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

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

Chapter ?? 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 ?? 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 $V(G,D)$:

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

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-

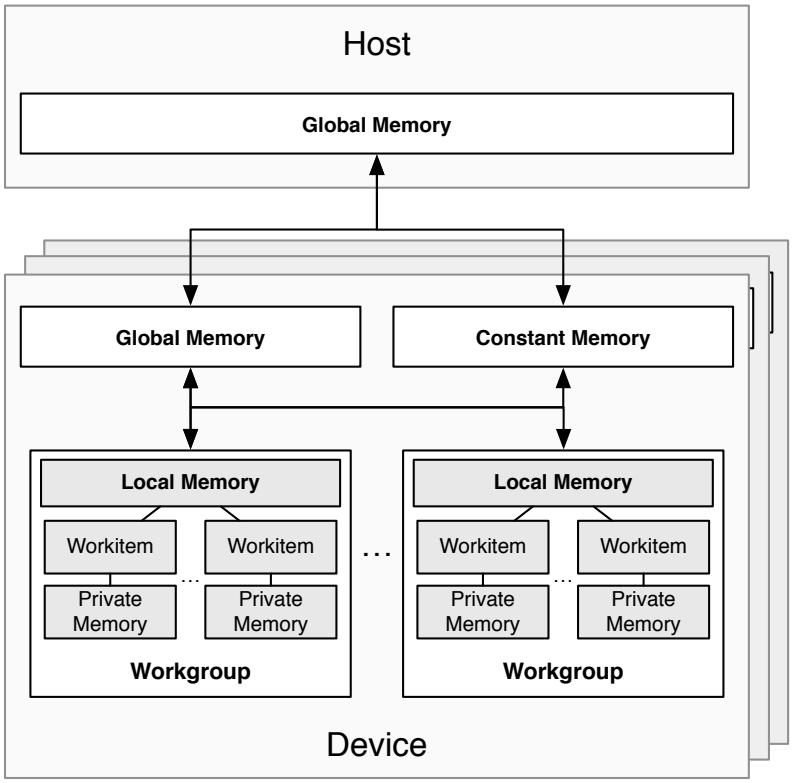


Figure 2.1: The OpenCL memory model. The host communicates with each device through transfers between global memory spaces. The capacity of each type of memory is dependent on the device hardware. In general, private memory is the fastest and smallest, and global memory is the largest and slowest.

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 produc-

tions, 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.

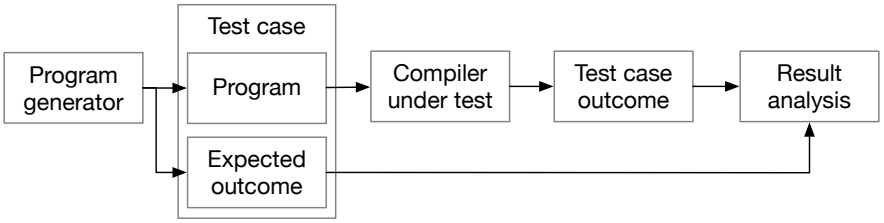
The complexity of optimizing compilers and programming languages renders formal verification of the entire compiler prohibitively expensive. Efforts have been made in this direction, for example CompCert [Ler17], a formally verified compiler for the C programming language, but this comes at the cost 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.

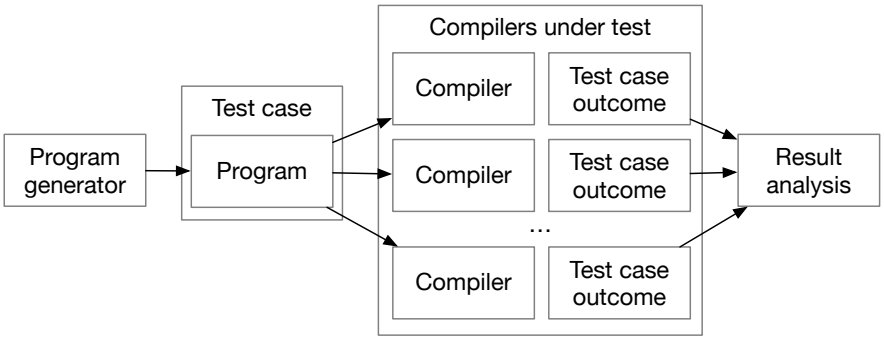
The random generation of programs to test compilers is a well established approach to the compiler validation problem. The main question of interest is in how to efficiently generate codes which trigger bugs. There are two main approaches: *program generation*, where inputs are synthesised from scratch;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

ical. 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+16].

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 compi-

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

As with iterative compilation, the main problem is in pruning and efficiently navigating the search space. In practise, Massalin found their system to scale only to subroutines typically less than 13 instructions [Mas87]. As such, 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× 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 the technique presented in this thesis 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. **Chen2019** review some of the proposed techniques [**Chen2019**] 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]. Henkel et al. use symbolic execution to abstract the program traces. Embeddings are then learned from these abstracted symbolic traces [Hen+18]. Yin et al. and Tufano et al. present techniques for learning representations of code edits [Tuf+19; Yin+18].

This feels o of place

Neural Code Comprehension builds on the experiments in Chapter **??** 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 for Programming Languages

Deep learning techniques for program generation and optimisation were reviewed in Section 3.2.2.3 and Section 3.3.2 respectively, but there have been many other applications of machine 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].

A 2017 survey by Allamanis et al. describes fast moving field of machine learning techniques for programming languages [All+17]. *AutoComment* mines the popular Q&A site StackOverflow to automatically generate code comments [WYT13]. *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 [RVK15]. Allamanis, Peng, and Sutton use attentional neural networks to generate summaries of source code [APS16]. *Nero* uses an encode-decoder architecture to predict method names in stripped binaries. The system takes as input a sequence of call sites from the execution of a binary and produces as output a predicted method name [**David2019**].

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

VFS15; Wu+14]. Allamanis raises concern about code duplicates in corpora of open-source programs used for machine learning [All18]. They find that corpora often contain a high percentage of duplicate or near-duplicate code. This impacts cases where the corpus is divided into training and test sets. Duplicate code appearing both in the training and test sets can lead to artificially high accuracies of models on the test set. The work in this thesis does not use open source corpus as a test set.

Machine learning has also been applied to other areas such as bug detection and static analysis. Heo, Oh, and Yi present a machine-learning technique to tune static analysis to be selectively unsound, based on anomaly detection techniques [HOY17]. Koc et al. present a classifier that attempts to predict whether a static analysis tool's error report is a false positive based on the program structures of previous reports that produced false error reports [Koc+17]. Lam et al. employ neural networks to relate keywords in bug reports to code tokens and terms in source files and documentation to accelerate bug localisation. **Wang2016c** employ a Deep Belief Network (DBN) to automatically learn semantic features from token vectors extracted from programs' Abstract Syntax Trees (ASTs). The features are then used for automatic defect detection [**Wang2016c**]. Chen et al. train two models on compiler test cases, one to predict whether a test case will trigger a compiler bug, the other to predict the execution of the test program. The outputs of these two models is used to schedule test cases so as to maximise the potential for exposing bugs in the shortest amount of time [Che+17]. *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].

Machine learning has been applied to the task of automatic software repair. Monperrus surveys the literature in the field [Mon18]. *DeepRepair* using an encoder-decoder architecture to sort code fragments according to similarity of suspicious elements [**White2019**]. Vasic et al. train a model to jointly localise and repair variable-misuse bugs using multi-headed pointer networks [Vas+19]. *SequenceR* uses sequence-to-sequence learning to generate patches [Che+18]. *Getafix* uses a hierarchical clustering algorithm that summarizes fix patterns into a hierarchy ranging from general to specific patterns [Bad+19].

CodeBuff uses a carefully designed set of features to learn abstract code formatting rules from a representative corpus of programs [TV16]. 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]. Oda et al. employ machine translation techniques to generate pseudo-code from source code [Oda+15].

3.5 Summary

This chapter has surveyed the relevant literature in the fields of program generation and optimisation. The next chapter presents a novel technique for generating executable programs using models trained on corpora of example programs.

Chapter 4

Synthesising Benchmarks for Predictive Modelling

4.1 Introduction

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Synthesizing Benchmarks for Predictive Modeling

Chris Cummins
Pavlos Petoumenos
University of Edinburgh, UK
{c.cummins,ppetoume}@inf.ed.ac.uk

Zheng Wang
Lancaster University, UK
z.wang@lancaster.ac.uk

Hugh Leather
University of Edinburgh, UK
hleather@inf.ed.ac.uk

Abstract

Predictive modeling 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.

We mine open source repositories for program fragments and apply deep learning techniques to automatically construct models for how humans write programs. We sample these models to generate an unbounded number of runnable training programs. The quality of the programs is such that even human developers struggle to distinguish our generated programs from hand-written code.

We use our generator for OpenCL programs, CLgen, to automatically synthesize 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 these weaknesses further increases performance by $4.30\times$.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—code generation, compilers, optimization

Keywords Synthetic program generation, OpenCL, Benchmarking, Deep Learning, GPUs

1. Introduction

Predictive modeling is a well researched method for building optimization heuristics that often exceed human experts and reduces development time [1–11]. Figure 1 shows the process by which these models are trained. A set of training programs are identified which 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 summarized 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.

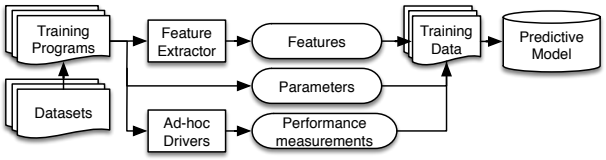


Figure 1. Training a predictive model.

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 the 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

978-1-5090-4931-8/17 © 2017 IEEE

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 [4, 8]. 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.

By contrast, our approach does not require an expert to define what human programs look like. Instead, we automatically infer the structure and likelihood of programs over a huge corpus of open source projects. From this corpus, we learn a probability distribution over sets of characters seen in human written code. Later, we sample from this distribution to generate new random programs which, because the distribution models human written code, are indistinguishable from human code. We can then populate our 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. Our approach is enabled by two recent developments:

The first is the breakthrough effectiveness of deep learning for modeling complex structure in natural languages [12, 13]. As we show, 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 furnishes us with 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 our program generator, CLgen, we also provide an accompanying host driver which generates datasets for, then executes and profiles synthesized programs.

We make the following contributions:

- We are the first to apply deep learning over source codes to synthesize compilable, executable benchmarks.
- A novel tool CLgen[1] for general-purpose benchmark synthesis using deep learning. CLgen automatically and rapidly generates thousands of human like programs for use in predictive modeling.
- We use CLgen to 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 $4.30\times$.

[1] `https://github.com/ChrisCummins/clgen`

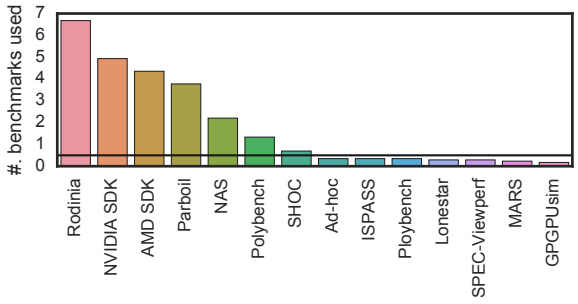


Figure 2. The average number of benchmarks used in GPGPU research papers, organized by origin. In this work we use the seven most popular benchmark suites.

2. Motivation

In this section we make the argument for synthetic benchmarks. We identified frequently used benchmark suites 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. We found the average number of benchmarks used in each paper to be 17, and that a small pool of benchmarks suites account for the majority of results, shown in Figure 2. We selected the 7 most frequently used benchmark suites (accounting for 92% of results), and evaluated the performance of the state of the art *Grewe et al.* [14] predictive model across each. The model predicts whether running a given OpenCL kernel on the GPU gives better performance than on the CPU. We describe the full experimental methodology in Section 7.

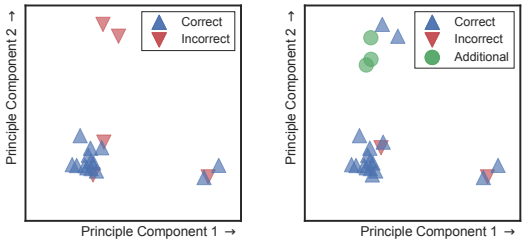
Table 1 summarizes our results. The performance of a model trained on one benchmark suite and used to predict the mapping for another suite is generally very poor. The benchmark suite which provides the best results, NVIDIA SDK, achieves on average only 49% of the optimal performance. The worst case 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 generalize 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 3 shows the feature space of the Parboil benchmark suite, showing whether, for each benchmark, the model was able to correctly predict the appropriate optimization. We used Principle Component Analysis to reduce the multi-dimensional feature space to aid visualization.

As we see in Figure 3a, there is a dense cluster of neighboring benchmarks, a smaller cluster of three benchmarks, and two outliers. The lack of neighboring observations means that the model is unable to learn a good heuristic for the two outliers, which leads to them being incorrectly optimized. In

	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 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.



(a) (b)

Figure 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 neighboring observations in (b) corrects this.

Figure 3b, we 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 optimized. We found such outliers in all of the benchmark suites of Table 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. Our novel approach, described in the next section, solves this problem by generating an unbounded number of programs to cover the feature space with fine granularity.

3. Overview of Our Approach

In this paper we present CLgen, a tool for synthesizing OpenCL benchmarks, and an accompanying host driver for executing synthetic benchmarks for gathering performance data for predictive modeling. While we demonstrate our approach using OpenCL, it is language agnostic. Our tool CLgen learns the semantics and structure from over a million lines of hand-written code from GitHub, and synthesizes programs through a process of iterative model sampling.

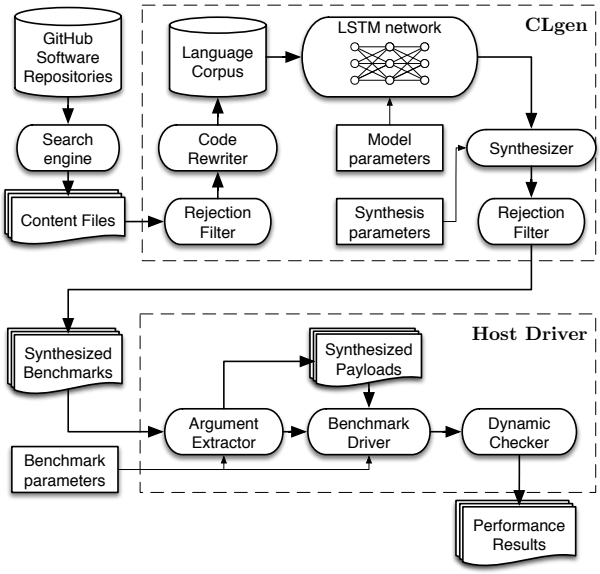


Figure 4. Benchmark synthesis and execution pipeline.

