Deep Learning over Programs

Chris Cummins

Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2019

Abstract

TODO: The crisis…

This thesis presents deep learning methodologies to simplify compiler construction. First, a generative model for source code is described, capable of producing executable programs derived from language models trained on open source corpora. The effectiveness of programs generated using these methodologies is then explored. TODO …

Acknowledgements

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 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2018.

- 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 International Symposium on Code Generation and Optimization (CGO)*, 2018.

(*Chris Cummins*)

Dedication Lorem Ipsum.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Listings

Chapter 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.

1.1 Machine Learning in Compilers

1.2 The Problem

1.3 Contributions

1.4 Structure

1.5 Summary

Chapter 2

Deep Learning and Methodologies

2.1 Introduction

2.2 Terminology

Notation: $\odot$ point-wise multiplication of tensors.

2.3 Machine Learning

2.3.1 Principal Component Analysis

2.3.2 Decision Trees

2.4 Deep Learning

2.4.1 Recurrent Neural Networks

2.4.1.1 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 maximize 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 maximize the probability of assigning the correct label to both training examples and samples from G, and train G to minimize $\log(1 - D(G(\boldsymbol{z})))$. 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 Evaluation Techniques

2.6 Summary

Chapter 3

Related Work

It is widely accepted that parallel programming is difficult, and the continued repetition of this claim has become something of a trite mantra for the parallelism research community. An interesting digression is to discuss some of the ways in which researchers have attempted to tackle this difficult problem, and why, despite years of research, it remains an ongoing challenge.

The most ambitious and perhaps daring field of parallelism research is that of automatic parallelisation, where the goal is to develop methods and systems to transform arbitrary sequential code into parallelised code. This is a well studied subject, with the typical approach being to perform these code transformations at the compilation stage. In Banerjee, Eigenmann, and Nicolau's thorough review [BEN93] on the subject, they outline the key challenges of automatic parallelisation:

- determining whether sequential code can be legally transformed for parallel execution; and

- identifying the transformation which will provide the highest performance improvement for a given piece of code.

Both of these challenges are extremely hard to tackle. For the former, the difficulties lie in performing accurate analysis of code behaviour. Obtaining accurate dynamic dependency analysis at compile time is an unsolved problem, as is resolving pointers and points-to analysis [Atk13; GLS01; Hin01].

The result of these challenges is that reliably performant, automatic parallelisation of arbitrary programs remains a far from reached goal; however, there are many note worthy variations on the theme which have been able to achieve some measure of success.

One such example is speculative parallelism, which circumvents the issue of having incomplete dependency information by speculatively executing code regions in parallel while performing dependency tests at runtime, with the possibility to fall back to "safe" sequential execution if correctness guarantees are not met [Pra10; TG10]. In [Jim+14], Jimborean et al. present a system which combines polyhedral transformations of user code with binary algorithmic skeleton implementations for speculative parallelisation, reporting speedups over sequential code of up to 15.62× on a 24 core processor.

Another example is PetaBricks, which is a language and compiler enabling parallelism through "algorithmic choice" [AC10; Ans09]. With PetaBricks, 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 autotuning techniques for enabling optimised multigrid programs [Cha+09], with the wider ambition that these autotuning techniques may be applied to all algorithmic choice programs [Ans14]. While this helps produce efficient parallel programs, it places a great burden on the developer, requiring them to provide enough contrasting implementations to make a search of the optimisation space fruitful.

Annotation-driven parallelism takes a similar approach. The user annotates their code to provide "hints" to the compiler, which can then perform parallelising transformations. A popular example of this is OpenMP, which uses compiler pragmas to mark code sections for parallel or vectorised execution [DE98]. Previous work has demonstrated code generators for translating OpenMP to OpenCL [GWO13] and CUDA [LME09]. Again, annotation-driven parallelism suffers from placing a burden on the developer to identify the potential areas for parallelism, and lacks the structure that algorithmic skeletons provide.

Algorithmic skeletons contrast the goals of automatic parallelisation by removing the challenge of identifying potential parallelism from compilers or users, instead allowing users to frame their problems in terms of well defined patterns of computation. This places the responsibility of providing performant, well tuned implementations for these patterns on the skeleton author.

3.1 Iterative Compilation & Machine Learning

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. One of the the first formalised publications of the technique appeared in 1998 by Bodin et al. [Bod+98]. Iterative compilation has since been demonstrated to be a highly effective form of empirical performance tuning for selecting compiler optimisations.

Given the huge number of possible compiler optimisations (there are 207 flags and parameters to control optimisations in GCC v4.9), it is often unfeasible to perform an exhaustive search of the entire optimisation space, leading to the development of methods for reducing the cost of evaluating configurations. These methods reduce evaluation costs either by shrinking the dimensionality or size of the optimisation space, or by guiding a directed search to traverse a subset of the space.

Machine learning has been successful applied to this problem, in [SMR03], using "meta optimisation" to tune compiler heuristics through an evolutionary algorithm to automate the search of the optimisation space. Fursin et al. continued this with Milepost GCC, the first machine learning-enabled self-tuning compiler [Fur+11]. A recent survey of the use of machine learning to improve heuristics quality by Burke et al. concludes that the automatic *generation* of these self-tuning heuristics but is an ongoing research challenge that offers the greatest generalisation benefits [Bur+13].

In [FT10; Fur+14; MF13], Fursin et al. advocate a collaborative and "big data" driven approach to autotuning, arguing that the challenges facing the widespread adoption of autotuning and machine learning methodologies can be attributed to: a lack of common, diverse benchmarks and datasets; 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, the Collective Mind knowledge system, which, while in early stages of ongoing development, is intended to provide a modular infrastructure for sharing autotuning performance data and related artifacts across the internet. In addition to sharing performance data, the approach taken in this thesis emphasises the collective *exploitation* of such performance data, so that data gathered from one device may be used to inform the autotuning decisions of another. This requires each device to maintain local caches of shared data to remove the network overhead that would be present from querying a single centralised data store during execution of a hot path. The current implementation of Collective Mind uses a NoSQL JSON format for storing performance data. The relational schema used in this thesis offers greater scaling performance and lower storage overhead as the amount of performance data grows.

Whereas iterative compilation requires an expensive offline training phase to search an optimisation space, dynamic optimisers perform this optimisation space exploration at runtime, allowing programs to respond to dynamic features “online”. This is a challenging task, as a random search of an optimisation space may result in configurations with vastly suboptimal performance. In a real world system, evaluating many suboptimal configurations will cause a significant slowdown of the program. Thus a requirement of dynamic optimisers is that convergence time towards optimal parameters is minimised.

Existing dynamic optimisation research has typically taken a low level approach to performing optimisations. Dynamo is a dynamic optimiser which performs binary level transformations of programs using information gathered from runtime profiling and tracing [BDB00]. While this provides the ability to respond to dynamic features, it restricts the range of optimisations that can be applied to binary transformations. These low level transformations cannot match the performance gains that higher level parameter tuning produces.

An interesting related tangent to iterative compilation is the development of so-called “superoptimisers”. In [Mas87], 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 superoptimiser 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 exponential growth in the size of the search space is far too expensive for all but the smallest of hot paths, typically less than 13 instructions. 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 superoptimiser 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.

As briefly discussed in Section **??**, the complex interactions between optimisations and heterogeneous hardware makes performance tuning for heterogeneous parallelism a difficult task. Performant GPGPU programs require careful attention from the developer to properly manage data layout in DRAM, caching, diverging control flow, and thread communication. The performance of programs depends heavily on fully utilising zero-overhead thread scheduling, memory bandwidth, and thread grouping. Ryoo et al. illustrate the importance of these factors by demonstrating speedups of up to 432× for matrix multiplication in CUDA by appropriate use of tiling and loop unrolling [Ryo+08a]. The importance of proper exploitation of local shared memory and synchronisation costs is explored in [Lee+10].

In [CW14], data locality optimisations are automated using a description of the hardware and a memory-placement-agnostic compiler. The authors demonstrate impressive speedups of up to 2.08×, although at the cost of requiring accurate memory hierarchy descriptor files for all targeted hardware. The descriptor files must be hand generated, requiring expert knowledge of the underlying hardware in order to properly exploit memory locality.

Data locality for nested parallel patterns is explored in [Lee+14]. The authors use an automatic mapping strategy for nested parallel skeletons on GPUs, which uses a custom intermediate representation and a CUDA code generator, achieving 1.24× speedup over hand optimised code on 7 of 8 Rodinia benchmarks.

Reduction of the GPGPU optimisation space is demonstrated in [Ryo+08b], using the common subset of optimal configurations across a set of training examples. This technique reduces the search space by 98%, although it does not guarantee that for a new program, the reduced search space will include the optimal configuration.

Magni, Dubach, and O'Boyle demonstrated that thread coarsening of OpenCL kernels can lead to speedups in program performance between 1.11× and 1.33× in [MDO14]. The authors achieve this using a machine learning model to predict optimal thread coarsening factors based on the static features of kernels, and an LLVM function pass to perform the required code transformations.

A framework for the automatic generation of OpenCL kernels from high-level programming concepts is described in [SFD15]. A set of rewrite rules is used to transform high-level expressions to OpenCL code, creating a space of possible implementations. This approach is ideologically similar to that of PetaBricks, in that optimisations are made through algorithmic choice, although in this case the transformations are performed automatically at the compiler level. The authors report performance on a par with that of hand written OpenCL kernels.

An enumeration of the optimisation space of Intel Thread Building Blocks in [CM08] shows that runtime knowledge of the available parallel hardware can have a significant impact on program performance. Collins, Fensch, and Leather exploit this in [CFL12], first using Principle Components Analysis to reduce the dimensionality of the space of possible optimisation parameters, followed by a search of parameter values to optimise program performance by a factor of 1.6× over values chosen by a human expert. In [Col+13], they extend this using static feature extraction and nearest neighbour classification to further prune the search space, achieving an average 89% of the oracle performance after evaluating 45 parameters.

Dastgeer developed a machine learning based autotuner for the SkePU skeleton library in [Das11]. Training data is used to predict the optimal execution device (i.e. CPU, GPU) for a given program by predicting execution time and memory copy overhead based on problem size. The autotuner only supports vector operations, and there is limited cross-architecture evaluation. In [DK15], the authors extend SkePU to improve the data consistency and transfer overhead of container types, reporting up to a 33.4× speedup over the previous implementation.

Stencil codes have a variety of computationally expensive uses from fluid dynamics to quantum mechanics. Efficient, tuned stencil kernels are highly sought after, with early work in 2003 by Bolz et al. demonstrating the capability of GPUs for massively parallel stencil operations [Bol+03]. In the resulting years, stencil codes have received much attention from the performance tuning research community.

Ganapathi et al. demonstrated early attempts at autotuning multicore stencil codes in [Gan+09], drawing upon the successes of statistical machine learning techniques in the compiler community, as discussed in Section 3.1. They present an autotuner 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, using 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. They present a compelling argument for the use of energy efficiency as an optimisation target in addition to runtime, citing that it was the power wall that lead to the multicore revolution in the first place. Their choice of only 2 benchmarks and 2 platforms makes the evaluation of their autotuner somewhat limited.

Berkeley et al. targeted 3D stencils code performance in [Ber+08]. Stencils are decomposed into core blocks, sufficiently small to avoid last level cache capacity misses.

These are then further decomposed to thread blocks, designed to exploit common locality threads may have within a shared cache or local memory. Thread blocks are divided into register blocks in order to take advantage of data level parallelism provided by the available registers. Data allocation is optimised on NUMA systems. The performance evaluation considers speedups of various optimisations with and without consideration for host/device transfer overhead.

Kamil et al. present an autotuning framework in [Kam+10] which accepts as input a Fortran 95 stencil expression and generates tuned shared-memory parallel implementations in Fortan, C, or CUDA. The system uses an IR to explore autotuning transformations, enumerating a subset of the optimisation space and recording only a single execution time for each configuration, reporting the fastest. They demonstrate their system on 4 architectures using 3 benchmarks, with speedups of up to 22× compared to serial implementations. The CUDA code generator does not optimise for the GPU memory hierarchy, using only global memory. As demonstrated in this thesis, improper utilisation of local memory can hinder program performance by two orders of magnitude. There is no directed search or cross-program learning.

In [ZM12], Zhang and Mueller present a code generator and autotuner for 3D Jacobi stencil codes. Using a DSL to express kernel functions, the code generator performs substitution from one of two CUDA templates to create programs for execution on GPUs. GPU programs are parameterised and tuned for block size, block dimensions, and whether input data is stored in read only texture memory. This creates an optimisation space of up to 200 configurations. In an evaluation of 4 benchmarks, the authors report impressive performance that is comparable with previous implementations of iterative Jacobi stencils on GPUs [HPS12; PF10]. The dominating parameter is shown to be block dimensions, followed by block size, then read only memory. The DSL presented in the paper is limited to expressing only Jacobi Stencils applications. Critically, their autotuner requires a full enumeration of the parameter space for each program. Since there is no indication of the compute time required to gather this data, it gives the impression that the system would be impractical for the needs of general purpose stencil computing. The autotuner presented in this thesis overcomes this drawback by learning parameter values which transfer to unseen stencils, without the need for an expensive tuning phase for each program and architecture.

In [CSB11], Christen, Schenk, and Burkhart presentf a DSL for expressing stencil codes, a C code generator, and an autotuner for exploring the optimisation space of blocking and vectorisation strategies. The DSL supports stencil operations on arbitrarily high-dimensional grids. The autotuner performs either an exhaustive, multi-run Powell search, Nelder Mead, or evolutionary search to find optimal parameter values. They evaluate their system on two CPUS and one GPU using 6 benchmarks. A comparison of tuning results between different GPU architectures would have been welcome, as the results presented in this thesis show that devices have different responses to optimisation parameters. The authors do not present a ratio of the available performance that their system achieves, or how the performance of optimisations vary across benchmarks or devices.

A stencil grid can be decomposed into smaller subsections so that multiple GPUs can operate on each subsection independently. This requires a small overlapping region where each subsection meets — the halo region — to be shared between devices. For iterative stencils, values in the halo region must be synchronised between devices after each iteration, leading to costly communication overheads. One possible optimisation is to deliberately increase the size of the halo region, allowing each device to compute updated values for the halo region, instead of requiring a synchronisation of shared state. This reduces the communication costs between GPUs, at the expense of introducing redundant computation. Tuning the size of this halo region is the goal of PARTANS [LFC13], an autotuning framework for multi-GPU stencil computations. Lutz, Fensch, and Cole explore the effect of varying the size of the halo regions using six benchmark applications, finding that the optimal halo size depends on the size of the grid, the number of partitions, and the connection mechanism (i.e. PCI express). The authors present an autotuner which determines problem decomposition and swapping strategy offline, and performs an online search for the optimal halo size. The selection of overlapping halo region size compliments the selection of workgroup size which is the subject of this thesis. However, PARTANS uses a custom DSL rather than the generic interface provided by SkelCL, and PARTANS does not learn the results of tuning across programs, or across multiple runs of the same program.

Using active learning to minimize the cost of iterative compilation [Ogi+17]. Continuous learning of compiler heuristics [Fur+11; TC13].

3.1.1 Feature Generation for Compilers

Machine learning has emerged as a viable means in automatically constructing heuristics for code optimization [Aga+06; Din+15; KC12; Mur+16; SA05; WO10]. 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 [Jia+10] and runtime information such as system load [WWO14] and performance counters [Dub+09]. 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 [SA05], or projected into a lower dimensional space [Col+13; Dub+07]. FEAST employs a range of existing feature selection methods to select useful candidate features [Tin+16]. 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 [PCA12]. They use a Support Vector Machine 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. 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 [LBO14; Nam+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 [LBO14] 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 [Nam+10] 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 [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 optimization, while our technique does not require the program to be profiled.

3.2 Deep Learning for Programming Langauges

Deep learning is a nascent branch of machine learning in which deep or multi-level graphs of processing layers are used to detect patterns in natural data [Bud15; LBH15]. It is proving especially successful for its ability to 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].

In past work I used the Long Short-Term Memory (LSTM) architecture of Recurrent Neural Network (RNN) [Mik+15; SSN12] to generate sequences of OpenCL code [Cum+17b]. The LSTM network architecture comprises recurrent 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 [Gra13; GS05]. Although this is the first application of deep learning for generating executable programs, 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 LSTMs for sequence modeling is demonstrated in [SVL14]. Sutskever, Vinyals, and Le apply two LSTM 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. Such sequence transformations should be considered for the purpose of program generation.

