Deep Learning for Compilers

First Year Review Document

by

Chris Cummins

Supervisors:
Hugh Leather, Pavlos Petoumenos, Richard Mayr

May 7, 2018

Institute for Computing Systems Architecture,
School of Informatics,
University of Edinburgh

Abstract

Continued advancements in machine learning have increasingly extended the state-of-the art in language modelling for natural language processing. Coupled with the increasing popularity of websites such as GitHub for hosting software projects, this raises the potential for large scale language modelling over open source code to build probabilistic models which capture both the semantics of a programming language and its common usage in real world applications. This document describes my work towards the development of systems for automatically generating programs in a given programming language. The aim of this work is improvements to predictive modelling for compiler optimisation and compiler testing. In my first year I have applied LSTMs to large corpuses of code mined from open source in order to generate executable OpenCL kernels. These generated programs have been shown to improve the performance of state-of-the-art predictive models; though the programs are typically short and limited to operating only on scalars and vectors of numerical values. This document describes the plans for future work to extend this initial proof-of-concept through the use of formal grammars to generate programs in arbitrary languages, capable of satisfying arbitrary properties of interest. This will enable the automatic generation of programs in any language for a which a large corpus of real world codes is available, with a range of applications including exploration of unknown parts of program feature spaces, and identifying bugs in compilers.

Contents

1 Introduction

1.1 Project aim

The aim of this project is the development of novel deep learning methods for the automatic generation of programs, with applications for improving predictive modelling and testing methods for compilers. The automatic generation of programs benefits predictive modelling if the generated programs enumerate a given program feature space, providing useful benchmarks for learning optimisation heuristics. In compiler testing, the automatic generation of programs is beneficial if there are programs produced whose computed results differ between compiler implementations.

Current approaches to program generation are limited to random instantiation of code templates and datasets, or inflexible formal grammar methods. These methods typically produce programs which are markedly unlike real hand-written programs. This limits their utility for predictive modelling and compiler testing — programs which do not resemble hand-written code may expose bugs in compilers which are not considered a priority to address; and, given their unusual nature, may fail to enumerate the parts of program feature spaces which are relevant to real human workloads.

The aim of my work is to develop a new approach to program generation, in which a stream of program fragments taken from open source code are analysed at large scale and used to build probabilistic language models from which new programs can be sampled. In modelling real hand written codes, these generated programs will be indistinguishable from human workloads, and will allow enumeration of the relevant part of program feature spaces, as well as identifying bugs in compilers which arise from common usage of a programming language.

1.2 Project objectives

1. *A system for automatic program generation through deep learning.* At its simplest, such a system would consist of a method for generating program code, and a tool for checking whether a given program is syntactically and semantically correct and executable. The foundation for developing this system is the application of language modelling to large corpuses of open source code, with the intent of learning not just the syntax and semantics of a particular programming language, but also the patterns and traits of programs which are most representative of real human workloads. The goal of this objective is the automatic generation of human-like workloads.

2. *A technique for the generation of programs which match a given property of interest.* Properties of interest could include: having static code features within a given range; computing different outputs when compiled using different compilers; or exceeding a minimum threshold of runtime on a particular architecture. This extension to the previous objective will allow for a directed exploration of a program feature space.

3. *An agent-based approach for per-program selection and ordering of compiler optimisation passes using reinforcement learning.* Both compiler optimisation pass selection and phase ordering are critical to the effectiveness of optimising compilers, and are difficult problems to tackle. Fixed phase orderings fail to exploit the available performance on a per-program basis, and the optimisation spaces are large and high dimensional, rendering exhaustive search infeasible. As a demonstration of the effectiveness of automatic program generation, the fixed phase ordering of a compiler may be replaced with an agent which would enable online tuning of the optimisation pipeline, using automatically generated programs to explore the high dimensional optimisation space.

4. *Dissemination of results in one or more top-tier conference or journal.* Each of the previously described objectives corresponds to one or more potential publications. Relevant conferences and journals to target for publication include: CGO, FSE, HiPC, ICPP, IJPP, LCTES, OOPSLA, PACT, PLDI, PPoPP, and TACO.

2 Literature survey

2.1 Deep learning

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) [Mik10; SSN12] to generate sequences of OpenCL code (Appendix C). 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; GS18]. 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+15].

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].

Software engineering 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, Zhang, Zhang, and Kim 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.

2.2 Iterative compilation

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, Kisuki, Knijnenburg, O'Boyle, and Rohou [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 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, Kashnikov, Memon, Chamski, Temam, Namolaru, Yom-Tov, Mendelson, Zaks, Courtois, Bodin, Barnard, Ashton, Bonilla, Thomson, Williams, and O'Boyle 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, Enmyren, and Kessler developed a machine learning based autotuner for the SkePU skeleton library in [DEK11]. 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, Petoumenos, Wang, and Leather 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 [Fur+14; FT10; MF13], Fursin, Miceli, Lokhmotov, Gerndt, Baboulin, Malony, Chamski, Novillo, and Del Vento 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.

Dynamic optimisers 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.

Superoptimisers 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.

