. find that replacing a cache-optimised B-Tree-Index implementation with a deep learning model yields up to 70% speedup with a $10\times$ reduction in memory on some real workloads [Kra+18]. Krishnan et al. 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 a set of high-quality features that can capture the important characteristics of programs. Given that there is an infinite number of these 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. Dubach et al. characterise programs using performance counters [Dub+09]. Jiang et al. extract program-level behaviours such as loop trip counts and the size of input files [Jia+10]. Berral et al. use additional runtime information such as system load [Ber+10].

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 initial large, high dimensional candidate feature space is pruned via feature selection, or projected into a lower dimensional space. Stephenson and Amarasinghe 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 [SA05]. Collins et al. use Principle Component Analysis (PCA) to reduce a four-dimensional feature space to two dimensions, reducing the size of the space to 0.05%. Dubach et al. also use PCA to reduce the dimensionality of their feature space, but determine and use the number of components that account for 95% of the total variance. In their case, 5 components. *FEAST* employs a range of existing feature selection methods to select useful candidate features [Tin+16].

Park, Cavazos, and Alvarez present a unique graph-based approach for feature representations [PCA12]. They use a Support Vector Machine (SVM) where the kernel is based on 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. The techniques presented in this thesis 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. These approaches closely tie the implementation of the predictive model to the compiler IR, which means changes to the IR will require modifications to the model. Leather, Bonilla, and O'Boyle uses genetic programming to search for features, requiring a huge grammar to be written, some 160kB in length [LBO14]. 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. Namolaru et al. 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.

Cavazos et al. present a reaction-based predictive model for software-hardware co-design [Cav+06]. Their approach profiles the target program using several carefully selected compiler options to see how 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 our technique does 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 applied deep neural networks for program feature generation and selection. This work is the first to do so.

3.3.2.2 Representing programs with embeddings

This thesis presents deep learning methodologies for learning over programs, inspired by natural language processing. With these techniques, program source code is tokenized 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. Allamanis et al. review some of the proposed techniques in [All+17] Babii, Janes, and Robbes explore the impact that choices in vocabulary have on time to convergence of software language models [BJR19].

The techniques in this thesis use a hybrid character/token-level vocabulary to tokenize source code. This is to prevent the blow-up in vocabulary size that occurs from using a purely token-based vocabulary. Cvitkovic, Singh, and Anandkumar 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 [CSA18].

Mou2016 derive an embedding space from the tokens in the source code of a program [**Mou2016**]. Wang, Singh, and Su propose an embedding space extracted from program traces, rather than the syntactic structure of the program [WSS17].

Neural Code Comprehension builds on the experiments in Chapter 6 of this thesis, using embeddings derived from a novel *Contextual Flow Graph* (XFG) representation which combines both data and control flow. The embeddings are assembled from LLVM byte-code, enabling support for any language for which there exists a front-end to LLVM [BJH18].

There are plenty of other _2vec papers to touch on

3.4 Deep Learning over Programs

Deep learning is a nascent branch of machine learning in which deep or multi-level systems of processing layers are used to detect patterns in natural data [LBH15; WRX17]. 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, including human parity in conversational speech recognition [Xio+16]; professional level performance in video games [Mni+15]; and autonomous vehicle control [LCW12]. The use of deep learning techniques for software has long been a goal of research [Whi+15].

up from
!

Learning to summarise / name code:

[All+17]

Naturalize employs techniques developed in the natural language processing domain to model coding conventions [All+14].

JSNice leverages probabilistic graphical models to predict program properties such as identifier names for JavaScript [**Raychev**].

Allamanis, Peng, and Sutton use attentional neural networks to generate summaries of source code [APS16].

Wong, Yang, and Tan mines Q&A site StackOverflow to automatically generate code comments [WYT13].

Procedure names [DAY19].

In recent years, machine learning techniques have been employed to model and learn from program source code on various tasks. These include mining coding idioms [AS14]

There is an increasing interest in mining source code repositories at large scale [AS13; Kal+09; Whi+15]. Previous studies have involved data mining of GitHub to analyse software engineering practices [Bai+14; GAL14; VFS15; Wu+14].

Concern about code duplicates in corpora of open-source programs are raised in [All18]. This impacts cases where the open-source corpus is divided into training/test set. The high percentage of near-duplicate code means that often the divide is muddied. The work in this thesis does not use open source corpus as a test set.

Bug detection:

Machine learning has also been applied to other areas such as improving bug finding static analysers [HOY17; Koc+17].

Identifying buffer overruns [Cho+17]

Processing bug reports [HLZ16; Lam+15].

DeepBugs combines binary classification of correct and incorrect code with semantic processing to name bugs [PS18].

Code2Inv uses reinforcement learning to learn loop invariants for program verification [Si+18].

Learning to represent edits and diffs [Yin+18] [Tuf+19].

Bug prioritisation:

Prioritising test programs [Che+17]

Example code generation / API usage / code completion:

Code generation [Gu+16].

Raychev, Vechev, and Yahav use statistical models to provide contextual code completion [RVY14].

Gu et al. use deep learning to generate example code for APIs as responses to natural language queries [Gu+16].

Pseudo-code generation [Oda+15].

In a recent work [Bun+17], Bunel *et al.* formulate code super-optimisation as a stochastic search problem to learn the distribution of different code transformations and expected performance improvement. As acknowledged by the authors, their approach can be improved by having temporal information of the code structures.

Learning to fix bugs:

Survey of automatic software repair [Mon18].

Repairing programs [Kou+17]

DeepRepair sorts code fragments according to similarity of suspsicious elements [Whi+19].

A slew of neural program repair tools:

Vasic et al. train a model to jointly loalize and repair variable-misuse bugs, uses multi-headed pointer networks [Vas+19]

.

Program repair [Che+18].

Corrective patches [HSN].

Getafix uses a hierarchical clustering algorithm that summarizes fix patterns into a hierarchy ranging from general to specific patterns [Bad+19].

Embeddings from AST [Hen+18].

Another instance of learning from a corpus of programs: *CodeBuff* uses a carefully designed set of features to learn abstract code formatting rules from a representative corpus of programs [TV16].

3.5 Summary

Chapter 4

Generating Benchmarks through Deep Learning

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 hand-written code. In this chapter, CLgen is used to automatically synthesise thousands of programs and show that learning over these improves the performance of a state-of-the-art predictive model by $1.27\times$. In addition, the fine covering of the feature space automatically exposes weaknesses in the feature design which are invisible with the sparse training examples from existing benchmark suites. Correcting

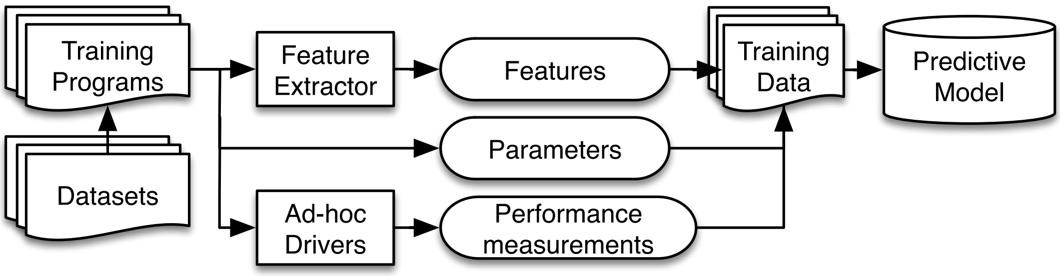


Figure 4.1: Training a predictive model.

these weaknesses further increases performance by 4.30×.

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

It is common for feature vectors to contain dozens of elements. This means that a large volume of training data is needed to have an 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 defines 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 [Cum+16b; FE15]. 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 is 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 [Gra13; SVL14]. Deep learning is capable not just of learning the macro syntactical and semantic structure of programs, but also the nuances of how humans typically write code. It is truly remarkable when one considers that it is given no prior knowledge of the syntax or semantics of the language.

The second is the increasing popularity of public and open platforms for hosting software projects and source code. This popularity provides the 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 we discover that it is also useful 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, we are 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 a potential value of this technique for feature designers.

This chapter is organised as follows: first, motivation is provided for the use of benchmark generators in Section 4.2. 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 Benchmark Generators

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 *Grewe et al.* [GWO13] predictive model was evaluated across each of the 7 most frequently used benchmark suites (accounting for 92% of results in the surveyed papers). The *Grewe et al.* 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 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. Principle Component Analysis is used to reduce the multi-dimensional feature space to aid

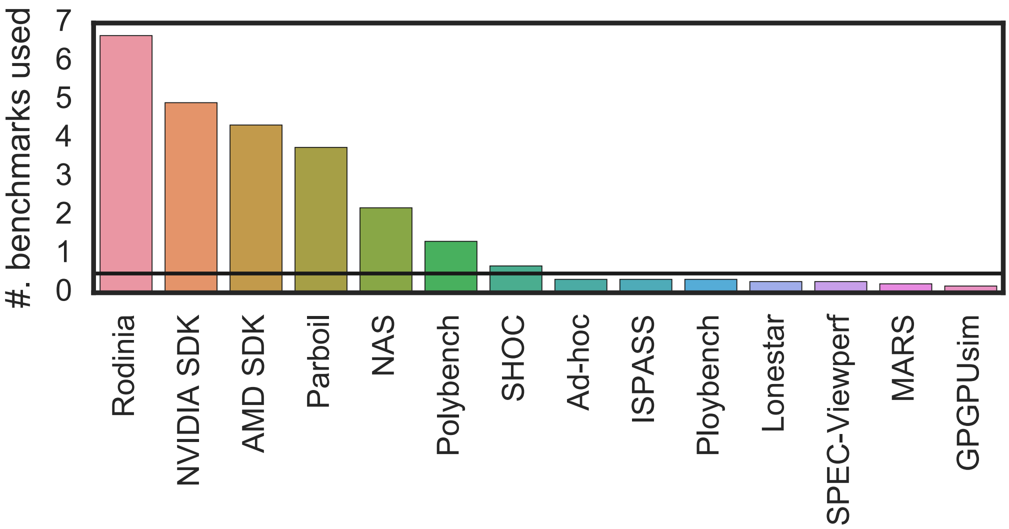


Figure 4.2: The average number of benchmarks used in GPGPU research papers published between 2013-2016 in CGO, HiPC, PACT, and PPoPP conferences.

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

Table 4.1: Performance relative to the optimal of the *Grewe et al.* 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.

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, I hand-selected benchmarks which are neighbouring in the feature space and retrained the model. 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.

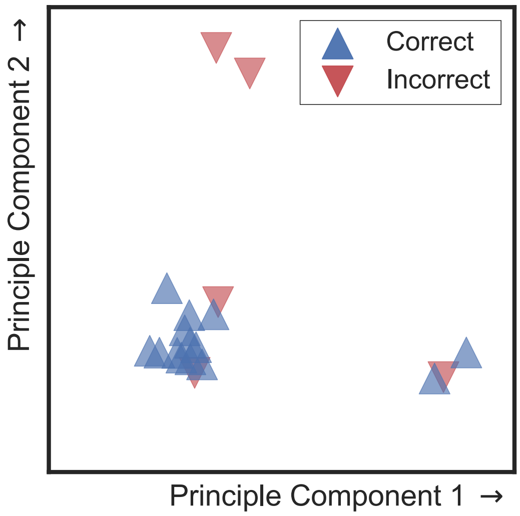
These results highlight the significant effect that the number and distribution of training programs has on the quality of predictive models. Without good coverage of the feature space, any machine learning methodology is unlikely to produce high quality heuristics, suitable for general use on arbitrary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

4.3 CLgen: Synthetic OpenCL Benchmark Generation

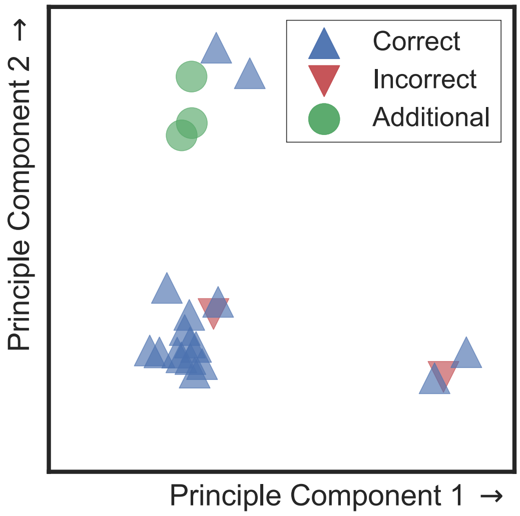
This section introduces CLgen, an undirected, general-purpose program synthesizer. It adopts and augments recent advanced techniques from deep learning to learn over massive codebases. In contrast to existing grammar and template based approaches, CLgen is entirely probabilistic. The system *learns* 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 hand-written 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



(a)



(b)

Figure 4.3: A two dimensional projection of the *Grewe et al.* 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.

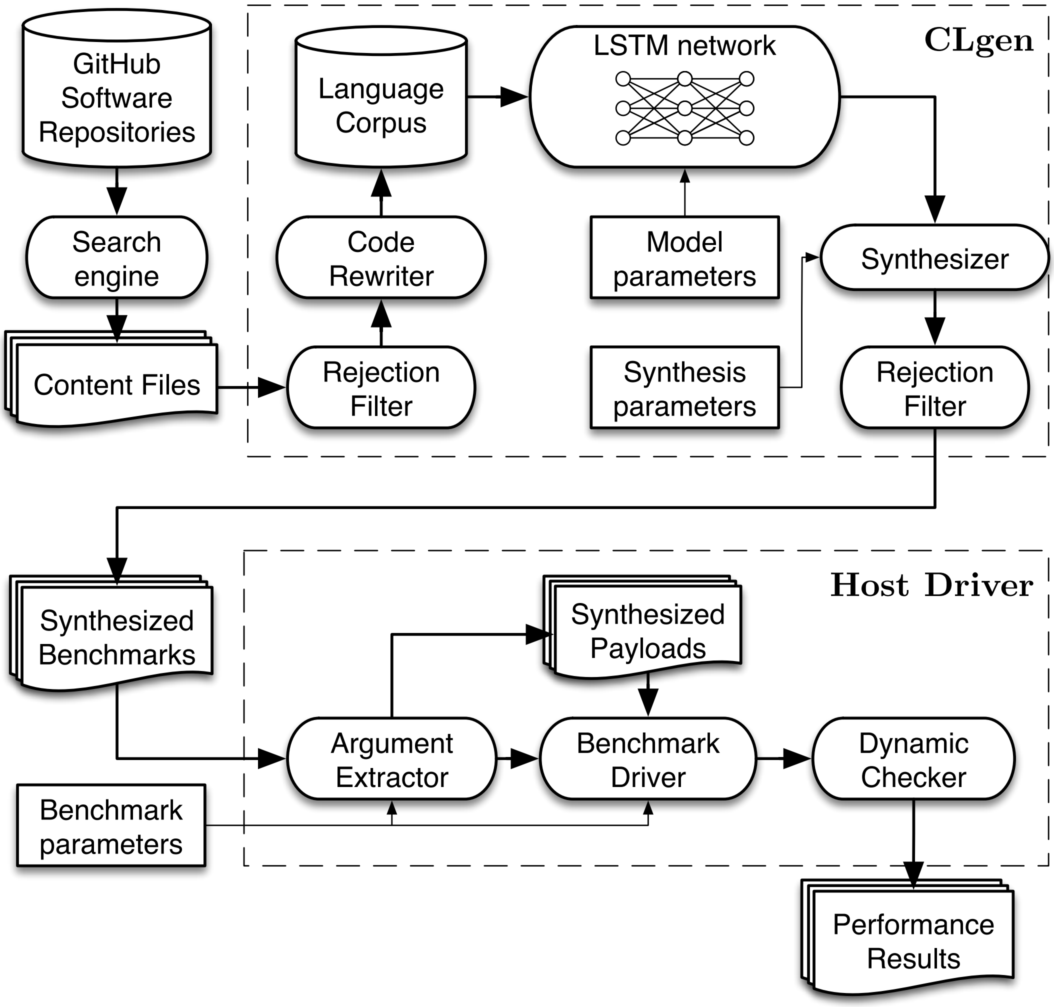


Figure 4.4: The benchmark synthesis and execution pipeline. Software is mined from GitHub; this is used to construct a language model from which new programs may be synthesised; a host driver is used to produce performance results.

better and more accurate predictive models.