We use a host driver to execute the synthesized programs to gather performance data for use in predictive modeling. Figure 4 provides an overview of the program synthesis and execution pipeline. Our approach extends the state of the art by providing a general-purpose solution for benchmark synthesis, leading to better and more accurate predictive models.

In the course of evaluating our technique against prior work we discovered that it is also useful for evaluating the quality of features. Since we are able to cover the space so much more finely than 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. We go on to show that doing this significantly increases the performance of the learned heuristics. We expect that our technique will be valuable for feature designers.

4. CLgen: Benchmark Synthesis

CLgen is an undirected, general-purpose program synthesizer for OpenCL. 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 neural networks which model the semantics and usage of a huge corpus of code fragments in the target programming language. This section describes the assembly of an OpenCL language corpus, the application of deep learning over this corpus, and the process of synthesizing programs.

4.1 An OpenCL Language Corpus

Deep learning requires large datasets [15]. For the purpose of modeling a programming language, this means assembling a very large collection of real, hand-written source codes. We assembled OpenCL codes 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 recognize it as a searchable programming language. We developed a search engine 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 dataset of 8078 "content files" which potentially contain OpenCL code, originating from 793 GitHub repositories.

We prune the raw dataset extracted from GitHub using a custom toolchain we developed for rejection filtering and code rewriting, built on LLVM.

Rejection Filter The rejection filter accepts as input a content file and returns whether or not it contains compilable, executable OpenCL code. To do this we attempt to compile the input to NVIDIA PTX bytecode and perform static analysis to ensure a minimum static instruction count of three. We discard any inputs which do not compile or contain fewer than three instructions.

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 analyzing 148k lines of compilation errors, we discovered a large number of failures 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, we developed a *shim header* which contains inferred values for common type definitions (e.g. `FLOAT_T`), and common constants (e.g. `WGSIZE`), shown in Listing 1.

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

```
1  /* Enable OpenCL features */
2  #define cl_clang_storage_class_specifiers
3  #define cl_khr_fp64
4  #include <clc/clc.h>
5
6  /* Inferred types */
7  typedef float FLOAT_T;
8  typedef unsigned int INDEX_TYPE;
   ... (36 more)
9
10 /* Inferred constants */
11 #define M_PI 3.14025
12 #define WG_SIZE 128
   ... (185 more)
```

Listing 1. The *shim* header file, providing inferred type aliases and constants for OpenCL on GitHub.

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. We consider these ambiguities to be *non-functional variance*, and developed a tool to normalize code of these variances so as to make the 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 [16], our rewrite method preserves program behavior. 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 Figure 5. A side effect of this process is a reduction in code size, largely due to the removal of comments and excess white space. The final language corpus contains 1.3 million lines of transformed OpenCL, consisting of 9487 kernel functions. Identifier rewriting reduces the bag-of-words vocabulary size by 84%.

4.2 Learning OpenCL

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

We use the Long Short-Term Memory (LSTM) architecture of Recurrent Neural Network [17, 18] to learn a character-level language model over the corpus of OpenCL compute kernels. The LSTM network architecture comprises recurrent

```
#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]); }}
```

(a) Example content file

```
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]);
  }
}
```

(b) Content file after code rewriting

Figure 5. The code rewriting process, which transforms code to make it more amenable to language modeling.

layers of *memory cells*, each consisting of an input, output, and forget gate, and an output layer providing normalized probability values from a 1-of-K coded vocabulary [19].

We use a 3-layer LSTM network with 2048 nodes per layer, implemented in Torch. We train this 17-million parameter model 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 parallelized across devices. The trained network can be deployed to lower-compute machines for use.

4.3 Synthesizing OpenCL

We synthesize OpenCL compute kernels by iteratively sampling the learned language model. We implemented two modes for model sampling: 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 synthesizes kernels matching this signature. In the second sampling mode this argument specification is omitted, allowing the model to synthesize compute kernels of arbitrary signatures, dictated by the distribution of argument types within the language corpus.

In either mode we generate a *seed* text, and sample the model, 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.

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
```

The same rejection filter described in Section 4.1 then either accepts or rejects the sample as a candidate synthetic benchmark. Listing 6 shows 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. We evaluate the quality of synthesized code in Section 6.

5. Benchmark Execution

We developed a host driver to gather performance data from synthesized 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 behavior.

5.1 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.

5.2 Dynamic Checker

For the purpose of performance benchmarking we are not interested in the correctness of computed values, but we define a class of programs as performing *useful work* if they predictably compute some result. We devised a low-overhead runtime behavior check to validate that a synthesized program does useful work based on the outcome of four executions of a tested program:

```
__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)];
}
```

(a) Vector operation with branching and synchronization.

```
__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;
}
```

(b) Zip operation which computes $c_i = 3a_i + 2b_i + 4$.

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

(c) Partial reduction over reinterpreted vector type.

Figure 6. Compute kernels synthesized with CLgen. All three kernel were synthesized from the same argument specification: three single-precision floating-point arrays and a read-only signed integer.

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:
   - $A_{1out} \neq A_{1in}$ 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. Our method is based on random differential testing [20], though we emphasize that this is not a general purpose approach and is tailored specifically for our use case. For example, we anticipate a false positive rate for kernels with subtle sources of non-determinism which more thorough methods may expose [21–23], however we deemed such methods unnecessary for our purpose of performance modeling.

6. Evaluation of Synthetic Programs

In this section we evaluate the quality of programs synthesized by CLgen by their likeness to hand-written code, and discuss limitations of the synthesis and execution pipeline.

6.1 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 we adopt a methodology from machine learning research based on the *Turing Test* [24–26]. We reason that if the output of CLgen is human like code, then a human judge will be unable to distinguish it from hand-written code.

We devised a double blind test 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. We applied the code rewriting process 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 [27], a program generator for differential testing[1].

We scored each participant's answers, finding the average score of the control group to be 96% (stdev. 9%), an unsurpris-

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

Raw Code Features		
`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

Combined Code Features	
`F1: transfer/(comp+mem)`	commun.-computation ratio
`F2: coalesced/mem`	% coalesced memory accesses
`F3: (localmem/mem)` $\times$ `wgsize`	ratio local to global mem accesses $\times$ #. work-items
`F4: comp/mem`	computation-mem ratio

(b) Combinations of raw features

Table 2. *Grewe et al.* model features.

	Version	#. benchmarks	#. kernels
NPB (SNU [29])	1.0.3	7	114
Rodinia [30]	3.1	14	31
NVIDIA SDK	4.2	6	12
AMD SDK	3.0	12	16
Parboil [31]	0.2	6	8
PolyBench [32]	1.0	14	27
SHOC [33]	1.1.5	12	48
Total	-	71	256

Table 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. Experimental platforms.

ing outcome as generated programs for testing have multiple "tells", for example, their only input is a single `ulong` pointer. There were no false positives (synthetic code labeled human) for CLSmith, only false negatives (human code labeled synthetic). With CLgen synthesized 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.

6.2 Limitations

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 3 use irregular data types as inputs. We will address this limitation through recursive program synthesis, whereby a call to a user-defined function or unrecognized type will trigger candidate functions and type definitions to be synthesized. 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 datasets 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.

7. Experimental Methodology

7.1 Experimental Setup

Predictive Model We reproduce the predictive model from Grewe, Wang, and O'Boyle [14]. 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 2b.

Benchmarks As in [14], we test our model on the NAS Parallel Benchmarks (NPB) [28]. We use the hand-optimized OpenCL implementation of Seo, Jo, and Lee [29]. In [14] 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, we use a much larger collection of 142 programs, summarized in Table 3. These additional programs are taken from all 7 of the most frequently used benchmark suites identified in Section 2. None of these programs were used to train CLgen. We synthesized 1,000 kernels with CLgen to use as additional benchmarks.

Platforms We evaluate our approach on two 64-bit CPU-GPU systems, detailed in Table 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.

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

7.2 Methodology

We replicated the methodology of [14]. 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.

We use *leave-one-out cross-validation* to evaluate predictive models. For each benchmark, a model is trained on data from all other benchmarks and used to predict the mapping

```
__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 2. 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 behavior and optimal device mapping. The addition of a branching feature fixes this.

for each kernel and dataset in the excluded program. We repeat this process with and without the addition of synthetic benchmarks in the training data. We do not test model predictions on synthetic benchmarks.

8. Experimental Results

We evaluate the effectiveness of our approach on two heterogeneous systems. We first compare the performance of a state of the art predictive model [14] with and without the addition of synthetic benchmarks, then show how the synthetic benchmarks 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 [27].

8.1 Performance Evaluation

Figure 7 shows speedups of the *Grewe et al.* predictive model over the NAS Parallel Benchmark suite with and without the addition of synthesized 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, we achieve an average speedup of $2.42\times$ with the addition of synthetic benchmarks, with prediction improvements over the baseline for 62.5% of benchmarks on AMD and 53.1% on NVIDIA.

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

8.2 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 generalize over a wider range of programs. We found this 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 generalize as a result of overspecializing to the NPB suite.

The first case is that `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 (we speculate this was a necessary dimensionality reduction in the presence of sparse training programs). To counter this we extended the model to use the raw feature values in addition to the combined features.

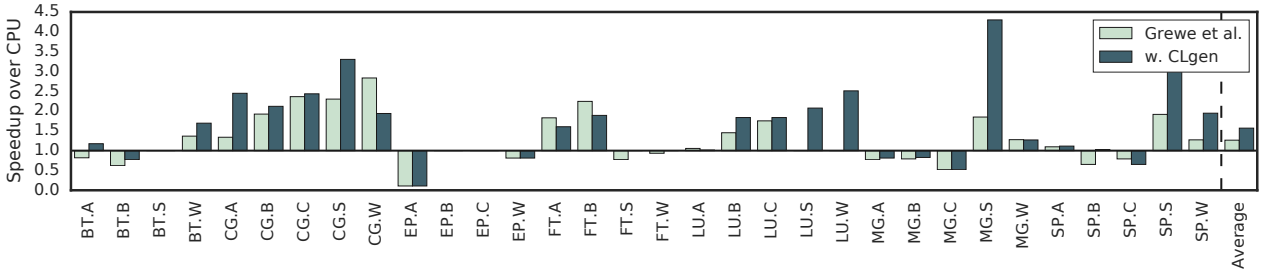
The second case is that some of our generated programs had identical feature values as in the benchmark set, but had different *behavior* (i.e. optimal mappings). Listing 2 shows one example of a CLgen benchmark which is indistinguishable in the feature space to one the of existing benchmarks — the Fast Walsh-Hadamard transform — but with different behavior. 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 minimized branching. To counter this we extended the predictive model with an additional feature containing a static count of branching operations in a kernel.

Figure 8 shows speedups of our extended model across all seven of the benchmark suites used in Section 2. Model performance, even on this tenfold increase of benchmarks, is good. There are three benchmarks on which the model performs poorly: `MatrixMul`, `cutcp`, and `pathfinder`. Each of those programs make heavy use of loops, which we believe the static code features of the model fail to capture. This could be addressed by extracting dynamic instruction counts using profiling, but we considered this beyond the scope of our work. It is not our goal to perfect the predictive model, but to show the performance improvements associated with training on synthetic programs. To this extent, we are successful, achieving average speedups of $3.56\times$ on AMD and $5.04\times$ on NVIDIA across a very large test set.

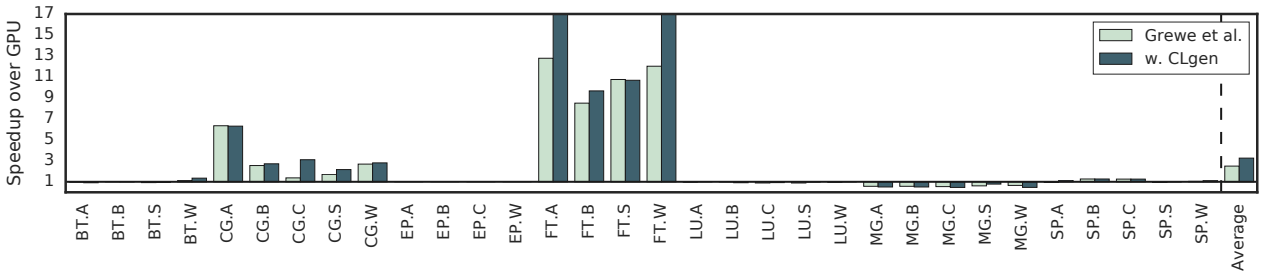
8.3 Comparison of Source Features

As demonstrated in Section 2, the predictive quality of a model for a given point in the feature space is improved with the addition of observations from neighboring points. By producing thousands of artificial programs modeled on the structure real OpenCL programs, CLgen is able to consistently and automatically generate programs which are close in the feature space to the benchmarks which we are testing on.

To quantify this effect we use the static code features of Table 2a, plus the branching feature discussed in the previous subsection, to measure the number of CLgen kernels generated with the same feature values as those of the benchmarks we examined in the previous subsections. We examine only static code features 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.



(a) AMD Tahiti 7970



(b) NVIDIA GTX 970

Figure 7. 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.

Figure 9 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 our original intuition: CLgen kernels, by emulating the way real humans write OpenCL programs, are concentrated in the same area of the feature space as real programs. Moreover, the number of CLgen kernels we generate is unbounded, allowing us to continually refine the exploration of the feature space, while the number of kernels available on GitHub 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. We conclude that the unique contribution of CLgen is its ability to generate many thousands of programs *that are appropriate for predictive modeling*.

9. Related Work

Our work lies at the intersections of a number of areas: program generation, benchmark characterization, and language modeling and learning from source code. There is no existing work which is similar to ours, in respect to learning from large corpuses of source code for benchmark generation.

GENESIS [34] is a language for generating synthetic training programs. Users annotate template programs with statistical distributions over features, which are instantiated to generate statistically controlled permutations of templates. Template based approaches provide domain-specific solutions for a constrained feature and program space, for example, generating permutations of Stencil codes [35, 36]. Our approach provides *general-purpose* program generation over unknown domains, in which the statistical distribution of generated programs is automatically inferred from real world code.

Random program generation is an effective method for software testing. Grammar-based *fuzz testers* have been developed for C [37] and OpenCL [27]. A mutation-based approach for the Java Virtual Machine is demonstrated in [38]. Goal-directed program generators have been used for a variety of domains, including generating linear transforms [39], MapReduce programs [40], and data structure implementations [41]. Program synthesis from input/output examples is used for simple algorithms in [42], string manipulation in [43], and geometry constructions in [44].

Machine learning has been applied to source code to aid software engineering. Naturalize employs techniques developed in the natural language processing domain to model coding conventions [45]. JSNice leverages probabilistic graphical models to predict program properties such as identifier names for Javascript [46].