GPUs 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, Rodrigues, Baghsorkhi, Stone, Kirk, and Hwu 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.

Stencils 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, Farmer, Grinspun, and Schroder 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, Datta, Fox, and Patterson 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, Datta, Murphy, Volkov, Williams, and Carter 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, Chan, Oliker, Shall, and Williams 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 A) 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 A).

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.

2.3 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; GA15] (Appendix A). 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 [Yan+11] and OpenCL [Lid+15]. Programs generated by grammar-based approaches are often unlike real hand-written 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+16].

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 Summary of progress

Progress in my first year has been focused on developing the initial proof-of-concept of this novel approach to program generation, and in extending and publishing previous work on predictive modeling for optimising GPU stencil computations.

3.1 Research outputs

Publications

1. Cummins, C, Petoumenos, P, Steuwer, M & Leather, H, "Autotuning OpenCL Workgroup Size for Stencil Patterns". In International Workshop on Adaptive Self-tuning Computing Systems (ADAPT). HiPEAC, Prague, Czech Republic, 18 January 2016 [**Appendix A**];

2. Cummins, C, Petoumenos, P, Steuwer, M & Leather, H , "Towards Collaborative Performance Tuning of Algorithmic Skeletons". In Workshop on High-Level Programming for Heterogeneous & Hierarchical Parallel Systems (HLPGPU). HiPEAC, Prague, Czech Republic, 19 January 2016 [**Appendix B**];

3. Cummins, C, Petoumenos, P, Wang, Z & Leather, H, "Synthesizing Benchmarks for Predictive Modeling". To appear in International Symposium on Code Generation and Optimization (CGO). Austin, TX, USA, 4–8 February 2017 [**Appendix C**].

Posters

1. Cummins, C, Petoumenous, P, Steuwer, M, Leather, H, "Humans Need Not Apply", Google PhD Student Summit on Compiler & Programming Technology, Munich, Germany, 7–9 December 2015;

2. Cummins, C, Petoumenos, P, Steuwer, M & Leather, H, "Autotuning OpenCL Workgroup Sizes", HiPEAC, Prague, Czech Republic, 18–20 January 2016.

3. Cummins, C, Petoumenos, P, Steuwer, M & Leather, H, "Autotuning OpenCL Workgroup Sizes", 37th ACM SIGPLAN conference on Programming Language Design & Implementation (PLDI), Santa Barbara, California, 13–17 June 2016.

4. Cummins, C, Petoumenos, P, Steuwer, M & Leather, H, "Autotuning OpenCL Workgroup Sizes", 12th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES), Fiuggi, Italy, 10–16 July 2016.

Talks

1. Cummins, C, "All the OpenCL on GitHub: Teaching an AI to code, one character at a time". Amazon Development Centre, Edinburgh, UK, 19 May 2016;

2. Cummins, C, "Building an AI that Codes", Ocado Technology, Hatfield, UK, 22 July 2016;

3. Cummins, C, "Machine Learning & Compilers", Codeplay Software, Edinburgh, UK, 9 September 2016.

3.2 Codeplay internship

From April through September I interned at Codeplay Software in Edinburgh. My role within Codeplay was as a member of the SYCL demos team, in which I contributed to SYCL support for TensorFlow and Eigen; developed a Python frontend for the C++ template library VisionCpp; and developed a toolset of metaprogramming utilities for SYCL.

Eigen[1] is a C++ template library which provides linear math operations on n-dimensional tensors. It is a key dependency of TensorFlow **??**, a popular library for distributed and parallelised machine learning. During my internship I imlpemented GPU memory management for tensors using SYCL, and support for broadcast operations.

SYCL is a single-source specification for heterogeneous parallelism in C++ [Khr15]. Codeplay is developing a compiler for this standard, ComputeCpp. Compiling a SYCL application with ComputeCpp is a two pass process. In the first pass, the device compiler produces SPIR code for execution on parallel devices. SPIR is an intermediate language which extends LLVM bytecode for parallel compute and graphics [Khr14]. The second compiler pass uses a host compiler and links against the generated SPIR bytecode. At runtime, the SYCL runtime schedules kernels for execution on parallel devices, and OpenCL platforms compile the SPIR bytecode to executable device code.

VisionCPP [Gol16] is a C++ template library for performance-portable vision processing using SYCL. Users declare trees of VisionCpp expressions, which at compile time may be fused into a single kernel for efficient execution on GPUs [Pot+15]. While working at Codeplay I implemented a Python interface for VisionCpp, which allows for simple construction of expression trees using the python object interface. When evaluated, python expression trees are lazily evaluated to a sequence of nodes, from which C++ code is generated. The ComputeCpp compiler is then invoked to generate a native binary for the expression tree, which is linked and loaded by the python runtime and called. Inputs and outputs are transferred between python and the native binary, allowing for a seemless interface of high level scripting language and effecitient native GPU-accelerated code. This is a research project with the potential for publication in a high level GPU programming workshop.

[1] `http://eigen.tuxfamily.org/`

4 Proposal

4.1 Thesis outline

Chapter 1: Introduction. An introduction to the main topic, summary of contributions, and motivating examples.