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

4.3.2 An OpenCL Language Corpus

Deep learning requires large data sets [LBH15]. For the purpose of modelling a programming language, this means assembling a 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 tool chain developed for rejection filtering and code rewriting, built on LLVM.

4.3.2.1 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 a 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 being rejected). Through analysing 148k lines of compilation errors, a large number of failures was 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 4.1.

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.

4.3.2.2 Code Rewriter

Programming languages have few of the issues of semantic interpretation present in natural language, though there remains 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

```
/* 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 4.1: An overview of the *shim* header file, providing inferred type aliases and constants for OpenCL on GitHub.

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 is pre-processed 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 [AS13], our 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 is enforced to ensure consistent use of braces, parentheses, and white space.

An example of the code rewriting process is shown in Listings **??** and 4.3. 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

```
#define DTYPE float
#define ALPHA(a) 3.5f * a
inline DTYPE ax(DTYPE x) { return ALPHA(x); }

__kernel void saxpy( /* SAXPY kernel */
    __global DTYPE *input1,
    __global DTYPE *input2,
    const int nelem)
{
     unsigned int idx = get_global_id(0);
     // = ax + y
     if (idx < nelem) {
         input2[idx] += ax(input1[idx]); }}
```

Listing 4.2: An example OpenCL content file downloaded from GitHub prior to code rewriting.

lines of transformed OpenCL, consisting of 9487 kernel functions. Identifier rewriting reduces the bag-of-words vocabulary size 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) architecture of Recurrent Neural Network [Mik10; SSN12] is used to learn a character-level language model over the corpus of OpenCL compute kernels. The LSTM network architecture comprises recurrent layers of *memory cells*, each consisting of an input, output, and forget gate, and an output layer providing normalised probability values from a 1-of-K coded vocabulary [GS05].

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

```
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 4.3: The example OpenCL content file of Listing 4.2 after code rewriting. Conditional compilation has been removed, the variables and functions renamed, and a code style enforced.

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 model is sampled, character by character, until the end of the compute kernel is reached, or until a predetermined maximum number of characters is reached. Algorithm 1 illustrates this process. The same rejection filter described in Section 4.3.2.1 then either accepts or rejects the sample as a candidate synthetic benchmark. Listings 4.4, 4.5, and 4.6 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.

Algorithm 1 Sampling a candidate kernel from a seed text.

Require: LSTM model M, maximum kernel length n.

Ensure: Completed sample string S.

```
S ← "__kernel void A(const int a) {"            ▷ Seed text
d ← 1                                             ▷ Initial code block depth
for i ← |S| to n do
    c ← predictcharacter(M, S)                    ▷ Generate new character
    if c = "{" then
        d ← d + 1                                 ▷ Entered code block, increase depth
    else if c = "}" then
        d ← d − 1                                 ▷ Exited code block, decrease depth
    end if
    S ← S + c                                     ▷ Append new character
    if depth = 0 then
        break                                     ▷ Exited function block, stop sampling
    end if
end for
```

```
__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 4.4: 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 4.5: CLgen-synthesised zip operation which computes $c_i = 3a_i + 2b_i + 4$.

4.4 Benchmark Execution

A host driver is used to gather performance data from synthesised CLgen code. The driver accepts as input an OpenCL kernel, generates *payloads* of user-configurable sizes, and executes the kernel using the generated payloads, providing dynamic checking of kernel behaviour.

4.4.1 Generating 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

For the purpose of performance benchmarking the correctness of computed values is of little interest, but I define a class of programs as performing *useful work* if they predictably compute some result. A low-overhead runtime behaviour check is used to

```
__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 4.6: CLgen-synthesised partial reduction over reinterpreted vector type.

validate that a synthesised program does useful work based on the outcome of four executions of a tested program:

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

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

4.5 Qualitative Evaluation of Generated Programs

This section evaluates the quality of programs synthesised by CLgen by their likeness to hand-written code.

Judging whether a piece of code has been written by a human is a challenging task for a machine, so I adopted a methodology from machine learning research based on the *Turing Test* [Gao+15; Vin+15; ZIE16]. If the output of CLgen is human-like code, it reasons then that a human judge will be unable to distinguish it from hand-written 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 hand-written code from GitHub. The samples from GitHub were vetted to ensure that they were indeed hand-written 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, with 10 of them receiving code generated by CLgen, and 5 of them acting as a control group, receiving code generated by CLSmith [Lid+15], a program generator for differential testing[1].

Each participant's answers was scored. The average score of the control group is 96% (stdev. 9%), an unsurprising outcome as programs generated using the CLSmith grammar for testing have multiple "tells"; for example, 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 hand-written programs, with human judges scoring no better than random chance.

4.6 Experimental Methodology

4.6.1 Experimental Setup

4.6.1.1 Predictive Model

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

4.6.1.2 Benchmarks

As in [GWO13], the model is tested on the NAS Parallel Benchmarks (NPB) [Bai+91]. The hand-optimised OpenCL implementation of Seo, Jo, and Lee [SJL11] is used.

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

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)`$\times$`wgsize`	Memory access ratio $\times$ #. work-items
`F4`	`comp/mem`	Computation-memory ratio

(b) Combinations of raw features

Table 4.2: Features used by *Grewe et al.* to predict CPU/GPU mapping of OpenCL kernels. The features are extracted using a custom analysis pass based using LLVM.

In [GWO13] the authors augment the training set of the predictive model with 47 additional kernels taken from 4 GPGPU benchmark suites. To more fully sample the program space, a much larger collection of 142 programs 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 **??**. None of these programs were used to train CLgen. 1,000 kernels were synthesised with CLgen to use as additional benchmarks.

4.6.1.3 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.

4.6.1.4 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

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

Table 4.3: List of benchmarks

	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.

configured to synthesise payloads between 128B-130MB, approximating that of the dataset sizes found in the benchmark programs.

4.6.2 Methodology

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

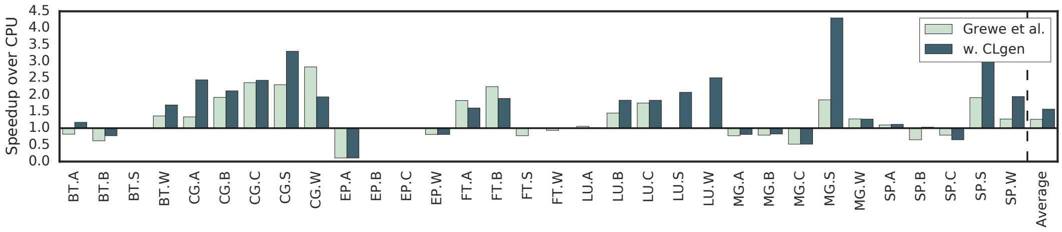
Models are evaluated using *leave-one-out cross-validation*. For each benchmark, a model is trained on data from all other benchmarks and used to predict the mapping for each kernel and dataset in the excluded program. The process is repeated with and without the addition of synthetic benchmarks in the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

4.7 Experimental Results

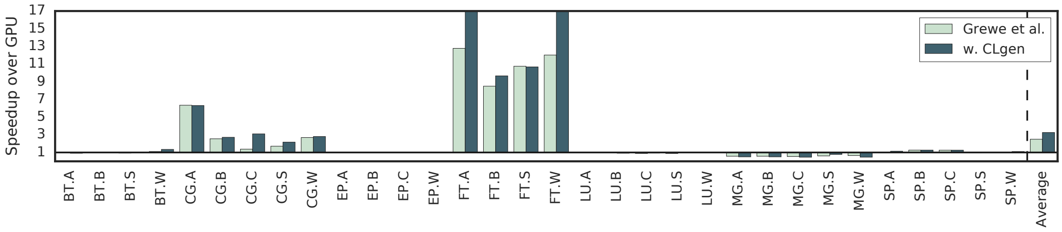
The effectiveness of synthetic benchmarks is evaluated on two heterogeneous systems. First the performance of a state-of-the-art predictive model [GWO13] is compared with and without the addition of synthetic benchmarks, then synthetic benchmarks are shown expose weaknesses in the feature design and how these can be addressed to develop a better model. Finally we compare the ability of CLgen to explore the program feature space against a state-of-the-art program generator [Lid+15].

4.7.1 Performance Evaluation

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



(a) AMD Tahiti 7970



(b) NVIDIA GTX 970

Figure 4.5: Speedup of programs using *Grewe et al.* 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.