The application of language modeling for generating executable programs is novel. In training on large corpuses of hand-written code, the language model learns the human biases which are present in common codes [CBN16]. While such human-induced basiases can prove controverial in social domains [Bol+16; JWC17], this enables the generation of programs which, unlike other approaches to program generation, are representative of real workloads.

Neural networks are computationally expensive, though their implementations can be generic and parallelised. Library implementations are available in Torch [CKF11], Caffe [Jia+14], and TensorFlow [Aba+16]. The increasing size and depth of computation graphs in deep learning has challenged the ability to compute results in reasonable time. Possible methods for reducing computational overhead involve fusing operations across layers in the graph using domain specific languages [Ash+15; Pot+15; Tru+16]; decoupling interfaces between layers using small networks to synthesise learning gradients during training [Jad+16]; and specialising hardware for computing data parallel workloads using FPGAs and ASICs [MS10].

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 [All+14]. JSNice leverages probabilistic graphical models to predict program properties such as identifier names for Javascript [RVK15]. Gu et al. use deep learning to generate example code for APIs as responses to natural language queries [Gu+16]. Allamanis, Peng, and Sutton use attentional neural networks to generate summaries of source code [APS16]. Wong, Yang, and Tan mines Q&A site StackOverflow to automatically generate code comments [WYT13]. Raychev, Vechev, and Yahav use statistical models to provide contextual code completion [RVY14].

There is an increasing interest in mining source code repositories at large scale [AS13; Kal+09; Whi+15]. Previous studies have involved data mining of GitHub to analyze software engineering practices [Bai+14; GAL14; VFS15; Wu+14]; however, no work so far has exploited mined source code for program generation.

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. One of the the first formalised publications of the technique appeared in 1998 by Bodin et al. [Bod+98]. Iterative compilation has since been demonstrated to be a highly effective form of empirical performance tuning for selecting compiler optimisations.

An enumeration of the optimisation space of Intel Thread Building Blocks in [CM08] shows that runtime knowledge of the available parallel hardware can have a significant impact on program performance. Collins, Fensch, and Leather exploit this in [CFL12], first using Principle Components Analysis to reduce the dimensionality of the optimisation space, followed by a search of parameter values to improve program performance by a factor of 1.6× over values chosen by a human expert. In [Col+13], they extend this using static feature extraction and nearest neighbour classification to further prune the search space, achieving an average 89% of the oracle performance after evaluating 45 parameters.

Frameworks for iterative compilation offer mechanisms for abstracting the iterative compilation process from the optimisation space. *OpenTuner* presents a generic framework for optimisation space exploration [Ans+13]. *CLTune* is a generic autotuner for OpenCL kernels [NC15]. Both frameworks implement *search*, however, the huge number of possible compiler optimisations makes such a search expensive to perform for every new configuration of program, architecture and dataset.

Machine learning has been used to guide iterative compilation and predict optimisations for code. Stephenson, Martin, and Reilly use "meta optimisation" to tune compiler heuristics through an evolutionary algorithm to automate the search of the optimisation space [SMR03]. Fursin et al. continued this with Milepost GCC, the first machine learning-enabled self-tuning compiler [Fur+11]. A survey of machine learning heuristics quality concludes that the automatic *generation* of self-tuning heuristics is an ongoing research challenge that offers the greatest generalisation benefits [Bur+13].

Dastgeer developed a machine learning based autotuner for the SkePU skeleton library in [Das11]. Training data is used to predict the optimal execution device (i.e. CPU, GPU) for a given program by predicting execution time and memory copy overhead based on problem size. The autotuner only supports vector operations, and there is limited cross-architecture evaluation. In [DK15], the authors extend SkePU to improve the data consistency and transfer overhead of container types, reporting up to a 33.4× speedup over the previous implementation.

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+15]. This reduces the cost of training compared to a random search. Wahib and Maruyama use machine learning to automate the selection of GPU kernel transformations [WM15].

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 autotuning techniques for enabling optimised multigrid programs [Cha+09], with the wider ambition that these autotuning techniques may be applied to all algorithmic choice programs [Ans14]. While this helps produce efficient programs, it places a great burden on the developer, requiring them to provide enough contrasting implementations to make a search of the optimisation space fruitful.

In [FT10; Fur+14; MF13], Fursin et al. advocate a "big data" driven approach to autotuning, arguing that the challenges facing widespread adoption of autotuning and machine learning methodologies can be attributed to: a lack of common, diverse benchmarks and datasets; 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, the Collective Mind knowledge system, which provides a modular infrastructure for sharing autotuning performance data and related artifacts across the internet.

Iterative compilation typically involves searching the optimisation space offline — dynamic optimisers perform this optimisation space exploration at runtime, allowing optimisations tailored to dynamic feature values. This is a challenging task, as a random search of an optimisation space may result in many configurations with suboptimal performance. In a real world system, evaluating many suboptimal configurations will cause significant slowdowns to a program. A resulting requirement of dynamic optimisers is that convergence time towards optimal parameters is minimised.

Existing dynamic optimisation research has typically taken a low level approach to performing optimisations. Dynamo is a dynamic optimiser which performs binary level transformations of programs using information gathered from runtime profiling and tracing [BDB00]. While this provides the ability to respond to dynamic features, it restricts the range of optimisations that can be applied to binary transformations. These low level transformations cannot match the performance gains that higher level parameter tuning produces.

In [Mas87], the smallest possible program which performs a specific function is found through an iterative enumeration of the entire instruction set. Starting with a program of a single instruction, the superoptimiser 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 exponential growth in the size of the search space is far too expensive for all but the smallest of hot paths, typically less than 13 instructions. 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 superoptimiser 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.

Performant GPGPU programs require careful attention from the developer to properly manage data layout in DRAM, caching, diverging control flow, and thread communication. The performance of programs depends heavily on fully utilising zero-overhead thread scheduling, memory bandwidth, and thread grouping. Ryoo et al. illustrate the importance of these factors by demonstrating speedups of up to 432× for matrix multiplication in CUDA by appropriate use of tiling and loop unrolling [Ryo+08a]. The importance of proper exploitation of local shared memory and synchronisation costs is explored in [Lee+10].

In [CW14], data locality optimisations are automated using a description of the hardware and a memory-placement-agnostic compiler. The authors demonstrate speedups of up to 2.08×, although at the cost of requiring accurate memory hierarchy descriptor files for all targeted hardware. The descriptor files must be hand generated, requiring expert knowledge of the underlying hardware in order to properly exploit memory locality.

Data locality for nested parallel patterns is explored in [Lee+14]. The authors use an automatic mapping strategy for nested parallel skeletons on GPUs, which uses a custom intermediate representation and a CUDA code generator, achieving 1.24× speedup over hand optimised code on 7 of 8 Rodinia benchmarks.

Reduction of the GPGPU optimisation space is demonstrated in [Ryo+08b], using the common subset of optimal configurations across a set of training examples. This technique reduces the search space by 98%, although it does not guarantee that for a new program, the reduced search space will include the optimal configuration.

Magni, Dubach, and O'Boyle demonstrated that thread coarsening of OpenCL kernels can lead to speedups in program performance between 1.11× and 1.33× in [MDO14]. The authors achieve this using a machine learning model to predict optimal thread coarsening factors based on the static features of kernels, and an LLVM function pass to perform the required code transformations.

A framework for the automatic generation of OpenCL kernels from high-level programming concepts is described in [SFD15]. A set of rewrite rules is used to transform high-level expressions to OpenCL code, creating a space of possible implementations. This approach is ideologically similar to that of PetaBricks, in that optimisations are made through algorithmic choice, although in this case the transformations are performed automatically at the compiler level. The authors report performance on a par with that of hand written OpenCL kernels.

Stencil codes have a variety of computationally expensive uses from fluid dynamics to quantum mechanics. Efficient, tuned stencil kernels are highly sought after, with early work in 2003 by Bolz et al. demonstrating the capability of GPUs for massively parallel stencil operations [Bol+03]. In the resulting years, stencil codes have received much attention from the performance tuning research community.

Ganapathi et al. demonstrated early attempts at autotuning multicore stencil codes in [Gan+09], drawing upon the successes of statistical machine learning techniques in the compiler community. They present an autotuner 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, using 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. They present a compelling argument for the use of energy efficiency as an optimisation target in addition to runtime, citing that it was the power wall that lead to the multicore revolution in the first place. Their choice of only 2 benchmarks and 2 platforms makes the evaluation of their autotuner somewhat limited.

Berkeley et al. targeted 3D stencils code performance in [Ber+08]. Stencils are decomposed into core blocks, sufficiently small to avoid last level cache capacity misses.

These are then further decomposed to thread blocks, designed to exploit common locality threads may have within a shared cache or local memory. Thread blocks are divided into register blocks to take advantage of data level parallelism provided by the available registers. Data allocation is optimised on NUMA systems. The performance evaluation considers speedups of various optimisations with and without consideration for host/device transfer overhead.

Kamil et al. present an autotuning framework in [Kam+10] which accepts as input a Fortran 95 stencil expression and generates tuned shared-memory parallel implementations in Fortan, C, or CUDA. The system uses an IR to explore autotuning transformations, enumerating a subset of the optimisation space and recording only a single execution time for each configuration, reporting the fastest. They demonstrate their system on 4 architectures using 3 benchmarks, with speedups of up to 22× compared to serial implementations. The CUDA code generator does not optimise for the GPU memory hierarchy, using only global memory. As demonstrated in this thesis, improper utilisation of local memory can hinder program performance by two orders of magnitude. There is no directed search or cross-program learning.

In [ZM12], Zhang and Mueller present a code generator and autotuner for 3D Jacobi stencil codes. Using a DSL to express kernel functions, the code generator performs substitution from one of two CUDA templates to create programs for execution on GPUs. GPU programs are parameterised and tuned for block size, block dimensions, and whether input data is stored in read only texture memory. This creates an optimisation space of up to 200 configurations. In an evaluation of 4 benchmarks, the authors report performance that is comparable with previous implementations of iterative Jacobi stencils on GPUs [HPS12; PF10]. The dominating parameter is shown to be block dimensions, followed by block size, then read only memory. The DSL presented in the paper is limited to expressing only Jacobi Stencils applications. Their autotuner requires a full enumeration of the parameter space for each program, which may be impractical for the needs of general purpose stencil computing. Previous work (Appendix **??**) overcomes this drawback by learning parameter values which transfer to unseen stencils, without the need for an expensive tuning phase for each program and architecture.

In [CSB11], Christen, Schenk, and Burkhart presents a DSL for expressing stencil codes, a C code generator, and an autotuner for exploring the optimisation space of blocking and vectorisation strategies. The DSL supports stencil operations on arbitrarily high-dimensional grids. The autotuner performs either an exhaustive, multi-run

Powell search, Nelder Mead, or evolutionary search to find optimal parameter values. They evaluate their system on two CPUS and one GPU using 6 benchmarks. The authors do not present a ratio of the available performance that their system achieves, or how the performance of optimisations vary across benchmarks or devices.

A stencil grid can be decomposed into smaller subsections so that multiple GPUs can operate on each subsection independently. This requires a small overlapping region where each subsection meets — the halo region — to be shared between devices. For iterative stencils, values in the halo region must be synchronised between devices after each iteration, leading to costly communication overheads. One possible optimisation is to deliberately increase the size of the halo region, allowing each device to compute updated values for the halo region, instead of requiring a synchronisation of shared state. This reduces the communication costs between GPUs, at the expense of introducing redundant computation. Tuning the size of this halo region is the goal of PARTANS [LFC13], an autotuning framework for multi-GPU stencil computations. Lutz, Fensch, and Cole explore the effect of varying the size of the halo regions using six benchmark applications, finding that the optimal halo size depends on the size of the grid, the number of partitions, and the connection mechanism (i.e. PCI express). The authors present an autotuner which determines problem decomposition and swapping strategy offline, and performs an online search for the optimal halo size. The selection of overlapping halo region size compliments the selection of workgroup size which is the subject of previous work (Appendix **??**).

In applications of machine learning for iterative compilation, a limiting factor of the effectiveness of learned models is the number of benchamrks used. The development of automatic program generation alleviates this problem by allowing an unbounded number of programs to enumerate the feature space at an increasingly granular scale.

Recently, deep neural networks [LBH15] have been shown to be a powerful tool for feature engineering in various tasks including image recognition [He+16; KSH12] and audio processing [Lee+09]. In the field of compiler optimization, no work so far has applied deep neural networks for program feature generation and selection. Our work is the first to do so.

Generative models for video [SMS15].

Limits of language modeling [Joz+16].

Sequence to sequence learning, NIPS paper [SVL14].

Deep Learning is an emerging branch of machine learning, offering breakthrough domains in a number of domains [WRX17].

Automatic inference from raw inputs in other fields. Review of representation learning [BCV13]. Object detection, image classification, text classification [Con+16], and even to the optimizations themselves [And+16]. We are the first to apply this approach to compilers.

End to end training of robotic control policies [Lev+16].

Exploring relevance of inferred features in model hidden layers [Yos+14]. Transfer learning is common in other domains, e.g. vision [Oqu+14; Raz+14]. Often using the hidden layers as fixed feature extractors (by using the output of one sub-model as input to a new model), rather than using the same model structure and transferring weights (as we did).

3.2.1 Mining Source Code

There is an increasing interest in mining source code repositories at large scale [AS13; Kal+09; Whi+15]. Previous studies have involved data mining of GitHub to analyze software engineering practices [Bai+14; GAL14; VFS15; Wu+14], for example code generation [Gu+16], code summarization [APS16], comment generation [WYT13], and code completion [RVY14]. However, no work so far has exploited mined source code for benchmark generation. This work is the first to do so.

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 [All+14] and idioms [AS14], API example code [Gu+16] and pseudo-code generation [Oda+15], and benchmark generation [Cum+17b]. 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.

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

3.3 Program Generation

GENESIS [CGA15] is a language for generating synthetic training programs. The essence of the approach is to construct a probabilistic grammar with embedded semantic actions that defines a language of possible programs. New programs may be created by sampling the grammar and, through setting probabilities on the grammar productions, the sampling is biased towards producing programs from one part of the space or another. This technique is potentially completely general, since a grammar can theoretically be constructed to match any desired program domain. However, despite being theoretically possible, it is not easy to construct grammars which are both suitably general and also produce programs that are in any way similar to human written programs. It has been shown to be successful over a highly restricted space of stencil benchmarks with little control flow or program variability [FE15; Gar15] (Appendix **??**). 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, learning from a corpus 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 [**Yang2012**] and OpenCL [Lid+15]. 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. A mutation-based approach for differential testing the Java Virtual Machine is demonstrated in [Che+16b].

Goal-directed program generators have been used for a variety of domains, including generating linear transforms [VDP09], MapReduce programs [Smi16], and data structure implementations [LEE16].

3.3.1 Training with Synthetic Benchmarks

The use of synthetic benchmarks for providing empirical performance evaluations dates back to as early as 1974 [CW76]. The *automatic generation* of such synthetic benchmarks is a more recent innovation, serving the purpose initially of stress-testing increasingly complex software systems for behaviour validation and automatic bug detection [GLM08; VCS00]. A range of techniques have been developed for these purposes, ranging from applying random mutations to a known dataset to generate test stimuli, to so-called "whitebox fuzz testing" which analyses program traces to explore the space of a program's control flow. Csmith is one such tool which generates randomised C source programs for the purpose of automatically detecting compiler bugs [**Yang2012**].

A method for the automatic generation of synthetic benchmarks for the purpose of *performance* tuning is presented in [CGA15]. Chiu, Garvey, and Abdelrahman use template substitution over a user-defined range of values to generate training programs with a statistically controlled range of features. A Perl preprocessor generates output source codes from an input description using a custom input language Genesis. Genesis is more flexible than the system presented in this thesis, supporting substitution of arbitrary sources. The authors describe an application of their tool for generating OpenCL stencil kernels, but do not report any performance results.

GENESIS [CGA15] 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 [Cum+16a; Gar15]. 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 [**Yang2012**] and OpenCL [Lid+15]. A mutation-based approach for the Java Virtual Machine is demonstrated in [Che+16b]. Goal-directed program generators have been used for a variety of domains, including generating linear transforms [VDP09], MapReduce programs [Smi16], and data structure implementations [LEE16]. Program synthesis from input/output examples is used for simple algorithms in [Zar+16], string manipulation in [Gul11], and geometry constructions in [GKT11].

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 [All+14]. JSNice leverages probabilistic graphical models to predict program properties such as identifier names for Javascript [RVK15].