Chapter 2: Literature survey. A thorough exposition of the relevant literature in the fields of compiler research and machine learning.

Chapter 3: Language modelling for program code. Defining machine learning systems for language modelling over corpuses of program source code. Background and application of language modelling approaches to programming languages. Description of source code transformations for machine learning, and an empirical evaluation of the effects of transformations on the effectiveness of learned LSTM models. This chapter will provide the foundational methodology for building probabilistic models of programs in a given programming languages.

Chapter 4: Automatic program generation through deep learning. Introducing novel techniques for generating and evaluating programs and their inputs, as described in the first project objective. Empirical evaluations for two use cases: compiler autotuning and differential testing. For each evaluation: experimental setup, methodology, results, and analysis. The basis of this chapter will be previous work for synthesising OpenCL kernels [Appendix C], with extensions for a grammar-based approach to program generation.

Chapter 5: Searching the program feature space. A methodology for performing directed searches of program feature spaces using deep learning program synthesis, satisfying the second project objective. This may be an iterative process of mutating code to converge on the goal features, or using a model-based approach which guarantees program synthesis with specific features by considering only well-formed sequences which result in the target feature values. This section will contain an empirical evaluation of convergence time and possible coverage of feature spaces.

Chapter 6: Learning the compiler optimisation pipeline. Applying an agent based approach to the compiler optimisation pipeline, the third project objective. This would replace the fixed ordering of optimisation passes in present compilers, and would use reinforcement learning combined with automatic program generation to explore the space of pass orderings and selection. The agent will be trained using synthesised programs, and empirically evaluated on benchmark suites using LLVM.

Chapter 7: Conclusions. Summary of contributions and research impact. Future research questions and directions for extended work.

4.2 Work plan

Year 2 The year will begin with extensions to previous work on CLgen[2]:

- adding support for alternate vocabularies, to enable learning models at the token or AST-level, as opposed to the current character-based approach;
- adding support for alternative encodings, enabling additional corpus transformations, for example, by reversing the character sequence, or interleaving characters from the start and end of sequences.
- implementing an iterative hill climb search of the program feature space using mutations to model seed text and random number generator state;

The goal of these extensions is incremental improvements to my current work on deep learning program synthesis (Appendix C), by demonstrating the ability to enumerate feature spaces, and reducing the rejection rate of synthesised programs. The deliverable for this work would be either an extended journal publication of Appendix C, or a new publication *"Directed exploration of program feature spaces"*. This outcome would depend on the extend to which these changes improve CLgen. At a minimum, the rejection rate would be measurably decreased so that programs can be generated a faster rate, and an iterative process of sample rejection would be shown to produce programs which converge towards specific feature values.

Upon completion of these first goals by the end of February 2017, a short period of time will be dedicated to analysis of the language models learned by CLgen. By modelling the syntax and semantics of source codes in a given language, it may be possible to extract the learned grammar of the language from the model's intermediate layers. The ability to automatically generate a formal grammar for a program language would have benefits for program synthesis, allowing the development of a grammar based system which operates on well-formed syntax trees, replacing the need for rejection tests to validate that a synthesised program is syntactically and semantically correct.

Should this brief exploratory analysis fail at extracting formal language grammars from learned models, then an alternative approach to reducing the rejection rate of generated programs would be the development of a system to iteratively and incrementally repair issues with rejected programs. This would extend the functionality of existing systems for statically verifying correctness of programs, allowing generated programs with small syntactic and semantic errors to be recovered.

The latter 3 months of the year will be used to test improvements on corpus generation. This will involve an empirical evaluation of the program rejection rate when sampling a model trained on previously generated programs. If the performance of the system improves under the these conditions, then it will allow for training language models on incrementally larger corpuses, and provide a mechanism for a program generator which is self-improving as the number of

[2] `http://chriscummins.cc/clgen/`

programs it generates increases. The qualitative evaluation of synthesised programs described in Appendix C will be repeated for models trained on generated programs.

Year 3 The first three months of my third year will be used to develop a formal grammar based approach to automatic program synthesis. Given a corpus of example programs in a given programming language, and a tool which verifies or rejects a program in the given language, this approach would instantiate the grammar probabilistically, using a language model trained on the corpus to determine the probability of a given production rule based on its probability distribution within the corpus.

By operating only on sequences of well-formed syntax trees, a grammar based approach has the potential to significantly improve the rate of program generation by pruning the space of possible output sequences to only those which produce correct programs. This would allow more strict checks to be imposed on generated programs, allowing focus on generating programs which are free from undefined behaviour.

The successul development of this system will be validated in a set of experiments to generate programs in three programming languages: OpenCL, C, and Java. Generated programs will be tested on multiple compilers, and the computed results of each compared. If the computed results differ and the programs are free from undefined behaviour, then a bug has been exposed in at least one of the disagreeing compilers. The intended deliverable for this work is a publication *"Differential testing compilers through deep learning"*.