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 sparse 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 *Grewe et al.* 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` 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 of CLgen-generated programs had identical feature values as in the benchmark set, but had different *behaviour* (i.e. optimal mappings). Listing 4.7 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 transform — 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 **??**. 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 make heavy use of loops, which it is believed the static code features of the model fail 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 goal to perfect the predictive model, but to show the performance improvements associated with training on synthetic pro-

```
__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 4.7: In the *Grewe et al.* feature space this CLgen program is indistinguishable from AMD's Fast Walsh–Hadamard transform benchmark, but has very different runtime behaviour and optimal device mapping. The addition of a branching feature fixes this.

grams. To this extent, the goal is succeeded, achieving average speedups of 3.56$\times$ on AMD and 5.04$\times$ on NVIDIA across a very large test set.

4.7.3 Comparison of Source Features

As demonstrated in Section **??**, 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 benchmarks that we are testing on.

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 we examined in the previous subsections. Only static code features are examined to allow comparison with the GitHub kernels for which we have no automated method to execute them and extract runtime features, and CLSmith generated programs.

Figure 4.7 plots the number of 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 confirms the original intuition: CLgen kernels, by emulating the way real humans

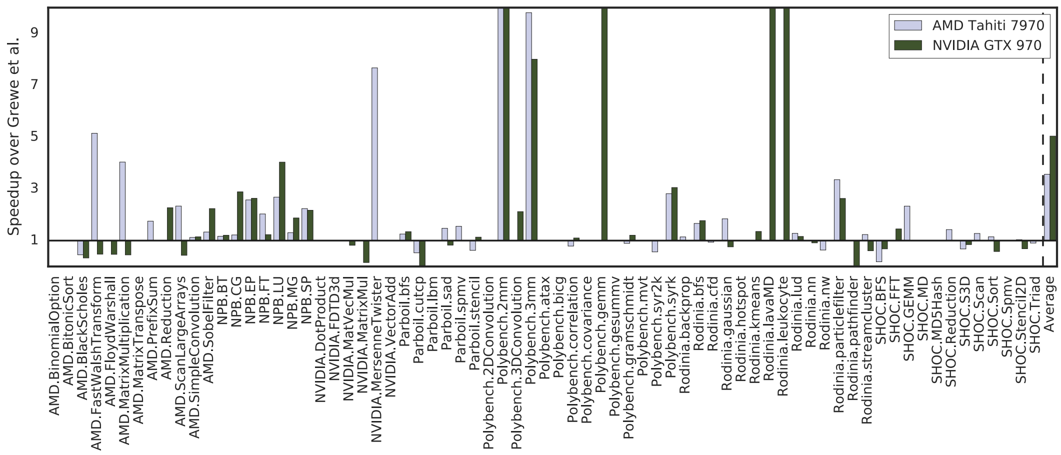


Figure 4.6: Speedups of predictions using an extended model over *Grewe et al.* on both experimental platforms. Synthetic benchmarks and the additional program features outperform the original predictive model by a factor $3.56\times$ on AMD and $5.04\times$ on NVIDIA.

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. I conclude that 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 is typically only a few dozen common benchmarks available for experiments. This chapter present a novel tool which is the first of it's kind — an entirely probabilistic program generator capable of generating 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 programs which to trained eyes are indistinguishable from hand-written code. The approach is tested using a state-of-the-art predictive model, improving its performance by a factor of $1.27\times$. Synthetic

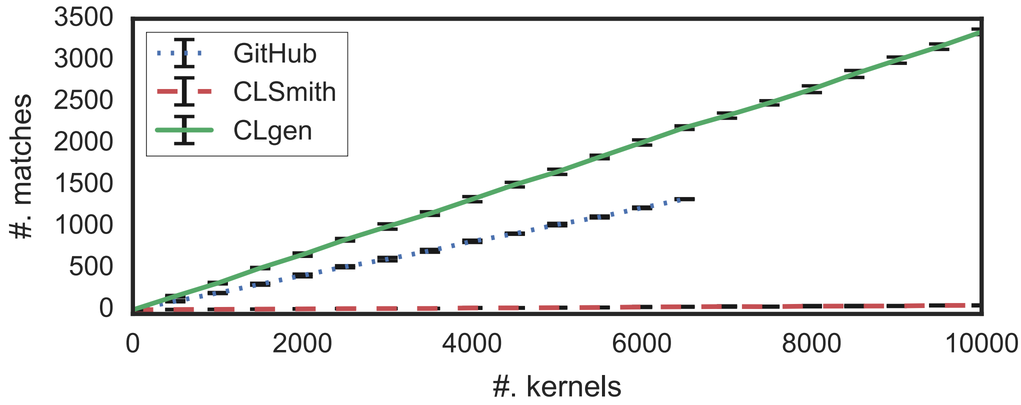


Figure 4.7: The number of kernels from GitHub, CLSmith, and CLgen with static code features matching the benchmarks. CLgen generates kernels that are closer in the feature space than CLSmith, and can continue to do so long after we have exhausted the extent of the GitHub data set. Error bars show the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

benchmarks automatically exposed weaknesses in the feature set which, when corrected, further improved the performance by $4.30\times$.

Chapter 5

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

Compiler fuzzing requires efficiently generating test cases that trigger compiler bugs. The state-of-the-art approach, CSmith [Yan+11], generates large random programs by defining and sampling a probabilistic grammar which covers a subset of the C programming language. Through this grammar, CSmith ensures that the generated code easily passes the compiler front-end and stresses the most complex part of the compiler, the middle-end. Complex static and dynamic analyses make sure that programs are free from undefined behaviour. The programs are then differentially tested.

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

This chapter introduces *DeepSmith*, a novel machine learning approach to accelerating compiler validation through the inference of generative models for compiler inputs. DeepSmith is a fast, effective, and low effort approach to the generation of random programs for compiler fuzzing. The methodology 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 OpenCL programming language. In 1,000 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[1] is an open source framework for compiler fuzzing. Figure 5.1 provides a high-level overview. In this work OpenCL is targeted, though the approach is language agnostic. 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, employing state-of-the-art deep learning techniques to build models for how humans write programs. The approach is inspired by breakthrough results in modelling

[1] DeepSmith available at: https://chriscummins.cc/deepsmith

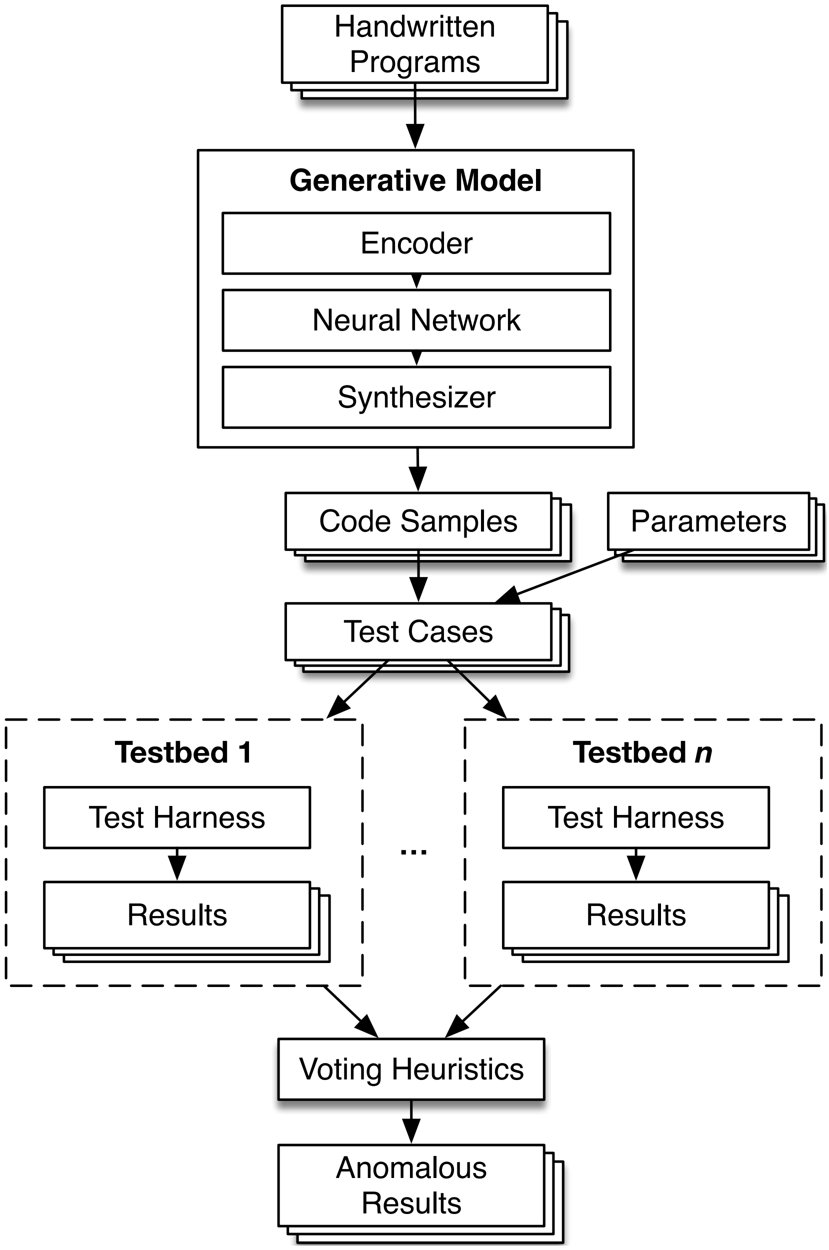


Figure 5.1: DeepSmith system overview.

challenging and high dimensional data sets through unsupervised learning [Bow+15; Rag+16; RJS17]. Contrary to existing tools, this approach does not require expert knowledge of the target language and is only a few hundred lines of code.

5.2.1.1 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.

5.2.1.2 Encoder

The textual representation of program codes must be encoded as numeric sequences for feeding as input to the machine learning model. Prior machine learning works have used character-level encodings, token-level encodings, or fixed length feature vectors. We extend the hybrid character/token-level encoding of [Cum+17], 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.

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 I reason that this is an acceptable trade-off. For languages where the corpus can be many orders of magnitude larger, for example, C or Java, models may be generated without these modifications.

5.2.1.3 Neural Network

The Long Short-Term Memory (LSTM) architecture of Recurrent Neural Network is used to model program code [HS97]. 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 [PMB13]. This makes them effective at learning complex relationships over long sequences [LBE15] which is important for modelling program code. In this approach, LSTM networks are employed to model the vocabulary distribution over the encoded corpus. 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.

s should be
ntified

5.2.1.4 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. 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 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.

5.2.2.1 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 online compilation of the 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.

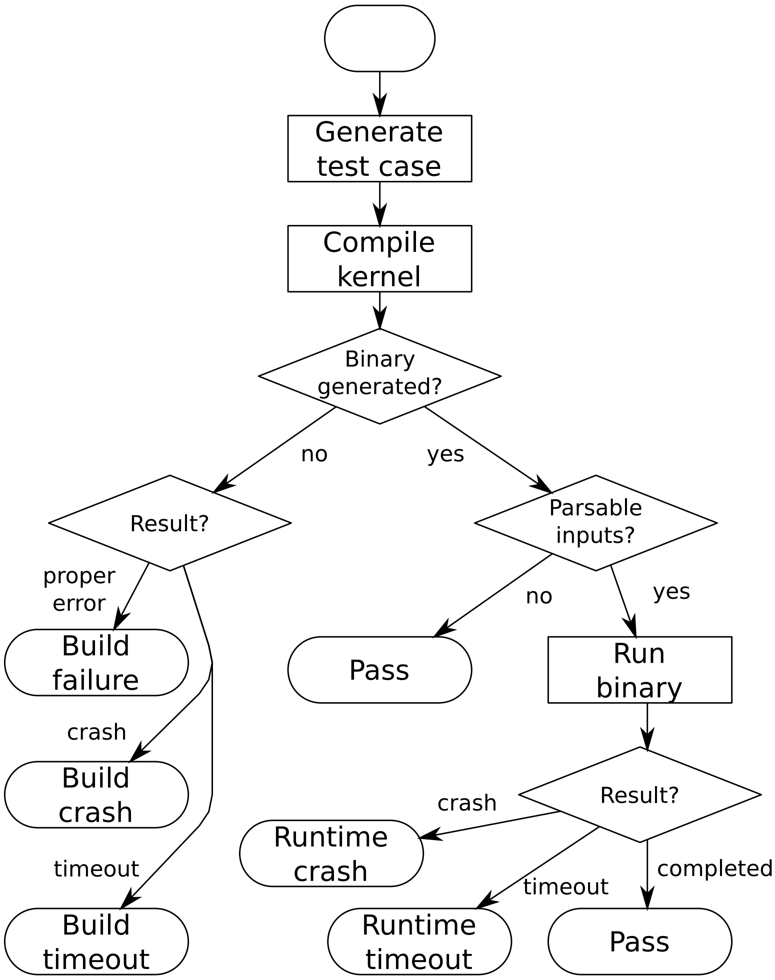


Figure 5.2: Test case execution, and possible results.

5.2.3 Voting Heuristics for Differential Testing

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

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. We look for test cases where there is a majority outcome – i.e. for which some fraction of the testbeds behave 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.

5.2.3.1 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 [BCD12] and Oclgrind [PM15]. Some compiler warnings provide 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 code 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 we 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

In this section we describe the experimental parameters used.

5.3.1 OpenCL Systems

We conducted testing of 10 OpenCL systems, summarised in Table 5.1. We covered a broad range of hardware: 3 GPUs, 4 CPUs, a co-processor, and an emulator. 7 of the compilers tested are commercial products, 3 of them are open source. Our 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, we create two testbeds. 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

For each generated program we create inputs as described in Section 5.2.2. In addition, we need to choose the number of threads to use. We generate two test cases, one using one thread, the other using 2048 threads. A test case is then a triple, consisting of *¡program, inputs, threads¿* settings.

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

(a)

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

(b)

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

5.3.4 Bug Search Time Allowance

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

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 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 [Lid+15]. DeepSmith is able to identify a broad range of defects, many of which CLSmith cannot, for only a fraction of the engineering effort. Finally, this section contains a quantitative analysis of compiler robustness over time, using the compiler crash rate of every LLVM release in the past two years as a metric of compiler robustness. The findings show that progress is good, compilers are becoming more robust, yet the introduction of new features and regressions ensures that compiler validation remains a moving target.

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

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

5.4.1 Compile-time Defects

OpenCL is typically compiled online, which amplifies the significance of detecting compile-time defects, as they may not be discovered until code has been shipped to customers. Numerous cases were found where DeepSmith kernels trigger a crash in the compiler (and as a result, the host process), or cause the compiler to loop indefinitely. In the testing time allotted 199 test cases were identified which trigger unreachable

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

(a) Reduced from 48 line kernel.

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

(b) Reduced from 52 line kernel.

```
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.

code failures, triggered 31 different compiler assertions, and produced 114 distinct stack traces from other compiler crashes.

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 to 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 defect. Figure 5.4b induces a crash in NVIDIA compilers due to an implicit conversion between global to constant address qualifiers. Worse, Testbeds 3± was 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

```
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.

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-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. 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 [Che+16]. 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 < ...)` `{...}`) 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 have 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.

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

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

Figure 5.6: Example OpenCL kernels which are 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 5.6d 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 5.7a shows the generation and execution times of DeepSmith and CLSmith test cases, excluding timeouts[2]. DeepSmith generation time grows linearly with program length, and is on average 2.45× faster than CLSmith. Test case execution is on average 4.46× faster than CLSmith.

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

[2] 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 we change the arbitrary timeout threshold, so for fairness timeouts have been excluded.

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

#.	Device	±	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

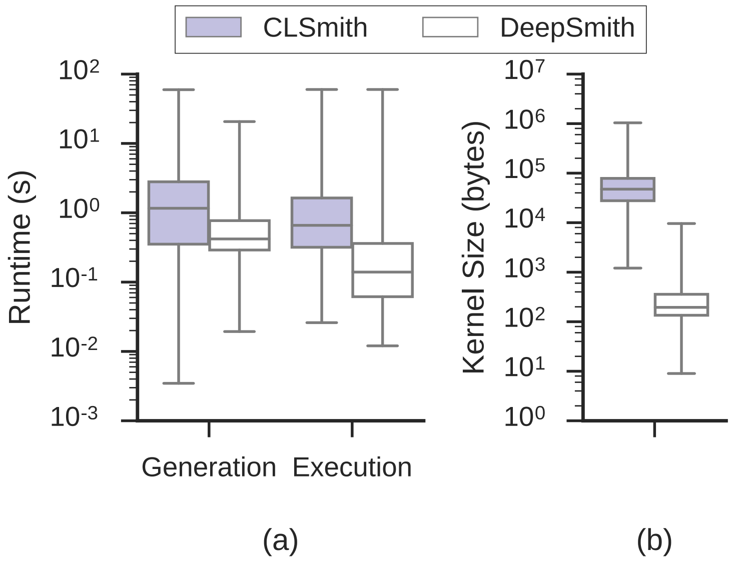


Figure 5.7: Comparison of runtimes (a) and test case sizes (b). DeepSmith test cases are on average evaluated $3.03\times$ faster than CLSmith ($2.45\times$, and $4.46\times$ for generation and execution, respectively), and are two orders of magnitude smaller. Timings do not include the cost of timeouts which would increase the performance gains of DeepSmith by nearly a factor of two.

5.4.3.2 Comparison of Test Cases

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

5.4.3.3 Comparison of Results

Both testing systems found anomalous results of all types. In 48 hours of testing, CLSmith discovered compile-time crashes (**bc**) in 8 of the 20 testbeds, DeepSmith crashed all of them. DeepSmith triggered 31 distinct compiler assertions, CLSmith 2. Both of the assertions triggered by CLSmith were also triggered by DeepSmith. DeepSmith also triggered 3 distinct *unreachable!* compile-time crashes, CLSmith triggered 0. The ratio of build failures is higher in the token-level generation of DeepSmith (51%) than the grammar-based generation of CLSmith (26%).

The Intel CPU Testbeds (4±, 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 5.8 and 5.9 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 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 5.10. 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 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 a greater potential to trigger miscompilations. CLSmith generated 33 programs with anomalous wrong-outputs. DeepSmith generated 30.

```
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 5.8: 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 5.9: 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 5.10: Example kernels which trigger compiler assertions which both CLSmith and DeepSmith exposed.

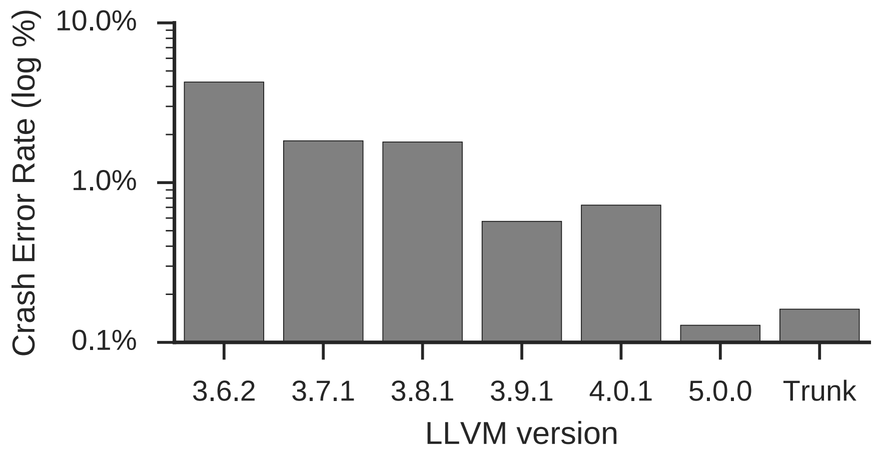


Figure 5.11: Crash rate of the Clang front-end of every LLVM release in the past 24 months compiling 75k DeepSmith kernels.

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 5.11 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.

	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.

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

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

Since LLVM will not release unless their compiler passes their own extensive test suites, this also reinforces the case for compiler fuzzing. DeepSmith provides an effective means for the generation of such fuzzers, at a fraction of the cost of existing techniques.