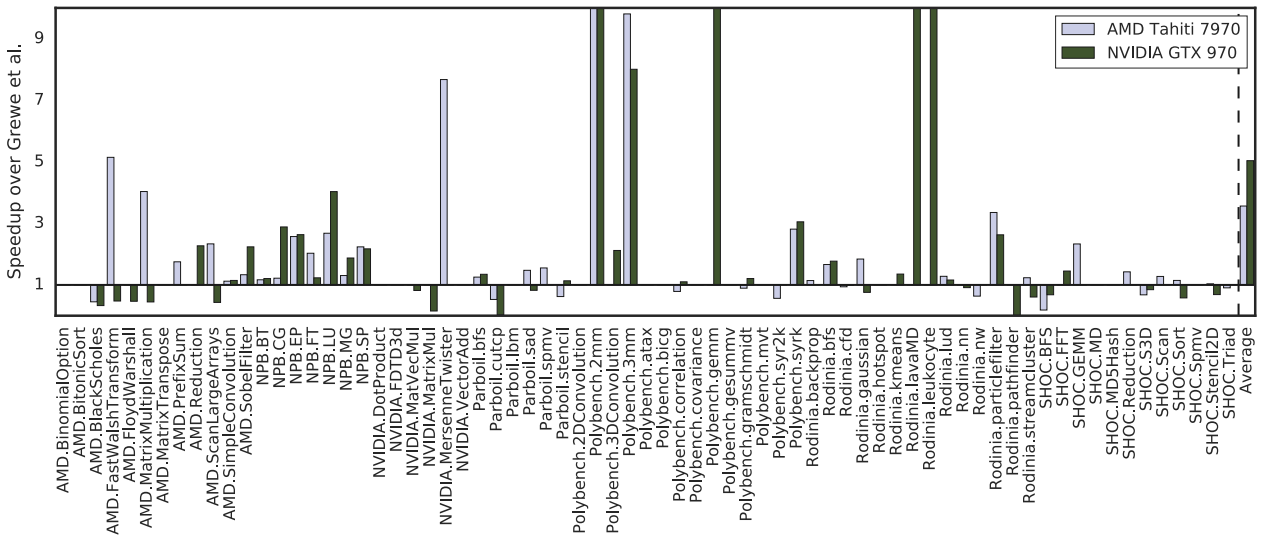


Figure 8. Speedups of predictions using our extended model over *Grewe et al.* on both experimental platforms. Synthetic benchmarks and the additional program features outperform the original predictive model by a factor $3.56\times$ on AMD and $5.04\times$ on NVIDIA.

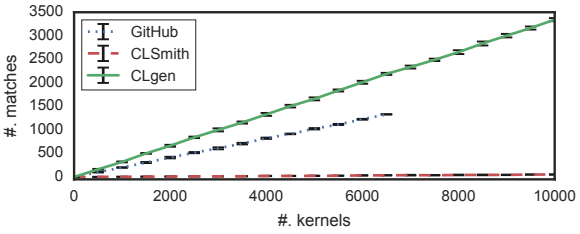


Figure 9. The number of kernels from GitHub, CLSmith, and CLgen with static code features matching the benchmarks. CLgen generates kernels that are closer in the feature space than CLSmith, and can continue to do so long after we have exhausted the extent of the GitHub dataset. Error bars show standard deviation of 10 random samplings.

There is an increasing interest in mining source code repositories at large scale [16, 47, 48]. Previous studies have involved data mining of GitHub to analyze software engineering practices [49–52], for example code generation [53], code summarization [54], comment generation [55], and code completion [56]. However, no work so far has exploited mined source code for benchmark generation. This work is the first to do so.

10. Conclusion

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. We 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. Our 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. Our tool generates programs which to trained eyes are indistinguishable from hand-written code. We tested our approach using a state of the art predictive model, improving its performance by a factor of $1.27\times$. We found that synthetic benchmarks exposed weaknesses in the feature set which, when corrected, further improved the performance by $4.30\times$. 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.

Acknowledgments

Our thanks to the volunteers at Codeplay Software Ltd and the University of Edinburgh for participating in the qualitative evaluation. This work was supported by the UK Engineering and Physical Sciences Research Council under grants EP/L01503X/1 (CDT in Pervasive Parallelism), EP/L000055/1 (ALEA), EP/M01567X/1 (SANDeRs), EP/M0 15823/1, and EP/M015793/1 (DIVIDEND). The code and data for this paper are available at:
`http://chriscummins.cc/cgo17`.

References

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

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

[3] A. Magni, C. Dubach, and M. O'Boyle. "Automatic Optimization of Thread-Coarsening for Graphics Processors". In: *PACT*. ACM, 2014, pp. 455–466.

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

[5] Z. Wang and M. O'Boyle. "Mapping Parallelism to Multi-cores: A Machine Learning Based Approach". In: *PPoPP*. 15. ACM, 2009, pp. 75–84.

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

[7] Z. Wang and M. O'Boyle. "Partitioning Streaming Parallelism for Multi-cores: A Machine Learning Based Approach". In: *PACT*. ACM, 2010, pp. 307–318.

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

[9] A. Collins, C. Fensch, and H. Leather. "Auto-Tuning Parallel Skeletons". In: *Parallel Processing Letters* 22.02 (June 2012), p. 1240005.

[10] H. Leather, E. Bonilla, and M. O'Boyle. "Automatic Feature Generation for Machine Learning Based Optimizing Compilation". In: *TACO* 11 (2014).

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

[12] A. Graves. "Generating Sequences with Recurrent Neural Networks". In: *arXiv:1308.0850* (2013).

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

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

[15] Y. LeCun, Y. Bengio, and G. Hinton. "Deep learning". In: *Nature* 521.7553 (2015), pp. 436–444.

[16] M. Allamanis and C. Sutton. "Mining Source Code Repositories at Massive Scale using Language Modeling". In: *MSR*. 2013, pp. 207–216.

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

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

[19] A. Graves and J. Schmidhuber. "Framewise Phoneme Classification with Bidirectional LSTM and Other Neural Network Architectures". In: *Neural Networks* 5.5 (18), pp. 602–610.

[20] W. M. McKeeman. "Differential Testing for Software". In: *DTJ* 10.1 (1998), pp. 100–107.

[21] A. Betts, N. Chong, and A. Donaldson. "GPUVerify: A Verifier for GPU Kernels". In: *OOPSLA*. 2012, pp. 113–131.

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

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

[24] 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).

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

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

[27] C. Lidbury, A. Lascu, N. Chong, and A. Donaldson. "Many-Core Compiler Fuzzing". In: *PLDI*. 2015, pp. 65–76.

[28] 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), pp. 63–73.

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

[30] 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.

[31] 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).

[32] 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.

[33] 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.

[34] A. Chiu, J. Garvey, and T. S. Abdelrahman. “Genesis: A Language for Generating Synthetic Training Programs for Machine Learning”. In: *CF*. ACM, 2015, p. 8.

[35] J. D. Garvey and T. S. Abdelrahman. “Automatic Performance Tuning of Stencil Computations on GPUs”. In: *ICPP* (2015).

[36] C. Cummins, P. Petoumenos, M. Steuwer, and H. Leather. “Autotuning OpenCL Workgroup Size for Stencil Patterns”. In: *ADAPT*. 2016.

[37] X. Yang, Y. Chen, E. Eide, and J. Regehr. “Finding and Understanding Bugs in C Compilers”. In: *PLDI*. 2011.

[38] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao. “Coverage-Directed Differential Testing of JVM Implementations”. In: *PLDI*. 2016.

[39] Y. Voronenko, F. De Mesmay, and M. Püschel. “Computer Generation of General Size Linear Transform Libraries”. In: *CGO*. IEEE, 2009, pp. 102–113.

[40] C. Smith. “MapReduce Program Synthesis”. In: *PLDI*. 2016.

[41] C. Loncaric, T. Emina, and M. D. Ernst. “Fast Synthesis of Fast Collections”. In: *PLDI*. 2016.

[42] W. Zaremba, T. Mikolov, A. Joulin, and R. Fergus. “Learning Simple Algorithms from Examples”. In: *ICML*. 2016.

[43] S. Gulwani. “Automating string processing in spreadsheets using input-output examples”. In: *POPL*. 2011.

[44] S. Gulwani, V. A. Korthikanti, and A. Tiwari. “Synthesizing geometry constructions”. In: *PLDI*. 2011.

[45] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. “Learning Natural Coding Conventions”. In: *FSE*. 2014, pp. 281–293.

[46] Veselin Raychev, Martin Vechev, and Andreas Krause. “Predicting Program Properties from “Big Code””. In: *POPL*. 2015.

[47] M. White, C. Vendome, M. Linares-Vasquez, and D. Poshyvanyk. “Toward Deep Learning Software Repositories”. In: *MSR*. 2015.

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

[49] 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, pp. 265–268.

[50] E. Guzman, D. Azócar, and Y. Li. “Sentiment Analysis of Commit Comments in GitHub: an Empirical Study”. In: *MSR*. 2014, pp. 352–355.

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

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

[53] X. Gu, H. Zhang, D. Zhang, and S. Kim. “Deep API Learning”. In: *arXiv:1605.08535* (2016).

[54] M. Allamanis, H. Peng, and C. Sutton. “A Convolutional Attention Network for Extreme Summarization of Source Code”. In: *arXiv:1602.03001* (2016).

[55] E. Wong, J. Yang, and L. Tan. “AutoComment: Mining Question and Answer Sites for Automatic Comment Generation”. In: *ASE*. IEEE, 2013, pp. 562–567.

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

A. Artifact description

A.1 Abstract

Our research artifact consists of interactive Jupyter notebooks. For your convenience, we provide two methods of validating our results: an 'AE' notebook which validates the main experiments of the paper, and a comprehensive 'Paper' notebook which replicates every experiment of the paper, including additional analysis. The most convenient method to evaluate our results is to access our pre-configured live server:

`http://[redacted]:8888/notebooks/AE.ipynb`

using the password `[redacted]`, and to follow the instructions contained within.

A.2 Description

A.2.1 Check-list (Artifact Meta Information)

- **Run-time environment:** A web browser.
- **Output:** OpenCL code, runtimes, figures and tables from the paper.
- **Experiment workflow:** Run (or install locally) Jupyter notebooks; interact with and observe results.
- **Experiment customization:** Edit code in Jupyter notebook; full API and CLI for CLgen.
- **Publicly available?:** Yes, code and data. See: `http://chriscummins.cc/cgo17/`

A.2.2 How Delivered

Jupyter notebooks which contain an annotated version of this paper, interleaved with the code necessary to replicate results. We provide three options to run the Jupyter notebooks:

1. Remote access to the notebook running on our pre-configured experimental platform.
2. Download our pre-packaged VirtualBox image with Jupyter notebook installed.
3. Install the project locally on your own machine.

A.3 Installation

Access the Jupyter notebooks using one of the three methods we provide. Once accessed, proceed to Section A.4.

A.3.1 Remote Access

The Jupyter notebooks are available at:
`http://[redacted]:8888`, password `[redacted]`.

A dashboard showing server load is available at:
`http://[redacted]:19999`

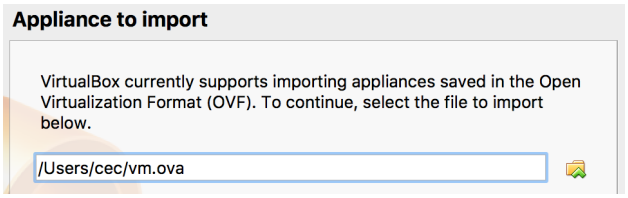
High system load may lead to inconsistent performance results; this may occur if multiple reviewers are accessing the server simultaneously.

A.3.2 Virtual Machine

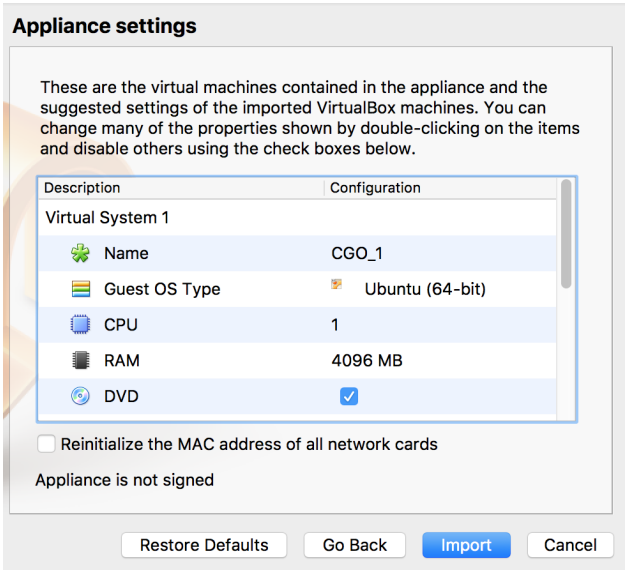
Copy our pre-configured 5.21 GB VirtualBox image using:

```
$ scp cgo@[redacted]:vm.ova ~
Password: [redacted]
```

Install the virtual machine using VirtualBox's "Import Appliance" command:



The image was prepared using VirtualBox 5.1.8. It has the following configuration: Ubuntu 16.04, 4 GB RAM, 10 GB hard drive, bridged network adapter with DHCP, US keyboard layout, GMT timezone.



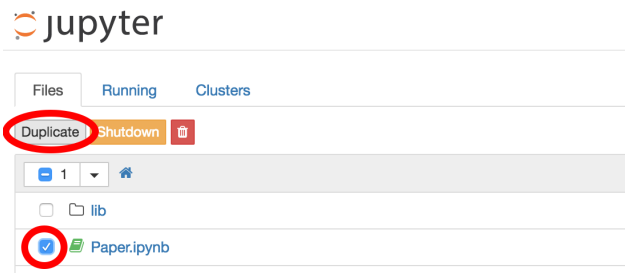
Start the machine and log in using username and password `cgo`. Once at the shell, run `launch`. This will start the Jupyter notebook server and print its address. You can access the notebooks at this address using the browser of the host device. Please note that the VirtualBox image does not have OpenCL, so new runtimes cannot be generated.

A.3.3 Local Install

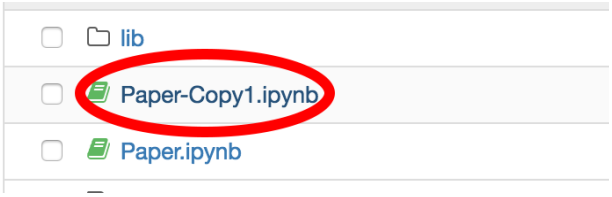
See `http://chriscummins.cc/cgo17/` for instructions. Note that we only support Ubuntu 16.04 or OS X, and sudo privileges are required to install the necessary requirements. Other Linux distributions may work but will require extra steps to install the correct package versions.

A.4 Experiment Workflow

1. Access the Jupyter notebook server using one of the three options described in Section A.3.
2. From the Jupyter server page, tick the checkbox next to one of the two notebooks: `AE.ipynb` for minimal artifact reproduction or `Paper.ipynb` for a comprehensive interactive paper.
3. Click the button "Duplicate".