Months 4 through 6 of year 3 will be used to develop an agent-based approach to compiler pass selection and ordering. This will involve extensions to the LLVM pass manager to support reinforcement learning, and use the automatic program generator developed previously to explore the program feature space. The implementation work for this project is minimal, with the majority of time required for running empirical evaluations of the system on benchmark suites. The second half of the year will be dedicated to thesis write up.

5 Conclusion

The focus of my research is on the development of systems for automatically generating programs in a given programming language. Such systems have large potential impact on the fields of predictive modelling for compiler optimisations and compiler testing. In my first year I have successfully demonstrated and published an initial proof-of-concept of a new approach to program generation using deep learning, which generates an unbounded number of programs that are indistinguishable from hand written code. Extending this work to generate programs in arbitrary languages and exhibiting arbitrary properties of interest will greatly improve the utility of this system, and enable improvements to compiler optimisations which are not possible using existing state-of-the-art program generators or hand written programs.

References

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

[All+14] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. "Learning Natural Coding Conventions". In: *FSE*. 2014, pages 281–293.

[APS16] M. Allamanis, H. Peng, and C. Sutton. "A Convolutional Attention Network for Extreme Summarization of Source Code". In: *arXiv:1602.03001* (2016).

[AS13] M. Allamanis and C. Sutton. "Mining Source Code Repositories at Massive Scale using Language Modeling". In: *MSR*. 2013, pages 207–216.

[Ans+09] A Ansel, C Chan, Y. L Wong, M Olszewski, Q Zhao, A Edelman, and S Amarasinghe. "PetaBricks: A Language and Compiler for Algorithmic Choice". In: *PLDI*. New York, NY, USA: ACM, 2009. URL: http://doi.acm.org/10.1145/1542476.1542481.

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

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

[Ash+15] A. Ashari, M. Boehm, B. Reinwald, and K. Campbell. "On Optimizing Machine Learning Workloads via Kernel Fusion". In: *PPoPP*. 2015, pages 173–182.

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

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. "Dynamo: A Transparent Dynamic Optimization System". In: *PLDI*. New York, NY, USA: ACM, 2000. URL: http://doi.acm.org/10.1145/349299.349303.

[Ber+08] L. Berkeley, K. Datta, M. Murphy, V. Volkov, S. Williams, and J. Carter. "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures". In: *SC*. 2008, page 4.

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

[Bol+16] T. Bolukbasi, K. Chang, J. Zou, V. Saligrama, and A. Kalai. “Man is to Computer Programmer as Woman is to Homemaker? Debiasing Word Embeddings”. In: *arXiv:1607.06520* (2016).

[Bol+03] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schroder. “Sparse matrix solvers on the GPU: conjugate gradients and multigrid”. In: *TOG* 22.3 (2003), pages 917–924.

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

[Bur+13] E. K. Burke, M. Gendreau, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, and R. Qu. “Hyper-heuristics: a survey of the state of the art”. In: *JORS* 64 (2013).

[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).

[Cha+09] C Chan, H Ansel, Y. L Wong, S Amarasinghe, and A Edelman. “Autotuning multigrid with PetaBricks”. In: *SC*. New York, New York, USA: ACM Press, 2009. URL: http://dl.acm.org/citation.cfm?doid=1654059.1654065.

[CW14] Guoyang Chen and Bo Wu. “PORPLE: An Extensible Optimizer for Portable Data Placement on GPU”. In: *MICRO*. IEEE, 2014, pages 88–100.

[Che+16] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao. “Coverage-Directed Differential Testing of JVM Implementations”. In: *PLDI*. 2016.

[CGA15] A. Chiu, J. Garvey, and T. S. Abdelrahman. “Genesis: A Language for Generating Synthetic Training Programs for Machine Learning”. In: *CF*. ACM, 2015, page 8.

[CSB11] Matthias Christen, Olaf Schenk, and Helmar Burkhart. “PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures”. In: *PDPS*. IEEE, May 2011, pages 676–687.

[CFL12] A. Collins, C. Fensch, and H. Leather. “Auto-Tuning Parallel Skeletons”. In: *Parallel Processing Letters* 22.02 (June 2012), page 1240005.

[Col+13] A. Collins, C. Fensch, H. Leather, and M. Cole. “MaSiF: Machine Learning Guided Auto-tuning of Parallel Skeletons”. In: *HiPC* (Dec. 2013), pages 186–195.

[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.

[DK15] U. Dastgeer and C. Kessler. “Smart Containers and Skeleton Programming for GPU-Based Systems”. In: *IJPP* (2015), pages 1–25.

[DEK11] Usman Dastgeer, Johan Enmyren, and Christoph W Kessler. "Auto-tuning SkePU: a multi-backend skeleton programming framework for multi-GPU systems". In: *IWMSE*. ACM, 2011, pages 25–32.

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

[Fur+11] G. Fursin, Y. Kashnikov, A. W. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois, F. Bodin, P. Barnard, E. Ashton, E. Bonilla, J. Thomson, C. K. I. Williams, and M. O'Boyle. "Milepost GCC: Machine Learning Enabled Self-tuning Compiler". In: *IJPP* 39.3 (2011), pages 296–327.