5.4.5 Extensibility of Language Model

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

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 random program generator is 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

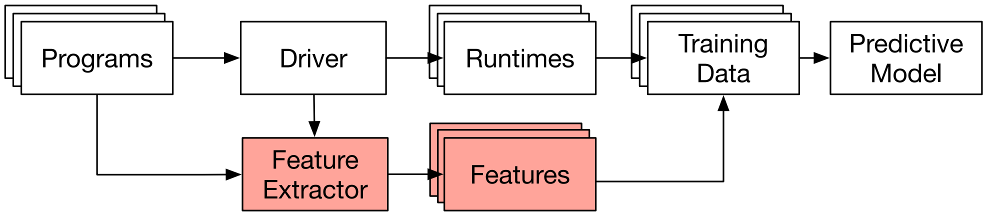
End-to-end Deep Learning of Optimisation Heuristics

6.1 Introduction

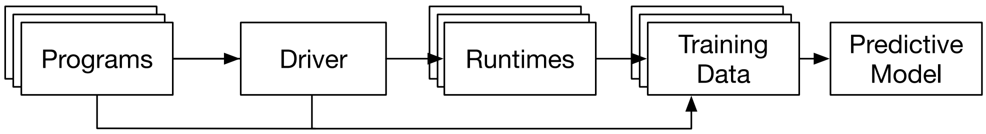
There are countless scenarios during the compilation and execution of a parallel program where decisions must be made as to how, or if, a particular optimisation should be applied. Modern compilers and runtimes are rife with hand coded *heuristics* which perform this decision making. The performance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

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

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



(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. For example, Section 4.7.2 identified one such feature, causing performance to be 40% lower on average.

To make heuristic construction fast and cheap, humans must be taken out of the loop. While techniques for automatic feature generation from the compiler IR have been proposed in the past [LBO14; Nam+10], they do not solve the problem in a practical way. They are deeply embedded into the compiler, require expert knowledge to guide the generation, have to be repeated from scratch for every new heuristic, and their search time can be prohibitive. The inciting motivation for this chapter is that such costly approaches are not necessary any more. Deep learning techniques have shown astounding successes in identifying complex patterns and relationships in images [He+16; KSH12], audio [Lee+09], and even computer code [All+14; AS14]. This chapter hypothesises that deep neural networks should be able to automatically extract features from source code. The experiments showed that even this was a conservative target: with deep neural networks one can bypass static feature extraction and learn optimisation heuristics directly on raw code.

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 neural networks which learn how code correlates with performance. Internally and without prior knowledge, the networks construct complex abstractions of the input program characteristics and correlations between those abstractions and performance. This chapter proposes replacing the need for compile-time or static code features, merging feature and heuristic construction into a single process of joint learning. The system admits auxiliary features to describe information unavailable at compile time, such as the sizes of runtime input parameters. Beyond these optional inclusions, the system is able to learn optimisation heuristics without human guidance.

By employing *transfer learning* [Yos+14], the proposed approach is able to produce high quality heuristics even when learning on a small number of programs. The properties of the raw code that are abstracted by the beginning layers of our 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 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 introduces DeepTune, a novel system for building optimisation heuristics. Section 6.3 describes the experimental setup of two case studies: heterogeneous device mapping and thread coarsening. Section 6.4 contains the results of the case studies. Finally Section 6.5 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 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 System 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[1]. The model is using Keras, with TensorFlow [Aba+16] and Theano [Ber+11] back-ends.

6.2.2 Language Model

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

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

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

[1] DeepTune is available at: `https://chriscummins.cc/deeptune`

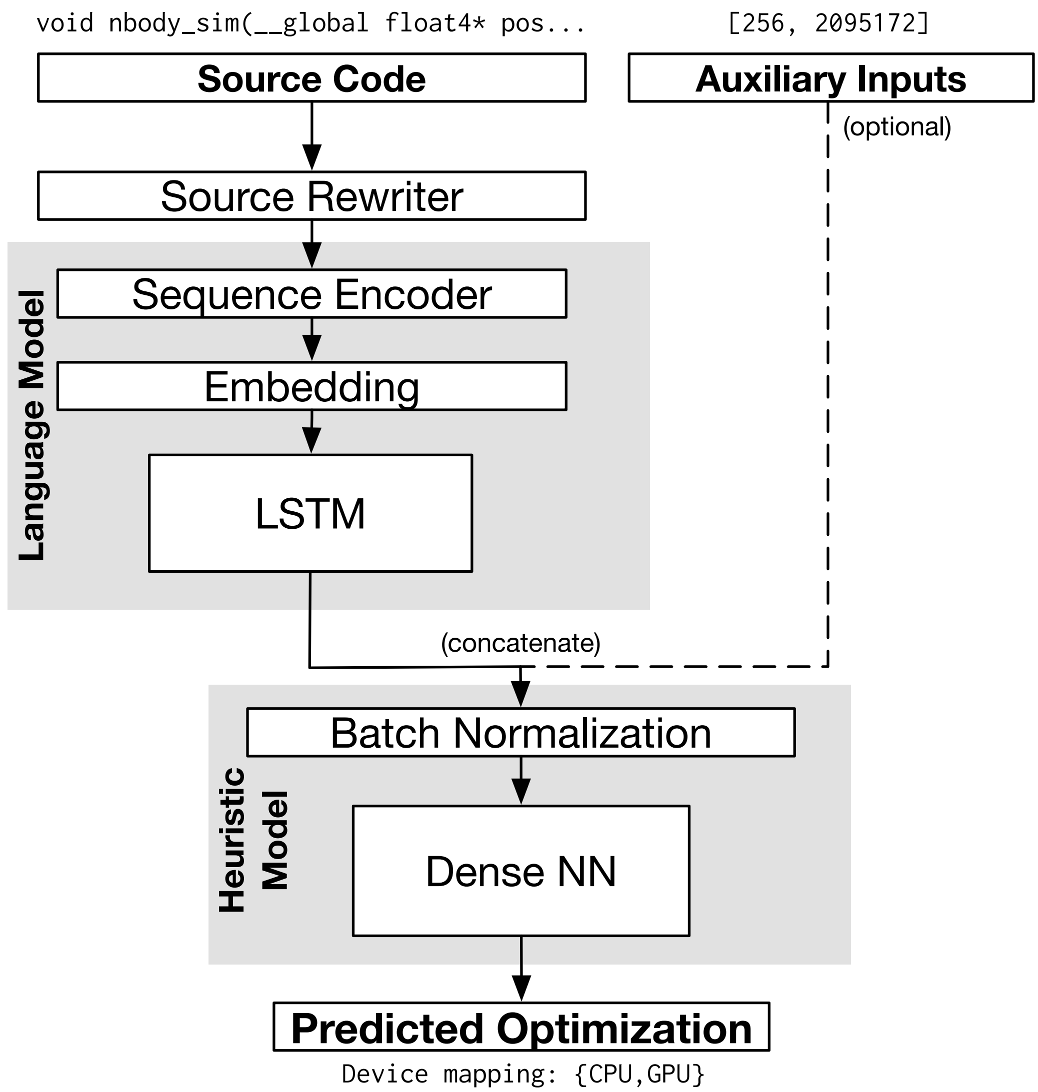


Figure 6.2: DeepTune architecture. Code properties are extracted from source code by the language model. They are fed, together with optional auxiliary inputs, to the heuristic model to produce the final prediction.

6.2.2.1 Source Re-writer

To begin with, a series of *source normalising* transformations are applied, extending the system described in Section 4.3.2.2. 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

A source code is encoded as a sequence of integers for interpretation by neural networks, where each integer is an index into a predetermined vocabulary. In [Joz+16], a character based vocabulary is used. This minimises the size of the vocabulary, but leads to long sequences which are harder to extract structure from. In [AS13], a token based vocabulary is used. This leads to shorter sequences, but causes an explosion in the vocabulary size, as every identifier and literal must be represented uniquely.

A hybrid, partially tokenised vocabulary approach is 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.