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 [CGA15] is a language for generating synthetic training programs. Users annotate template programs with statistical distributions over features, which are in-

stantiated 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 [Cum+16a; Gar15]. 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 [**Yang2012**] and OpenCL [Lid+15]. A mutation-based approach for the Java Virtual Machine is demonstrated in [Che+16b]. Goal-directed program generators have been used for a variety of domains, including generating linear transforms [VDP09], MapReduce programs [Smi16], and data structure implementations [LEE16]. Program synthesis from input/output examples is used for simple algorithms in [Zar+16], string manipulation in [Gul11], and geometry constructions in [GKT11].

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 [All+14]. JSNice leverages probabilistic graphical models to predict program properties such as identifier names for Javascript [RVK15].

There is an increasing interest in mining source code repositories at large scale [AS13; Kal+09; Whi+15]. Previous studies have involved data mining of GitHub to analyze software engineering practices [Bai+14; GAL14; VFS15; Wu+14], for example code generation [Gu+16], code summarization [APS16], comment generation [WYT13], and code completion [RVY14]. However, no work so far has exploited mined source code for benchmark generation. This work is the first to do so.

The random generation of test cases is a well established approach to the compiler validation problem. Prior approaches are surveyed in [BS97; KP05] and empirically contrasted in [Che+16a]. 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.

3.3.1.0.1 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 [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 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 [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 unlike our approach, no distribution is learned over the grammar; program enumeration is uniformly random.

3.3.1.0.2 Program Mutation Equivalence Modulo Inputs (EMI) testing [LAS14; SLS16] 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 [HHZ12] 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 [ZSS17] again works by transforming existing code. It identifies algorithmic patterns in short pieces of code and enumerates all the possible permutations of variable usage. 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.

3.3.1.0.3 Machine Learning There is an increasing interest in applying machine learning to software testing. Most similar to our work is Learn&fuzz [GPS17], 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 [Wan+17] learns a probabilistic context-sensitive grammar over a corpus of programs to generate input seeds for mutation testing. The generated seeds are shown to improve the

Table 3.1: The number of DeepSmith programs that trigger Solidity compiler crashes from 12 hours of testing.

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

code coverage of AFL [Zal] 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 [HOY17; Koc+17], repairing programs [Kou+17; Whi+], prioritizing test programs [Che+17], identifying buffer overruns [Cho+17], and processing bug reports [HLZ16; Lam+15]. 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.

Machine learning has emerged as a viable means in automatically constructing heuristics for code optimization [Cum+16b; KC12; Mur+16; Ogi+17; RGW17; WO10]. 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 [Jia+10] and runtime information such as system load [WWO14] and performance counters [Dub+09]. 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 [SA05; TMW17], or projected into a lower dimensional space [Col+13; Dub+07]. FEAST employs a range of existing feature selection methods to select useful candidate features [Tin+16]. 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 [PCA12]. 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 [LBO14; Nam+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 [LBO14] 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 [Nam+10] 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 codesign [Cav+06]. 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 [All+14] and idioms [AS14], API example code [Gu+16] and pseudo-code generation [Oda+15], and benchmark generation [Cum+17b]. 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 [He+16; KSH12] and audio processing [Lee+09]. No work so far has applied deep neural networks for program feature generation. Our work is the first to do so.

Chapter 4

CLgen: Generating Compiler Benchmarks through Deep Learning

4.1 Introduction

Predictive modeling is a well researched method for building optimization heuristics that often exceed human experts and reduces development time [CFL12; Cum+16b; FE15; LBO14; MDO14; MSD16; Ogi+14; Wan+14; WO09; WO10; WWO14]. Figure 6.1 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.

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

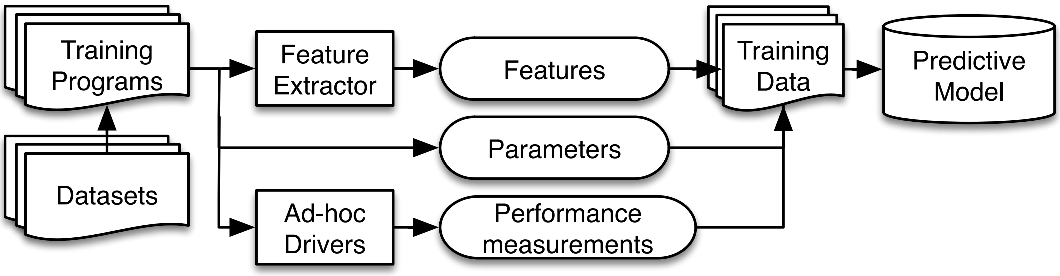


Figure 4.1: Training a predictive model.

defines a language of possible programs. New programs may be created by sampling the grammar and, through setting probabilities on the grammar productions, the sampling is biased towards producing programs from one part of the space or another. This technique is potentially completely general, since a grammar can theoretically be constructed to match any desired program domain. However, despite being theoretically possible, it is not easy to construct grammars which are both suitably general and also produce programs that are in any way similar to human written programs. It has been shown to be successful over a highly restricted space of stencil benchmarks with little control flow or program variability [Cum+16b; FE15]. But, it is not clear how much effort it will take, or even if it is possible for human experts to define grammars capable of producing human like programs in more complex domains.

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 [Gra13; SVL14]. 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

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

Table 4.1: Performance relative to the optimal of the *Grewe et al.* predictive model across different benchmark suites on an AMD GPU. The columns show the suite used for training; the rows show the suite used for testing.

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×, and expose limitations in the feature design of the model which, after correcting, further increases performance by 4.30×.

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

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

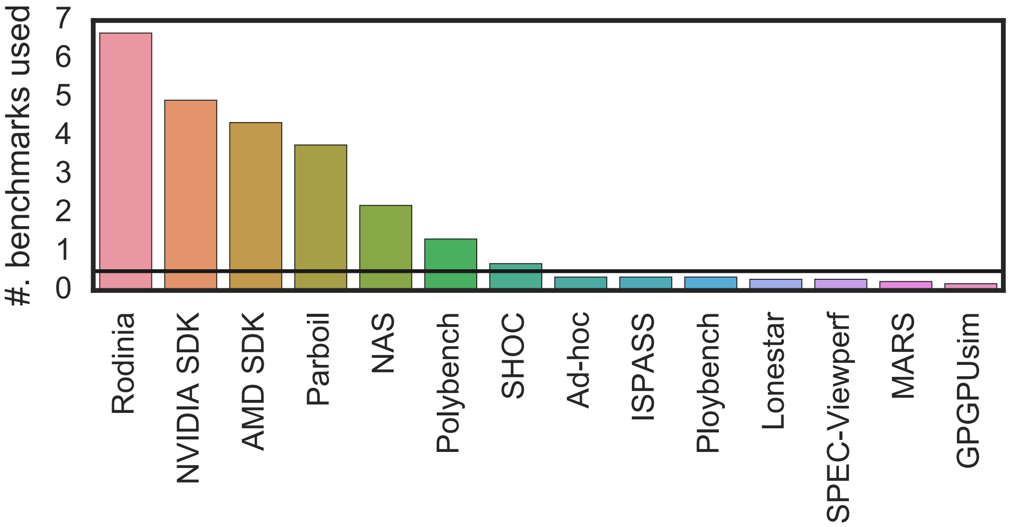
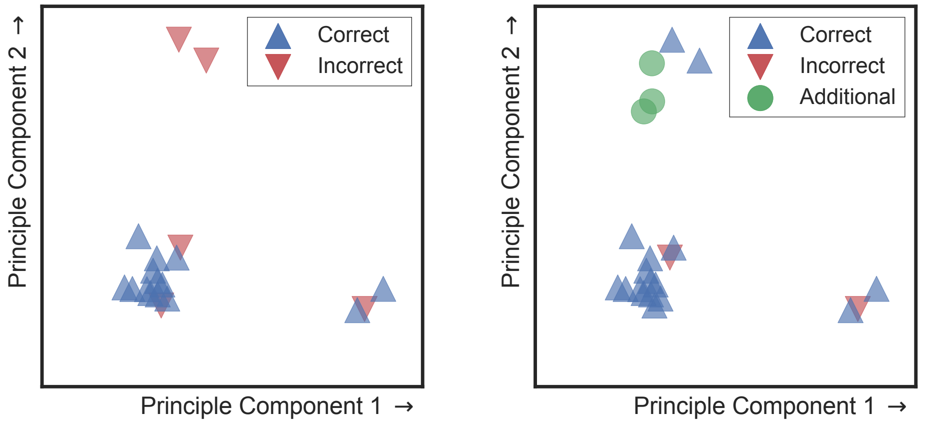


Figure 4.2: The average number of benchmarks used in GPGPU research papers, organized by origin. In this work we use the seven most popular benchmark suites.

paper to be 17, and that a small pool of benchmarks suites account for the majority of results, shown in Figure 4.2. 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.* [GWO13] 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 6.3.

Table 4.1 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 4.3 shows the feature space of the Parboil benchmark suite, showing whether, for each benchmark, the model was able to correctly predict the appropriate optimization. We used Principle Component Analysis to reduce the multi-dimensional feature space to aid visualization.



(a)

(b)

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

As we see in Figure 4.3a, 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 Figure 4.3b, 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 4.1.

These results highlight the significant effect that the number and distribution of training programs has on the quality of predictive models. Without good coverage of the feature space, any machine learning methodology is unlikely to produce high quality heuristics, suitable for general use on arbitrary real applications, or even applications from different benchmark suites. 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.

4.2 CLgen: Benchmark Synthesis

This section introduces 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. We use a host driver to execute the synthesized programs to gather performance data for use in predictive modeling. Figure 4.4 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.

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.

4.2.1 An OpenCL Language Corpus

Deep learning requires large datasets [LBH15]. For the purpose of modeling a programming language, this means assembling a very large collection of real, handwritten 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

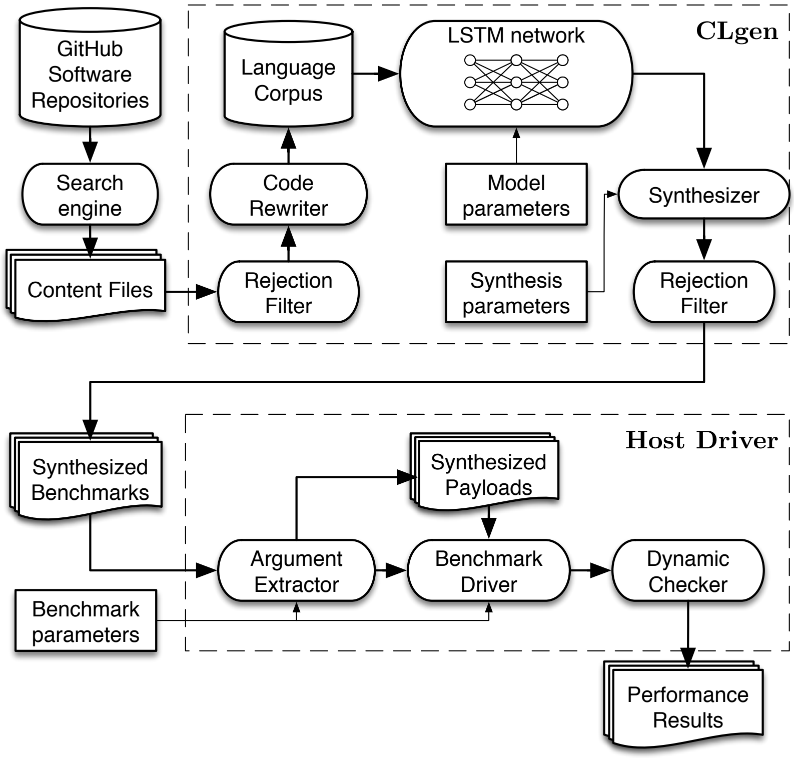


Figure 4.4: Benchmark synthesis and execution pipeline.

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.

4.2.1.0.1 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 4.1.

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

4.2.1.0.2 Code Rewriter Programming languages have few of the issues of semantic interpretation present in natural language, though there remains many sources of variance at the syntactic level. For example, the presence and content of comments in code, and the choice of identifying names given to variables. 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 [AS13], 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 4.5. 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%.

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

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

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

Listing 4.1: The *shim* header file, providing inferred type aliases and constants for OpenCL on GitHub.

4.2.2 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 [Mik+15; SSN12] to learn a character-level language model over the corpus of OpenCL compute kernels. The LSTM network architecture comprises recurrent layers of *memory cells*, each consisting of an input, output, and forget gate, and an output layer providing normalized probability values from a 1-of-K coded vocabulary [GS05].

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.

```
#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 4.5: The code rewriting process, which transforms code to make it more amenable to language modeling.

Algorithm 1 Sampling a candidate kernel from a seed text.

```
Require: LSTM model $M$, maximum kernel length $n$.
Ensure: Completed sample string $S$.
$S \leftarrow$ "__kernel void A(const int a) {"      ▷ Seed text
$d \leftarrow 1$                                      ▷ Initial code block depth
for $i \leftarrow |S|$ to $n$ do
    $c \leftarrow predictcharacter(M, S)$             ▷ Generate new character
    if $c =$ "{" then
        $d \leftarrow d + 1$                          ▷ Entered code block, increase depth
    else if $c =$ "}" then
        $d \leftarrow d - 1$                          ▷ Exited code block, decrease depth
    end if
    $S \leftarrow S + c$                              ▷ Append new character
    if $depth = 0$ then
        break                                         ▷ Exited function block, stop sampling
    end if
end for
```

4.2.3 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. The same rejection filter described in Section 4.2.1.0.1 then either accepts or rejects the sample as a candidate synthetic benchmark. Listing 4.6 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 5.4.