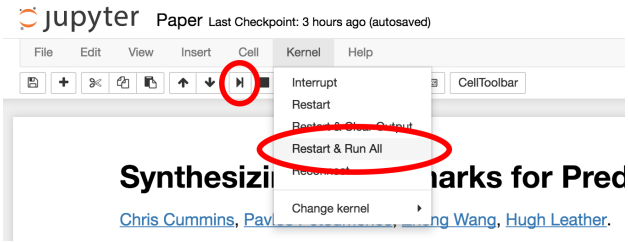


4. Click on the name of the newly created copy, e.g. `Paper-Copy1.ipynb` or `AE-Copy3.ipynb`.



5. Repeatedly press the *play* button (tooltip is "run cell, select below") to step through each cell of the notebook.

   OR select "Kernel" > "Restart & Run All" from the menu to run all of the cells in order.



A.5 Evaluation and Expected Result

Each code cell within the Jupyter notebook generates an output. Expected results are described in text cells.

We include both the code necessary to evaluate the data used in the paper, and the code necessary to generate and evaluate new data. For example, we include the large neural network trained on all of the OpenCL on GitHub (which took 3 weeks to train), along with a small dataset to train a new one.

A.6 Experiment Customization

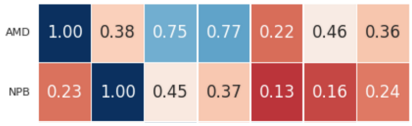
The experiments are fully customizable. The Jupyter notebook can be edited "on the fly". Simply type your changes into the cells and re-run them. For example, in Table 1 of the `Paper.ipyn` notebook we cross-validate the performance of predictive models on an AMD GPU:

```
In [7]: data = amd_benchmarks
        metrics = cgo13.classification(data, group_by="suite")
        results = np.zeros((len(metrics), len(metrics[0])))

        for j in range(len(metrics)):
            for i in range(len(metrics[j])):
                results[j, i] = metrics[j][i].oracle
        results = results.T

        groups = sorted(set(data["Group"]))

        ax = sns.heatmap(results, annot=True, fmt=".2f", linewidths=.5,
                         xticklabels=shortlabels(groups),
                         yticklabels=shortlabels(groups), cbar=False,
                         cmap="RdBu", center=.5, vmin=0, vmax=1)
        plt.xlabel("Training set")
        plt.ylabel("Testing set")
        viz.finalise(figsize=(5, 5), tight=True)
```



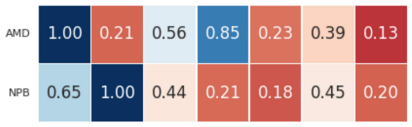
To replicate this experiment using the NVIDIA GPU, change the first line of the appropriate code cell to read `data = nvidia_benchmarks` and re-run the cell:

```
In [11]: data = nvidia_benchmarks
         metrics = cgo13.classification(data, group_by="suite")
         results = np.zeros((len(metrics), len(metrics[0])))

         for j in range(len(metrics)):
             for i in range(len(metrics[j])):
                 results[j, i] = metrics[j][i].oracle
         results = results.T

         groups = sorted(set(data["Group"]))

         ax = sns.heatmap(results, annot=True, fmt=".2f", linewidths=.5,
                          xticklabels=shortlabels(groups),
                          yticklabels=shortlabels(groups), cbar=False,
                          cmap="RdBu", center=.5, vmin=0, vmax=1)
         plt.xlabel("Training set")
         plt.ylabel("Testing set")
         viz.finalise(figsize=(5, 5), tight=True)
```



Note that some of the cells depend on the values of prior cells and must be executed in sequence.

CLgen has a documented API and command line interface. You can create new corpuses, train new networks, sample kernels, etc.

A.7 Notes

For more information about CLgen, visit:
`http://chriscummins.cc/clgen`
For more information about Artifact Evaluation, visit:
`http://ctuning.org/ae/submission-20161020.html`

4.2 Conclusion

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Chapter 5

Compiler Fuzzing through Deep Learning

5.1 Introduction

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Compiler Fuzzing through Deep Learning

Chris Cummins,
Pavlos Petoumenos, Hugh Leather
University of Edinburgh
United Kingdom
{c.cummins,ppetoume,hleather}@inf.ed.ac.uk

Alastair Murray
Codeplay Software
Edinburgh, United Kingdom
alastair.murray@codeplay.com

ABSTRACT

Random program generation — fuzzing — is an effective technique for discovering bugs in compilers but successful fuzzers require extensive development effort for every language supported by the compiler, and often leave parts of the language space untested.

We introduce DeepSmith, a novel machine learning approach to accelerating compiler validation through the inference of generative models for compiler inputs. Our approach *infers* a learned model of the structure of real world code based on a large corpus of open source code. Then, it uses the model to automatically generate tens of thousands of realistic programs. Finally, we apply established differential testing methodologies on them to expose bugs in compilers. We apply our approach to the OpenCL programming language, automatically exposing bugs with little effort on our side. In 1,000 hours of automated testing of commercial and open source compilers, we discover bugs in all of them, submitting 67 bug reports. Our 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. Our 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. With 18 lines of code we extended our program generator to a second language, uncovering crashes in Solidity compilers in 12 hours of automated testing.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

Deep Learning; Differential Testing; Compiler Fuzzing.

ACM Reference Format:
Chris Cummins, Pavlos Petoumenos, Hugh Leather and Alastair Murray. 2018. Compiler Fuzzing through Deep Learning. In *ISSTA'18: International Symposium on Software Testing and Analysis, July 16–21, 2018, Amsterdam, Netherlands.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3213846.3213848

ISSTA'18, July 16–21, 2018, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5699-2/18/07…$15.00
https://doi.org/10.1145/3213846.3213848

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 optimizing 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 [6, 7, 16]. When fuzzing, randomly generated valid or semi-valid inputs are fed to the compiler. Any kind of unexpected behavior, 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 behavior or a gold standard compiler from which to create reference outputs. In the absence of those, Differential Testing [22] 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 behavior, 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 [32], 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 behavior. The programs are then differentially tested.

While CSmith has been successfully used to identify hundreds of bugs in 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 [20] — 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.

What we propose is a fast, effective, and low effort approach to the generation of random programs for compiler fuzzing. Our methodology uses recent advances in deep learning to automatically

construct 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. Our approach essentially frames the generation of random programs as a language modeling 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.

In this work we primarily target OpenCL, an open standard for programming heterogeneous systems, though our approach is largely language agnostic. We chose OpenCL 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. We provide preliminary results supporting DeepSmith's potential for multi-lingual compiler fuzzing.

We make the following contributions:

- a novel, automatic, and fast approach for the generation of expressive random programs for compiler fuzzing. We *infer* programming language syntax, structure, and use from real-world examples, not through an expert-defined grammar. Our system needs two orders of magnitude less code than the state-of–the-art, and takes less than a day to train;
- we discover a similar number of bugs as the state-of–the-art, but also find bugs which prior work cannot, covering more components of the compiler;
- in modeling real handwritten code, our 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.

2 DEEPSMITH

DeepSmith[1] is our open source framework for compiler fuzzing. Figure 1 provides a high-level overview. In this work we target OpenCL, 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.

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, we treat the problem as an unsupervised machine learning task, employing state-of-the-art deep learning techniques to build models for how humans write programs. Our approach is inspired by breakthrough results in modeling challenging and high dimensional datasets through unsupervised learning [4, 27, 28]. Contrary to existing tools, our approach does not require expert knowledge of the target language and is only a few hundred lines of code.

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

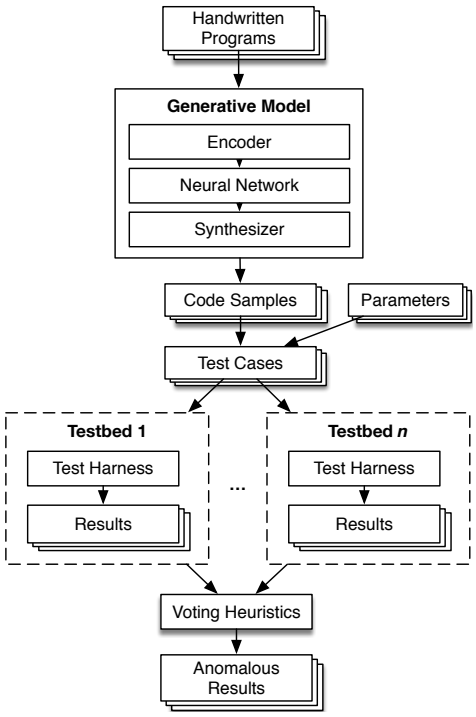


Figure 1: DeepSmith system overview.

Handwritten Programs. The generative model needs to be trained on a *seed corpus* of example programs. We automated the assembly of this corpus by mining 10k OpenCL kernels from open source repositories on GitHub. We used an *oracle compiler* (LLVM 3.9) to statically check the source files, 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 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.

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 [9], 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.

We additionally employed semantic-preserving transformations 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 behavior. 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. We 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.

Neural Network. We use the Long Short-Term Memory (LSTM) architecture of Recurrent Neural Network to model program code [12]. 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 modeling 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 [24]. This makes them effective at learning complex relationships over long sequences [21] which is important for modeling program code. Our LSTM networks model the vocabulary distribution over the encoded corpus. After initial experiments using different model parameters, we found 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. We provided the model with no prior knowledge of the structure or syntax of a programming language.

Program Generation. The trained network is sampled to generate new programs. The model is seeded with the start of a kernel (identified in OpenCL using the keywords `kernel void`), and sampled token-by-token. A "bracket depth" counter is incremented or decremented upon production of `{` or `}` tokens respectively, so that the end of the kernel can be detected and sampling halted. The generated sequence of tokens is then decoded back to text and used for compiler testing.

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, we used the test harness of CLSmith. 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.

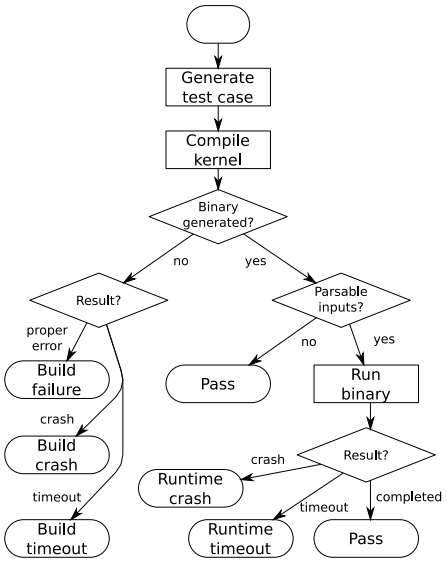


Figure 2: Test case execution, and possible results.

We desired a more flexible harness so as to test a more expressive range of programs, capable of supporting multi-argument kernels and generating data to use as inputs.

We developed a harness which first determines the expected arguments from the function prototype and generates host data for them. At the moment, we support scalars and arrays of all OpenCL primitive and vector types. For a kernel execution across n threads, buffers of size n are allocated for pointer arguments and populated with values $[1 \ldots n]$; scalar inputs are given value n, since we observe that most kernels use these 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 our driver cannot create example inputs. In this case, a "compile-only" stub is used, which only compiles the kernel, without generating input data or executing the compiled kernel.

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

Test Harness Output Classes. Executing a test case on a testbed leads to one of seven possible outcomes, illustrated in Figure 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.

#.	Platform	Device	Driver	OpenCL	Operating system	Device Type	Open Source?	Bug Reports Submitted
1	NVIDIA CUDA	GeForce GTX 1080	375.39	1.2	Ubuntu 16.04 64bit	GPU		8
2	NVIDIA CUDA	GeForce GTX 780	361.42	1.2	openSUSE 13.1 64bit	GPU		1
3	Beignet	Intel HD Haswell GT2	1.3	1.2	Ubuntu 16.04 64bit	GPU	Yes	13
4	Intel OpenCL	Intel E5-2620 v4	1.2.0.25	2.0	Ubuntu 16.04 64bit	CPU		6
5	Intel OpenCL	Intel E5-2650 v2	1.2.0.44	1.2	CentOS 7.1 64bit	CPU		1
6	Intel OpenCL	Intel i5-4570	1.2.0.25	1.2	Ubuntu 16.04 64bit	CPU		5
7	Intel OpenCL	Intel Xeon Phi	1.2	1.2	CentOS 7.1 64bit	Accelerator		3
8	POCL	POCL (Intel E5-2620)	0.14	1.2	Ubuntu 16.04 64bit	CPU	Yes	22
9	Codeplay	ComputeAorta (Intel E5-2620)	1.14	1.2	Ubuntu 16.04 64bit	CPU		1
10	Oclgrind	Oclgrind Simulator	16.10	1.2	Ubuntu 16.04 64bit	Emulator	Yes	7

Table 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 optimizations, the other without.

2.3 Voting Heuristics for Differential Testing

We employ established Differential Testing methodologies 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 [22]. However, we extend this approach 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 behavior (examples may be found in Section 4.1). For all other outcomes, *differential tests* are required to confirm anomalous behavior. 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. We use the presence of the majority increasing the likelihood that there is a 'correct' behavior for the test case. In this work, we choose the majority fraction to be $\lceil \frac{2}{3} n \rceil$, where n is the number of testbeds.

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

False Positives for Anomalous Runtime Behavior. Generated programs may contain undefined or non-deterministic behavior which will incorrectly be labeled as anomalous. CSmith circumvents this problem by performing complex analyses during generation so as to minimize the chance of producing programs with undefined behavior. Although similar analyses could be created as filters for our system, we take a simpler approach, filtering only the few types of non-deterministic behavior we have actually observed to happen in practice.

We filter data races, out-of-bounds and uninitialized accesses with GPUverify [2] and Oclgrind [26]. Some compiler warnings provide strong indication of non-deterministic behavior (e.g. comparison between pointer and integer) – we check for these warnings and filter 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 allows 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 behavior we observed comes from division by zero and related mathematical functions which require non-zero values. We apply a simple detection and filtering heuristic – we change the input values and check to see if the output remains anomalous. While theoretically insufficient, in practice we found that no false positives remained.

3 EXPERIMENTAL SETUP

In this section we describe the experimental parameters used.

3.1 OpenCL Systems

We conducted testing of 10 OpenCL systems, summarized in Table 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.

3.2 Testbeds

For each OpenCL system, we create two testbeds. In the first, the compiler is run with optimizations disabled. In the second, optimizations are enabled. Each testbed is then a triple, consisting of *<device, driver, is_optimized>* settings. This mechanism gives 20 testbeds to evaluate.

3.3 Test Cases

For each generated program we create inputs as described in Section 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.