insert maximal munch algorithm, Algorithm **??**

This process continues until every character in the corpus has been consumed. The resulting derived vocabulary consists of 128 symbols which we use to encode new program sources. Figure 6.3c shows the vocabulary derived for a single input source code Figure 6.3b.

```
//#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.

Algorithm 2 Deriving a vocabulary from a string.

Require: Candidate vocabulary V_c, string S.

Ensure: Vocabulary V.

$i \leftarrow 1$
while $S \neq \phi$ **do**
  $i \leftarrow i + 1$
  $c \leftarrow substr(S, 0, i)$
  if $c \notin V_c$ **then**
    $c \leftarrow substr(S, 0, i - 1)$
    $S \leftarrow substr(S, i - 1, |S|)$
    $V \leftarrow V \cup \{c\}$
    $i \leftarrow 1$
  end if
end while

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 [BDK14; Mik+13]. An embedding layer maps each token in the integer encoded vocabulary to a vector of real values. Given a vocabulary size V and embedding dimensionality D, an embedding matrix $\boldsymbol{W}_E \in \mathbb{R}^{V \times D}$ is learned during training, so that an integer encoded sequences of tokens $\boldsymbol{t} \in \mathbb{N}^L$ is mapped to the matrix $\boldsymbol{T} \in \mathbb{R}^{L \times D}$. We use an embedding dimensionality $D = 64$.

6.2.2.4 Sequence Characterisation

Once source codes have been encoded into sequences of embedding vectors, neural networks are used to extract a fixed size vector which characterises the entire sequence.

This is comparable to the hand engineered feature extractors used in prior works, but is a *learned* process that occurs entirely — and automatically — within the hidden layers of the network.

The Long Short-Term Memory (LSTM) architecture is used for sequence characterisation [HS97]. LSTMs 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 [PMB13]), LSTMs employ a *forget gate* with a linear activation function, allowing them to retain activations for arbitrary durations. This makes them effective at learning complex relationships over long sequences [LBE15], an especially important capability for modelling program code, as dependencies in sequences frequently occur over long ranges (for example, a variable may be declared as an argument to a function and used throughout).

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

6.2.3 Auxiliary Inputs

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

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 we did not normalise, then scaling the auxiliary inputs could affect the training of the heuristic model. Normalisation occurs in batches. The batch normalisation method of [IS15] is used, in which each scalar of the heuristic model's inputs $x_1 \ldots x_n$

is normalised to a mean 0 and standard deviation of 1:

$$x_i' = \gamma_i \frac{x_i - E(x_i)}{\sqrt{Var(x_i)}} + \beta_i \tag{6.1}$$

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

The final component of DeepTune is comprised of two fully connected 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 training of deep networks [NH10]. 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 works, the key difference being that instead of having to manually create and extract features from programs, the raw program codes themselves are used.

The model is trained with Stochastic Gradient Descent (SGD), using the Adam optimiser [KB15]. For training data $X_1 \ldots X_n$, SGD attempts to find the model parameters Θ that minimise the output of a loss function:

$$\Theta = \arg\min_{\Theta} \frac{1}{n} \sum_{i=1}^{n} \ell(X_i, \Theta) \tag{6.2}$$

where loss function $\ell(x, \Theta)$ computes the logarithmic difference between the predicted and expected values.

To reduce training time, multiple inputs are *batched* together and fed into the 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 padding token, 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.

6.3 Experimental Methodology

DeepTune is applied to two heterogeneous compiler-based machine learning tasks and its performance compared to state-of-the-art approaches that use expert selected features.

6.3.1 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 we execute the program to maximise performance?

6.3.1.1 State-of-the-art

In [GWO13], *Grewe et al.* develop a predictive model 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.

6.3.1.2 Expert Chosen Features

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

6.3.1.3 Experimental Setup

The predictive model of *Grewe et al.* [GWO13] is replicated. The same experimental setup is used as in Section 4.6 in which the experiments are extended to a larger set of 71 programs, summarised in Table 6.2a. The programs were evaluated on two CPU-GPU platforms, detailed in Table 6.3a.

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 *Grewe et al.* to predict heterogeneous device mappings for OpenCL kernels.

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

(a) Case Study A: OpenCL Heterogeneous Mapping

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

(b) Case Study B: OpenCL Thread Coarsening Factor

Table 6.2: Benchmark programs.

	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

(a) Case Study A: OpenCL Heterogeneous Mapping

	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

(b) Case Study B: OpenCL Thread Coarsening Factor

Table 6.3: Experimental platforms.

6.3.1.4 DeepTune Configuration

Figure 6.4a shows the 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.

6.3.1.5 Model Evaluation

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

6.3.2 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 architec-

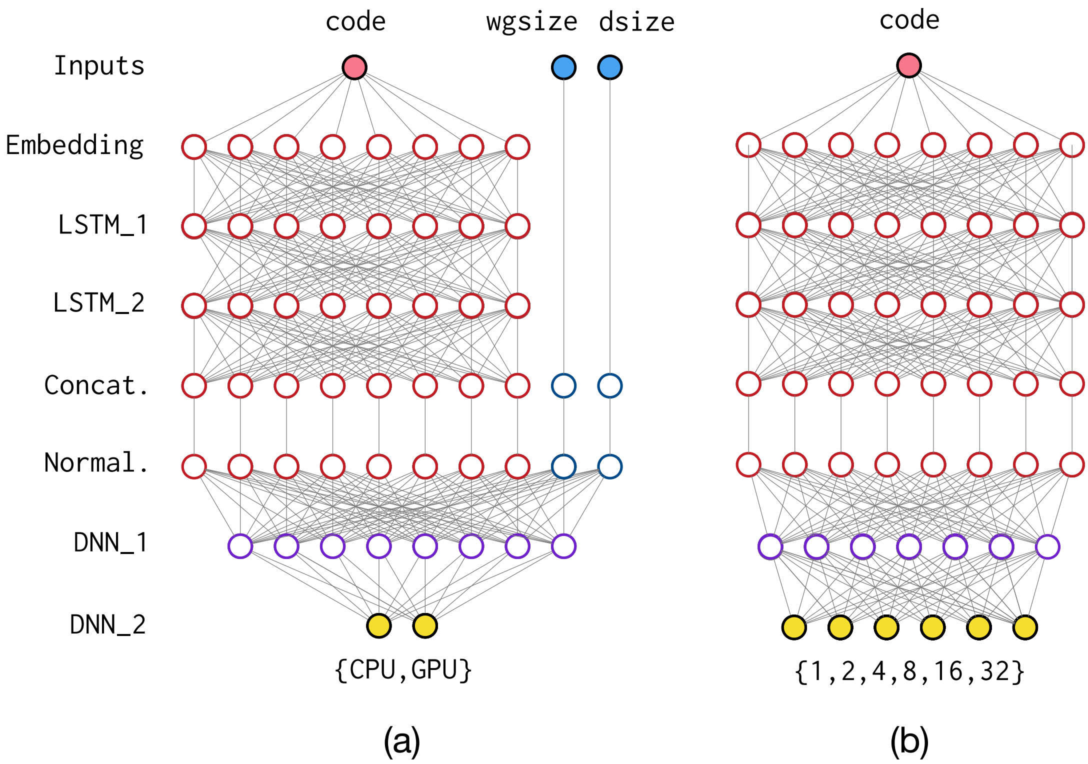
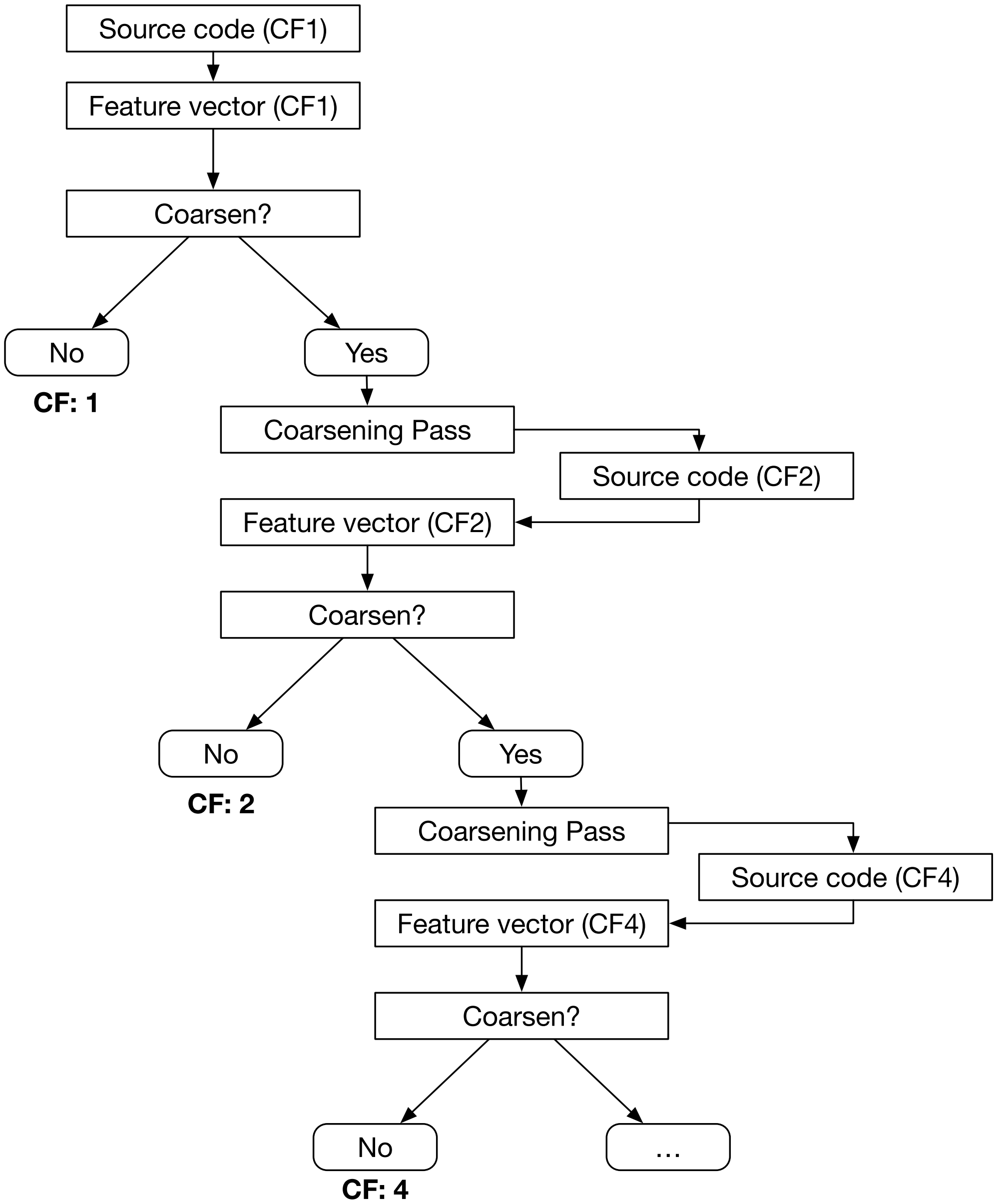


Figure 6.4: DeepTune neural networks, configured for (a) heterogeneous mapping, and (b) thread coarsening factor. The design stays almost the same regardless of the optimisation problem. The only changes are the extra input for (a) and size of the output layers.

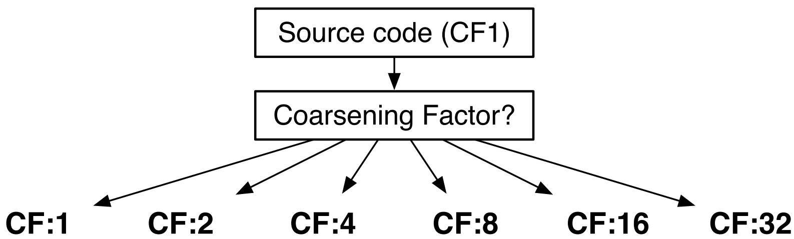
tures.

6.3.2.1 State-of-the-art

Magni et al. present a predictive model for OpenCL thread coarsening in [MDO14]. They implement an iterative heuristic which determines whether a given program would benefit from coarsening. If yes, then the program is coarsened, 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 6.5a. They select from one of six possible coarsening factors: $(1,2,4,8,16,32)$, divided into 5 binary choices.



(a) *Magni et al.* cascading binary model.



(b) Proposed approach.

Figure 6.5: Two approaches for predicting coarsening factor (CF) of OpenCL kernels. *Magni et al.* 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 *Magni et al.* for predicting thread coarsening. From these values, they compute relative deltas for each iteration of coarsening, then use PCA for selection.

6.3.2.2 Expert Chosen Features

Magni et al. followed a very comprehensive feature engineering process. 17 candidate features were assembled from previous studies of performance counters and computed theoretical values [MDO13; Sim+12]. For each candidate feature they compute its coarsening *delta*, reflecting the change in each feature value caused by coarsening: $f_\Delta = (f_{after} - f_{before})/f_{before}$, adding it to the feature set. Then they use Principle Component Analysis (PCA) on the 34 candidates and selected the first 7 principle components, accounting for 95% of variance in the space.

6.3.2.3 Experimental Setup

The experimental setup of *Magni et al.* [MDO14] is replicated. The thread coarsening optimisation is evaluated on 17 programs, listed in Table 6.2b. Four different GPU architectures are used, listed in Table 6.3b.

6.3.2.4 DeepTune Configuration

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

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

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

6.3.3 Comparison of Case Studies

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 6.5 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 Experimental Results

This section evaluates the effectiveness of DeepTune for two distinct optimisation tasks: predicting the optimal device to run a given program, and predicting thread coarsening factors.

First DeepTune is compared against two expert-tuned predictive models, showing that DeepTune outperforms the state-of-the-art in both cases. Section 6.4.3 describes leveraging knowledge learned from training DeepTune for one heuristic to boost training for the other heuristic, further improving performance. Finally, Section 6.4.4 analyses the working mechanism of DeepTune.

6.4.1 Case Study A: OpenCL Heterogeneous Mapping

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 against a static single-device approach and the *Grewe et al.* 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 6.6 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 *Grewe et al.* model achieves an average accuracy of 73%, a significant improvement over the static mapping. By automatically extracting useful feature representations from the source code, DeepTune gives an average accuracy of 82%, an improvement over both schemes.

Using the static mapping as a baseline, the relative performance of each program is computed using the device selected by the *Grewe et al.* and DeepTune models. Figure 6.7 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.

6.4.2 Case Study B: OpenCL Thread Coarsening Factor

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×.

Figure 6.8 shows speedups achieved by the *Magni et al.* and DeepTune models for all programs and platforms. The performance of programs without coarsening is used as baseline. On the four experimental platforms (AMD HD 5900, Tahiti 7970,

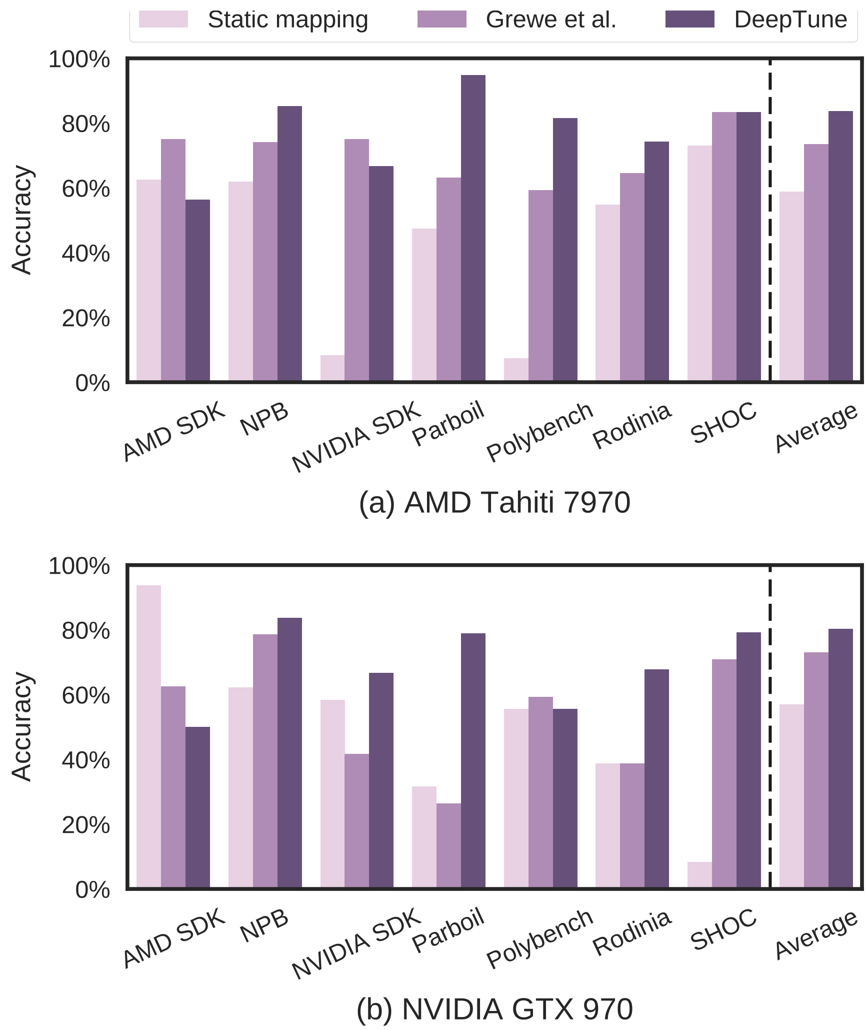


Figure 6.6: Accuracy of optimisation heuristics for heterogeneous device mapping, aggregated by benchmark suite. The optimal static mapping achieves 58% accuracy. The *Grewe et al.* and DeepTune predictive models achieve accuracies of 73% and 84%, respectively.

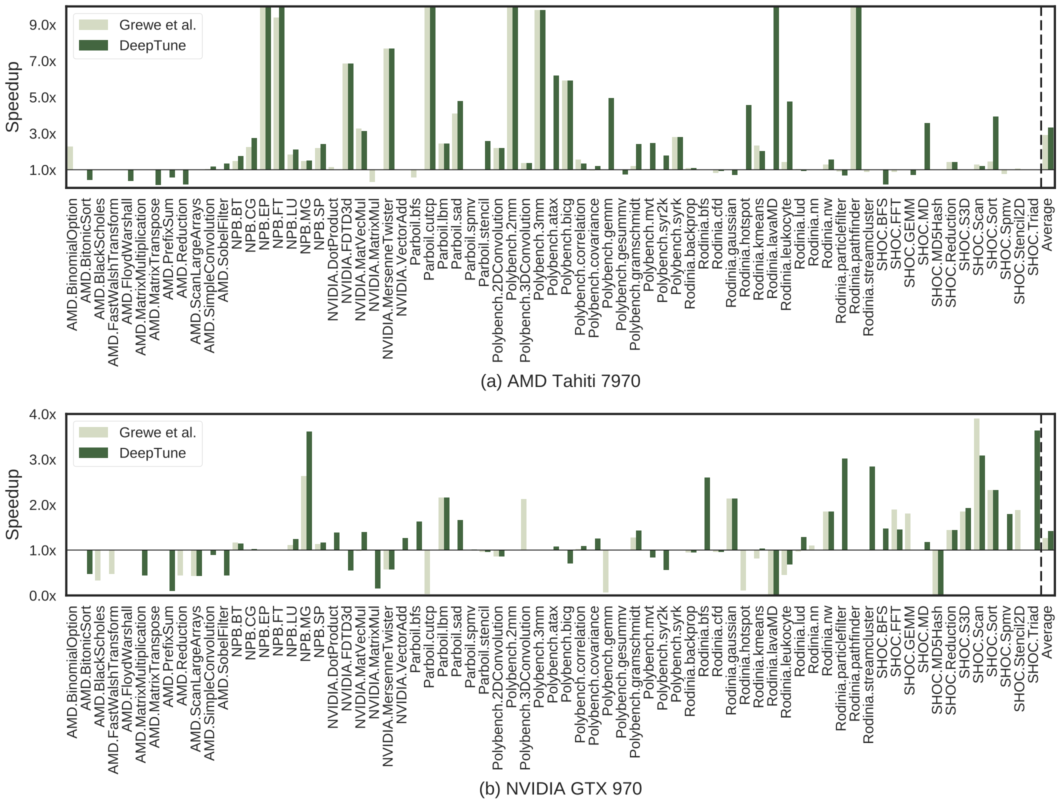


Figure 6.7: Speedup of predicted heterogeneous mappings over the best static mapping for both platforms. In (a) DeepTune achieves an average speedup of 3.43x over static mapping and 18% over *Grewe et al.* In (b) the speedup is 1.42x and 13% respectively.

NVIDIA GTX 480, and Tesla K20c), the *Magni et al.* 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 used by *Magni et al.* (only 17 programs in total). As a result, the models suffer from sparse training data. Chapter 4 presents a methodology for overcoming data sparsity using additional programs; 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.

6.4.3 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 *Grewe et al.* and *Magni et al.* 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 neural network classification, information learned at the early layers of neural networks (i.e. closer to the input layer) will be useful for multiple tasks. The later the network layers are (i.e. closer to the output layer), the more specialised the layers become [ZF14].

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

As shown in Figure 6.8, 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

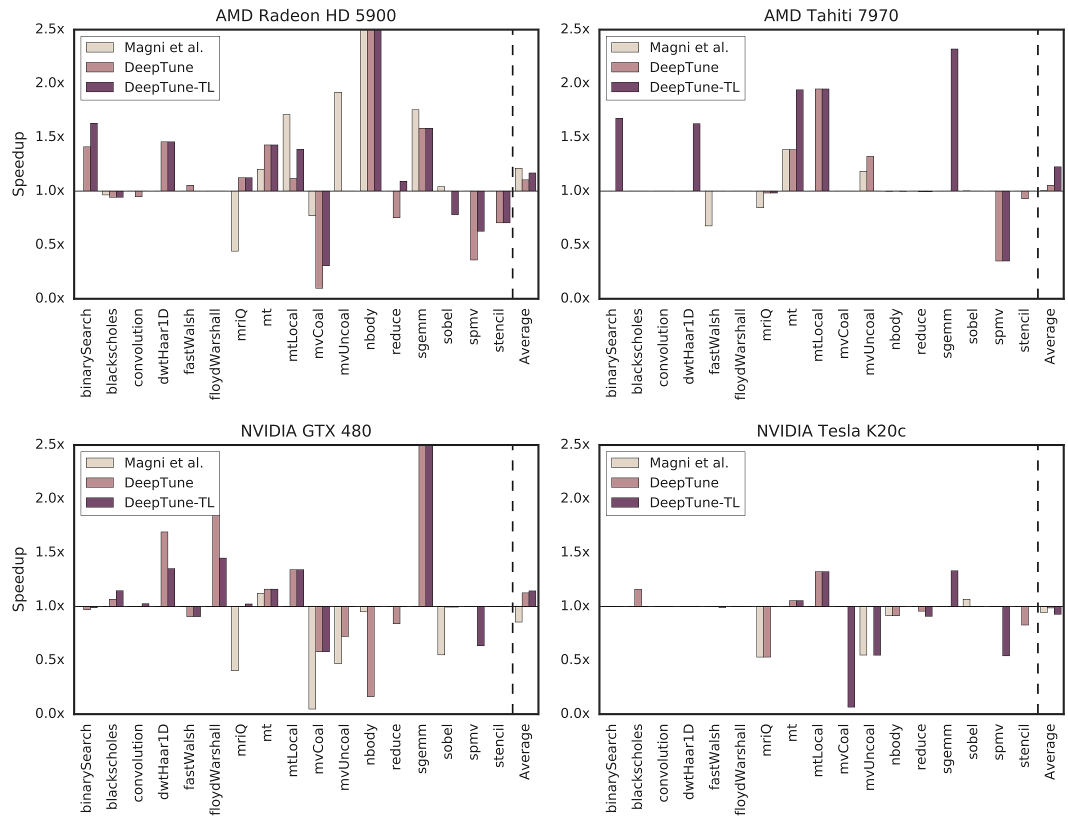


Figure 6.8: Speedups of predicted coarsening factors for each platform. DeepTune outperforms *Magni et al.* on three of the four platforms. Transfer learning improves DeepTune speedups further, by 16% on average.

12% performance improvement over *Magni et al.* 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.