```
__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;
```

4.3 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.

4.3.1 Generating Payloads

A *payload* encapsulates all of the arguments of an OpenCL compute kernel. After parsing the input kernel to derive argument types, a rule-based approach is used to generate synthetic payloads. For a given global size S_g: host buffers of S_g elements are allocated and populated with random values for global pointer arguments, device-only buffers of S_g elements are allocated for local pointer arguments, integral arguments are given the value S_g, and all other scalar arguments are given random values. Host to device data transfers are enqueued for all non-write-only global buffers, and all non-read-only global buffers are transferred back to the host after kernel execution.

4.3.2 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:

1. Create 4 equal size payloads $A1_{in}$, $B1_{in}$, $A2_{in}$, $B2_{in}$, subject to restrictions: $A1_{in} = A2_{in}$, $B1_{in} = B2_{in}$, $A1_{in} \neq B1_{in}$.

2. Execute kernel k 4 times: $k(A1_{in}) \rightarrow A1_{out}$, $k(B1_{in}) \rightarrow B1_{out}$, $k(A2_{in}) \rightarrow A2_{out}$, $k(B2_{in}) \rightarrow B2_{out}$.

3. Assert:

   - $A1_{out} \neq A1_{in}$ and $B1_{out} \neq B1_{in}$, else k has no output (for these inputs).
   - $A1_{out} \neq B1_{out}$ and $A2_{out} \neq B2_{out}$, else k is input insensitive t (for these inputs).
   - $A1_{out} = A2_{out}$ and $B1_{out} = B2_{out}$, 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 [McK98], 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 [BCD12; PM15; SD16], however we deemed such methods unnecessary for our purpose of performance modeling.

4.4 Qualitative Evaluation of Generated Programs

In this section we evaluate the quality of programs synthesized by CLgen by their likeness to hand-written code.

Judging whether a piece of code has been written by a human is a challenging task for a machine, so we adopt a methodology from machine learning research based on the *Turing Test* [Gao+15; Vin+15; ZIE16]. 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 [Lid+15], a program generator for differential testing[2].

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

[2] 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)`×`wgsize`	ratio local to global mem accesses × #. work-items
`F4:`	`comp/mem`	computation-mem ratio

(b) Combinations of raw features

Table 4.2: *Grewe et al.* model features.

better than random chance.

4.5 Experimental Methodology

4.5.1 Experimental Setup

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

4.5.1.0.2 Benchmarks As in [GWO13], we test our model on the NAS Parallel Benchmarks (NPB) [Bai+91]. We use the hand-optimized OpenCL implementation of Seo, Jo, and Lee [SJL11]. In [GWO13] the authors augment the training set of the predictive model with 47 additional kernels taken from 4 GPGPU benchmark suites. To more fully sample the program space, we use a much larger collection of 142 programs, summarized in Table 6.2. These additional programs are taken from all 7 of the most frequently used benchmark suites identified in Section **??**. None of these programs were used to train CLgen. We synthesized 1,000 kernels with CLgen to use as

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

Table 4.3: List of benchmarks.

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

Table 4.4: Experimental platforms.

additional benchmarks.

4.5.1.0.3 Platforms We evaluate our approach on two 64-bit CPU-GPU systems, detailed in Table 5.1. One system has an AMD GPU and uses OpenSUSE 12.3; the other is equipped with an NVIDIA GPU and uses Ubuntu 16.04. Both platforms were unloaded.

4.5.1.0.4 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.

4.5.2 Methodology

We replicated the methodology of [GWO13]. Each experiment is repeated five times and the average execution time is recorded. The execution time includes both device compute time and the data transfer overheads.

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 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.

4.6 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 [GWO13] 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 [Lid+15].

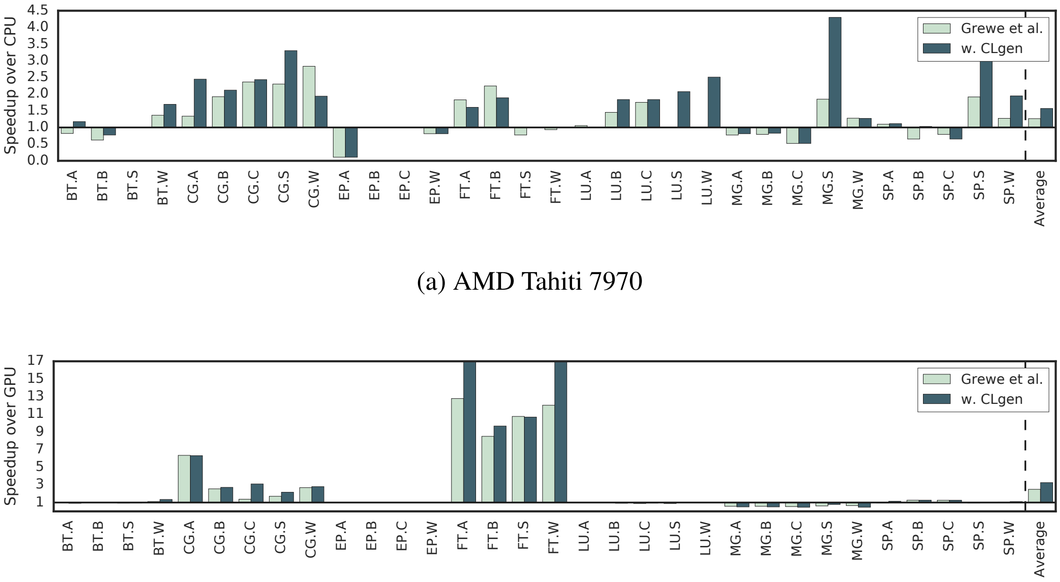
4.6.1 Performance Evaluation

Figure 4.7 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× 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.

4.6.2 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.*



(a) AMD Tahiti 7970

(b) NVIDIA GTX 970

Figure 4.7: 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.

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

Listing 4.2: 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.

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 4.2 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 4.8 shows speedups of our extended model across all seven of the benchmark suites used in Section **??**. Model performance, even on this tenfold increase of benchmarks, is good. There are three benchmarks on which the model performs poorly: `MatrixMul`, `cutcp`, and `pathfinder`. Each of those programs make heavy use of loops, which 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× on AMD and 5.04× on NVIDIA across a very large test set.

4.6.3 Comparison of Source Features

As demonstrated in Section **??**, the predictive quality of a model for a given point in the feature space is improved with the addition of observations from 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 4.2a, plus the branching feature discussed in the previous subsection, to measure the number of CLgen ker-

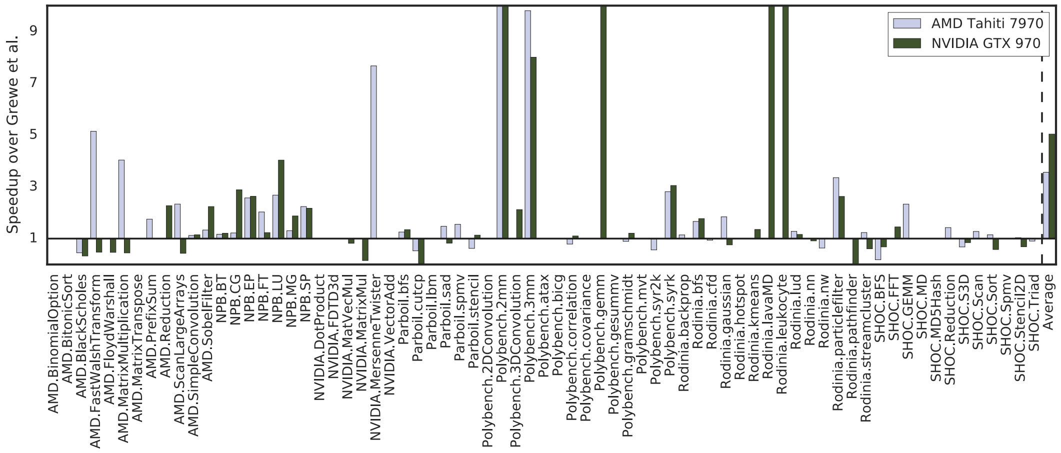


Figure 4.8: Speedups of predictions using our extended model over *Grewe et al.* on both experimental platforms. Synthetic benchmarks and the additional program features outperform the original predictive model by a factor $3.56\times$ on AMD and $5.04\times$ on NVIDIA.

nels 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.

Figure 6.4 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*.

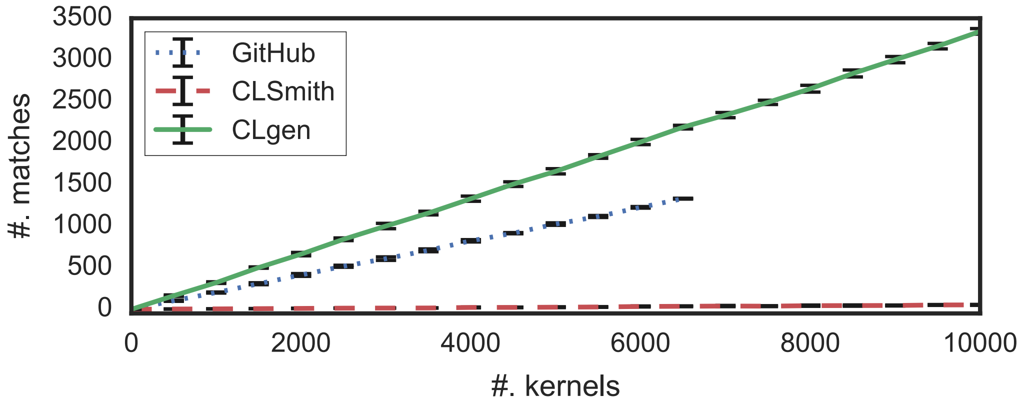


Figure 4.9: The number of kernels from GitHub, CLSmith, and CLgen with static code features matching the benchmarks. CLgen generates kernels that are closer in the feature space than CLSmith, and can continue to do so long after we have exhausted the extent of the GitHub dataset. Error bars show standard deviation of 10 random samplings.

4.7 Summary

The quality of predictive models is bound by the quantity and quality of programs used for training, yet there is typically only a few dozen common benchmarks available for experiments. 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×. We found that synthetic benchmarks exposed weaknesses in the feature set which, when corrected, further improved the performance by 4.30×. 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.

Chapter 5

Synthesizing Test Cases for Compiler Validation

5.1 Introduction

Compilers should produce correct code for valid inputs, and meaningful errors for invalid inputs. Failure to do so can hinder software development or even cause catastrophic runtime errors. Still, properly testing compilers is hard. Modern 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 [Che+13; Che+16a; KP05]. 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 [McK98] can be used. The generated code and a set of inputs form a *test case* which is compiled and executed on multiple *testbeds*. If the test case should have deterministic 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 [Yan+11], generates large random programs by defining and sampling a probabilistic grammar which covers a subset of the C programming language. Through this grammar, CSmith ensures that the generated code easily passes the compiler front-end and stresses the most complex part of the compiler, the middle-end. Complex static and dynamic analyses make sure that programs are free from undefined 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 [Lid+15] — a superficially simple task — took 9 months and an additional 8k lines of code. Given the difficulty of defining a new grammar, typically only a subset of the language is implemented.

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.

5.2 DeepSmith: Compiler Fuzzing through Deep Learning

DeepSmith[1] is our open source framework for compiler fuzzing. Figure 6.2 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.

5.2.1 Generative Model

Generating test cases for compilers is hard because their inputs are highly structured. Producing text with the right structure requires expert knowledge and a significant engineering effort, which has to be repeated from scratch for each new language. Instead, 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 [Bow+15; Rag+16; RJS17]. Contrary to existing tools, our approach does not require expert knowledge of the target language and is only a few hundred lines of code.

5.2.1.0.1 Handwritten Programs The generative model needs to be trained on a *seed corpus* of example programs. We automated the assembly of this corpus by min-

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

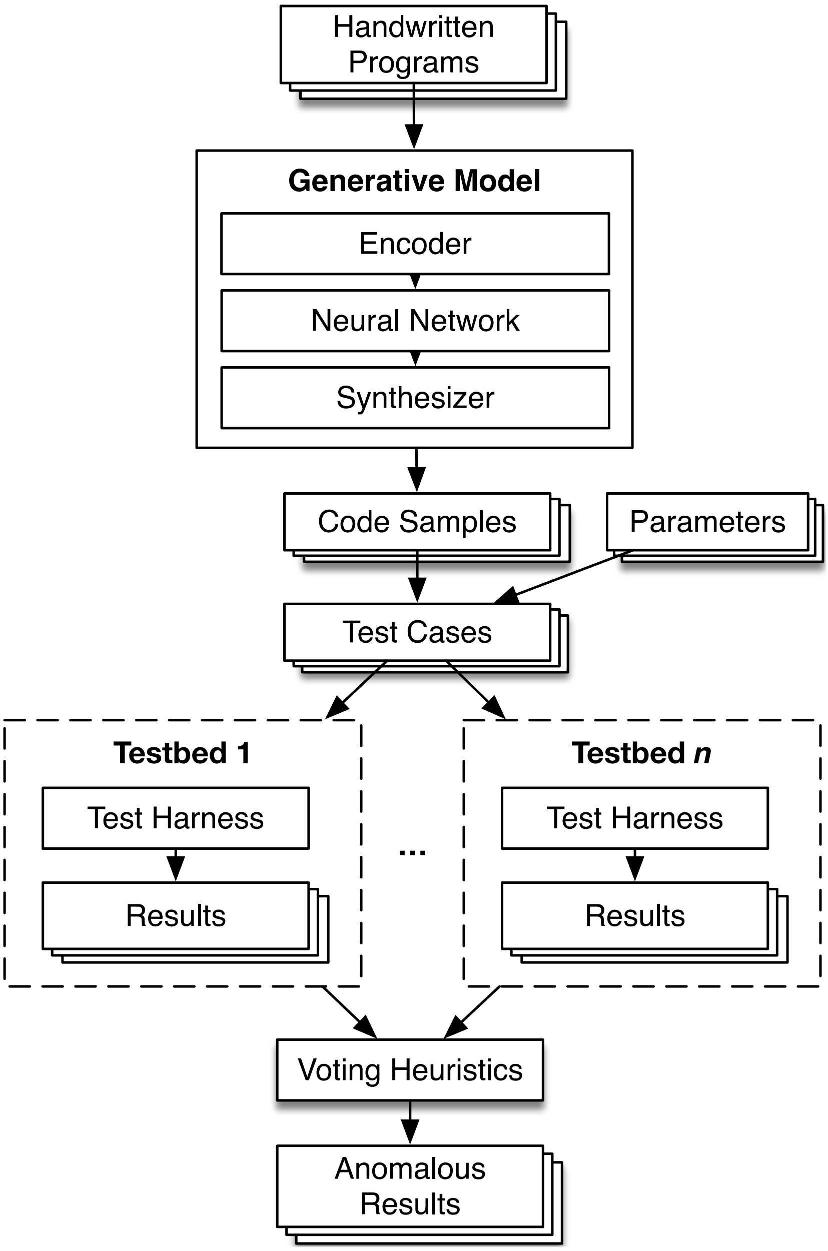


Figure 5.1: DeepSmith system overview.

ing 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 5.4.4).

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

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

5.2.1.0.3 Neural Network We use the Long Short-Term Memory (LSTM) architecture of Recurrent Neural Network to model program code [Chi+11]. 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 [PMB13]. This makes them effective at learning complex relationships over long sequences [LBE15] 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.

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

5.2.2 Test Harness

OpenCL is an embedded compute kernel language, requiring host code to compile, execute, and transfer data between the host and device. For the purpose of compiler fuzzing, this requires a *test harness* to run the generated OpenCL programs. At first, 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. 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*

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

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

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 5.2. A *build failure* occurs when online compilation of the OpenCL kernel fails, usually accompanied by an error diagnostic. A *build crash* or *build timeout* outcome occurs if the compiler crashes or fails to produce a binary within 60 seconds, respectively. For compile-only test cases, a *pass* is achieved if the compiler produces a binary. For test cases in which the kernel is executed, kernel execution leads to one of three potential outcomes: *runtime crash* if the program

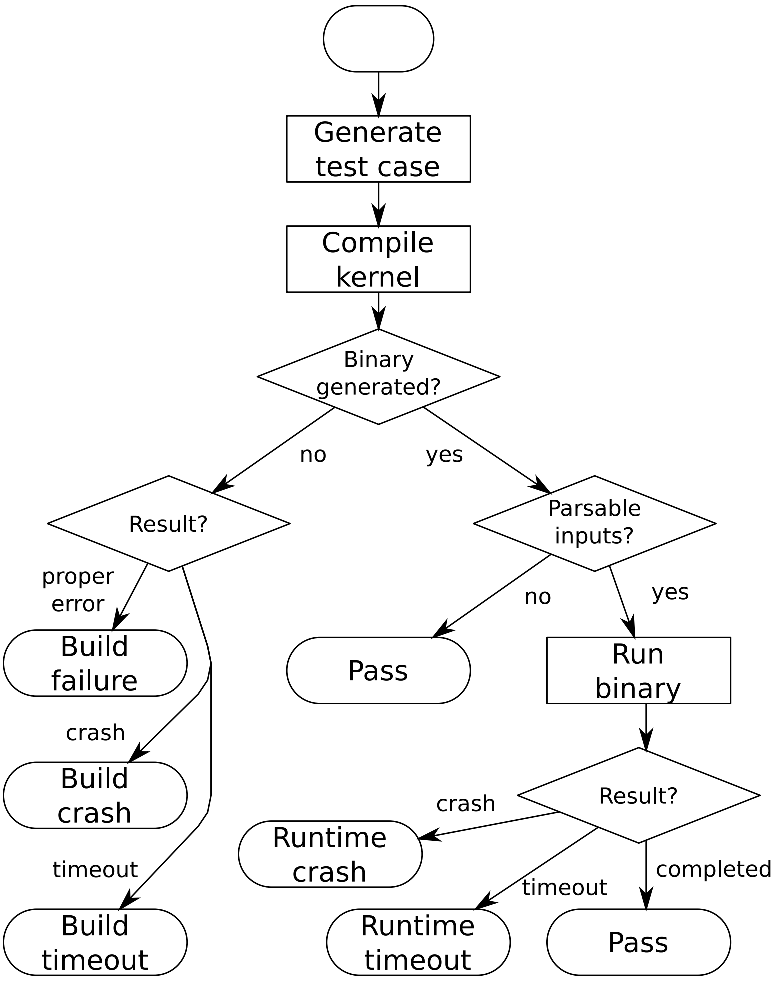


Figure 5.2: Test case execution, and possible results.

crashes, *timeout* if the kernel fails to terminate within 60 seconds, or *pass* if the kernel terminates gracefully and computes an output.

5.2.3 Voting Heuristics for Differential Testing

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 [McK98]. 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 5.4.1). 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.

5.2.3.0.1 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 [BCD12] and Oclgrind [PM15]. 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.

5.3 Experimental Setup

In this section we describe the experimental parameters used.

5.3.1 OpenCL Systems

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

5.3.2 Testbeds

For each OpenCL system, we create two testbeds. In the first, the compiler is run with 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.

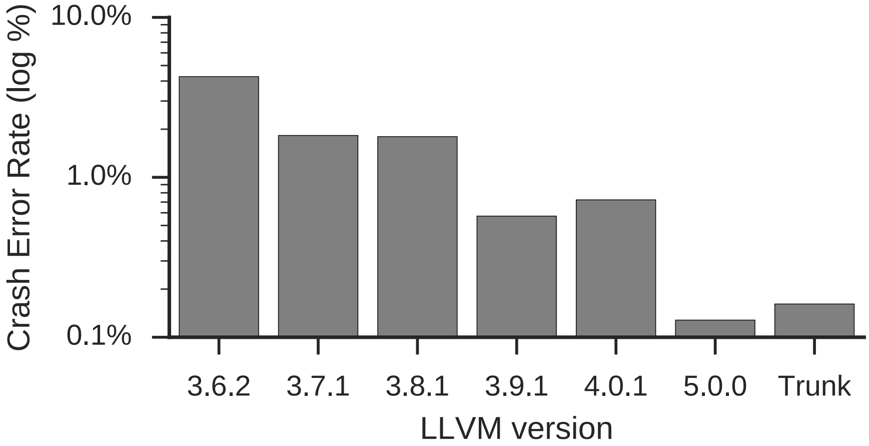


Figure 5.3: Crash rate of the Clang front-end of every LLVM release in the past 24 months compiling 75k DeepSmith kernels.

5.3.3 Test Cases

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

5.3.4 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.

5.4 Evaluation

We report on the results of DeepSmith testing of the 10 OpenCL systems from Table 5.1, 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 [Lid+15]. DeepSmith is able to identify a broad range of defects, many of which CLSmith cannot, for only a fraction of the engineering effort. Finally, we provide a quantitative analysis of compiler ro-

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

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

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

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

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

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

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

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

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

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

Figure 5.4: Example kernels which crash compilers.

Table 5.2: The number of DeepSmith programs which trigger distinct Clang front-end assertions, and the number of programs which trigger unreachables.

	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

bustness 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 5.1, suffixed with $+$, $-$, or $\pm$ to denote optimizations on, off, or either, respectively.

5.4.1 Compile-time Defects

OpenCL is typically compiled online, which amplifies the significance of detecting compile-time defects, as they may not be discovered until code has been shipped to customers. 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.

5.4.1.0.1 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*.

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

(a) Reduced from 48 line kernel.

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

(b) Reduced from 52 line kernel.

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

(c) Reduced from 68 line kernel.

Figure 5.5: Example codes which crash parsers.

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 5.4a raises an assertion during semantic analysis of the AST when used as an array index.

Type errors were an occasional cause of compile-time defect. Figure 5.4b induces a crash in NVIDIA compilers due to an implicit conversion between global to constant address qualifiers. Worse, we found that Testbeds 3± would loop indefinitely on some kernels containing implicit conversions from a pointer to an integer, as shown in Figure 5.6a. 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 5.4c and 5.4d 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 5.4e silently crashes 6 of the 10 compilers, which, to the best of our knowledge, makes DeepSmith the first random program generator to induce a defect through exploiting compiler-specific functionality.

5.4.1.0.2 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 5.5 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.

5.4.1.0.3 Compiler Hangs As expected, some compile-time defects are optimization sensitive. Testbed 1+ hangs on large loop bounds, shown in Figure 5.6b. All commercial Intel compilers we tested hang during optimization of non-terminating loops (Figure 5.6c).

Testbeds 7± loop indefinitely during compilation of the simple kernel in Figure 5.6d.

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

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

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

5.4.2 Runtime Defects

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

5.4.2.0.1 Thread-dependent Flow Control A common pattern in OpenCL is to obtain the thread identity, often as an `int`, and to compare this against some fixed value to determine whether or not to complete a unit of work (46% of OpenCL kernels on GitHub use this (*tid* → int, `if (tid < ...)` {...}) pattern). DeepSmith, having