3.4 Bug Search Time Allowance

We compare both our fuzzer and CLSmith. We allow 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.

```
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 3: Example kernels which crash compilers.

4 EVALUATION

We report on the results of DeepSmith testing of the 10 OpenCL systems from Table 1, in which each ran for 48 hours. We found bugs in all the compilers we tested — every compiler crashed, and every compiler generated programs which either crash or silently compute the wrong result. To date, we have submitted 67 bug reports to compiler vendors. We first provide a qualitative analysis of compile-time and runtime defects found, followed by a quantitative comparison of our approach against the state-of-the-art in OpenCL compiler fuzzing — CLSmith [20]. DeepSmith is able to identify a broad range of defects, many of which CLSmith cannot, for only a fraction of the engineering effort. Finally, we provide 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. We find 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 save space. No test case reduction, either manual or automatic, was needed.

For the remainder of the paper we identify testbeds using the OpenCL system number from Table 1, suffixed with +, −, or ± to denote optimizations on, off, or either, respectively.

```
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 4: Example codes which crash parsers.

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. We found numerous cases 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 we have identified 199 test cases which trigger unreachable code failures, triggered 31 different compiler assertions, and produced 114 distinct stack traces from other compiler crashes.

Semantic Analysis Failures. Compilers should produce meaningful diagnostics when inputs are invalid, yet we discovered dozens of compiler defects 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. We believe that our approach contributes 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 3a 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 3b induces a crash in NVIDIA compilers due to an implicit conversion between global to constant address qualifiers. Worse, we found that Testbeds 3± would loop indefinitely on some kernels containing implicit conversions from a pointer to an integer, as shown in Figure 5a. While spinning, the compiler would utilize 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 3c and 3d pass the type checker and semantic analysis, but trigger compiler assertions during code generation.

An interesting yet unintended byproduct 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 builtins. We found the quality of error handling on these builtins to vary wildly. For example, Figure 3e silently crashes 6 of the 10 compilers, which, to the best of our knowledge, makes

```
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 optimization of kernels with large loop bounds. Testbeds 1− and 2± compile in under 1 second.

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

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

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

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

Figure 5: Example kernels which hang compilers.

DeepSmith the first random program generator to induce a defect through exploiting compiler-specific functionality.

Parser Failures. Parser development is a mature and well understood practice. We uncovered parser errors in several compilers. Each of the code samples in Figure 4 induce crash errors during parsing of compound statements in both Testbeds 5± and 7±. For space, we have hand-reduced the listings to minimal code samples, which we have reported to Intel. Each reduction took around 6 edit-compile steps, taking less than 10 minutes. In total, we have generated 100 distinct programs which crash compilers during parsing.

Compiler Hangs. As expected, some compile-time defects are optimization sensitive. Testbed 1+ hangs on large loop bounds, shown in Figure 5b. All commercial Intel compilers we tested hang during optimization of non-terminating loops (Figure 5c).

Testbeds 7± loop indefinitely during compilation of the simple kernel in Figure 5d.

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.

```
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 optimize 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 6: Example kernels which are miscompiled.

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

4.2 Runtime Defects

Prior work on compiler test case generation has focused on extensive stress-testing of compiler middle-ends to uncover miscompilations [6]. 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 we provide examples of bugs which cannot currently be discovered by CLSmith.

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 modeled 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 optimizer of Testbeds 4+ and 6+ which causes the `if` branch in Figure 6a to be erroneously executed when the kernel is compiled with optimizations enabled. We have reported this issue to Intel. CLSmith does not permit the thread identity to modify control flow, rendering such productions impossible.

Figure 6b shows a simple program in which thread identity determines the program output. We found that this test case would 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.

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 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.

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 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.

4.3 Comparison to State-of-the-art

In this section, we provide a quantitative comparison of the bug-finding capabilities of DeepSmith and CLSmith.

Results Overview. Table 2 shows the results of 48 hours of consecutive testing for all Testbeds. 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 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 optimization level generally does not affect testing throughput significantly, with the exception of Testbed 7+. Optimization of large structs is expensive on Testbed 7+, and CLSmith test cases use global structs extensively. This is a known issue — in [20] 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 we have chosen to exclude it.

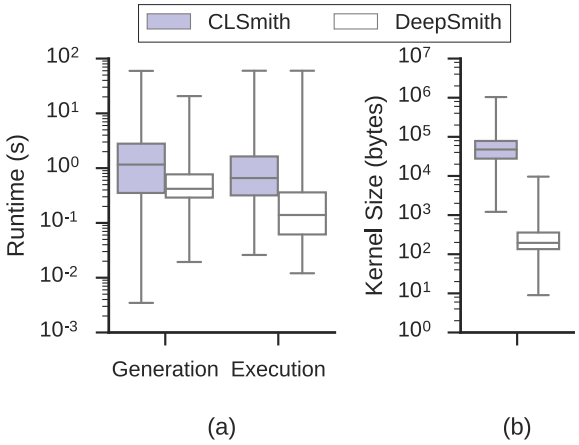


Figure 7: Comparison of runtimes (a) and test case sizes (b). DeepSmith test cases are on average evaluated 3.03× faster than CLSmith (2.45×, and 4.46× for generation and execution, respectively), and are two orders of magnitude smaller. Timings do not include the cost of timeouts which would increase the performance gains of DeepSmith by nearly a factor of two.

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 [25], though it requires on average 100 minutes for each test case using a parallelized implementation (and over 6 hours if this parallelization 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.

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. Figure 8 provides examples. Many of these crashes are optimization sensitive, and are more likely to occur when optimizations are enabled. CLSmith was able to induce a crash in only one of the Intel testbeds with optimizations disabled. DeepSmith crashed all of the compilers with both optimizations enabled and disabled.

			CLSmith							DeepSmith						
#.	Device	±	bc	bto	abf	arc	awo	✓	total	bc	bto	abf	arc	awo	✓	total
1	GeForce GTX 1080	−	0	0	0	2	2	15628	15632	27	0	3	0	5	62105	62140
		+	0	71	0	6	9	14007	14093	20	1	1	0	7	57361	57390
2	GeForce GTX 780	−	0	0	0	28	5	18220	18253	27	0	3	0	9	87129	87168
		+	26	14	0	0	3	17654	17697	32	1	1	0	9	82666	82709
3	Intel HD Haswell GT2	−	2714	2480	0	0	3	1121	6318	574	200	2	0	12	136977	137765
		+	2646	2475	0	0	3	1075	6199	569	200	5	0	10	135430	136214
4	Intel E5-2620 v4	−	0	27	1183	0	0	16313	17523	57	0	9	1	0	107982	108049
		+	487	87	1130	0	0	17350	19054	320	147	7	3	0	113616	114093
5	Intel E5-2650 v2	−	0	11	0	0	0	17887	17898	152	2	0	0	0	90882	91036
		+	112	175	0	0	0	14626	14913	170	117	0	0	1	90478	90766
6	Intel i5-4570	−	0	14	1226	0	0	17118	18358	73	0	9	2	1	111240	111325
		+	526	63	1180	0	0	19185	20954	318	140	7	2	1	117049	117517
7	Intel Xeon Phi	−	4	84	0	0	8	13265	13361	68	4	0	0	1	37171	37244
		+	42	1474	0	0	2	3258	4776	77	47	0	0	0	37501	37625
8	POCL (Intel E5-2620)	−	0	0	0	675	0	17250	17925	54	1	2	89	3	85318	85467
		+	0	3	0	99	5	13980	14087	46	0	1	104	4	81267	81422
9	ComputeAorta (Intel E5-2620)	−	0	0	0	0	0	18479	18479	51	0	1	3	1	112324	112380
		+	0	0	0	300	11	18625	18936	59	0	0	48	4	115323	115434
10	Oclgrind Simulator	−	0	0	0	0	0	5287	5287	2081	0	0	0	1	73261	75343
		+	0	0	0	0	0	5334	5334	2265	0	0	0	0	77959	80224

Table 2: Results from 48 hours of testing using CLSmith and DeepSmith. System #. as per Table 1. ± denotes optimizations 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.

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 9. 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 optimizations 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.

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 3e. To evaluate the impact of Clang, we used debug+assert builds of every LLVM release in the past 24 months and processed 75,000 DeepSmith kernels through the Clang front-end (this includes the lexer, parser, and type checker, but not code generation).

Figure 10 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 3 shows the 7 distinct assertions triggered during this experiment. Assertion 1 (*Uncorrected typos!*) is raised on all compiler versions — see Figure 3a 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. We have submitted bug reports for each of the three assertions triggered in the development trunk, as well as for two distinct unreachables.

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

```
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.**

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

(f) ***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);
}
```

(g) ***Intel OpenCL RemoveDuplicationBarrier*** **pass.**

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

(h) ***X86 DAG->DAG Instruction Selection*** **pass.**

Figure 8: Example 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 9: Example kernels which trigger compiler assertions which both CLSmith and DeepSmith exposed.

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. To explore this, we report 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.

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

Initial Results. We ran the generator and harness loop for 12 hours on four testbeds: the Solidity reference compiler `solc` with optimizations on or off, and `solc-js`, which is an Emscripten compiled version of the `solc` compiler. Our results are summarized in Table 4. We found numerous cases 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, we believe these preliminary results indicate the utility of our approach for simplifying test case generation.

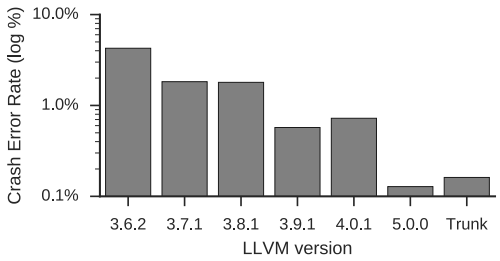


Figure 10: Crash rate of the Clang front-end of every LLVM release in the past 24 months compiling 75k DeepSmith kernels.

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

Table 3: The number of DeepSmith programs which trigger distinct Clang front-end assertions, and the number of programs which trigger unreachables.

5 RELATED WORK

The random generation of test cases is a well established approach to the compiler validation problem. Prior approaches are surveyed in [3, 16] and empirically contrasted in [6]. 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 synthesized from scratch; and *program mutation*, where existing codes are modified so as to identify anomalous behavior.

Program Generation. 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 [22]. Subsequent works have presented increasingly complex generators which improve in some metric of interest, generally expressiveness or probability of correctness. CSmith [32] 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 behavior 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. Subsequent generators influenced by CSmith, like Orange3 [23], focus on features and bug types beyond the scope of CSmith, arithmetic bugs in the case of Orange3. Glade [1] derives a grammar from a corpus of example programs. The derived grammar is enumerated to produce new programs, though unlike our approach, no distribution is learned over the grammar; program enumeration is uniformly random.

Program Mutation. Equivalence Modulo Inputs (EMI) testing [19, 29] follows a different approach to test case generation. Starting with existing code, it inserts or deletes statements that will not be executed, so functionality should remain the same. If it is affected, it is due to a compiler bug. While a powerful technique able to find hard to detect bugs, it relies on having a very large number of programs to mutate. As such, it still requires an external code generator. Similarly to CSmith, EMI favors very long test programs. LangFuzz [13] also uses mutation but does this by inserting code segments which have previously exposed bugs. This increases the chances of discovering vulnerabilities in scripting language engines. Skeletal program enumeration [34] again works by transforming existing code. It identifies algorithmic patterns in short pieces of code and enumerates all the possible permutations of variable usage. Compared to all these, our fuzzing approach is low cost, easy to develop, portable, capable of detecting a wide range of errors, and focusing by design on bugs that are more likely to be encountered in a production scenario.

Compiler	±	Silent Crashes	Assertion 1	Assertion 2
solc	−	204	1	
	+	204	1	
solc-js	−	3628	1	1
	+	908	1	1

Table 4: The number of DeepSmith programs that trigger Solidity compiler crashes from 12 hours of testing.

Machine Learning. There is an increasing interest in applying machine learning to software testing. Most similar to our work is Learn&fuzz [10], 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. Unlike compiler testing, PDF test cases require no inputs and no pre-processing of the training corpus. Skyfire [30] learns a probabilistic context-sensitive grammar over a corpus of programs to generate input seeds for mutation testing. The generated seeds are shown to improve the code coverage of AFL [33] when fuzzing XSLT and XML engines, though the seeds are not directly used as test cases. Machine learning has also been applied to other areas such as improving bug finding static analyzers [11, 15], repairing programs [17, 31], prioritizing test programs [5], identifying buffer overruns [8], and processing bug reports [14, 18]. To the best of our knowledge, no work so far has succeeded in finding compiler bugs by exploiting the learned syntax of mined source code for test case generation. Ours is the first to do so.

6 CONCLUSIONS

We present a novel framework for compiler fuzzing. By posing the generation of random programs as an unsupervised machine learning problem, we dramatically reduce the cost and human effort required to engineer a random program generator. 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.

We demonstrated our approach by targeting the challenging many-core domain of OpenCL. Our implementation, DeepSmith, has uncovered dozens of bugs in OpenCL implementations. We have exposed bugs in parts of the compiler where current approaches have not, for example in missing error handling. We provided a preliminary exploration of the extensibility of our approach. Our test cases are small, two orders of magnitude shorter than the state-of-the-art, and easily interpretable.

7 ACKNOWLEDGEMENTS

This work was supported by the UK EPSRC under grants EP/L01503X/1 (CDT in Pervasive Parallelism), and EP/P003915/1 (SUMMER).