6.4.4 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 subsection attempts to illuminate the inner workings, using a single

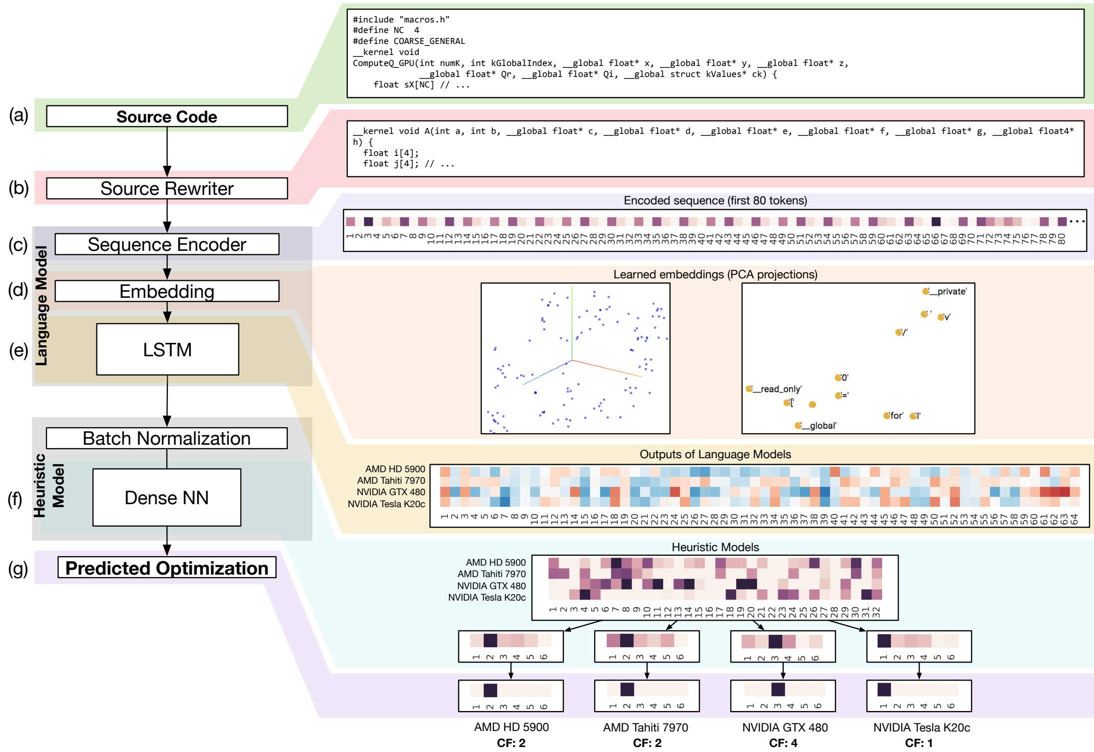


Figure 6.9: Visualising the internal state of DeepTune when predicting coarsening factor for Parboil's `mriQ` benchmark on four different architectures. The activations in each layer of the four models increasingly diverge the lower down the network.

example from Case Study B: predicting the thread coarsening factor for Parboil's `mriQ` benchmark on four different platforms.

Figure 6.9 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 6.9b. The rewritten source code is tokenised and encoded in a 1-of-k vocabulary. Figure 6.9c 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 6.9d shows a PCA projection of the embedding space for one of the platforms, showing multiple clusters of tokens.

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 6.9e 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 experiments in [BJH18] support this reasoning, where a platform-agnostic language model achieves slightly poorer performance.

As information flows through the network, the layers become progressively more specialised to the specific platform. This can be seen in Figure 6.9f, 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 6.9g 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%.

6.5 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

7.1 Contributions

This section summarises the main contributions of this thesis for XXX.

7.1.1 Workload Characterisation

7.1.2 Compiler Optimisations

7.1.3 Compiler Testing

7.2 Critical Analysis

The problems addressed in this thesis are not new.

7.2.1 Limitations of Generative Models

Extensibility to other languages largely untested.

Generating multi-function programs.

Our new approach enables 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 it has limitations. Our method of seeding the language models with the start of a function means that we cannot support user defined types, or calls to user-defined functions. This means that we only consider scalars and arrays as inputs; while 6 (2.3%) of the benchmark kernels from Table 6.2 use irregular data types as inputs. We will address this limitation through recursive program synthesis,

CGO'17 CLgen limitations

whereby a call to a user-defined function or unrecognised type will trigger candidate functions and type definitions to be synthesised. Currently we only run single-kernel benchmarks. We will extend the host driver to explore multi-kernel schedules and interleaving of kernel executions. Our host driver generates data sets from uniform random distributions, as do many of the benchmark suites. For cases where non-uniform inputs are required (e.g. profile-directed feedback), an alternate methodology for generating inputs must be adopted.

Contents of GitHub corpus was only very lightly vetted. I could have used the dynamic checker to ensure that programs work. Inspecting the corpus reveals some spurious programs, e.g. test cases for an OpenCL static analysis tool which deliberately contain runtime defects.

Turing test of 4.5 could be used to pit competing generative models against one another, given sufficient volunteers.

7.2.2 Limitations of Sequential Classification

7.3 Future Work

O'17 con- ions sec-

Our hope for this work is to demonstrate a proof of concept for an exciting new avenue of program generation, and that the full release of CLgen will expedite discovery in other domains. In future work we will extend the approach to multiple programming languages, and investigate methods for performing an automatic directed search of feature spaces.

CT'17 con- ions sec-

In future work, we will extend our heuristic construction approach by automatically learning dynamic features over raw data; apply unsupervised learning techniques [Le+12] over unlabelled source code to further improve learned representations of programs; and deploy trained DeepTune heuristic models to low power embedded systems using quantisation and compression of neural networks [HMD15].

Graph-level representations with RNNS [JJ18]. Graph surveys [Wu+18; ZCZ18].

Table of GitHub corpus size by programming language. There's room for machine learning in lots of languages! E.g. Haskell, Java, C/C++, Solidity, Python, OpenCL

Bibliography

[Aba+16] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. "TensorFlow: A system for large-scale machine learning". In: *arXiv:1605.08695* (2016).

[Aga+06] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. "Using Machine Learning to Focus Iterative Optimization". In: *CGO*. IEEE, 2006.

[All+14] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. "Learning Natural Coding Conventions". In: *FSE*. ACM, 2014.

[All+17] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton. "A Survey of Machine Learning for Big Code and Naturalness". In: *arXiv:1709.06182* (2017).

[All18] M. Allamanis. "The Adverse Effects of Code Duplication in Machine Learning Models of Code". In: *arXiv:1812.06469* (2018).

[Ans+09] A. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. "PetaBricks: A Language and Compiler for Algorithmic Choice". In: *PLDI*. ACM, 2009.

[Ans+13] J. Ansel, S. Kamil, K. Veeramachaneni, U. O. Reilly, and S. Amarasinghe. "OpenTuner: An Extensible Framework for Program Autotuning". In: *PACT*. ACM, 2013.

[Ans14] J. Ansel. "Autotuning Programs with Algorithmic Choice". PhD thesis. Massachusetts Institute of Technology, 2014.

[APS16] M. Allamanis, H. Peng, and C. Sutton. "A Convolutional Attention Network for Extreme Summarization of Source Code". In: *ICML*. 2016.

[AR12] J. Ansel and U. O. Reilly. “SiblingRivalry: Online Autotuning Through Local Competitions”. In: *CASES*. ACM, 2012.

[AS13] M. Allamanis and C. Sutton. “Mining Source Code Repositories at Massive Scale using Language Modeling”. In: *MSR*. 2013.

[AS14] M. Allamanis and C. Sutton. “Mining Idioms from Source Code”. In: *FSE*. ACM, 2014.

[Ash+17] A. H. Ashouri, A. Bignoli, G. Palermo, C. Silvano, S. Kulkarni, and J. Cavazos. “MiCOMP: Mitigating the Compiler Phase-ordering Problem Using Optimization Sub-sequences and Machine Learning”. In: *TACO* (2017).

[Ash+18] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano. “A Survey on Compiler Autotuning using Machine Learning”. In: *CSUR* 51.5 (2018).

[Bad+19] J. Bader, A. Scott, M. Pradel, and S. Chandra. “Getafix: Learning to Fix Bugs Automatically”. In: *arXiv:1902.06111* (2019).

[Bai+14] R. Baishakhi, D. Posnett, V. Filkov, and P. Devanbu. “A Large Scale Study of Programming Languages and Code Quality in Github”. In: *FSE*. ACM, 2014.

[Bai+91] D. H. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. “The NAS Parallel Benchmarks”. In: *IJHPCA* 5.3 (1991).

[Bas+17] O. Bastani, R. Sharma, A. Aiken, and P. Liang. “Synthesizing Program Input Grammars”. In: *PLDI*. 2017.

[BCD12] A. Betts, N. Chong, and A. Donaldson. “GPUVerify: A Verifier for GPU Kernels”. In: *OOPSLA*. ACM, 2012.

[BDB00] V. Bala, E. Duesterwald, and S. Banerjia. “Dynamo: A Transparent Dynamic Optimization System”. In: *PLDI*. New York, NY, USA: ACM, 2000.

[BDK14] M. Baroni, G. Dinu, and G. Kruszewski. “Don’t Count, Predict! A Systematic Comparison of Context-Counting vs . Context-Predicting Semantic Vectors”. In: *ACL*. 2014.

[Ber+10] Josep Ll Berral, Íñigo Goiri, Ramón Nou, Ferran Julià, Jordi Guitart, Ricard Gavaldà, and Jordi Torres. "Towards energy-aware scheduling in data centers using machine learning". In: *e-Energy*. 2010.

[Ber+11] J. Bergstra, F. Bastien, O. Breuleux, P. Lamblin, R. Pascanu, O. Delalleau, G. Desjardins, D. Warde-Farley, I. Goodfellow, A. Bergeron, and Y. Bengio. "Theano: Deep Learning on GPUs with Python". In: *BigLearning Workshop*. 2011.

[BJ05] R. H. Bell and Lizy K. John. "Improved automatic testcase synthesis for performance model validation". In: *SC*. 2005.

[BJH18] T. Ben-nun, A. S. Jakobovits, and T. Hoefler. "Neural Code Comprehension: A Learnable Representation of Code Semantics". In: *NeurIPS*. 2018.

[BJR19] H. Babii, A. Janes, and R. Robbes. "Modeling Vocabulary for Big Code Machine Learning". In: *arXiv:1904.01873* (2019).

[Bod+98] F. Bodin, T. Kisuki, P. M. W. Knijnenburg, M. O'Boyle, and E. Rohou. "Iterative compilation in a non-linear optimisation space". In: *PACT*. ACM, 1998.

[Bow+15] S. R. Bowman, L. Vilnis, O. Vinyals, A. M. Dai, R. Jozefowicz, and S. Bengio. "Generating Sentences from a Continuous Space". In: *arXiv:1511.06349* (2015).

[Bro+18] M. Brockschmidt, M. Allamanis, A. L. Gaunt, and O. Polozov. "Generative Code Modeling with Graphs". In: *arXiv:1805.08490* (2018).

[BS97] A. S. Boujarwah and K. Saleh. "Compiler Test Case Generation Methods: A Survey and Assessment". In: *Information and Software Technology* 39.9 (1997).

[Bun+17] R. Bunel, A. Desmaison, M. P. Kumar, and P. H. S. Torr. "Learning to Superoptimize Programs". In: *ICLR*. 2017.

[Cav+06] J. Cavazos, C. Dubach, F. Agakov, E. Bonilla, M. O'Boyle, G. Fursin, and O. Temam. "Automatic Performance Model Construction for the Fast Software Exploration of New Hardware Designs". In: *CASES*. 2006.

[CFL12] A. Collins, C. Fensch, and H. Leather. "Auto-Tuning Parallel Skeletons". In: *Parallel Processing Letters* 22.02 (June 2012).

[CGA15] A. Chiu, J. Garvey, and T. S. Abdelrahman. "Genesis: A Language for Generating Synthetic Training Programs for Machine Learning". In: *CF*. ACM, 2015.

[Cha+09] C. Chan, H. Ansel, Y. L. Wong, S. Amarasinghe, and A. Edelman. "Autotuning multigrid with PetaBricks". In: *SC*. 2009.

[Che+09] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. "Rodinia: A Benchmark Suite for Heterogeneous Computing". In: *IISWC*. IEEE, Oct. 2009.

[Che+13] Y. Chen, A. Groce, C. Zhang, W. Wong, X. Fern, E. Eide, and J. Regehr. "Taming Compiler Fuzzers". In: *PLDI* (2013).

[Che+16] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie. "An Empirical Comparison of Compiler Testing Techniques". In: *ICSE*. 2016.

[Che+17] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, and B. Xie. "Learning to Prioritize Test Programs for Compiler Testing". In: *ICSE*. 2017.

[Che+18] Z. Chen, S. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, and M. Monperrus. "SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair". In: *arXiv:1901.01808* (2018).

[Che+19] L. Cheng, Y. Zhang, Y. Zhang, C. Wu, Z. Li, Y. Fu, and H. Li. "Optimizing seed inputs in fuzzing with machine learning". In: *arXiv:1902.02538* (2019).

[Cho+17] M. Choi, S. Jeong, H. Oh, and J. Choo. "End-to-End Prediction of Buffer Overruns from Raw Source Code via Neural Memory Networks". In: *arXiv:1703.02458* (2017).

[Col+13] A. Collins, C. Fensch, H. Leather, and M. Cole. "MaSiF: Machine Learning Guided Auto-tuning of Parallel Skeletons". In: *HiPC*. IEEE, 2013.

[CSA18] Milan Cvitkovic, Badal Singh, and Anima Anandkumar. "Deep Learning On Code with an Unbounded Vocabulary". In: *Machine Learning* 4 (2018).

[Cum+15] C. Cummins, P. Petoumenos, M. Steuwer, and H. Leather. "Autotuning OpenCL Workgroup Size for Stencil Patterns". In: *ADAPT*. 2015.

[Cum+16a] C. Cummins, P. Petoumenos, M. Steuwer, and H. Leather. "Autotuning OpenCL Workgroup Size for Stencil Patterns". In: *ADAPT*. 2016.

[Cum+16b] C. Cummins, P. Petoumenos, M. Steuwer, and H. Leather. "Towards Collaborative Performance Tuning of Algorithmic Skeletons". In: *HLPGPU*. 2016.

[Cum+17] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather. "End-to-end Deep Learning of Optimization Heuristics". In: *PACT*. IEEE, 2017.