```
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 5.6: Example kernels which hang compilers.

```
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 5.7: Example kernels which are miscompiled.

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 5.7a 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 5.7b 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.

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

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

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

5.4.3 Comparison to State-of-the-art

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

5.4.3.0.1 Results Overview Table 5.3 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

Table 5.3: Results from 48 hours of testing using CLSmith and DeepSmith. System #. as per Table 5.1. ± 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							DeepSmith				
#.	**Device**	±	**bc**	**bto**	**abf**	**arc**	**awo**	✓	**total**	**bc**	**bto**	**abf**	**arc**	**awo**
1	GeForce GTX 1080	−	0	0	0	2	2	15628	15632	27	0	3	0	5
		+	0	71	0	6	9	14007	14093	20	1	1	0	7
2	GeForce GTX 780	−	0	0	0	28	5	18220	18253	27	0	3	0	9
		+	26	14	0	0	3	17654	17697	32	1	1	0	9
3	Intel HD Haswell GT2	−	2714	2480	0	0	3	1121	6318	574	200	2	0	12
		+	2646	2475	0	0	3	1075	6199	569	200	5	0	10
4	Intel E5-2620 v4	−	0	27	1183	0	0	16313	17523	57	0	9	1	0
		+	487	87	1130	0	0	17350	19054	320	147	7	3	0
5	Intel E5-2650 v2	−	0	11	0	0	0	17887	17898	152	2	0	0	0
		+	112	175	0	0	0	14626	14913	170	117	0	0	1
6	Intel i5-4570	−	0	14	1226	0	0	17118	18358	73	0	9	2	1
		+	526	63	1180	0	0	19185	20954	318	140	7	2	1
7	Intel Xeon Phi	−	4	84	0	0	8	13265	13361	68	4	0	0	1
		+	42	1474	0	0	2	3258	4776	77	47	0	0	0
8	POCL (Intel E5-2620)	−	0	0	0	675	0	17250	17925	54	1	2	89	3
		+	0	3	0	99	5	13980	14087	46	0	1	104	4
9	ComputeAorta (Intel E5-2620)	−	0	0	0	0	0	18479	18479	51	0	1	3	1
		+	0	0	0	300	11	18625	18936	59	0	0	48	4
10	Oclgrind Simulator	−	0	0	0	0	0	5287	5287	2081	0	0	0	1
		+	0	0	0	0	0	5334	5334	2265	0	0	0	0

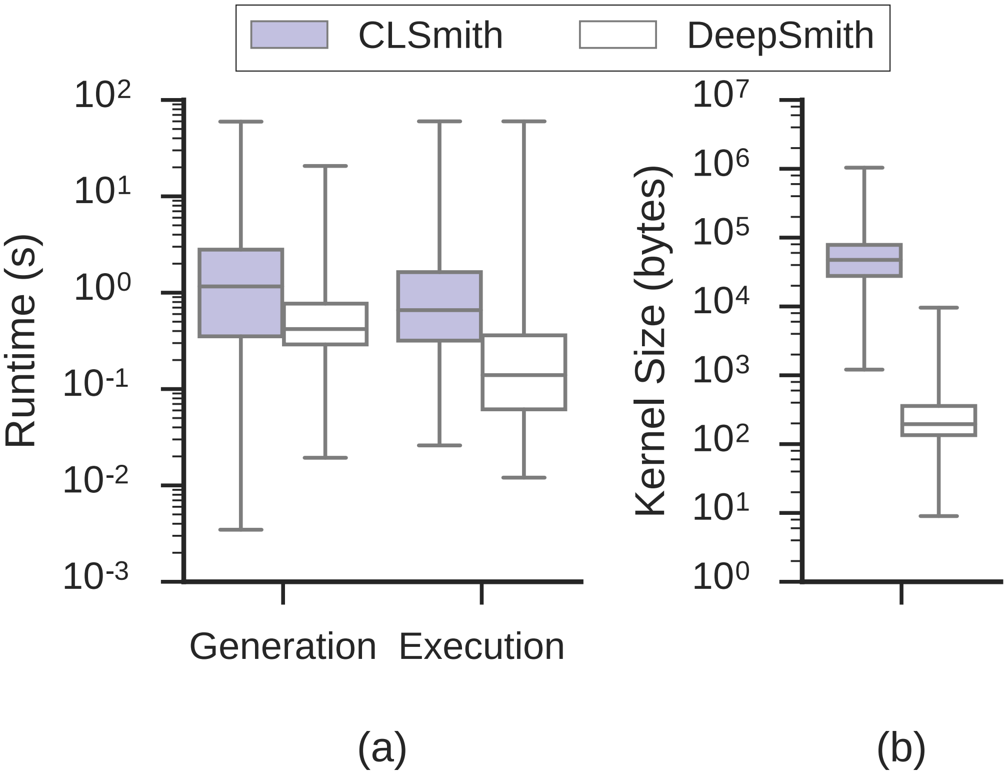


Figure 5.8: Comparison of runtimes (a) and test case sizes (b). DeepSmith test cases are on average evaluated $3.03\times$ faster than CLSmith ($2.45\times$, and $4.46\times$ for generation and execution, respectively), and are two orders of magnitude smaller. Timings do not include the cost of timeouts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

less likely to timeout (execute for 60 seconds without termination), and the test cases which do not timeout execute faster.

Figure 5.8a shows the generation and execution times of DeepSmith and CLSmith test cases, excluding timeouts[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 [Lid+15] the authors omit large-scale testing on this device for this reason. The use of structs in handwritten OpenCL is comparatively rare — only 7.1% of kernels on GitHub use them.

5.4.3.0.2 Comparison of Test Cases The average CLSmith program is 1189 lines long (excluding headers). CLSmith test cases require reduction in order to expose the underlying bug. An automated approach to OpenCL test case reduction is presented in [PDL16], though it requires on average 100 minutes for each test case using a 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.

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

The Intel CPU Testbeds (4±, 5±, 6±, and 7±) would occasionally emit a stack trace upon crashing, identifying the failure point in a specific compiler pass. CLSmith

[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.

triggered such crashes in 4 distinct passes. DeepSmith triggered crashes in 10 distinct passes, including 3 of the 4 in which CLSmith did. Figure 5.9 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 produced many **bto** results across 13 Testbeds. Given the large kernel size, it is unclear how many of those are infinite loops or simply a result of slow compilation of large kernels. The average size of CLSmith **bto** kernels is 1558 lines. Automated test case reduction — in which thousands of permutations of a program are executed — may be prohibitively expensive for test cases with very long runtimes. DeepSmith produced **bto** results across 11 Testbeds and with an average kernel size of 9 lines, allowing for rapid identification of the underlying problem.

The integrated GPU Testbeds (3±) frequently failed to compile CLSmith kernels, resulting in over 10k **bc** and **bto** results. Of the build crashes, 68% failed silently, and the remainder were caused by the same two compiler assertions for which DeepSmith generated 4 line test cases, shown in Figure 5.10. DeepSmith also triggered silent build crashes in Testbeds 3±, and a further 8 distinct compiler assertions.

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

ComputeAorta (Testbeds 9±) defers kernel compilation so that it can perform 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.

5.4.4 Compiler Stability Over Time

The Clang front-end to LLVM supports OpenCL, and is commonly used in OpenCL drivers. This in turn causes Clang-related defects to potentially affect multiple compil-