[Fur+14] G. Fursin, R. Miceli, A. Lokhmotov, M. Gerndt, M. Baboulin, A. D. Malony, Z. Chamski, D. Novillo, and D. Del Vento. "Collective Mind: Towards practical and collaborative auto-tuning". In: *Scientific Programming* 22.4 (2014), pages 309–329.

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

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

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

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

[Gol16] M. Goli. "VisionCPP: A SYCL-based Computer Vision Framework". In: *IWOCL*. 2016.

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

[GS18] A. Graves and J. Schmidhuber. "Framewise Phoneme Classification with Bidirectional LSTM and Other Neural Network Architectures". In: *Neural Networks* 5.5 (18), pages 602–610.

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

[Gu+16] X. Gu, H. Zhang, D. Zhang, and S. Kim. "Deep API Learning". In: *arXiv:1605.08535* (2016).

[GAL14] E. Guzman, D. Azócar, and Y. Li. "Sentiment Analysis of Commit Comments in GitHub: an Empirical Study". In: *MSR*. 2014, pages 352–355.

[HPS12] Justin Holewinski, Louis-Noël Pouchet, and P Sadayappan. "High-performance Code Generation for Stencil Computations on GPU Architectures". In: *SC*. 2012, pages 311–320. URL: http://doi.acm.org/10.1145/2304576.2304619.

[Jad+16] M. Jaderberg, W. M. Czarnecki, S. Osindero, O. Vinyals, A. Graves, and K. Kavukcuoglu. "Decoupled Neural Interfaces using Synthetic Gradients". In: *arXiv:1608.05343* (2016).

[Jia+14] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. "Caffe: Convolutional Architecture for Fast Feature Embedding". In: *ACMMM*. 2014.

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

[JNR02] Rajeev Joshi, Greg Nelson, and Keith Randall. "Denali: a goal-directed superoptimizer". In: *PLDI*. Volume 37. 5. ACM, 2002, page 304.

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

[Kam+10] S. Kamil, C. Chan, L. Oliker, J. Shall, and S. Williams. "An auto-tuning framework for parallel multicore stencil computations". In: *IPDPS* (2010).

[Khr15] Khronos OpenCL Group Inc. *SYCL Specification Vesion 1.2*. Technical report. 2015.

[Khr14] Khronos OpenCL Group Inc. *The SPIR Specification Version 1.2*. Technical report. 2014.

[LBH15] Y. LeCun, Y. Bengio, and G. Hinton. "Deep learning". In: *Nature* 521.7553 (2015), pages 436–444.

[Lee+14] Hyoukjoong Lee, Kevin J Brown, Arvind K Sujeeth, Tiark Rompf, and Kunle Olukotun. "Locality-Aware Mapping of Nested Parallel Patterns on GPUs". In: *MICRO*. IEEE, 2014, pages 63–74.

[Lee+10] Victor W. Lee, Per Hammarlund, Ronak Singhal, Pradeep Dubey, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, and Srinivas Chennupaty. "Debunking the 100X GPU vs. CPU myth". In: *ACM SIGARCH Computer Architecture News* 38 (2010), page 451.

[Lid+15] C. Lidbury, A. Lascu, N. Chong, and A. Donaldson. "Many-Core Compiler Fuzzing". In: *PLDI*. 2015, pages 65–76.

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

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

[LFC13] Thibaut Lutz, Christian Fensch, and Murray Cole. “PARTANS: An Autotuning Framework for Stencil Computation on Multi-GPU Systems”. In: *TACO* 9.4 (2013), page 59.

[MDO14] A. Magni, C. Dubach, and M. O’Boyle. “Automatic Optimization of Thread-Coarsening for Graphics Processors”. In: *PACT*. ACM, 2014, pages 455–466.

[Mas87] Henry Massalin. *Superoptimizer – A Look at the Smallest Program*. 1987.

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

[Mik10] Tomas Mikolov. “Recurrent Neural Network based Language Model”. In: *Interspeech*. 2010.

[MS10] J. Misra and I. Saha. “Artificial neural networks in hardware: A survey of two decades of progress”. In: *Neurocomputing* 74.1-3 (2010), pages 239–255.

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

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

[Ogi+15] W. F. Ogilvie, P. Petoumenos, Z. Wang, and H. Leather. “Intelligent Heuristic Construction with Active Learning”. In: *CPC*. 2015.

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

[Pot+15] R. Potter, P. Keir, R.J. Bradford, and A. Murray. “Kernel composition in SYCL”. In: *IWOCL*. 2015.

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

[RVK15] Veselin Raychev, Martin Vechev, and Andreas Krause. “Predicting Program Properties from “Big Code””. In: *POPL*. 2015.

[Reg+12] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. “Test-case reduction for C compiler bugs”. In: *PLDI*. 2012, pages 335–346.

[Ryo+08a] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. “Optimization principles and application performance evaluation of a multithreaded GPU using CUDA”. In: *PPoPP* (2008), page 73.

[Ryo+08b] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, Sara S. Baghsorkhi, Sain-Zee Ueng, John a. Stratton, and Wen-mei W. Hwu. "Program optimization space pruning for a multithreaded GPU". In: *CGO*. New York, New York, USA: IEEE, 2008, pages 195–204. URL: http://portal.acm.org/citation.cfm?doid=1356058.1356084.