[Cum15] C. Cummins. "Autotuning Stencils Codes with Algorithmic Skeletons". PhD thesis. University of Edinburgh, 2015.

[CW76] H. J. Curnow and B. A. Wichmann. "A Syntetic Benchmark". In: *Computer* 19.1 (1976).

[Dan+10] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. "The Scalable HeterOgeneous Computing (SHOC) Benchmark Suite". In: *GPGPU*. ACM, 2010.

[DAY19] Y. David, U. Alon, and E. Yahav. "Neural Reverse Engineering of Stripped Binaries". In: *arXiv:1902.09122* (2019).

[DDD18] C. De Boom, B. Dhoedt, and T. Demeester. "Character-level Recurrent Neural Networks in Practice: Comparing Training and Sampling Schemes". In: *arXiv:1801.00632* (2018).

[DEK11] U. Dastgeer, J. Enmyren, and C. W. Kessler. "Auto-tuning SkePU: a Multi-Backend Skeleton Programming Framework for Multi-GPU Systems". In: *IWMSE*. ACM, 2011.

[Din+15] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U. O'Reilly, and S. Amarasinghe. "Autotuning Algorithmic Choice for Input Sensitivity". In: *PLDI*. ACM, 2015.

[DR16] M. Dhok and M. K. Ramanathan. "Directed Test Generation to Detect Loop Inefficiencies". In: *FSE*. ACM, 2016.

[Dub+07] C. Dubach, J. Cavazos, B. Franke, G. Fursin, M. O'Boyle, and O. Temam. "Fast Compiler Optimisation Evaluation Using Code-Feature Based Performance Prediction". In: *CF*. ACM, 2007.

[Dub+09] C. Dubach, T. M. Jones, E. V. Bonilla, G. Fursin, and M. O'Boyle. "Portable Compiler Optimisation Across Embedded Programs and Microarchitectures using Machine Learning". In: *MICRO*. ACM, 2009.

[EK10] J Enmyren and CW Kessler. "SkePU: a multi-backend skeleton programming library for multi-GPU systems". In: *HLPP*. ACM, 2010.

[FE15] T. L. Falch and A. C. Elster. "Machine Learning Based Auto-tuning for Enhanced OpenCL Performance Portability". In: *IPDPSW*. IEEE, 2015.

[Fer+12] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware". In: *ASPLOS*. ACM, 2012.

[FT10] G. Fursin and O. Temam. "Collective Optimization: A Practical Collaborative Approach". In: *TACO* 7.4 (2010).

[Fur+11] G. Fursin, Y. Kashnikov, A. W. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois, F. Bodin, P. Barnard, E. Ashton, E. Bonilla, J. Thomson, C. K. I. Williams, and M. O'Boyle. "Milepost GCC: Machine Learning Enabled Self-tuning Compiler". In: *IJPP* 39.3 (2011).

[Fur+14] G. Fursin, R. Miceli, A. Lokhmotov, M. Gerndt, M. Baboulin, A. D. Malony, Z. Chamski, D. Novillo, and D. Del Vento. "Collective Mind: Towards practical and collaborative auto-tuning". In: *Scientific Programming* 22.4 (2014).

[FVS11] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. "A Comprehensive Performance Comparison of CUDA and OpenCL". In: *ICPP*. IEEE, 2011.

[GA15] J. D. Garvey and T. S. Abdelrahman. "Automatic Performance Tuning of Stencil Computations on GPUs". In: *ICPP*. IEEE, 2015.

[GAL14] E. Guzman, D. Azócar, and Y. Li. "Sentiment Analysis of Commit Comments in GitHub: an Empirical Study". In: *MSR*. 2014.

[Gan+09] A. Ganapathi, K. Datta, A. Fox, and D. Patterson. "A Case for Machine Learning to Optimize Multicore Performance". In: *HotPar*. 2009.

[Gao+15] H. Gao, J. Mao, J. Zhou, Z. Huang, L. Wang, and W. Xu. “Are You Talking to a Machine? Dataset and Methods for Multilingual Image Question Answering”. In: *arXiv:1505.05612* (2015).

[GEB15] L. A. Gatys, A. S. Ecker, and M. Bethge. “A Neural Algorithm of Artistic Style”. In: *arXiv:1508.06576* (2015).

[Geo+18] K. Georgiou, C. Blackmore, S. Xavier-de-Souza, and K. Eder. “Less is More: Exploiting the Standard Compiler Optimization Levels for Better Performance and Energy Consumption”. In: *SCOPES*. 2018.

[GH11] Chris Gregg and Kim Hazelwood. “Where is the data? Why you cannot debate CPU vs. GPU performance without the answer”. In: *ISPASS*. 2011.

[Goo+14] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. “Generative Adversarial Networks”. In: *arXiv:1406.2661* (2014).

[Gos+10] N. Goswami, R. Shankar, M. Joshi, and T. Li. “Exploring GPGPU workloads: Characterization methodology, analysis and microarchitecture evaluation implications”. In: *IISWC*. 2010.

[GPS17] P. Godefroid, H. Peleg, and R. Singh. “Learn&Fuzz: Machine Learning for Input Fuzzing”. In: *ASE*. 2017.

[Gra+12] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. “Auto-tuning a High-Level Language Targeted to GPU Codes”. In: *InPar*. 2012.

[Gra13] A. Graves. “Generating Sequences with Recurrent Neural Networks”. In: *arXiv:1308.0850* (2013).

[Gre+15a] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber. “LSTM: A Search Space Odyssey”. In: *arXiv:1503.04069* (2015).

[Gre+15b] K. Gregor, I. Danihelka, A. Graves, D. J. Rezende, and D. Wierstra. “DRAW: A Recurrent Neural Network For Image Generation”. In: *arXiv:1502.04623* (2015).

[GS05] A. Graves and J. Schmidhuber. “Framewise Phoneme Classification with Bidirectional LSTM and Other Neural Network Architectures”. In: *Neural Networks* 5.5 (2005).

[Gu+16] X. Gu, H. Zhang, D. Zhang, and S. Kim. "Deep API Learning". In: *FSE*. ACM, 2016.

[GWO13] D. Grewe, Z. Wang, and M. O'Boyle. "Portable Mapping of Data Parallel Programs to OpenCL for Heterogeneous Systems". In: *CGO*. IEEE, 2013.

[He+16] K. He, X. Zhang, S. Ren, and J. Sun. "Deep Residual Learning for Image Recognition". In: *CVPR*. IEEE, 2016.

[Hen+18] J. Henkel, S. K. Lahiri, B. Liblit, and T. Reps. "Code Vectors: Understanding Programs Through Embedded Abstracted Symbolic Traces". In: *FSE*. 2018.

[HHZ12] C. Holler, K. Herzig, and A. Zeller. "Fuzzing with Code Fragments". In: *Usenix* (2012).

[HKP11] J. Han, M. Kamber, and J. Pei. *Data mining: concepts and techniques*. Elsevier, 2011.

[HLZ16] X. Huo, M. Li, and Z. Zhou. "Learning Unified Features from Natural and Programming Languages for Locating Buggy Source Code". In: *IJCAI*. 2016.

[HMD15] S. Han, H. Mao, and W. J. Dally. "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding". In: *arXiv:1510.00149* (2015).

[HOY17] K. Heo, H. Oh, and K. Yi. "Machine-Learning-Guided Selectively Unsound Static Analysis". In: *ICSE*. 2017.

[HS97] S. Hochreiter and J. Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (1997).

[HSN] H. Hata, E. Shihab, and G. Neubig. "Learning to Generate Corrective Patches using Neural Machine Translation". In: *arXiv:1812.07170* ().

[IS15] S. Ioffe and C. Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *arXiv:1502.03167* (2015).

[Jia+10] Y. Jiang, Z. Z. Zhang, K. Tian, F. Mao, M. Gethers, X. Shen, and Y. Gao. "Exploiting Statistical Correlations for Proactive Prediction of Program Behaviors". In: *CGO* (2010).

[JJ18] Y. Jin and J. F. JaJa. “Learning Graph-Level Representations with Recurrent Neural Networks”. In: *arXiv:1805.07683* (2018).

[JNR02] R. Joshi, G. Nelson, and K. Randall. “Denali: a goal-directed superoptimizer”. In: *PLDI*. ACM, 2002.

[Jos+08] A. M. Joshi, L. Eeckhout, L. K. Johnz, and C. Isen. “Automated microprocessor stressmark generation”. In: *HPCA*. 2008.

[Jou+17] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. Mackean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and G. H. Yoon. “In-Datacenter Performance Analysis of a Tensor Processing Unit TM”. In: *ISCA*. 2017.

[Joz+16] R. Jozefowicz, O. Vinyals, M. Schuster, N. Shazeer, and Y. Wu. “Exploring the Limits of Language Modeling”. In: *arXiv:1602.02410* (2016).

[Kal+09] E. Kalliamvakou, L. Singer, G. Gousios, D. M. German, K. Blincoe, and D. Damian. “The Promises and Perils of Mining GitHub”. In: *MSR*. 2009.

[KB15] D. P. Kingma and J. L. Ba. “Adam: a Method for Stochastic Optimization”. In: *ICLR* (2015).

[KC12] S. Kulkarni and J. Cavazos. “Mitigating the Compiler Optimization Phase-Ordering Problem using Machine Learning”. In: *OOPSLA*. ACM, 2012.

[Koc+17] U. Koc, P. Saadatpanah, J. S. Foster, and A. A. Porter. “Learning a Classifier for False Positive Error Reports Emitted by Static Code Analysis Tools”. In: *MAPL*. 2017.

[Kom+10] Kazuhiko Komatsu, Katsuto Sato, Yusuke Arai, Kentaro Koyama, Hiroyuki Takizawa, and Hiroaki Kobayashi. "Evaluating performance and portability of OpenCL programs". In: *iWAPT*. 2010.

[Kou+17] M. Koukoutos, M. Raghothaman, E. Kneuss, and V. Kuncak. "On Repair with Probabilistic Attribute Grammars". In: *arXiv:1707.04148* (2017).

[KP05] A. S. Kossatchev and M. A. Posypkin. "Survey of Compiler Testing Methods". In: *Programming and Computer Software* 31.1 (2005).

[Kra+18] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. "The Case for Learned Index Structures". In: *SIGMOD*. ACM, 2018.

[Kri+18] S. Krishnan, Z. Yang, K. Goldberg, J. M. Hellerstein, and I. Stoica. "Learning to Optimize Join Queries With Deep Reinforcement Learning". In: *arXiv:1808.03196* (2018).

[KSH12] A. Krizhevsky, I. Sutskever, and G. E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *NIPS*. 2012.

[Lam+15] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. "Combining Deep Learning with Information Retrieval to Localize Buggy Files for Bug Reports". In: *ASE*. 2015.

[LAS14] V. Le, M. Afshari, and Z. Su. "Compiler Validation via Equivalence Modulo Inputs". In: *PLDI*. 2014.

[LBE15] Z. C. Lipton, J. Berkowitz, and C. Elkan. "A Critical Review of Recurrent Neural Networks for Sequence Learning". In: *arXiv:1506.00019* (2015).

[LBH15] Y. LeCun, Y. Bengio, and G. Hinton. "Deep learning". In: *Nature* 521.7553 (2015).

[LBO14] H. Leather, E. Bonilla, and M. O'Boyle. "Automatic Feature Generation for Machine Learning Based Optimizing Compilation". In: *TACO* 11.1 (2014).

[LCW12] T. Lozano-Perez, I. J. Cox, and G. T. Wilfong. *Autonomous Robot Vehicles*. Springer, 2012.

[Le+12] Q. V. Le, R. Monga, M. Devin, G. Corrado, K. Chen, M. A. Ranzato, J. Dean, and A. Y. Ng. "Building High-level Features Using Large Scale Unsupervised Learning". In: *ICML*. 2012.

[Lee+09] H. Lee, Y. Largman, P. Pham, and A. Y. Ng. "Unsupervised Feature Learning for Audio Classification using Convolutional Deep Belief Networks". In: *NIPS*. 2009.

[Lee+10] V. W. Lee, P. Hammarlund, R. Singhal, P. Dubey, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, S. Satish, M. Smelyanskiy, and S. Chennupaty. "Debunking the 100X GPU vs. CPU myth". In: *ACM SIGARCH Computer Architecture News* 38 (2010).

[Lem+18] C. Lemieux, R. Padhye, K. Sen, and D. Song. "PerfFuzz: Automatically Generating Pathological inputs". In: *ISSTA*. ACM, 2018.

[Ler17] X. Leroy. *The CompCert C Verified Compiler*. 2017.

[LFC13] T. Lutz, C. Fensch, and M. Cole. "PARTANS: An Autotuning Framework for Stencil Computation on Multi-GPU Systems". In: *TACO* 9.4 (2013).

[Lid+15] C. Lidbury, A. Lascu, N. Chong, and A. Donaldson. "Many-Core Compiler Fuzzing". In: *PLDI*. 2015.

[Liu+19] X. Liu, X. Li, R. Prajapati, and D. Wu. "DeepFuzz: Automatic Generation of Syntax Valid C Programs for Fuzz Testing". In: *AAAI*. 2019.

[Mas87] H. Massalin. "Superoptimizer – A Look at the Smallest Program". In: *ASPLOS*. ACM, 1987.

[Mat+19] B. Mathis, A. Kampmann, R. Gopinath, M. Höschele, M. Mera, and A. Zeller. "Parser-Directed Fuzzing". In: *PLDI*. 2019.

[McK98] W. M. McKeeman. "Differential Testing for Software". In: *DTJ* 10.1 (1998).

[MDO13] A. Magni, C. Dubach, and M. O'Boyle. "A Large-Scale Cross-Architecture Evaluation of Thread-Coarsening". In: *SC*. 2013.

[MDO14] A. Magni, C. Dubach, and M. O'Boyle. "Automatic Optimization of Thread-Coarsening for Graphics Processors". In: *PACT*. ACM, 2014.

[MF13] A. W. Memon and G. Fursin. "Crowdtuning: systematizing auto-tuning using predictive modeling and crowdsourcing". In: *PARCO*. 2013.

[Mik+13] T. Mikolov, K. Chen, G. Corrado, and J. Dean. "Distributed Representations of Words and Phrases and their Compositionality". In: *NIPS*. 2013.