```
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);
```

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

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

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

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

Figure 5.10: Example kernels which trigger compiler assertions which both CLSmith and DeepSmith exposed.

ers, for example the one in Figure 5.4e. 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 5.3 shows that the crash rate of the Clang front-end is, for the most part, steadily decreasing over time. The number of failing compiler crashes decreased tenfold between 3.6.2 and 5.0.0. Table 5.2 shows the 7 distinct assertions triggered during this experiment. Assertion 1 (*Uncorrected typos!*) is raised on all compiler versions — see Figure 5.4a for an example. The overall rate at which the assertion is triggered has decreased markedly, although there are slight increases between some releases. Notably, the current development trunk has the second lowest crash rate, but is joint first in terms of the number of unique assertions. Assertions 3 (*Addr == 0 —— hasTargetSpecificAddressSpace()*) and 4 (*isScalarType()*) were triggered by some kernels in the development trunk but not under any prior release. 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.

5.4.5 Extensibility of Language Model

A large portion of the DeepSmith architecture is language-agnostic, requiring only a corpus, encoder, and harness for each new language. This potentially significantly lowers the barrier-to-entry compared with prior grammar-based fuzzers. 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.

5.4.5.0.1 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.

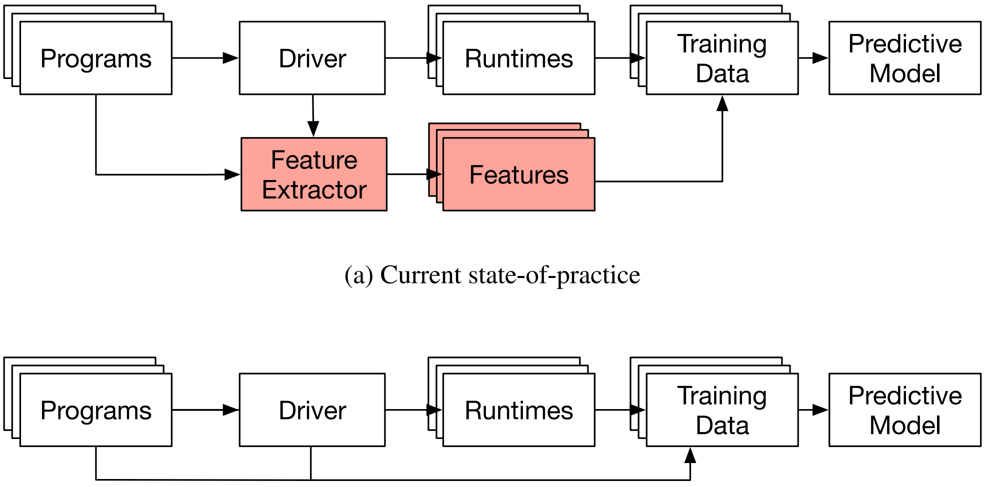
5.4.5.0.2 Initial Results We ran the generator and harness loop for 12 hours on four testbeds: the Solidity reference compiler `solc` with optimizations on or off, and `solc-js`, which is an Emscripten compiled version of the `solc` compiler. Our results are summarized in Table 3.1. We found numerous 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.

5.5 Summary

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.



(a) Current state-of-practice

(b) Our proposal

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

Chapter 6

End-to-end Deep Learning of Optimization Heuristics

6.1 Introduction

There are countless scenarios during the compilation and execution of a parallel program where decisions must be made as to how, or if, a particular 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 [Aga+06; Cum+16a; FE15; MSD16; SA05]. A *predictive model* is trained, using supervised machine learning, on empirical performance data and important quantifiable properties, or *features*, of representative programs. The model learns the correlation between these features and the 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 [GWO13; MDO14].

Still, experts are not completely removed from the design process, which is shown in Figure 6.1a. Selecting the appropriate features is a manual undertaking which requires a deep understanding of the system. The designer essentially decides which compile or runtime characteristics affect 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 [Cum+17b] the authors discovered that [GWO13] 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 [LBO14; Nam+10], they do not solve the problem in a practical way. They are deeply embedded into the compiler, require expert knowledge to guide the generation, have to be repeated from scratch for every new heuristic, and their search time can be prohibitive. 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 [He+16; KSH12], audio [Lee+09], and even computer code [All+14; AS14]. 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 6.1b 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* [Yos+14], 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.

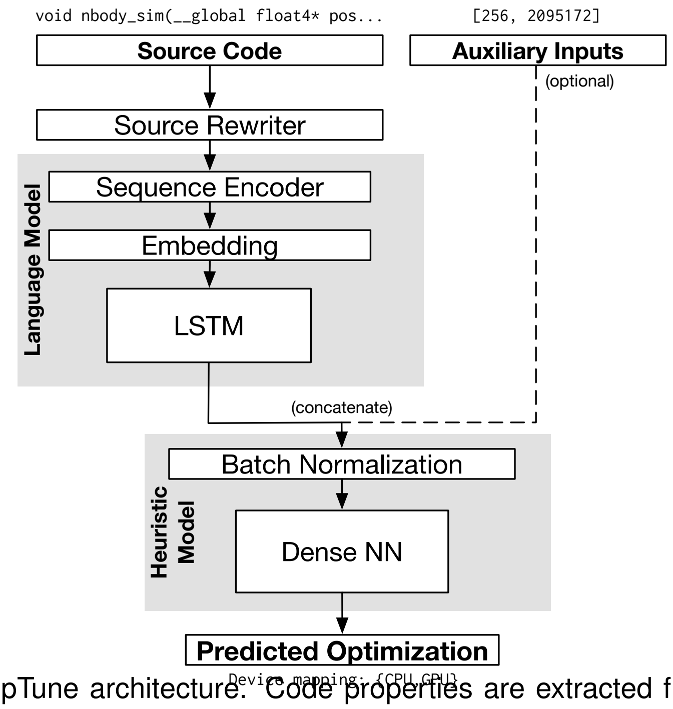


Figure 6.2: DeepTune architecture. Code properties are extracted from source code by the language model. They are fed, together with optional auxiliary inputs, to the heuristic model to produce the final prediction.

6.2 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.

6.2.1 System Overview

Figure 6.2 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 [Aba+16] and Theano [Ber+11] backends.

6.2.2 Language Model

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

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

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

6.2.2.0.1 Source Rewriter To begin with, we apply a series of *source normalizing* transformations extended from our previous work [Cum+17b]. 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 6.3a and 6.3b show the source rewriting applied to a simple program.

6.2.2.0.2 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 [Cum+17b], 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 [AS13], a token based vocabulary is used. This leads to shorter sequences,

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

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 6.3c shows the vocabulary derived for a single input source code Figure 6.3b.

6.2.2.0.3 Embedding During encoding, tokens in the vocabulary are mapped to unique integer values, e.g. `float` → 0, `int` → 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 [BDK14; Mik+13]. An embedding layer maps each token in the integer encoded vocabulary to a vector of real values. Given a vocabulary size V and embedding dimensionality D, an embedding matrix $\boldsymbol{W}_E \in \mathbb{R}^{V \times D}$ is learned during training, so that an integer encoded sequences of tokens $\boldsymbol{t} \in \mathbb{N}^L$ is mapped to the matrix $\boldsymbol{T} \in \mathbb{R}^{L \times D}$. We use an embedding dimensionality $D = 64$.

6.2.2.0.4 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 —

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

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

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

(b) The *streamcluster* kernel after source rewriting. Variable and function names are 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 6.3: Deriving a tokenized 1-of-k vocabulary encoding from an OpenCL source code.

and automatically — within the hidden layers of the network.

We use the the Long Short-Term Memory (LSTM) architecture [Chi+11] 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 [PMB13]), LSTMs employ a *forget gate* with a linear activation function, allowing them to retain activations for arbitrary durations. This makes them effective at learning complex relationships over long sequences [LBE15], an especially important capability for 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.

6.2.3 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 [Din+15; SA05]. When present, the values of auxiliary inputs are concatenated with the output of the language model, and fed into a heuristic model.

6.2.4 Heuristic Model

The heuristic model takes the learned representations of the source code and auxiliary inputs (if present), and uses these values to make the final 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, in which each scalar of the heuristic model's inputs $x_1 \ldots x_n$ is normalized to a mean 0 and standard deviation of 1:

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

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

The final component of DeepTune is comprised of two fully connected neural network layers. The first layer consists of 32 neurons. The second layer consists of a single neuron for each possible heuristic decision. Each neuron applies an activation function $f(x)$ over its inputs. We use rectifier activation functions $f(x) = \max(0, x)$ for the first layer due to their improved performance during training of deep networks [NH10]. 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.

6.2.5 Training the network

DeepTune is trained in the same manner as prior works, the key difference being that instead of having to manually create and extract features from programs, we simply use the raw program codes themselves.