[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.

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

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

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

[Tru+16] L. Truong, R. Barik, E. Totoni, H. Liu, C. Markley, A. Fox, and T. Shpeisman. "Latte: a language, compiler, and runtime for elegant and efficient deep neural networks". In: *PLDI*. 2016.

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

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

[VDP09] Y. Voronenko, F. De Mesmay, and M. Püschel. "Computer Generation of General Size Linear Transform Libraries". In: *CGO*. IEEE, 2009, pages 102–113.

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

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

[WYT13] E. Wong, J. Yang, and L. Tan. "AutoComment: Mining Question and Answer Sites for Automatic Comment Generation". In: *ASE*. IEEE, 2013, pages 562–567.

[Wu+14] Y. Wu, J. Kropczynski, P. C. Shih, and J. M. Carroll. "Exploring the Ecosystem of Software Developers on GitHub and Other Platforms". In: *CSCW*. 2014, pages 265–268.

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

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

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

[ZM12] Yongpeng Zhang and Frank Mueller. “Auto-generation and Auto-tuning of 3D Stencil Codes on GPU clusters”. In: *CGO*. IEEE, 2012, pages 155–164.

Autotuning OpenCL Workgroup Size for Stencil Patterns

Chris Cummins Pavlos Petoumenos Michel Steuwer Hugh Leather
University of Edinburgh
c.cummins@ed.ac.uk, ppetoume@inf.ed.ac.uk, michel.steuwer@ed.ac.uk, hleather@inf.ed.ac.uk

Abstract

Selecting an appropriate workgroup size is critical for the performance of OpenCL kernels, and requires knowledge of the underlying hardware, the data being operated on, and the implementation of the kernel. This makes portable performance of OpenCL programs a challenging goal, since simple heuristics and statically chosen values fail to exploit the available performance. To address this, we propose the use of machine learning-enabled autotuning to automatically predict workgroup sizes for stencil patterns on CPUs and multi-GPUs.

We present three methodologies for predicting workgroup sizes. The first, using classifiers to select the optimal workgroup size. The second and third proposed methodologies employ the novel use of regressors for performing classification by predicting the runtime of kernels and the relative performance of different workgroup sizes, respectively. We evaluate the effectiveness of each technique in an empirical study of 429 combinations of architecture, kernel, and dataset, comparing an average of 629 different workgroup sizes for each. We find that autotuning provides a median 3.79× speedup over the best possible fixed workgroup size, achieving 94% of the maximum performance.

1. Introduction

Stencil codes have a variety of computationally demanding uses from fluid dynamics to quantum mechanics. Efficient, tuned stencil implementations are highly sought after, with early work in 2003 by Bolz et al. demonstrating the capability of GPUs for massively parallel stencil operations [1]. Since then, the introduction of the OpenCL standard has introduced greater programmability of heterogeneous devices by providing a vendor-independent layer of abstraction for data parallel programming of CPUs, GPUs, DSPs, and other devices [2]. However, achieving portable performance of OpenCL programs is a hard task — OpenCL kernels are sensitive to properties of the underlying hardware, to the implementation, and even to the *dataset* that is operated upon. This forces developers to laboriously hand tune performance on a case-by-case basis, since simple heuristics fail to exploit the available performance.

In this paper, we demonstrate how machine learning-enabled autotuning can address this issue for one such optimisation parameter of OpenCL programs — that of workgroup size. The 2D optimisation space of OpenCL kernel workgroup sizes is complex and non-linear, making it resistant to analytical modelling. Successfully applying machine learning to such a space requires plentiful training data, the careful selection of features, and an appropriate classification approach. The approaches presented in this paper use features extracted from the architecture and kernel, and training data collected from synthetic benchmarks to predict workgroup sizes for unseen programs.

2. The SkelCL Stencil Pattern

Introduced in [3], SkelCL is an Algorithmic Skeleton library which provides OpenCL implementations of data parallel patterns for heterogeneous parallelism using CPUs and multi-GPUs. Figure 1 shows the components of the SkelCL stencil pattern, which applies a user-provided *customising function* to each element of a 2D matrix. The value of each element is updated based on its current value and the value of one or more neighbouring elements, called the *border region*. The border region describes a rectangular region about each cell, and is defined in terms of the number of cells in the border region to the north, east, south, and west of each cell. Where elements of a border region fall outside of the matrix bounds, values are substituted from either a predefined padding value, or the value of the nearest cell within the matrix, determined by the user.