[Mik10] T. Mikolov. "Recurrent Neural Network based Language Model". In: *Interspeech*. 2010.

[Mni+15] V. Mnih, K. Kavukcuoglu, D. Silver, A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015).

[Mon18] M. Monperrus. "Automatic Software Repair: a Bibliography". In: *CSUR* 51.1 (2018).

[MPL16] P. Mpeis, P. Petoumenos, and H. Leather. "Iterative compilation on mobile devices". In: *ADAPT*. 2016.

[MS10] J. Misra and I. Saha. "Artificial neural networks in hardware: A survey of two decades of progress". In: *Neurocomputing* 74.1-3 (2010).

[MSD16] P. Micolet, A. Smith, and C. Dubach. "A Machine Learning Approach to Mapping Streaming Workloads to Dynamic Multicore Processors". In: *LCTES*. ACM, 2016.

[Nam+10] M. Namolaru, A. Cohen, G. Fursin, A. Zaks, and A. Freund. "Practical Aggregation of Semantical Program Properties for Machine Learning Based Optimization". In: *CASES*. 2010.

[NC15] C. Nugteren and V. Codreanu. "CLTune: A Generic Auto-Tuner for OpenCL Kernels". In: *MCSoC*. 2015.

[NH10] V. Nair and G. E. Hinton. "Rectified Linear Units Improve Restricted Boltzmann Machines". In: *ICML*. 2010.

[NHI13] E. Nagai, A. Hashimoto, and N. Ishiura. "Scaling up Size and Number of Expressions in Random Testing of Arithmetic Optimization of C Compilers". In: *SASIMI*. 2013.

[NLS16] A. Neelakantan, Q. V. Le, and I. Sutskever. "Neural Programmer: Inducing Latent Programs with Gradient Descent". In: *ICLR*. 2016.

[Nvi07] Nvidia. *Compute unified device architecture programming guide*. Tech. rep. 2007.

[Oda+15] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura. "Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation". In: *ASE*. IEEE, 2015.

[Ogi+14] W. F. Ogilvie, P. Petoumenos, Z. Wang, and H. Leather. "Fast Automatic Heuristic Construction Using Active Learning". In: *LCPC*. 2014.

[Ogi+17] W. F. Ogilvie, P. Petoumenos, Z. Wang, and H. Leather. "Minimizing the cost of iterative compilation with active learning". In: *CGO* (2017).

[Oul+08] E. Ould-Ahmed-Vall, K. A. Doshi, C. Yount, and J. Woodlee. "Characterization of SPEC CPU2006 and SPEC OMP2001: Regression models and their transferability". In: *ISPASS*. IEEE, 2008.

[Owe+06] John D. Owens, David Luebke, Naga Govindraju, Mark Harris, Jens Kruger, Aaron E Lefohn, and Timothy J Purcell. "A Survey of General Purpose Computation on Graphics Hardware". In: *Computer Graphics Forum*. 2006.

[Pan+18] R. Panda, S. Song, J. Dean, and L. K. John. "Wait of a Decade: Did SPEC CPU 2017 Broaden the Performance Horizon?" In: *HPCA*. IEEE, 2018.

[Pas+17] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. "Automatic differentiation in PyTorch". In: (2017).

[PCA12] E. Park, J. Cavazos, and M. A. Alvarez. "Using Graph-Based Program Characterization for Predictive Modeling". In: *CGO*. IEEE, 2012.

[PDL16] M. Pflanzer, A. Donaldson, and A. Lascu. "Automatic Test Case Reduction for OpenCL". In: *IWOCL*. 2016.

[PE06] Z. Pan and R. Eigenmann. "Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning ". In: *CGO*. IEEE, 2006.

[Ped+18] L. Pedrosa, R. Iyer, A. Zaostrovnykh, J. Fietz, and K. Argyraki. "Automated Synthesis of Adversarial Workloads for Network Functions". In: *SIGCOMM*. ACM, 2018.

[Pet+17] T. Petsios, J. Zhao, A. D. Keromytis, and Suman Jana. "SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities". In: *CCS*. ACM, 2017.

[PJ13] S. Purini and L. Jain. “Finding Good Optimization Sequences Covering Program Space”. In: *TACO* (2013).

[PJJ07] Aashish Phansalkar, Ajay Joshi, and Lizy K. John. “Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite”. In: *ACM SIGARCH Computer Architecture News* 35.2 (2007).

[PM15] J. Price and S. Mcintosh-Smith. “Oclgrind: An Extensible OpenCL Device Simulator”. In: *IWOCL*. ACM, 2015.

[PMB13] R. Pacanu, T. Mikolov, and Y. Bengio. “On the Difficulties of Training Recurrent Neural Networks”. In: *ICML*. 2013.

[PS18] M. Pradel and K. Sen. “DeepBugs: A Learning Approach to Name-based Bug Detection”. In: *OOPSLA*. 2018.

[PSP18] H. Peng, Y. Shoshitaishvili, and M. Payer. “T-Fuzz: Fuzzing by Program Transformation”. In: *SP*. 2018.

[Rag+16] M. Raghu, B. Poole, J. Kleinberg, S. Ganguli, and J. Sohl-Dickstein. “On the expressive power of deep neural networks”. In: *arXiv:1606.05336* (2016).

[Reg+12] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. “Test-Case Reduction for C Compiler Bugs”. In: *PLDI*. 2012.

[RJS17] A. Radford, R. Jozefowicz, and I. Sutskever. “Learning to Generate Reviews and Discovering Sentiment”. In: *arXiv:1704.01444* (2017).

[Rul+10] S. Rul, H. Vandierendonck, J. D. Haene, and K. D. Bosschere. “An Experimental Study on Performance Portability of OpenCL Kernels”. In: *SAAHPC*. 2010.

[RVY14] V. Raychev, M. Vechev, and E. Yahav. “Code Completion with Statistical Language Models”. In: *PLDI*. 2014.

[Ryo+08a] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu. “Optimization principles and application performance evaluation of a multithreaded GPU using CUDA”. In: *PPoPP*. 2008.

[Ryo+08b] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S. Ueng, J. A. Stratton, and W. W. Hwu. “Program optimization space pruning for a multithreaded GPU”. In: *CGO*. IEEE, 2008.

[Ryo+15] J. H. Ryoo, S. J. Quirem, M. Lebeane, R. Panda, S. Song, and L. K. John. "GPGPU Benchmark Suites: How Well Do They Sample the Performance Spectrum?" In: *ICPP* (2015).

[SA05] M. Stephenson and S. Amarasinghe. "Predicting Unroll Factors Using Supervised Classification". In: *CGO*. IEEE, 2005.

[SD16] T. Sorensen and A. Donaldson. "Exposing Errors Related to Weak Memory in GPU Applications". In: *PLDI*. 2016.

[SD17] M. Steuwer and C. Dubach. "Lift: A Functional Data-Parallel IR for High-Performance GPU Code Generation". In: *CGO*. IEEE, 2017.

[SGS10] John E. Stone, David Gohara, and Guochun Shi. "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems". In: *CS&E* 12.3 (2010).

[She+18] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana. "NEUZZ: Efficient Fuzzing with Neural Program Learning". In: *arXiv:1807.05620* (2018).

[Si+18] X. Si, H. Dai, M. Raghothaman, M. Naik, and L. Song. "Learning Loop Invariants for Program Verification". In: *NeurIPS*. 2018.

[Sim+12] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc. "A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications". In: *PPoPP*. ACM, 2012.

[SJL11] S. Seo, G. Jo, and J. Lee. "Performance Characterization of the NAS Parallel Benchmarks in OpenCL". In: *IISWC*. IEEE, 2011.

[SLS16] C. Sun, V. Le, and Z. Su. "Finding Compiler Bugs via Live Code Mutation". In: *OOPSLA*. 2016.

[SMR03] M. Stephenson, M. Martin, and U. O. Reilly. "Meta Optimization: Improving Compiler Heuristics with Machine Learning". In: *PLDI*. 2003.

[SSN12] M. Sundermeyer, R. Schl, and H. Ney. "LSTM Neural Networks for Language Modeling". In: *Interspeech*. 2012.

[Str+12] J. A. Stratton, C. Rodrigues, I. Sung, N. Obeid, L. Chang, N. Anssari, G. D. Liu, and W. W. Hwu. "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing". In: *Center for Reliable and High-Performance Computing* (2012).

[SVL14] I. Sutskever, O. Vinyals, and Q. V. Le. "Sequence to Sequence Learning with Neural Networks". In: *NIPS*. 2014.

[TA03] S. Triantafyllis and D. I. August. "Compiler Optimization-Space Exploration". In: *CGO*. IEEE, 2003.

[TC13] M. Tartara and S. Crespi Reghizzi. "Continuous learning of compiler heuristics". In: *TACO* 9.4 (Jan. 2013).

[Tin+16] P. Ting, C. Tu, P. Chen, Y. Lo, and S. Cheng. "FEAST: An Automated Feature Selection Framework for Compilation Tasks". In: *arXiv:1610.09543* (2016).

[TPG18] L. D. Toffola, M. Pradel, and T. R. Gross. "Synthesizing programs that expose performance bottlenecks". In: *CGO*. 2018.

[Tuf+19] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk. "On Learning Meaningful Code Changes via Neural Machine Translation". In: *arXiv:1901.09102* (2019).

[TV16] P. Terence and J. Vinju. "Towards a Universal Code Formatter through Machine Learning". In: *SLE*. 2016.

[Vas+19] M. Vasic, A. Kanade, P. Maniatis, D. Bieber, and R. Singh. "Neural Program Repair by Jointly Learning to Localize and Repair". In: *ICLR*. 2019.

[VFS15] B. Vasilescu, V. Filkov, and A. Serebrenik. "Perceptions of Diversity on GitHub: A User Survey". In: *Chase* (2015).

[Vin+15] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. "Show and Tell: A Neural Image Caption Generator". In: *CVPR* (2015).

[Wan+14] Z. Wang, G. Tournavitis, B. Franke, and M. O'Boyle. "Integrating Profile-driven Parallelism Detection and Machine-learning-based Mapping". In: *TACO* (2014).

[Wan+17] J. Wang, B. Chen, L. Wei, and Y. Liu. "Skyfire: Data-Driven Seed Generation for Fuzzing". In: *S&P*. 2017.

[Wei+18] J. Wei, J. Chen, Y. Feng, K. Ferles, and Isil Dillig. "Singularity: Pattern Fuzzing for Worst Case Complexity". In: *ESEC/FSE*. ACM, 2018.

[Whi+15] M. White, C. Vendome, M. Linares-Vasquez, and D. Poshyvanyk. "Toward Deep Learning Software Repositories". In: *MSR*. 2015.

[Whi+19] M. White, M. Tufano, M. Martínez, M. Monperrus, and D. Poshyvanyk. "Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities". In: *SANER*. 2019.

[WO09] Z. Wang and M. O'Boyle. "Mapping Parallelism to Multi-cores: A Machine Learning Based Approach". In: *PPoPP*. 15. ACM, 2009.

[WO10] Z. Wang and M. O'Boyle. "Partitioning Streaming Parallelism for Multi-cores: A Machine Learning Based Approach". In: *PACT*. ACM, 2010.

[WO18] Z. Wang and M. O'Boyle. "Machine learning in Compiler Optimization". In: *Proceedings of the IEEE* 1.23 (2018).

[WRX17] H. Wang, B. Raj, and E. P. Xing. "On the Origin of Deep Learning". In: *arXiv:1702.07800* (2017).

[WSS17] K. Wang, R. Singh, and Z. Su. "Dynamic Neural Program Embeddings for Program Repair". In: *arXiv:1711.07163* (2017).

[Wu+14] Y. Wu, J. Kropczynski, P. C. Shih, and J. M. Carroll. "Exploring the Ecosystem of Software Developers on GitHub and Other Platforms". In: *CSCW*. 2014.

[Wu+18] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. "A Comprehensive Survey on Graph Neural Networks". In: *arXiv:1901.00596* (2018).

[WWO14] Y. Wen, Z. Wang, and M. O'Boyle. "Smart Multi-Task Scheduling for OpenCL Programs on CPU/GPU Heterogeneous Platforms". In: *HiPC*. IEEE, 2014.

[WYT13] E. Wong, J. Yang, and L. Tan. "AutoComment: Mining Question and Answer Sites for Automatic Comment Generation". In: *ASE*. IEEE, 2013.

[Xio+13] W. Xiong, Z. Yu, Z. Bei, J. Zhao, F. Zhang, Y. Zou, X. Bai, Y. Li, and C. Xu. "A Characterization of Big Data Benchmarks". In: *Big Data*. IEEE, 2013.

[Xio+16] W. Xiong, J. Droppo, X. Huang, F. Seide, M. Seltzer, A. Stolcke, D. Yu, and G. Zweig. "Achieving Human Parity in Conversational Speech Recognition". In: *arXiv:1610.05256* (2016).

[Yan+11] X. Yang, Y. Chen, E. Eide, and J. Regehr. "Finding and Understanding Bugs in C Compilers". In: *PLDI*. 2011.

[Yin+18] P. Yin, G. Neubig, M. Allamanis, M. Brockschmidt, and A. L. Gaunt. "Learning to Represent Edits". In: *arXiv:1810.13337* (2018).

[Yos+14] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson. "How Transferable are Features in Deep Neural Networks?" In: *NIPS*. 2014.

[Zal] M. Zalewski. *American Fuzzy Lop*.

[ZCZ18] Z. Ziwei, P. Cui, and W. Zhu. "Deep Learning on Graphs: A Survey". In: *arXiv:1812.04202* (2018).

[ZF14] M. D. Zeiler and R. Fergus. "Visualizing and Understanding Convolutional Networks". In: *ECCV*. 2014.

[ZIE16] R. Zhang, P. Isola, and A. A. Efros. "Colorful Image Colorization". In: *arXiv:1603.08511* (2016).

[ZSS17] Q. Zhang, C. Sun, and Z. Su. "Skeletal Program Enumeration for Rigorous Compiler Testing". In: *PLDI*. 2017.