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

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

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

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

length is only to improve training time. In production use, sequences do not need to be padded, allowing classification of arbitrary length codes.

6.3 Experimental Methodology

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.

6.3.1 Case Study A: OpenCL Heterogeneous Mapping

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

6.3.1.0.1 State-of-the-art In [GWO13], Grewe *et al.* develop a predictive model for mapping OpenCL kernels to the optimal device in CPU/GPU heterogeneous systems. They use supervised learning to construct decision trees, using a combination of static and dynamic kernel features. The static program features are extracted using a custom LLVM pass; the dynamic features are taken from the OpenCL runtime.

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

6.3.1.0.3 Experimental Setup We replicate the predictive model of Grewe *et al.* [GWO13]. We replicated the experimental setup of [Cum+17b] in which the experiments are extended to a larger set of 71 programs, summarized in Table 6.2a. The programs were evaluated on two CPU-GPU platforms, detailed in Table **??**.

6.3.1.0.4 DeepTune Configuration Figure 6.4a 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.

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

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

(a) Case Study A: OpenCL Heterogeneous Mapping

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

(b) Case Study B: OpenCL Thread Coarsening Factor

Table 6.2: Benchmark programs.

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

6.3.2 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 cer-

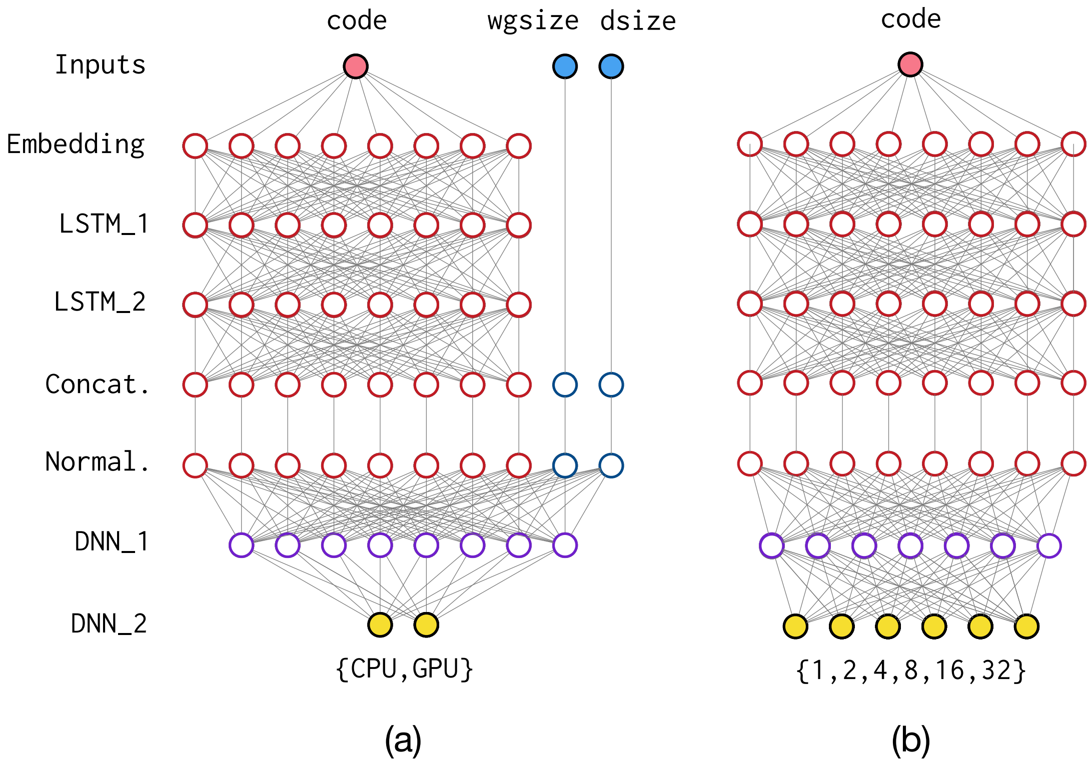
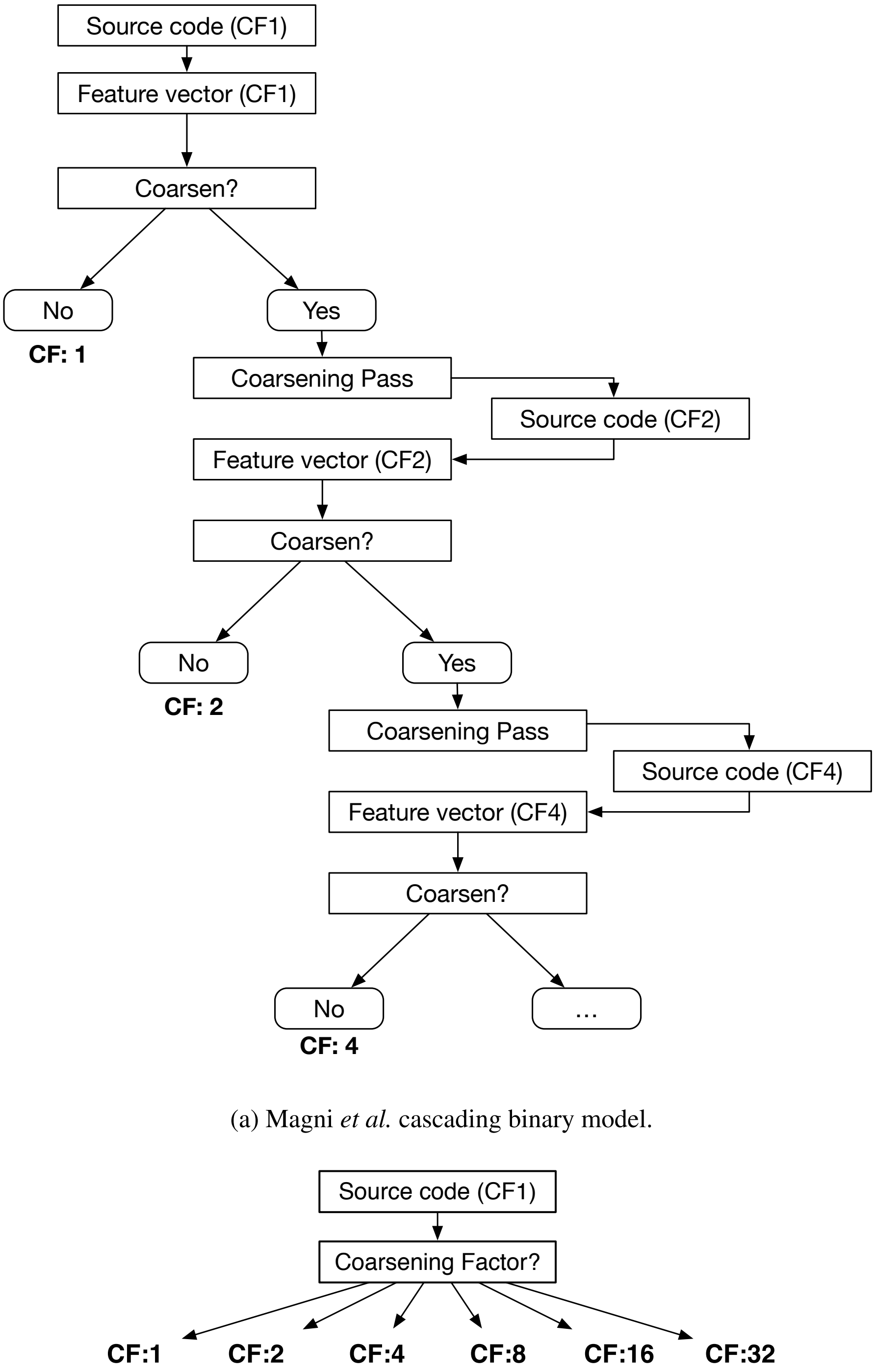


Figure 6.4: DeepTune neural networks, configured for (a) heterogeneous mapping, and (b) thread coarsening factor. The design stays almost the same regardless of the optimization problem. The only changes are the extra input for (a) and size of the output layers.

tain combinations of programs and architectures, for example programs with a large potential for Instruction Level Parallelism on Very Long Instruction Word architectures.

6.3.2.0.1 State-of-the-art Magni *et al.* present a predictive model for OpenCL thread coarsening in [MDO14]. They implement an iterative heuristic which determines whether a given program would benefit from coarsening. If yes, then the program is coarsened, and the process repeats, allowing further coarsening. In this manner, the problem is reduced from a multi-label classification problem into a series of binary decisions, shown in Figure 6.5a. They select from one of six possible coarsening factors: $(1, 2, 4, 8, 16, 32)$, divided into 5 binary choices.

6.3.2.0.2 Expert Chosen Features Magni *et al.* followed a very comprehensive feature engineering process. 17 candidate features were assembled from previous stud-



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

(b) Our approach.

Figure 6.5: Two approaches for predicting coarsening factor (CF) of OpenCL kernels. Magni *et al.* reduce the multi-label classification problem to a series of binary decisions, by iteratively applying the optimization and computing new feature vectors. Our approach simply predicts the coarsening factor directly from the source code.

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

Table 6.3: 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.

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

6.3.2.0.3 Experimental Setup We replicate the experimental setup of Magni *et al.* [MDO14]. The thread coarsening optimization is evaluated on 17 programs, listed in Table 6.2b. Four different GPU architectures are used, listed in Table **??**.

6.3.2.0.4 DeepTune Configuration Figure 6.4b shows the neural network configuration. We use the OpenCL kernel as input, and directly predict the coarsening factor.

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

Because [MDO14] 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.

6.3.3 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 6.4 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

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

lines of code: the first, adding the two auxiliary inputs; the second, increasing the size of the output layer for Case Study B from two neurons to six. The description of these differences is larger than the differences themselves.

6.4 Experimental Results

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.

6.4.1 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 6.6 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 6.7 shows these speedups. Both predictive models significantly outperform the static mapping; the Grewe *et al.* model achieves an average speedup of 2.91× on AMD and 1.26× on NVIDIA (geomean 1.18×). In 90% of cases, DeepTune matches or outperforms the predictions of the Grewe *et al.* model, achieving an average speedup of 3.34× on AMD and 1.41× on NVIDIA (geomean 1.31×). This 14% improvement in performance comes at a greatly reduced cost, requiring no intervention by humans.

6.4.2 Case Study B: OpenCL Thread Coarsening Factor

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

Figure 6.8 shows speedups achieved by the Magni *et al.* and DeepTune models for all programs and platforms. 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×, 1.01×, 0.86×, and 0.94×, respectively. DeepTune outperforms this, achieving speedups of 1.10×, 1.05×, 1.10×, and 0.99×.

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

The disappointing results for both predictive models 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

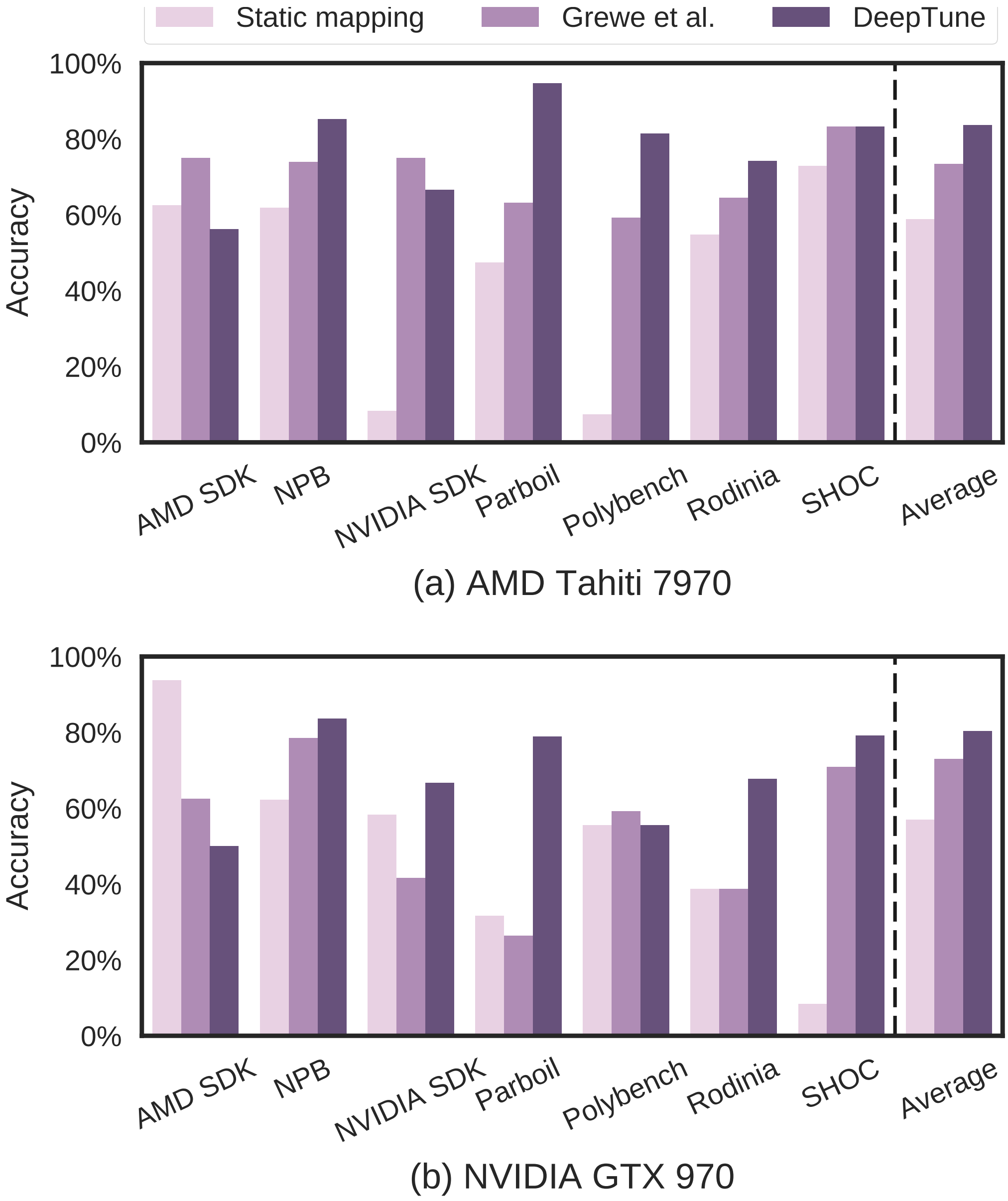
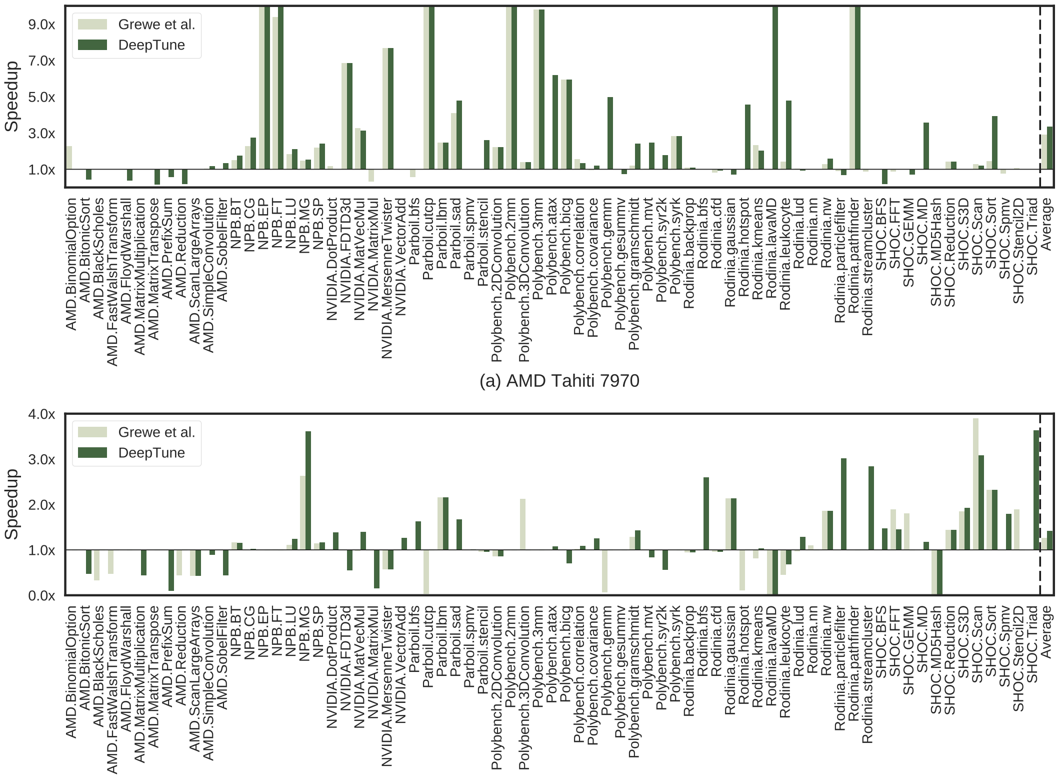


Figure 6.6: Accuracy of optimization heuristics for heterogeneous device mapping, aggregated by benchmark suite. The optimal static mapping achieves 58% accuracy. The Grewe *et al.* and DeepTune predictive models achieve accuracies of 73% and 84%, respectively.



(a) AMD Tahiti 7970

(b) NVIDIA GTX 970

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

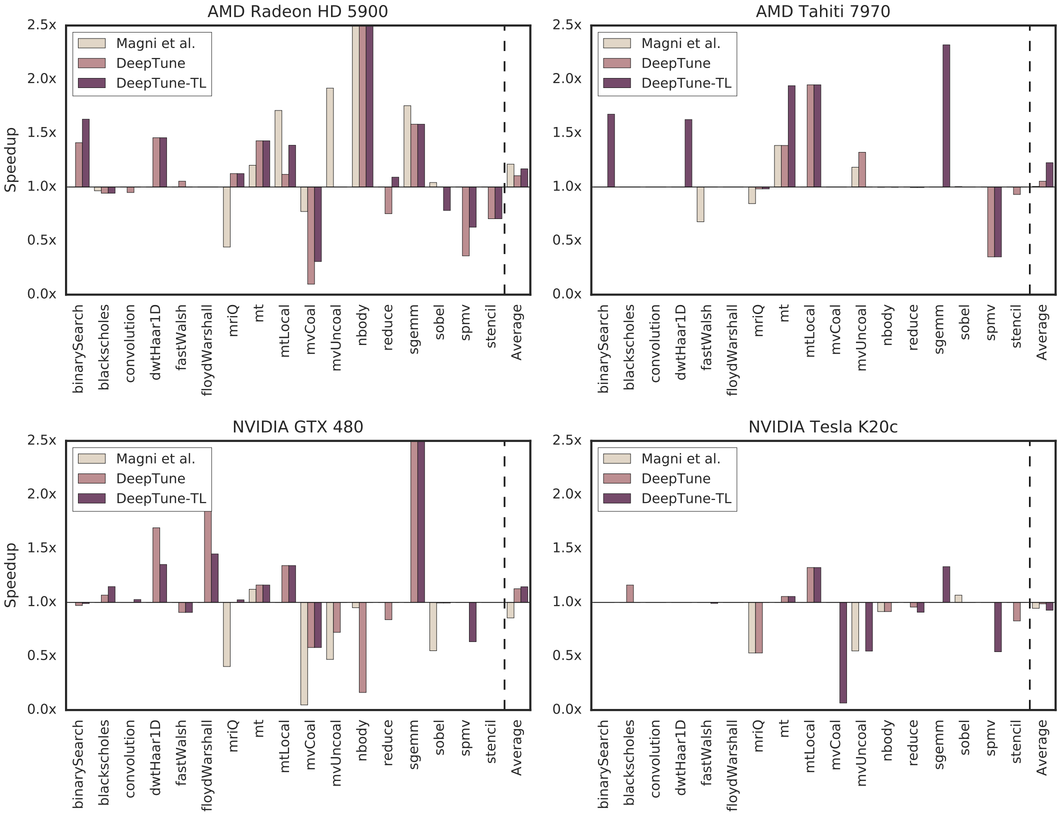


Figure 6.8: Speedups of predicted coarsening factors for each platform. DeepTune outperforms Magni *et al* on three of the four platforms. Transfer learning improves DeepTune speedups further, by 16% on average.

test a novel strategy for training optimization heuristics on a small number of programs by exploiting knowledge learned from other optimization domains.

6.4.3 Transfer Learning Across Problem Domains

There are inherent differences between the tasks of building heuristics for heterogeneous mapping and thread coarsening, evidenced by the contrasting choices of features and models in Grewe *et al.* and Magni *et al.* However, in both cases, the first role of DeepTune is to extract meaningful abstractions and representations of OpenCL code. Prior research in deep learning has shown that models trained on similar inputs for different tasks often share useful commonalities. The idea is that in neural network classification, information learned at the early layers of neural networks (i.e. closer to the input layer) will be useful for multiple tasks. The later the network layers are (i.e. closer to the output layer), the more specialized the layers become [ZF14].

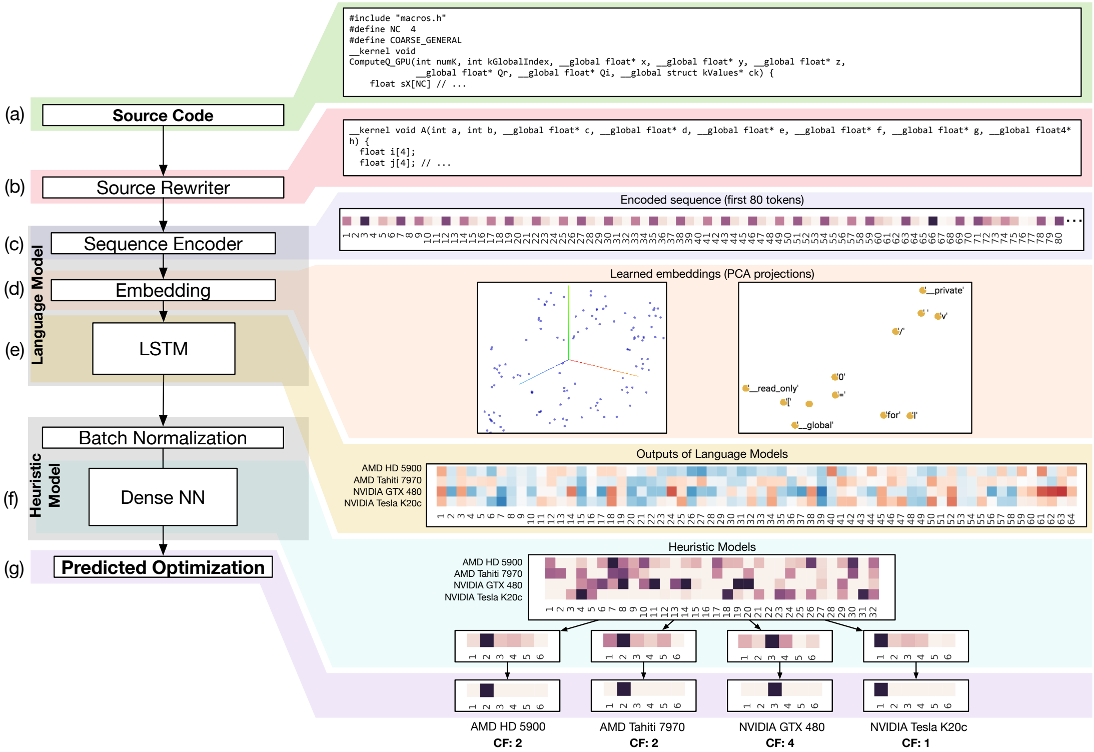
We hypothesized that this would be the case for DeepTune, enabling the novel transfer of information *across different optimization domains*. To test this, we extracted the language model — the `Embedding`, and `LSTM_{1,2}` layers — trained for the heterogeneous mapping task and *transferred* it over 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 6.8, our newly trained model, DeepTune-TL has improved performance for 3 of the 4 platforms: 1.17×, 1.23×, 1.14×, 0.93×, providing an average 12% performance improvement over Magni *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.

6.4.4 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 fea-