When a SkelCL stencil pattern is executed, each of the matrix elements are mapped to OpenCL work-items; and this collection of work-items is divided into *workgroups* for execution on the target hardware. A work-item reads the value of its corresponding matrix element and the surrounding elements defined by the border region. Since the border regions of neighbouring elements overlap, each element in the matrix is read multiple times. Because of this, a *tile* of elements of the size of the workgroup and the perimeter border region is allocated as a contiguous block in local memory. This greatly reduces the latency of repeated memory accesses

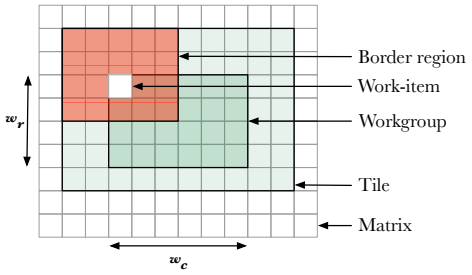


Figure 1: The components of a stencil: an input matrix is decomposed into workgroups, consisting of $w_r \times w_c$ elements. Each element is mapped to a work-item. Each work-item operates on its corresponding element and a surrounding border region (in this example, 1 element to the south, and 2 elements to the north, east, and west).

performed by the work-items. As a result, changing the workgroup size affects both the number of workgroups which can be active simultaneously, and the amount of local memory required for each workgroup. While the user defines the size, type, and border region of the matrix being operated upon, it is the responsibility of the SkelCL stencil implementation to select an appropriate workgroup size to use.

3. Autotuning Workgroup Size

Selecting the appropriate workgroup size for an OpenCL kernel depends on the properties of the kernel itself, underlying architecture, and dataset. For a given *scenario* (that is, a combination of kernel, architecture, and dataset), the goal of this work is to harness machine learning to *predict* a performant workgroup size to use, based on some prior knowledge of the performance of workgroup sizes for other scenarios. In this section, we describe the optimisation space and the steps required to apply machine learning. The autotuning algorithms are described in Section 4.

3.1 Constraints

The space of possible workgroup sizes W is constrained by properties of both the architecture and kernel. Each OpenCL device imposes a maximum workgroup size which can be statically checked through the OpenCL Device API. This constraint reflects architectural limitations of how code is mapped to the underlying execution hardware. Typical values are powers of two, e.g. 1024, 4096, 8192. Additionally, the OpenCL runtime enforces a maximum workgroup size on a per-kernel basis. This value can be queried at runtime once a program has been compiled for a specific execution device. Factors which affect a kernel's maximum workgroup size include the number of registers required, and the available number of SIMD execution units for each type of executable instruction.

While in theory, any workgroup size which satisfies the device and kernel workgroup size constraints should provide a valid program, in practice we find that some combinations of scenario and workgroup size cause a `CL_OUT_OF_RESOURCES` error to be thrown when the kernel is launched. We refer to these workgroup sizes as *refused parameters*. Note that in many OpenCL implementations, this error type acts as a generic placeholder and may not necessarily indicate that the underlying cause of the error was due to finite resources constraints. We define the space of *legal* workgroup sizes for a given scenario s as those which satisfy the architectural and kernel constraints, and are not refused:

$$W_{legal}(s) = \{w | w \in W, w < W_{\max}(s)\} - W_{refused}(s) \quad (1)$$

Where $W_{\max}(s)$ can be determined at runtime prior to the kernels execution, but the set $W_{refused}(s)$ can only be discovered emergently. The set of *safe* parameters are those which are legal for all scenarios:

$$W_{safe} = \cap \{W_{legal}(s) | s \in S\} \quad (2)$$

3.2 Stencil and Architectural Features

Since properties of the architecture, program, and dataset all contribute to the performance of a workgroup size for a particular scenario, the success of a machine learning system depends on the ability to translate these properties into meaningful explanatory variables — *features*. For each scenario, 102 features are extracted describing the architecture, kernel, and dataset.

Architecture features are extracted using the OpenCL Device API to query properties such as the size of local memory, maximum work group size, and number of compute units. Kernel features are extracted from the source code stencil kernels by compiling first to LLVM IR bitcode, and using statistics passes to obtain static instruction counts for each type of instruction present in the kernel, as well as the total number of instructions. These instruction counts are divided by the total number of instructions to produce instruction *densities*. Dataset features include the input and output data types, and the 2D matrix dimensions.

3.3 Training Data

Training data is collected by measuring the runtimes of stencil programs using different workgroup sizes. These stencil programs are generated synthetically using a statistical template substitution engine, which allows a larger exploration of the program space than is possible using solely hand-written benchmarks. A stencil template is parameterised first by stencil shape (one parameter for each of the four directions), input and

Algorithm 1 Prediction using classifiers

Require: scenario s
Ensure: workgroup size w

procedure BASELINE(s)
  $w \leftarrow \text{classify}(f(s))$
  if $w \in W_{legal}(s)$ **then**
    return w
  else
    return $\arg\max_{w \in W_{safe}} \left(\prod_{s \in S_{training}} p(s,w) \right)^{1/|S_{training}|}$
  end if
end procedure

procedure RANDOM(s)
  $w \leftarrow \text{classify}(f(s))$
  while $w \notin W_{legal}(s)$ **do**
    $W \leftarrow \left\{ w | w < W_{max}(s), w \notin W_{refused}(s) \right\}$
    $w \leftarrow$ random selection $w \in W$
  end while
  return w
end procedure