REFERENCES

[1] O. Bastani, R. Sharma, A. Aiken, and P. Liang. Synthesizing Program Input Grammars. In *PLDI*, 2017.
[2] A. Betts, N. Chong, and A. Donaldson. GPUVerify: A Verifier for GPU Kernels. In *OOPSLA*. ACM, 2012.
[3] A. S. Boujarwah and K. Saleh. Compiler Test Case Generation Methods: A Survey and Assessment. *Information and Software Technology*, 39(9), 1997.
[4] S. R. Bowman, L. Vilnis, O. Vinyals, A. M. Dai, R. Jozefowicz, and S. Bengio. Generating Sentences from a Continuous Space. *arXiv:1511.06349*, 2015.
[5] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, and B. Xie. Learning to Prioritize Test Programs for Compiler Testing. In *ICSE*, 2017.
[6] 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.
[7] Y. Chen, A. Groce, C. Zhang, W. Wong, X. Fern, E. Eide, and J. Regehr. Taming Compiler Fuzzers. *PLDI*, 2013.
[8] M. Choi, S. Jeong, H. Oh, and J. Choo. End-to-End Prediction of Buffer Overruns from Raw Source Code via Neural Memory Networks. *arXiv:1703.02458*, 2017.
[9] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather. End-to-end Deep Learning of Optimization Heuristics. In *PACT*. IEEE, 2017.
[10] P. Godefroid, H. Peleg, and R. Singh. Learn&Fuzz: Machine Learning for Input Fuzzing. In *ASE*, 2017.
[11] K. Heo, H. Oh, and K. Yi. Machine-Learning-Guided Selectively Unsound Static Analysis. In *ICSE*, 2017.
[12] S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8), 1997.
[13] C. Holler, K. Herzig, and A. Zeller. Fuzzing with Code Fragments. *Usenix*, 2012.
[14] X. Huo, M. Li, and Z. Zhou. Learning Unified Features from Natural and Programming Languages for Locating Buggy Source Code. In *IJCAI*, 2016.
[15] 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.
[16] A. S. Kossatchev and M. A. Posypkin. Survey of Compiler Testing Methods. *Programming and Computer Software*, 31(1), 2005.
[17] M. Koukoutos, M. Raghothaman, E. Kneuss, and V. Kuncak. On Repair with Probabilistic Attribute Grammars. *arXiv:1707.04148*, 2017.
[18] 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.
[19] V. Le, M. Afshari, and Z. Su. Compiler Validation via Equivalence Modulo Inputs. In *PLDI*, 2014.
[20] C. Lidbury, A. Lascu, N. Chong, and A. Donaldson. Many-Core Compiler Fuzzing. In *PLDI*, 2015.
[21] Z. C. Lipton, J. Berkowitz, and C. Elkan. A Critical Review of Recurrent Neural Networks for Sequence Learning. *arXiv:1506.00019*, 2015.
[22] W. M. McKeeman. Differential Testing for Software. *DTJ*, 10(1), 1998.
[23] 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.
[24] R. Pacanu, T. Mikolov, and Y. Bengio. On the Difficulties of Training Recurrent Neural Networks. In *ICML*, 2013.
[25] M. Pflanzer, A. Donaldson, and A. Lascu. Automatic Test Case Reduction for OpenCL. In *IWOCL*, 2016.
[26] J. Price and S. Mcintosh-Smith. Oclgrind: An Extensible OpenCL Device Simulator. In *IWOCL*. ACM, 2015.
[27] A. Radford, R. Jozefowicz, and I. Sutskever. Learning to Generate Reviews and Discovering Sentiment. *arXiv:1704.01444*, 2017.
[28] M. Raghu, B. Poole, J. Kleinberg, S. Ganguli, and J. Sohl-Dickstein. On the expressive power of deep neural networks. *arXiv:1606.05336*, 2016.
[29] C. Sun, V. Le, and Z. Su. Finding Compiler Bugs via Live Code Mutation. In *OOPSLA*, 2016.
[30] J. Wang, B. Chen, L. Wei, and Y. Liu. Skyfire: Data-Driven Seed Generation for Fuzzing. In *S&P*, 2017.
[31] M. White, M. Tufano, M. Martínez, M. Monperrus, and D. Poshyvanyk. Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities. *arXiv:1707.04742*.
[32] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and Understanding Bugs in C Compilers. In *PLDI*, 2011.
[33] M. Zalewski. American Fuzzy Lop.
[34] Q. Zhang, C. Sun, and Z. Su. Skeletal Program Enumeration for Rigorous Compiler Testing. In *PLDI*, 2017.

5.2 Conclusion

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Chapter 6

End-to-end Deep Learning of Optimisation Heuristics

6.1 Introduction

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

End-to-end Deep Learning of Optimization Heuristics

Chris Cummins, Pavlos Petoumenos
School of Informatics
University of Edinburgh
United Kingdom
{c.cummins,ppetoume}@inf.ed.ac.uk

Zheng Wang
School of Computing and Communications
Lancaster University
United Kingdom
z.wang@lancaster.ac.uk

Hugh Leather
School of Informatics
University of Edinburgh
United Kingdom
hleather@inf.ed.ac.uk

***Abstract*—Accurate automatic optimization heuristics are necessary for dealing with the complexity and diversity of modern hardware and software. Machine learning is a proven technique for learning such heuristics, but its success is bound by the quality of the features used. These features must be hand crafted by developers through a combination of expert domain knowledge and trial and error. This makes the quality of the final model directly dependent on the skill and available time of the system architect.**

Our work introduces a better way for building heuristics. We develop a deep neural network that learns heuristics over raw code, entirely without using code features. The neural network simultaneously constructs appropriate representations of the code and learns how best to optimize, removing the need for manual feature creation. Further, we show that our neural nets can transfer learning from one optimization problem to another, improving the accuracy of new models, without the help of human experts.

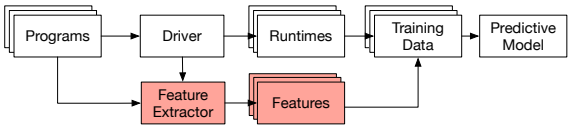
We compare the effectiveness of our automatically generated heuristics against ones with features hand-picked by experts. We examine two challenging tasks: predicting optimal mapping for heterogeneous parallelism and GPU thread coarsening factors. In 89% of the cases, the quality of our fully automatic heuristics matches or surpasses that of state-of-the-art predictive models using hand-crafted features, providing on average 14% and 12% more performance with no human effort expended on designing features.

***Keywords*-Optimization Heuristics; Machine Learning; Compiler Optimizations; Heterogeneous Systems**

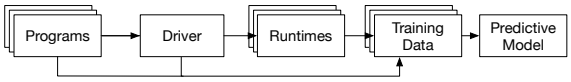
I. Introduction

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 optimization 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 suboptimal decisions. Researchers have focused on machine learning as a means to constructing high quality heuristics that often outperform their handcrafted equivalents [1–5]. A *predictive model* is trained, using supervised machine learning, on empirical performance data and important quantifiable properties, or *features*, of representative programs.



(a) Current state-of-practice



(b) Our proposal

Figure 1: Building a predictive model. The model is originally trained on performance data and features extracted from the source code and the runtime behavior. We propose bypassing feature extraction, instead learning directly over raw program source code.

The model learns the correlation between these features and the optimization decision that maximizes performance. The learned correlations are used to predict the best optimization 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 [6, 7].

Still, experts are not completely removed from the design process, which is shown in Figure 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 optimization decisions and expresses them in ways that make it easy to model their relationship to performance. Failing to identify an important feature has a negative effect on the resulting heuristic. For example, in [8] the authors discovered that [6] did not identify one such feature, causing performance to be 40% lower on average.

To make heuristic construction fast and cheap, we must take humans out of the loop. While techniques for automatic feature generation from the compiler IR have been proposed in the past [9, 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. Our insight was that such

costly approaches are not necessary any more. Deep learning techniques have shown astounding successes in identifying complex patterns and relationships in images [11, 12], audio [13], and even computer code [14, 15]. We hypothesized that deep neural networks should be able to automatically extract features from source code. Our experiments showed that even this was a conservative target: with deep neural networks we can bypass static feature extraction and learn optimization heuristics directly on raw code.

Figure 1b shows our 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. Our work replaces the need for compile-time or static code features, merging feature and heuristic construction into a single process of joint learning. Our system admits auxiliary features to describe information unavailable at compile time, such as the sizes of runtime input parameters. Beyond these optional inclusions, we are able to learn optimization heuristics without human guidance.

By employing *transfer learning* [16], our 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 optimization problem. We reuse these parts of the network across heuristics, and, in the process, we speed up learning considerably.

We evaluated our approach 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, our approach was able to outperform them by 14% and 12%, which indicates a better identification of important program characteristics, without any expert help. We make the following contributions:

- We present a methodology for building compiler heuristics without any need for feature engineering.
- A novel tool DeepTune for automatically constructing optimization heuristics without features. DeepTune outperforms existing state-of-the-art predictive models by 14% and 12% in two challenging optimization domains.
- We apply, for the first time, *transfer learning* on compile-time and runtime optimizations, improving the heuristics by reusing training information across different optimization problems, even if they are unrelated.

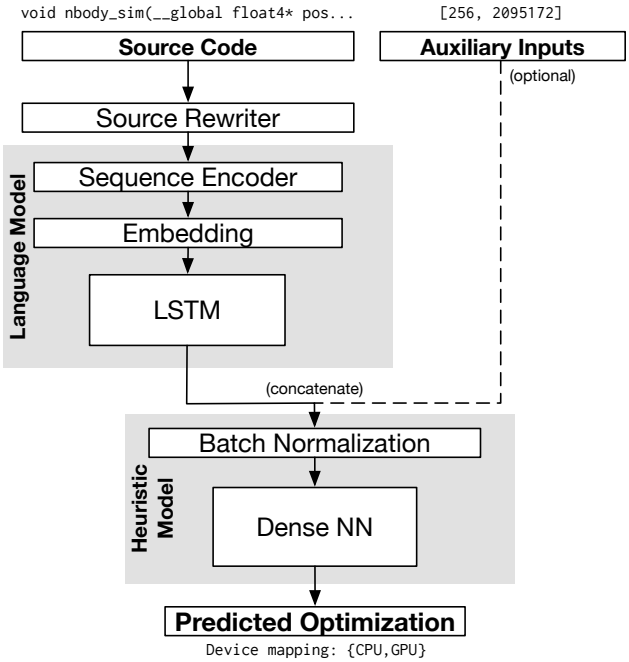


Figure 2: DeepTune architecture. Code properties are extracted from source code by the language model. They are fed, together with optional auxiliary inputs, to the heuristic model to produce the final prediction.

II. DeepTune: Learning On Raw Program Code

DeepTune is an end-to-end machine learning pipeline for optimization heuristics. Its primary input is the source code of a program to be optimized, and through a series of neural networks, it directly predicts the optimization which should be applied. By learning on source code, our approach is not tied to a specific compiler, platform, or optimization 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.

A. System Overview

Figure 2 provides an overview of the system. A source rewriter 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. We then optionally concatenate this vector 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 optimize the program.

DeepTune is open source[1]. We implemented the model in Keras, with TensorFlow [17] and Theano [18] backends.

B. 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, we employ state-of-the-art language modeling techniques, coupled with a series of generic, language agnostic code transformations.

Source Rewriter: To begin with, we apply a series of *source normalizing* transformations extended from our previous work [8]. 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 normalization is to simplify the task of modeling 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 3a and 3b show the source rewriting applied to a simple program.

Sequence Encoder: We encode source code as a sequence of integers for interpretation by neural networks, where each integer is an index into a predetermined vocabulary. In [8], a character based vocabulary is used. This minimizes the size of the vocabulary, but leads to long sequences which are harder to extract structure from. In [19], 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.

We designed a hybrid, partially tokenized approach. 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.

We first assembled a candidate vocabulary V_c for the OpenCL programming language containing the 208 data types, keywords, and language builtins of the OpenCL programming language. We then derived 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. Beginning with the first character in the corpus, our algorithm consumes the longest matching sequence from the candidate vocabulary. 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 3c shows the vocabulary derived for a single input source code Figure 3b.

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

```
1  //#define Elements
2  __kernel void memset_kernel(__global char * mem_d,
       ↪ short val, int number_bytes){
3      const int thread_id = get_global_id(0);
4      mem_d[thread_id] = val;
5  }
```

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

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

(b) The *streamcluster* kernel after source rewriting. Variable and function names are normalized, 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 3: Deriving a tokenized 1-of-k vocabulary encoding from an OpenCL source code.

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 colors.

To mitigate this, we use an *embedding*, 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 [20, 21]. 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 $D = 64$.

Sequence Characterization: Once source codes have been encoded into sequences of embedding vectors, neural networks are used to extract a fixed size vector which characterizes 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.

We use the the Long Short-Term Memory (LSTM) architecture [22] for sequence characterization. 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 [23]), 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 [24], an especially important capability for modeling 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).

We use a two layer LSTM network. The network receives a sequence of embedding vectors, and returns a single output vector, characterizing the entire sequence.

C. Auxiliary Inputs

We support an arbitrary number of additional real valued *auxiliary inputs* which can be optionally used to augment the source code input. We provide these inputs as a means of increasing the flexibility of our system, for example, to support applications in which the optimization heuristic depends on dynamic values which cannot be statically determined from the program code [3, 25]. When present, the values of auxiliary inputs are concatenated with the output of the language model, and fed into a heuristic model.

D. 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 optimization prediction.

We first normalize the values. Normalization 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 normalize, then scaling the auxiliary inputs could affect the training of the heuristic model. Normalization occurs in batches. We use the normalization method of [26], in which each scalar of the heuristic model's inputs $x_1 \dots x_n$ is normalized to a mean 0 and standard deviation of 1:

$$x'_i = \gamma_i \frac{x_i - E(x_i)}{\sqrt{Var(x_i)}} + \beta_i \tag{1}$$

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. We use rectifier activation functions $f(x) = \max(0, x)$ for the first layer due to their improved performance during training of deep networks [27]. For the output layer, we use sigmoid activation functions $f(x) = \frac{1}{1+e^{-x}}$ 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. We take the $\arg\max$ of the output layer to find the decision with the largest activation. For example, for a binary optimization heuristic the final layer will consist of two neurons, and the predicted optimization is the neuron with the largest activation.

E. 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, we simply use the raw program codes themselves.

The model is trained with Stochastic Gradient Descent (SGD), using the Adam optimizer [28]. For training data $X_1 \dots X_n$, SGD attempts to find the model parameters Θ that minimize the output of a loss function:

$$\Theta = \arg\min_{\Theta} \frac{1}{n} \sum_{i=1}^{n} \ell\left(X_i, \Theta\right) \tag{2}$$

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

To reduce training time, multiple inputs are *batched* together and are 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. We pad all sequences up to a fixed length of 1024 tokens using a special padding token, allowing matrices of `batch_size` × `max_seq_len` tokens to be processed simultaneously. We note that 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.

III. Experimental Methodology

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

A. 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

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
workgroup size	dynamic	#. work-items per kernel

(b) Values used in feature computation

Table I: Features used by Grewe *et al.* to predict heterogeneous device mappings for OpenCL kernels.

	Version	#. benchmarks	#. kernels
NPB (SNU [30])	1.0.3	7	114
Rodinia [31]	3.1	14	31
NVIDIA SDK	4.2	6	12
AMD SDK	3.0	12	16
Parboil [32]	0.2	6	8
PolyBench [33]	1.0	14	27
SHOC [34]	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 [32]	0.2	4	4
Total	-	17	17

(b) Case Study B: OpenCL Thread Coarsening Factor

Table II: 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 III: Experimental platforms.

and a choice of execution devices, the question then is on which device should we execute the program to maximize performance?

State-of-the-art: In [6], 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.

Expert Chosen Features: Table Ia shows the features used by their work. Each feature is an expression built upon the code and runtime metrics given in Table Ib.

Experimental Setup: We replicate the predictive model of Grewe *et al.* [6]. We replicated the experimental setup of [8] in which the experiments are extended to a larger set of 71 programs, summarized in Table IIa. The programs were evaluated on two CPU-GPU platforms, detailed in Table IIIa.

DeepTune Configuration: Figure 4a shows the neural network configuration of DeepTune for the task of predicting optimal device mapping. We use the OpenCL kernel source code as input, and the two dynamic values *workgroup size* and *data size* available to the OpenCL runtime.

Model Evaluation: We use *stratified 10-fold cross-validation* to evaluate the quality of the predictive models [29]. 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 dataset.