```
#include "macros.h"
#define NC  4
#define COARSE_GENERAL
__kernel void
ComputeQ_GPU(int numK, int kGlobalIndex, __global float* x, __global float* y, __global float* z,
             __global float* Qr, __global float* Qi, __global struct kValues* ck) {
    float sX[NC] // ...
```

```
__kernel void A(int a, int b, __global float* c, __global float* d, __global float* e, __global float* f, __global float* g, __global float4*
h) {
  float i[4];
  float j[4]; // ...
```

Figure 6.9: 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.

tures. 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 6.9 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 6.9b. The rewritten source code is tokenized and encoded in a 1-of-*k* vocabulary. Figure 6.9c 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 related tokens together. As such, they may differ between the four platforms. Figure 6.9d shows a PCA projection of the embedding space for one of the platforms, showing multiple clusters of tokens. By honing in on one of the clusters and annotating each point with its corresponding token, 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 6.9e 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 6.9f, which shows the two layers of the heuristic model. The activations within these increasingly diverge. The mean variance of activations across platforms increases threefold compared to the language model, from 0.039 to 0.107. Even the activations of the AMD HD 5900 and AMD Tahiti 7970 platforms are dissimilar, despite the final predicted coarsening factor for both platforms being the same. In Figure 6.9g 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%.

6.5 Summary

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 [Le+12] 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 [HMD15].

Chapter 7

Conclusions

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.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

7.1 Contributions

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

7.1.1 Workload Characterization

7.1.2 Compiler Optimizations

7.1.3 Compiler Testing

7.2 Critical Analysis

7.2.1 Limitations of Generative Models

Extensibility to other languages largely untested.

Generating multi-function programs.

Our new approach enables the synthesis of more human-like programs than current state of the art program generators, and without the expert guidance required by template based generators, but it has limitations. Our method of seeding the language models with the start of a function means that we cannot support user defined types, or calls to user-defined functions. This means that we only consider scalars and arrays as inputs; while 6 (2.3%) of the benchmark kernels from Table 6.2 use irregular data types as inputs. We will address this limitation through recursive program synthesis, 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.

7.2.2 Limitations of Sequential Classification

7.3 Future Work

Bibliography

[Aba+16] M. Abadi et al. “TensorFlow: A system for large-scale machine learning”. In: *arXiv:1605.08695* (2016).

[AC10] J. Ansel and C. Chan. “PetaBricks”. In: *XRDS: Crossroads, The ACM Magazine for Students* 17.1 (Sept. 2010). URL: `http://dl.acm.org/citation.cfm?doid=1836543.1836554`.

[Aga+06] F. Agakov et al. “Using Machine Learning to Focus Iterative Optimization”. In: *CGO*. IEEE, 2006.

[All+14] M. Allamanis et al. “Learning Natural Coding Conventions”. In: *FSE*. ACM, 2014.

[And+16] M. Andrychowicz et al. “Learning to Learn by Gradient Descent by Gradient Descent”. In: *NIPS*. 2016. arXiv: `1606.04474`.

[Ans+09] A. Ansel et al. “PetaBricks: A Language and Compiler for Algorithmic Choice”. In: *PLDI*. ACM, 2009.

[Ans+13] J. Ansel et al. “OpenTuner: An Extensible Framework for Program Autotuning”. In: *PACT*. ACM, 2013.

[Ans09] Jason Ansel. “PetaBricks: a language and compiler for algorithmic choice”. PhD thesis. MIT, 2009.

[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.

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

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

[Ash+15] A. Ashari et al. “On Optimizing Machine Learning Workloads via Kernel Fusion”. In: *PPoPP*. 2015.

[Atk13] C. E. Atkin-granville. “Parallelism Detection using Dynamic Instrumentation in a Virtual Machine”. PhD thesis. University of Edinburgh, 2013.

[Bai+14] R. Baishakhi et al. “A Large Scale Study of Programming Languages and Code Quality in Github”. In: *FSE*. ACM, 2014.

[Bai+91] D. H. Bailey et al. “The NAS Parallel Benchmarks”. In: *IJHPCA* 5.3 (1991).

[Bas+17] O. Bastani et al. “Synthesizing Program Input Grammars”. In: *PLDI*. 2017. arXiv: `1608.01723`. URL: `http://arxiv.org/abs/1608.01723`.

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

[BCV13] Y. Bengio, A. Courville, and P. Vincent. “Representation Learning: A Review and New Perspectives”. In: *TPAMI* 35.8 (2013). arXiv: `1206.5538`.

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

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

[BEN93] Utpal Banerjee, Rudolf Eigenmann, and Alexandru Nicolau. “Automatic program parallelization”. In: *Proceedings of the IEEE* 81.2 (1993).

[Ber+08] L. Berkeley et al. “Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures”. In: *SC*. 2008.

[Ber+11] J. Bergstra et al. “Theano: Deep Learning on GPUs with Python”. In: *BigLearning Workshop*. 2011.

[Bod+98] F. Bodin et al. “Iterative compilation in a non-linear optimisation space”. In: *PACT*. ACM, 1998.

[Bol+03] J. Bolz et al. “Sparse matrix solvers on the GPU: conjugate gradients and multigrid”. In: *TOG* 22.3 (2003).

[Bol+16] T. Bolukbasi et al. “Man is to Computer Programmer as Woman is to Homemaker? Debiasing Word Embeddings”. In: *arXiv:1607.06520* (2016).

[Bow+15] S. R. Bowman et al. “Generating Sentences from a Continuous Space”. In: *arXiv:1511.06349* (2015).

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

[Bud15] N. Buduma. *Fundamentals of Deep Learning*. O’Reilly, 2015.

[Bun+17] R. Bunel et al. “Learning to Superoptimize Programs”. In: *ICLR*. 2017. arXiv: `1611.01787`.

[Bur+13] E. K. Burke et al. “Hyper-heuristics: a survey of the state of the art”. In: *JORS* 64 (2013).

[Cav+06] J. Cavazos et al. “Automatic Performance Model Construction for the Fast Software Exploration of New Hardware Designs”. In: *CASES*. 2006.

[CBN16] A. Caliskan-islam, J. J. Bryson, and A. Narayanan. “Semantics derived automatically from language corpora necessarily contain human biases”. In: *arXiv:1608.07187* (2016).

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

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

[Cha+09] C. Chan et al. “Autotuning multigrid with PetaBricks”. In: *SC*. 2009.

[Che+09] S. Che et al. “Rodinia: A Benchmark Suite for Heterogeneous Computing”. In: *IISWC*. IEEE, Oct. 2009.

[Che+13] Y. Chen et al. “Taming Compiler Fuzzers”. In: *PLDI* (2013).

[Che+16a] J. Chen et al. “An Empirical Comparison of Compiler Testing Techniques”. In: *ICSE*. 2016.

[Che+16b] Y. Chen et al. “Coverage-Directed Differential Testing of JVM Implementations”. In: *PLDI*. 2016.

[Che+17] J. Chen et al. “Learning to Prioritize Test Programs for Compiler Testing”. In: *ICSE*. 2017.

[Chi+11] M. Chi et al. “An evaluation of pedagogical tutorial tactics for a natural language tutoring system: A reinforcement learning approach”. In: *IJAIED* 21.1-2 (2011).

[Cho+17] M. Choi et al. “End-to-End Prediction of Buffer Overruns from Raw Source Code via Neural Memory Networks”. In: *arXiv:1703.02458* (2017).

[CKF11] R. Collobert, K. Kavukcuoglu, and C. Farabet. “Torch7: A Matlab-like Environment for Machine Learning”. In: *BigLearn*. 2011.

[CM08] G. Contreras and M. Martonosi. “Characterizing and improving the performance of Intel Threading Building Blocks”. In: *IISWC*. IEEE, Oct. 2008.

[Col+13] A. Collins et al. “MaSiF: Machine Learning Guided Auto-tuning of Parallel Skeletons”. In: *HiPC*. IEEE, 2013.

[Con+16] A. Conneau et al. “Very Deep Convolutional Networks for Natural Language Processing”. In: *arXiv:16006.01781* (2016).

[CSB11] M. Christen, O. Schenk, and H. Burkhart. “PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures”. In: *PDPS*. IEEE, May 2011.

[Cum+16a] C. Cummins et al. “Autotuning OpenCL Workgroup Size for Stencil Patterns”. In: *ADAPT*. 2016. arXiv: `1511.02490`.

[Cum+16b] C. Cummins et al. “Towards Collaborative Performance Tuning of Algorithmic Skeletons”. In: *HLPGPU*. 2016.

[Cum+17a] C. Cummins et al. “End-to-end Deep Learning of Optimization Heuristics”. In: *PACT*. IEEE, 2017.

[Cum+17b] C. Cummins et al. “Synthesizing Benchmarks for Predictive Modeling”. In: *CGO*. IEEE, 2017.

[CW14] G. Chen and B. Wu. “PORPLE: An Extensible Optimizer for Portable Data Placement on GPU”. In: *MICRO*. ACM, 2014.

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

[Dan+10] A. Danalis et al. “The Scalable HeterOgeneous Computing (SHOC) Benchmark Suite”. In: *GPGPU*. ACM, 2010.

[Das11] U. Dastgeer. “Skeleton Programming for Heterogeneous GPU-based Systems”. PhD thesis. 2011.

[DE98] Leonardo Dagum and Rameshm Enon. “OpenMP: an industry standard API for shared-memory programming”. In: *CSE*. 1998.

[Din+15] Y. Ding et al. “Autotuning Algorithmic Choice for Input Sensitivity”. In: *PLDI*. ACM, 2015.

[DK15] U. Dastgeer and C. Kessler. “Smart Containers and Skeleton Programming for GPU-Based Systems”. In: *IJPP* (2015).

[Dub+07] C. Dubach et al. “Fast Compiler Optimisation Evaluation Using Code-Feature Based Performance Prediction”. In: *CF*. ACM, 2007.

[Dub+09] C. Dubach et al. “Portable Compiler Optimisation Across Embedded Programs and Microarchitectures using Machine Learning”. In: *MICRO*. ACM, 2009.

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

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

[Fur+11] G. Fursin et al. “Milepost GCC: Machine Learning Enabled Self-tuning Compiler”. In: *IJPP* 39.3 (2011).

[Fur+14] G. Fursin et al. “Collective Mind: Towards practical and collaborative auto-tuning”. In: *Scientific Programming* 22.4 (2014).

[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 et al. “A Case for Machine Learning to Optimize Multicore Performance”. In: *HotPar*. 2009.

[Gao+15] H. Gao et al. “Are You Talking to a Machine? Dataset and Methods for Multilingual Image Question Answering”. In: *arXiv:1505.05612* (2015).

[Gar15] J. D. Garvey. “Automatic Performance Tuning of Stencil Computations on GPUs”. PhD thesis. University of Toronto, 2015.

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

[GKT11] S. Gulwani, V. A. Korthikanti, and A. Tiwari. “Synthesizing geometry constructions”. In: *PLDI*. 2011.

[GLM08] P. Godefroid, M. Y. Levin, and D. Molnar. *Automated Whitebox Fuzz Testing*. 2008. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.151.9430%7B%5C&%7Drep=rep1%7B%5C&%7Dtype=pdf.

[GLS01] Rakesh Ghiya, Daniel Lavery, and David Sehr. “On the importance of points-to analysis and other memory disambiguation methods for C programs”. In: *PLDI*. Vol. 36. 5. 2001.

[Goo+14] I. Goodfellow et al. “Generative Adversarial Networks”. In: *arXiv:1406.2661* (2014).

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

[Gra+12] S. Grauer-Gray et al. “Auto-tuning a High-Level Language Targeted to GPU Codes”. In: *InPar*. 2012.

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

[Gre+15a] K. Greff et al. “LSTM: A Search Space Odyssey”. In: *arXiv:1503.04069* (2015). arXiv: 1503.04069. URL: http://arxiv.org/abs/1503.04069.

[Gre+15b] K. Gregor et al. “DRAW: A Recurrent Neural Network For Image Generation”. In: *arXiv:1502.04623* (2015).

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

[Gu+16] X. Gu et al. “Deep API Learning”. In: *FSE*. ACM, 2016.

[Gul11] S. Gulwani. “Automating string processing in spreadsheets using input-output examples”. In: *POPL*. 2011.

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

[He+16] K. He et al. “Deep Residual Learning for Image Recognition”. In: *CVPR*. IEEE, 2016.

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

[Hin01] Michael Hind. “Pointer Analysis: Haven’t We Solved This Problem Yet?” In: *PASTE*. ACM, 2001.

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

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

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

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

[HPS12] J. Holewinski, L. Pouchet, and P. Sadayappan. “High-performance Code Generation for Stencil Computations on GPU Architectures”. In: *SC*. 2012.

[Jad+16] M. Jaderberg et al. “Decoupled Neural Interfaces using Synthetic Gradients”. In: *arXiv:1608.05343* (2016).

[Jia+10] Y. Jiang et al. “Exploiting Statistical Correlations for Proactive Prediction of Program Behaviors”. In: *CGO* (2010).

[Jia+14] Y. Jia et al. “Caffe: Convolutional Architecture for Fast Feature Embedding”. In: *ACMMM*. 2014.

[Jim+14] A. Jimborean et al. “Dynamic and Speculative Polyhedral Parallelization Using Compiler-Generated Skeletons”. In: *IJPP* 42.4 (2014).

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

[Joz+16] R. Jozefowicz et al. “Exploring the Limits of Language Modeling”. In: *arXiv:1602.02410* (2016).

[JWC17] K. Joseph, W. Wei, and K. C. Carley. “Girls rule, boys drool: Extracting semantic and affective stereotypes on Twitter”. In: *CSCW*. 2017.

[Kal+09] E. Kalliamvakou et al. “The Promises and Perils of Mining GitHub”. In: *MSR*. 2009.

[Kam+10] S. Kamil et al. “An auto-tuning framework for parallel multicore stencil computations”. In: *IPDPS* (2010).

[KB15] D. P. Kingma and J. L. Ba. “Adam: a Method for Stochastic Optimization”. In: *ICLR* (2015). arXiv: `1412.6980`.

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

[Koc+17] U. Koc et al. “Learning a Classifier for False Positive Error Reports Emitted by Static Code Analysis Tools”. In: *MAPL*. 2017.

[Kou+17] M. Koukoutos et al. “On Repair with Probabilistic Attribute Grammars”. In: *arXiv:1707.04148* (2017).

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

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

[Lam+15] A. N. Lam et al. “Combining Deep Learning with Information Retrieval to Localize Buggy Files for Bug Reports”. In: *ASE*. 2015.

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

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

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

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

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

[Le+12] Q. V. Le et al. "Building High-level Features Using Large Scale Unsupervised Learning". In: *ICML*. 2012. arXiv: `1112.6209`.

[Lee+09] H. Lee et al. "Unsupervised Feature Learning for Audio Classification using Convolutional Deep Belief Networks". In: *NIPS*. 2009.

[Lee+10] V. W. Lee et al. "Debunking the 100X GPU vs. CPU myth". In: *ACM SIGARCH Computer Architecture News* 38 (2010).

[Lee+14] H. Lee et al. "Locality-Aware Mapping of Nested Parallel Patterns on GPUs". In: *MICRO*. ACM, 2014.

[LEE16] C. Loncaric, T. Emina, and M. D. Ernst. "Fast Synthesis of Fast Collections". In: *PLDI*. 2016.

[Lev+16] S. Levine et al. "End-to-End Training of Deep Visuomotor Policies". In: *JMLR* 17 (2016). arXiv: `1504.00702`.

[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 et al. "Many-Core Compiler Fuzzing". In: *PLDI*. 2015.

[LME09] Seyong Lee, S J Min, and Rudolf Eigenmann. "OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization". In: *PPoPP* (2009). URL: `http://dl.acm.org/citation.cfm?id=1504194`.

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

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

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

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

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

[Mik+13] T. Mikolov et al. "Distributed Representations of Words and Phrases and their Compositionality". In: *NIPS*. 2013.

[Mik+15] T. Mikolov et al. "Recurrent Neural Network based Language Model". In: *Interspeech*. 2015.

[Mni+15] V. Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015).

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

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

[Mur+16] S. Muralidharan et al. "Architecture-Adaptive Code Variant Tuning". In: *ASPLOS*. ACM, 2016.

[Nam+10] M. Namolaru et al. "Practical Aggregation of Semantical Program Properties for Machine Learning Based Optimization". In: *CASES*. 2010.

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

[NH10] V. Nair and G. E. Hinton. "Rectified Linear Units Improve Restricted Boltzmann Machines". In: *ICML*. 2010. arXiv: `1111.6189v1`.

[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.

[Oda+15] Y. Oda et al. "Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation". In: *ASE*. IEEE, 2015.

[Ogi+14] W. F. Ogilvie et al. "Fast Automatic Heuristic Construction Using Active Learning". In: *LCPC*. 2014.

[Ogi+15] W. F. Ogilvie et al. "Intelligent Heuristic Construction with Active Learning". In: *CPC*. 2015.

[Ogi+17] W. F. Ogilvie et al. "Minimizing the cost of iterative compilation with active learning". In: *CGO* (2017).

[Oqu+14] M. Oquab et al. "Learning and Transferring Mid-Level Image Representations using Convolutional Neural Networks". In: *CVPR*. IEEE, 2014.

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

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

[PF10] E. H. Phillips and M. Fatica. “Implementing the Himeno benchmark with CUDA on GPU clusters”. In: *IPDPS*. 2010.

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

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

[Pot+15] R. Potter et al. “Kernel composition in SYCL”. In: *IWOCL*. 2015.

[Pra10] Prakash Prabhu. “Safe Programmable Speculative Parallelism”. In: *PLDI*. ACM, 2010.

[Rag+16] M. Raghu et al. “On the expressive power of deep neural networks”. In: *arXiv:1606.05336* (2016).

[Raz+14] A. S. Razavian et al. “CNN Features off-the-shelf: an Astounding Baseline for Recognition”. In: *CVPRW*. IEEE, 2014. arXiv: `1403.6382`.

[Reg+12] J. Regehr et al. “Test-Case Reduction for C Compiler Bugs”. In: *PLDI*. 2012.

[RGW17] J. Ren, L. Gao, and Z. Wang. “Optimise Web Browsing on Heterogeneous Mobile Platforms: A Machine Learning Based Approach”. In: *INFOCOM*. 2017.

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

[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 et al. “Optimization principles and application performance evaluation of a multithreaded GPU using CUDA”. In: *PPoPP*. 2008.

[Ryo+08b] S. Ryoo et al. “Program optimization space pruning for a multithreaded GPU”. In: *CGO*. IEEE, 2008.

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

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

[SFD15] M. Steuwer, C. Fensch, and C. Dubach. “Patterns and Rewrite Rules for Systematic Code Generation From High-Level Functional Patterns to High-Performance OpenCL Code”. In: *arXiv:1502.02389* (2015).

[Sim+12] J. Sim et al. “A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications”. In: *PPoPP*. ACM, 2012.

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

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

[Smi16] C. Smith. “MapReduce Program Synthesis”. In: *PLDI*. 2016.

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

[SMS15] N. Srivastava, E. Mansimov, and R. Salakhutdinov. “Unsupervised Learning of Video Representations using LSTMs”. In: *ICML*. 2015.

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

[Str+12] J. A. Stratton et al. “Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing”. In: *Center for Reliable and High-Performance Computing* (2012).

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

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

[TG10] Oliver Trachsel and Thomas R Gross. “Variant-based Competitive Parallel Execution of Sequential Programs”. In: *CF*. ACM, 2010.

[Tin+16] P. Ting et al. “FEAST: An Automated Feature Selection Framework for Compilation Tasks”. In: *arXiv:1610.09543* (2016).

[TMW17] B. Taylor, V. S. Marco, and Z. Wang. “Adaptive Optimization for OpenCL Programs on Embedded Heterogeneous Systems”. In: *LCTES*. 2017.

[Tru+16] L. Truong et al. “Latte: a language, compiler, and runtime for elegant and efficient deep neural networks”. In: *PLDI*. 2016.

[VCS00] P. Verplaetse, J. Van Campenhout, and D. Stroobandt. “On synthetic benchmark generation methods”. In: *ISCAS*. Vol. 4. FEBRUARY. 2000.

[VDP09] Y. Voronenko, F. De Mesmay, and M. Püschel. “Computer Generation of General Size Linear Transform Libraries”. In: *CGO*. IEEE, 2009.

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

[Vin+15] O. Vinyals et al. “Show and Tell: A Neural Image Caption Generator”. In: *CVPR* (2015).

[Wan+14] Z. Wang et al. “Integrating Profile-driven Parallelism Detection and Machine-learning-based Mapping”. In: *TACO* (2014).

[Wan+17] J. Wang et al. “Skyfire: Data-Driven Seed Generation for Fuzzing”. In: *S&P*. 2017.

[Whi+] M. White et al. “Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities”. In: *arXiv:1707.04742* (). arXiv: `arXiv:1707.04742v1`.

[Whi+15] M. White et al. “Toward Deep Learning Software Repositories”. In: *MSR*. 2015.

[WM15] M. Wahib and N. Maruyama. “Automated GPU Kernel Transformations in Large-Scale Production Stencil Applications”. In: *HPDC*. 2015.

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

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

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

[Wu+14] Y. Wu et al. “Exploring the Ecosystem of Software Developers on GitHub and Other Platforms”. In: *CSCW*. 2014.

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

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

[Xio+16] W. Xiong et al. "Achieving Human Parity in Conversational Speech Recognition". In: *arXiv:1610.05256* (2016).

[Yan+11] X. Yang et al. "Finding and Understanding Bugs in C Compilers". In: *PLDI*. 2011.

[Yos+14] J. Yosinski et al. "How Transferable are Features in Deep Neural Networks?" In: *NIPS*. 2014.

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

[Zar+16] W. Zaremba et al. "Learning Simple Algorithms from Examples". In: *ICML*. 2016.

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

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

[ZM12] Y. Zhang and F. Mueller. "Auto-generation and Auto-tuning of 3D Stencil Codes on GPU clusters". In: *CGO*. IEEE, 2012.

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