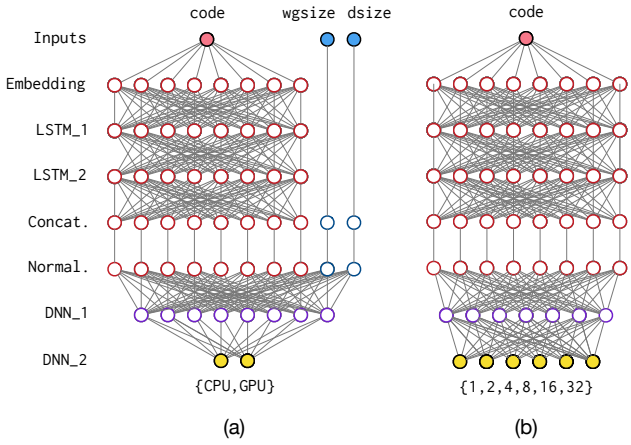
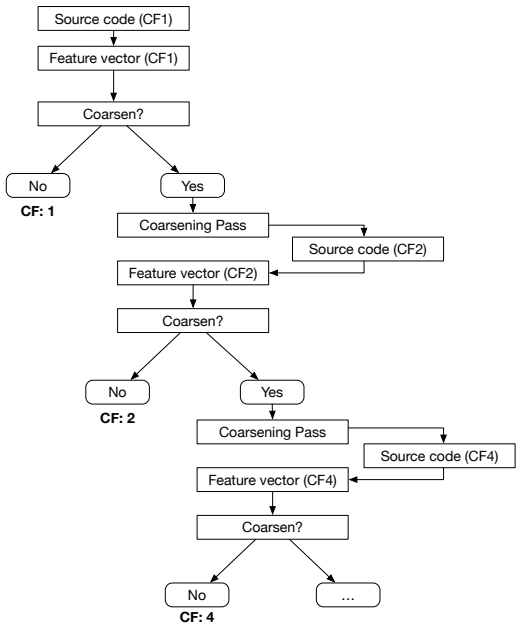


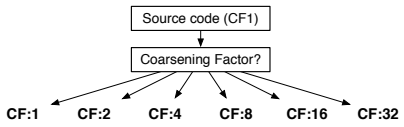
Figure 4: DeepTune neural networks, configured for (a) heterogeneous mapping, and (b) thread coarsening factor. The design stays almost the same regardless of the optimization problem. The only changes are the extra input for (a) and size of the output layers.

B. Case Study B: OpenCL Thread Coarsening Factor

Thread coarsening is an optimization for parallel programs in which the operations of two or more threads are fused together. This optimization can prove beneficial on certain combinations of programs and architectures, for example programs with a large potential for Instruction Level Parallelism on Very Long Instruction Word architectures.



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



(b) Our approach.

Figure 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 optimization and computing new feature vectors. Our approach simply predicts the coarsening factor directly from the source code.

State-of-the-art: Magni *et al.* present a predictive model for OpenCL thread coarsening in [7]. 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 5a. They select from one of six possible coarsening factors: $(1, 2, 4, 8, 16, 32)$, divided into 5 binary choices.

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 [35, 36]. 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.

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 IV: 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.

	#. 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 Norm .	66	64	264	256
DNN_1	32	32	2,144	2,080
DNN_2	2	6	66	198
Total			76,778	76,838

Table V: The size and number of parameters of the DeepTune components of Figure 4, configured for heterogeneous mapping (HM) and coarsening factor (CF).

Experimental Setup: We replicate the experimental setup of Magni *et al.* [7]. The thread coarsening optimization is evaluated on 17 programs, listed in Table IIb. Four different GPU architectures are used, listed in Table IIIb.

DeepTune Configuration: Figure 4b shows the neural network configuration. We use the OpenCL kernel as input, and directly predict the coarsening factor.

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.

Because [7] does not describe the parameters of the neural network, we perform an additional, *nested* cross-validation process to find the optimal model parameters. For every program in the training set, we evaluate 48 combinations of network parameters. We select the best performing configuration 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. We do not perform this tuning of hyper-parameters for DeepTune.

C. Comparison of Case Studies

For the two different optimization heuristics, the authors arrived at very different predictive model designs, with very different features. By contrast, we take exactly the same approach 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 V shows the similarity of our models. The only difference between our network design is the auxiliary inputs for Case Study A and the different number of optimization 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.

IV. Experimental Results

We evaluate the effectiveness of DeepTune for two distinct optimization tasks: predicting the optimal device to run a given program, and predicting thread coarsening factors.

We first compare DeepTune against two expert-tuned predictive models, showing that DeepTune outperforms the state-of-the-art in both cases. We then show that by leveraging knowledge learned from training DeepTune for one heuristic, we can boost training for the other heuristic, further improving performance. Finally, we analyze the working mechanism of DeepTune.

A. Case Study A: OpenCL Heterogeneous Mapping

Selecting the optimal execution device for OpenCL kernels is essential for maximizing performance. For a CPU/GPU heterogeneous system, this presents a binary choice. In this experiment, we compare our approach 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 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, we compute the relative performance of each program using the device selected by the Grewe *et al.* and DeepTune models. Figure 7 shows these speedups. Both predictive models significantly outperform the static mapping; the Grewe *et al.* model

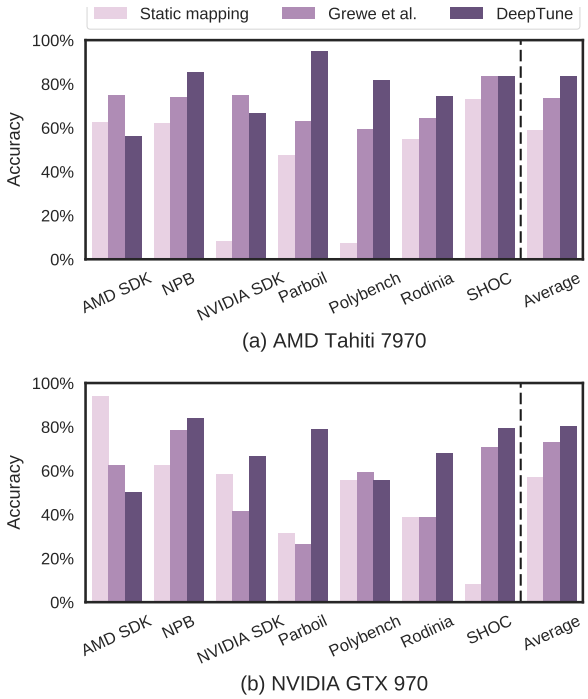


Figure 6: Accuracy of optimization heuristics for heterogeneous device mapping, aggregated by benchmark suite. The optimal static mapping achieves 58% accuracy. The Grewe *et al.* and DeepTune predictive models achieve accuracies of 73% and 84%, respectively.

achieves an average speedup of $2.91\times$ on AMD and $1.26\times$ on NVIDIA (geomean $1.18\times$). In 90% of cases, DeepTune matches or outperforms the predictions of the Grewe *et al.* model, achieving an average speedup of $3.34\times$ on AMD and $1.41\times$ on NVIDIA (geomean $1.31\times$). This 14% improvement in performance comes at a greatly reduced cost, requiring no intervention by humans.

B. 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 $1.36\times$.

Figure 8 shows speedups achieved by the Magni *et al.* and DeepTune models for all programs and platforms. We use as baseline the performance of programs without coarsening. On the four experimental platforms (AMD HD 5900, Tahiti 7970, NVIDIA GTX 480, and Tesla K20c), the Magni *et al.* model achieves average speedups of $1.21\times$, $1.01\times$, $0.86\times$, and $0.94\times$, respectively. DeepTune outperforms this, achieving speedups of $1.10\times$, $1.05\times$, $1.10\times$, and $0.99\times$.

Some programs — especially those with large divergent regions or indirect memory accesses — respond 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

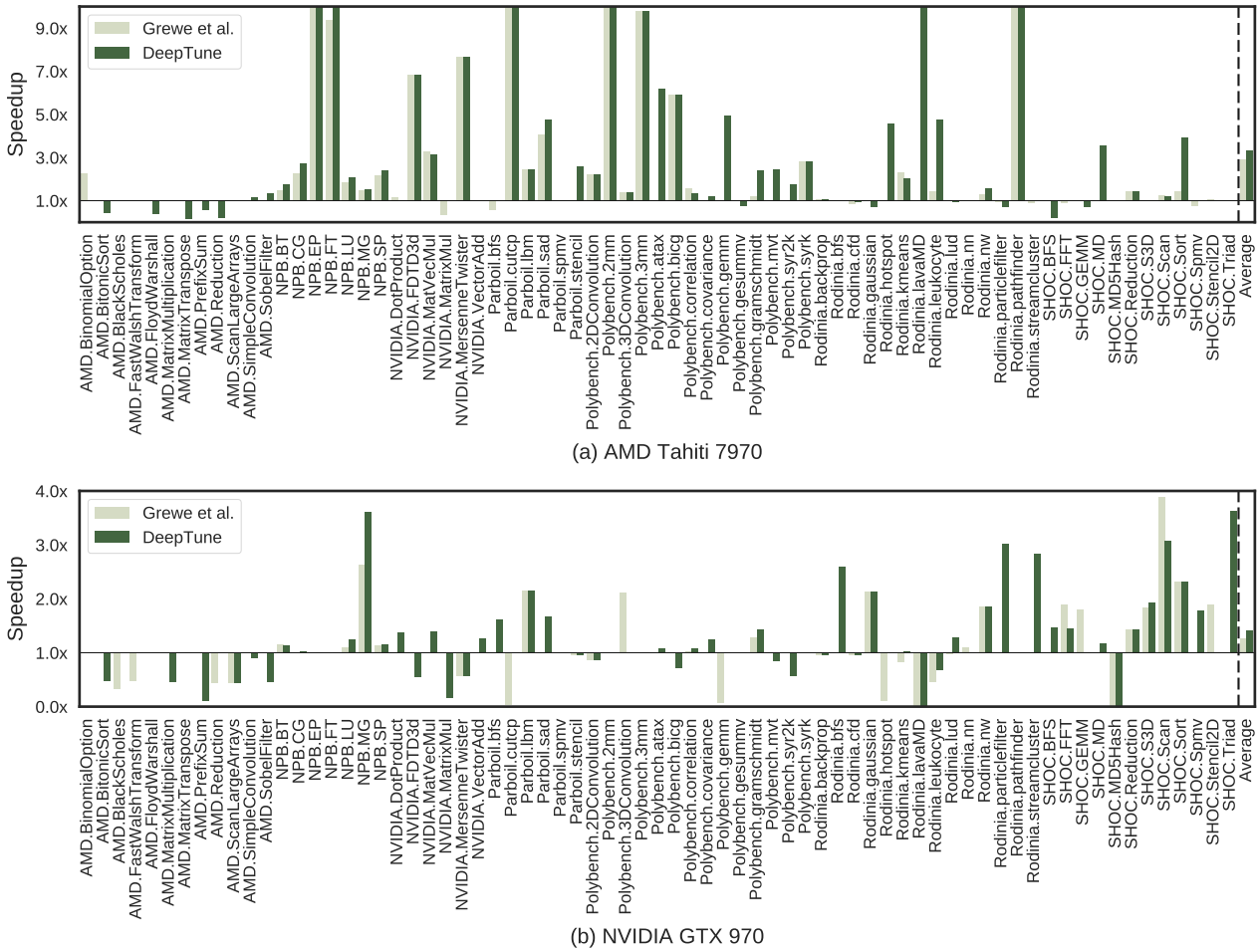


Figure 7: Speedup of predicted heterogeneous mappings over the best static mapping for both platforms. In (a) DeepTune achieves an average speedup of 3.43x over static mapping and 18% over Grewe *et al*. In (b) the speedup is 1.42x and 13% respectively.

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 can 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. Prior research has shown that data sparsity can be overcome using additional programs; in the following subsection we describe and test a novel strategy for training optimization heuristics on a small number of programs by exploiting knowledge learned from other optimization domains.

C. 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 specialized the layers become [37].

We hypothesized that this would be the case for DeepTune, enabling the novel transfer of information *across different optimization domains*. To test this, we extracted the language model — the `Embedding`, and `LSTM_{1,2}` layers — trained for the heterogeneous mapping task and *transferred* it over to the new task of thread coarsening. Since DeepTune keeps the same design for both optimization problems, this is as simple as copying the learned weights of the three layers. Then we trained the model as normal.

As shown in Figure 8, our newly trained model, DeepTune-TL has improved performance for 3 of the 4 plat-

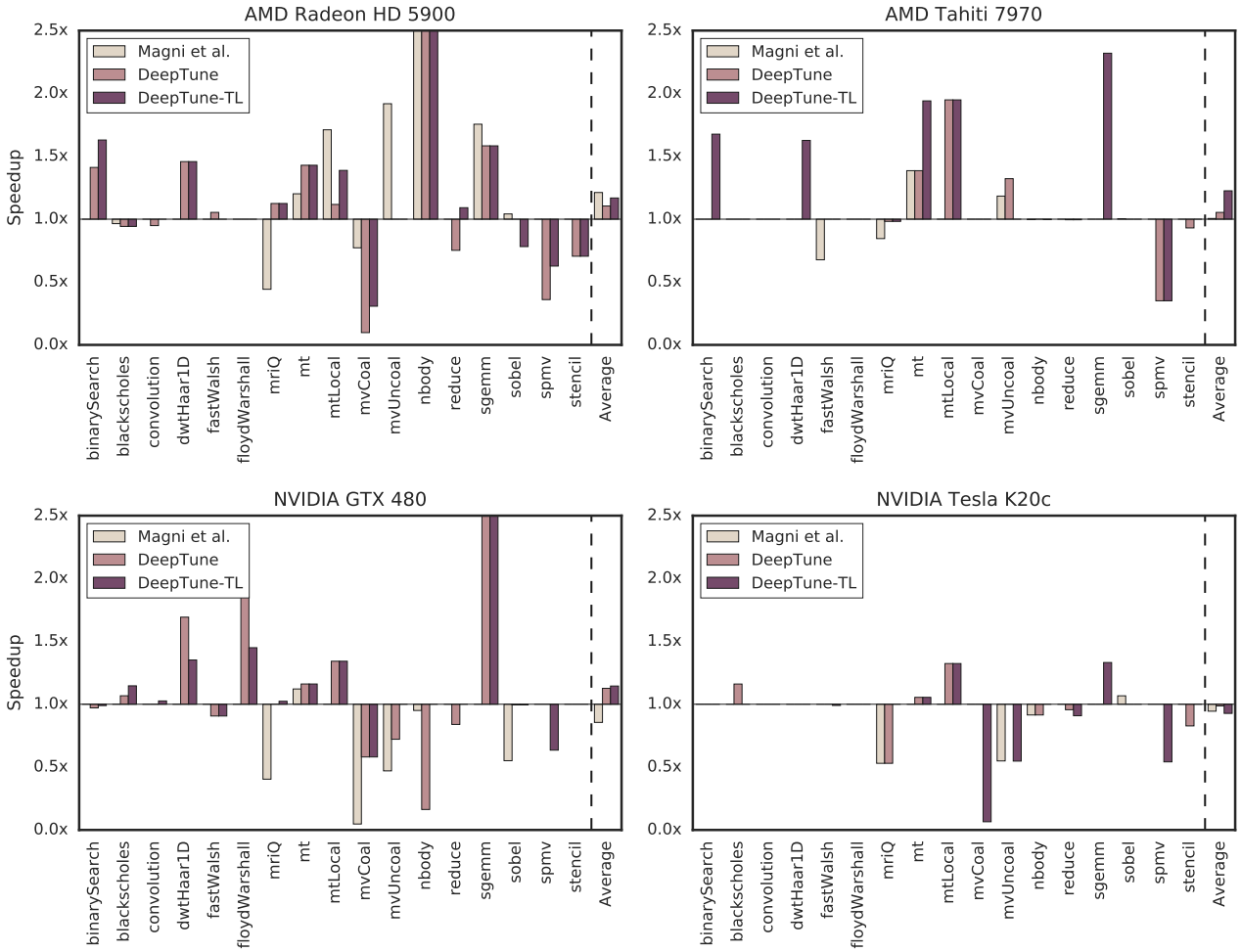


Figure 8: Speedups of predicted coarsening factors for each platform. DeepTune outperforms Magni *et al* on three of the four platforms. Transfer learning improves DeepTune speedups further, by 16% on average.

forms: 1.17×, 1.23×, 1.14×, 0.93×, providing an average 12% performance improvement over Magni *et al.* In 81% of cases, the use of transfer learning matched or improved the optimization 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, we match or improve performance in the majority of cases, but over-coarsening on three of the programs causes a modest reduction in average performance. We suspect that for this platform, further performance results are necessary due to its unusual optimization profile.

D. DeepTune Internal Activation States

We have shown that DeepTune automatically outperforms state-of-the-art predictive models for which experts have invested a great amount of time in engineering features. In this subsection we attempt to illuminate the inner workings, using a single example from Case Study B: predicting the thread coarsening factor for Parboil's `mriQ` benchmark on four different platforms.

Figure 9 shows the DeepTune configuration, with visual overlays showing the internal state. From top to bottom, we begin first with the input, which 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 9b. The rewritten source code is tokenized and encoded in a 1-of-k vocabulary. Figure 9c shows the first 80 elements of this encoded sequence as a heatmap in which each cell's color 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

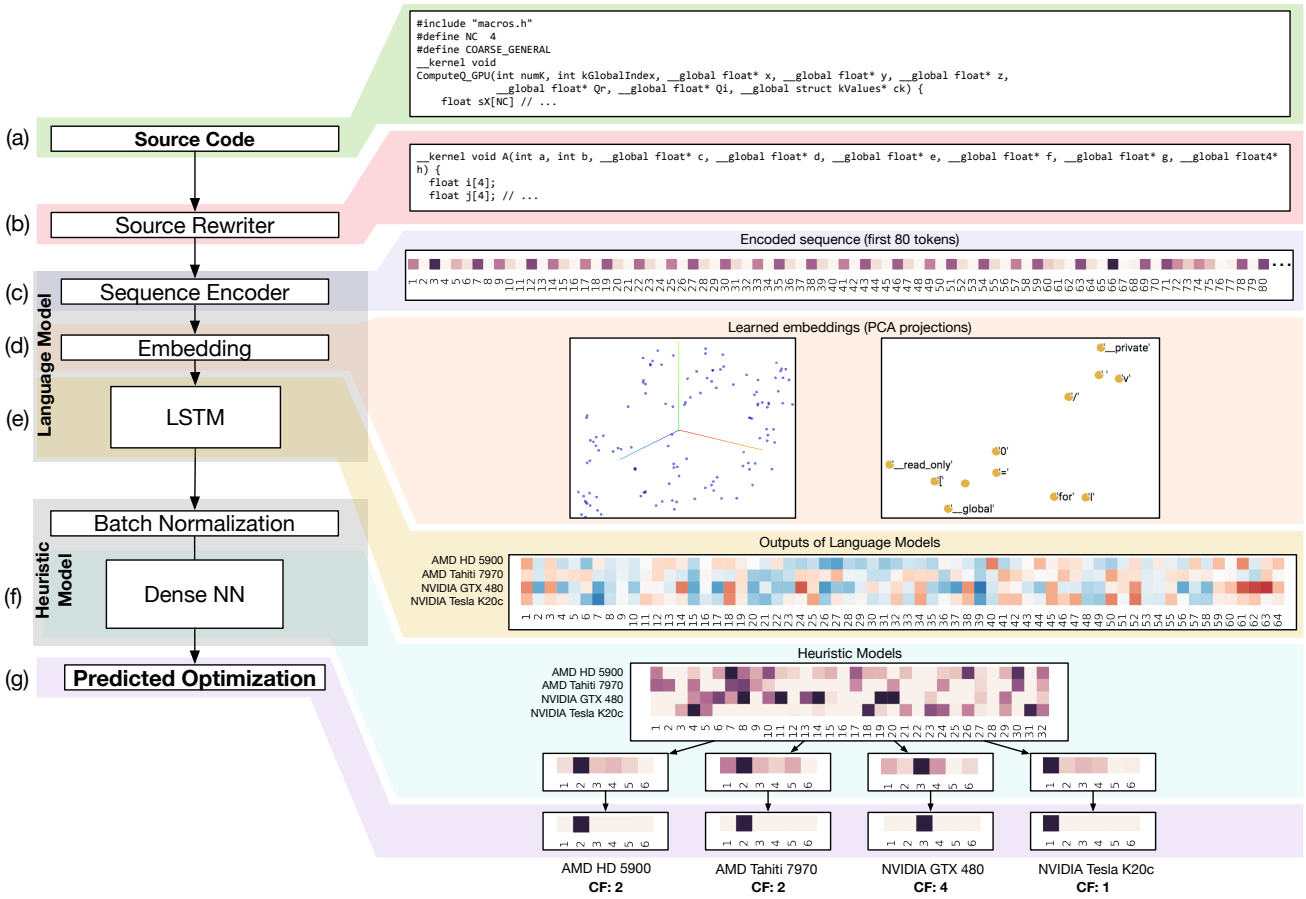


Figure 9: Visualizing the internal state of DeepTune when predicting coarsening factor for Parboil's `mriQ` benchmark on four different architectures. The activations in each layer of the four models increasingly diverge the lower down the network.

related tokens together. As such, they may differ between the four platforms. Figure 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, we see 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 characterize the entire sequence. Figure 9e shows the neurons in this layer for each of the four platforms, using a red-blue heatmap to visualize 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 specialized to the target platform.

As information flows through the network, the layers become progressively more specialized to the specific platform. We see this in Figure 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. In Figure 9g we take the largest activation of the output layer 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%.

V. Related Work

Machine learning has emerged as a viable means in automatically constructing heuristics for code optimization [38–43]. Its great advantage is that it can adapt to changing hardware platforms as it has no a priori assumptions about their behavior. The success of machine learning based code optimization has required having a set of high-quality features that can capture the important characteristics of the target program. 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 program features have been used in compiler-based machine learning. These include static code

structures [44] and runtime information such as system load [45] and performance counters [46]. 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 [3, 47], or projected into a lower dimensional space [48, 49]. FEAST employs a range of existing feature selection methods to select useful candidate features [50]. Unlike these approaches, DeepTune extracts features and reduces the dimensionality of the feature space completely internally and without expert guidance.

Park *et al.* present a unique graph-based approach for feature representations [51]. They use a Support Vector Machine where the kernel is based on a graph similarity metric. Their technique still requires hand coded features at the basic block level, but thereafter, graph similarity against each of the training programs takes the place of global features. Being a kernel method, it requires that training data graphs be shipped with the compiler, which may not scale as the size of the training data grows with the number of instances, and some training programs may be very large. Finally, their graph matching metric is expensive, requiring $O(n^3)$ to compare against each training example. By contrast, our method does 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 [9, 10]. 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. The work of [10] uses genetic programming to search for features, and required a huge grammar to be written, some 160kB in length. Although much of this can be created from templates, selecting the right range of capabilities and search space bias is non trivial and up to the expert. The work of [9] expresses the space of features via logic programming over relations that represent information from the IRs. It greedily searches for expressions that represent good features. However, their 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 [52]. Their approach profiles the targetprogram using several carefully selected compiler options to see how programruntime changes under these options for a given micro-architecture setting. Theythen use the program "reactions" to predict the best available applicationspeedup. While their approach does not use static code features, developers mustcarefully select a few settings from a large number of candidate options forprofiling, because poorly chosen options can significantly affect the quality ofthe model. Moreover, the program must be run several times before optimization,while our technique does not require the program to be profiled.

In recent years, machine learning techniques have been employed to model and learn from program source code on various tasks. These include mining coding conventions [15] and idioms [14], API example code [53] and pseudo-code generation [54], and benchmark generation [8]. Our work is the first attempt to extend the already challenging task of modeling distributions over source code to learning distributions over source code with respect to code optimizations.

Recently, deep neural networks have been shown to be a powerful tool for feature engineering in various tasks including image recognition [11, 12] and audio processing [13]. No work so far has applied deep neural networks for program feature generation. Our work is the first to do so.

VI. Conclusions

Applying machine learning to compiler and runtime optimizations requires generating features first. This is a time consuming process, it needs supervision by an expert, and even then we cannot be sure that the selected features are optimal. In this paper we present a novel tool for building optimization heuristics, DeepTune, which forgoes feature extraction entirely, relying on powerful language modeling 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.

Our 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 we do not tailor our model design or parameters for the optimization task at hand, yet we achieve performance on par with and in most cases *exceeding* state-of-the-art predictive models.

We used DeepTune to automatically construct heuristics for two challenging compiler and runtime optimization problems, find that, in both cases, we outperform state-of-the-art predictive models by 14% and 12%. We have also shown that the DeepTune architecture allows us to exploit information learned from another optimization problem to give the learning a boost. Doing so provides up to a 16% performance improvement when training using a handful of programs. We suspect this approach will be useful in other domains for which training data are a scarce resource.

In future work, we will extend our heuristic construction approach by automatically learning dynamic features over raw data; apply unsupervised learning techniques [56] over unlabeled source code to further improve learned representations of programs; and deploy trained DeepTune heuristic models to low power embedded systems using quantization and compression of neural networks [57].

ACKNOWLEDGMENTS

This work was supported by the UK Engineering and Physical Sciences Research Council under grants EP/L01503X/1 (CDT in Pervasive Parallelism), EP/M01567X/1 (SANDeRs), EP/M015793/1 (DIVIDEND), and EP/P003915/1 (SUMMER). The code and data for this paper are available at: https://chriscummins.cc/pact17.

REFERENCES

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

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

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

[4] 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.

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

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

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

[8] C. Cummins, P. Petoumenos, W. Zang, and H. Leather. "Synthesizing Benchmarks for Predictive Modeling." In: *CGO*. IEEE, 2017.

[9] 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.

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

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

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

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

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

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

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

[17] 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).

[18] 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.

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

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

[21] 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.

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

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

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

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

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

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

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

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

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

[31] 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.

[32] 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).

[33] 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.

[34] 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.

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

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

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

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

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

[40] S. Muralidharan, A. Roy, M. Hall, M. Garland, and P. Rai. "Architecture-Adaptive Code Variant Tuning." In: *ASPLOS*. ACM, 2016.

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

[42] J. Ren, L. Gao, and Z. Wang. "Optimise Web Browsing on Heterogeneous Mobile Platforms: A Machine Learning Based Approach." In: *INFOCOM*. 2017.

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

[44] 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).

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

[46] 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.

[47] B. Taylor, V. S. Marco, and Z. Wang. "Adaptive Optimization for OpenCL Programs on Embedded Heterogeneous Systems." In: *LCTES*. 2017.

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

[49] 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.

[50] 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).

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

[52] 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.

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

[54] 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 (T)." In: *ASE*. IEEE, 2015.

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

[56] 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.

[57] 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).

APPENDIX
ARTIFACT DESCRIPTION

A. Abstract

Our research artifact consists of interactive Jupyter notebooks. The notebooks enable users to replicate all experiments in the paper, evaluate results, and plot figures.

B. Description

1) Check-list (Artifact Meta Information):

- **Run-time environment:** Ubuntu Linux and a web browser.
- **Hardware:** Users with an NVIDIA GPU may enable CUDA support to speed up computation of experiments.
- **Output:** Trained neural networks, predictive model evaluations, figures and tables from the paper.
- **Experiment workflow:** Install and run Jupyter notebook server; interact with and observe results in web browser.
- **Experiment customization:** Edit code and parameters in Jupyter notebooks.
- **Publicly available?:** Yes, code and data. See: https://chriscummins.cc/pact17/

2) How Delivered: A publicly available git repository containing Jupyter notebooks and experimental data.

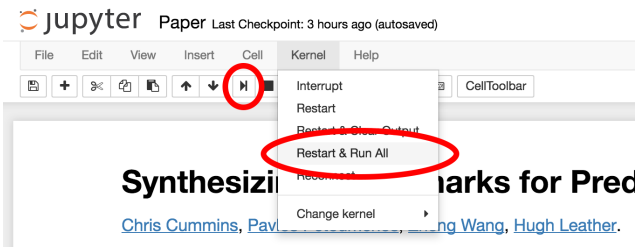
C. Installation

See https://chriscummins.cc/pact17/ for instructions. The `code` directory contains the Jupyter notebooks. Following the build instructions described in `code/README.md`, the full installation process is:

```
$ ./bootstrap.sh | bash
$ ./configure
$ make
```

D. Experiment Workflow

1) Launch the Jupyter server using the command: `make run`.
2) In a web browser, navigate to: `http://localhost:8000`.
3) Select a Jupyter notebook to open it.
4) Repeatedly press the *play* button (tooltip is "run cell, select below") to step through each cell of the notebook.
   OR select "Kernel" > "Restart & Run All" from the menu to run all of the cells in order.



E. Evaluation and Expected Result

Code cells within Jupyter notebooks display their output inline, and may be compared against the values in the paper. Expected results are described in text cells.

F. Experiment Customization

The experiments are fully customizable. The Jupyter notebook can be edited "on the fly". Simply type your changes into the cells and re-run them.
Note that some of the code cells depend on the values of prior cells, so must be executed in sequence. Select "Kernel" > "Restart & Run All" from the menu to run all of the cells in order.

G. Notes

For more information about DeepTune, visit:
https://chriscummins.cc/deeptune
For more information about Artifact Evaluation, visit:
http://cTuning.org/ae/submission-20170414.html

6.2 Conclusion

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

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 **??** 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 **??** 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 conions 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 conions 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.

[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.

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

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

[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.

[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.

[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.

[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+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).

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

[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).

[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.

[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.

[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.

[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).

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

[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.

[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.

[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.

[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).

[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.

[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+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).

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

[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).

[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.

[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+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.

[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).

[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.

[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.

[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.

[RVK15] V. Raychev, M. Vechev, and A. Krause. “Predicting Program Properties from ”Big Code””. In: *POPL*. 2015.

[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.

[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.

[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.

[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+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.

[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).

[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).

[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).

[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.