procedure NEARESTNEIGHBOUR(s)
  $w \leftarrow \text{classify}(f(s))$
  while $w \notin W_{legal}(s)$ **do**
    $d_{min} \leftarrow \infty$
    $w_{closest} \leftarrow$ null
    for $c \in \left\{ w | w < W_{\max}(s), w \notin W_{refused}(s) \right\}$ **do**
      $d \leftarrow \sqrt{(c_r - w_r)^2 + (c_c - w_c)^2}$
      if $d < d_{min}$ **then**
        $d_{min} \leftarrow d$
        $w_{closest} \leftarrow c$
      end if
    end for
    $w \leftarrow w_{closest}$
  end while
  return w
end procedure

output data types (either integers, or single or double precision floating points), and *complexity* — a simple boolean metric for indicating the desired number of memory accesses and instructions per iteration, reflecting the relatively bi-modal nature of stencil codes, either compute intensive (e.g. finite difference time domain and other PDE solvers), or lightweight (e.g. Game of Life and Gaussian blur).

4. Machine Learning Methods

The aim of this work is to design a system which predicts performant workgroup sizes for *unseen* scenarios, given a set of prior performance observations. This section presents three contrasting methods for achieving this goal.

4.1 Predicting Oracle Workgroup Sizes

The first approach is detailed in Algorithm 1. By considering the set of possible workgroup sizes as a hypothesis space, we train a classifier to predict, for a given set of features, the *oracle* workgroup size. The oracle workgroup size $\Omega(s)$ is the workgroup size which provides the lowest mean runtime $t(s,w)$ for a scenario s:

$$\Omega(s) = \underset{w \in W_{legal}(s)}{\arg\min} \; t(s,w) \tag{3}$$

Training a classifier for this purpose requires pairs of stencil features $f(s)$ to be labelled with their oracle workgroup size for a set of training scenarios $S_{training}$:

$$D_{training} = \{ (f(s), \Omega(s)) \, | s \in S_{training} \} \tag{4}$$

After training, the classifier predicts workgroup sizes for unseen scenarios from the set of oracle workgroup sizes from the training set. This is a common and intuitive approach to autotuning, in that a classifier predicts the best parameter value based on what worked well for the training data. However, given the constrained space of workgroup sizes, this presents the problem that future scenarios may have different sets of legal workgroup sizes to that of the training data, i.e.:

$$\bigcup_{\forall s \in S_{future}} W_{legal}(s) \not\subseteq \{ \Omega(s) | s \in S_{training} \} \tag{5}$$

This results in an autotuner which may predict workgroup sizes that are not legal for all scenarios, either because they exceed $W_{\max}(s)$, or because parameters are refused, $w \in W_{refused}(s)$. For these cases, we evaluate the effectiveness of three *fallback handlers*, which will iteratively select new workgroup sizes until a legal one is found:

1. *Baseline* — select the workgroup size which provides the highest average case performance from the set of safe workgroup sizes.
2. *Random* — select a random workgroup size which is expected from prior observations to be legal.
3. *Nearest Neighbour* — select the workgroup size which from prior observations is expected to be legal, and has the lowest Euclidian distance to the prediction.

Algorithm 2 Prediction using regressors

Require: scenario s, regressor $R(x,w)$, fitness function $\Delta(x)$
Ensure: workgroup size w

$W \leftarrow \{ w | w < W_{\max}(s) \} - W_{refused}(s)$ ▷ Candidates.
$w \leftarrow \underset{w \in W}{\arg\max} \Delta(R(f(s), w))$ ▷ Select best candidate.
while $w \notin W_{legal}(s)$ **do**
  $W_{refused}(s) = W_{refused}(s) + \{w\}$
  $W \leftarrow W - \{w\}$ ▷ Remove candidate from selection.
  $w \leftarrow \underset{w \in W}{\arg\max} \Delta(R(f(s), w))$ ▷ Select best candidate.
end while
return w

4.2 Predicting Kernel Runtimes

A problem of predicting oracle workgroup sizes is that, for each training instance, an exhaustive search of the

optimisation space must be performed in order to find the oracle workgroup size. An alternative approach is to instead predict the expected *runtime* of a kernel given a specific workgroup size. Given training data consisting of $(f(s), w, t)$ tuples, where $f(s)$ are scenario features, w is the workgroup size, and t is the observed runtime, we train a regressor $R(f(s), w)$ to predict the runtime of scenario and workgroup size combinations. The selected workgroup size $\bar{\Omega}(s)$ is then the workgroup size from a pool of candidates which minimises the output of the regressor. Algorithm 2 formalises this approach of autotuning with regressors. A fitness function $\Delta(x)$ computes the reciprocal of the predicted runtime so as to favour shorter over longer runtimes. Note that the algorithm is self correcting in the presence of refused parameters — if a workgroup size is refused, it is removed from the candidate pool, and the next best candidate is chosen. This removes the need for fallback handlers. Importantly, this technique allows for training on data for which the oracle workgroup size is unknown, meaning that a full exploration of the space is not required in order to gather a training instance, as is the case with classifiers.

4.3 Predicting Relative Performance

Accurately predicting the runtime of arbitrary code is a difficult problem. It may instead be more effective to predict the relative performance of two different workgroup sizes for the same kernel. To do this, we predict the *speedup* of a workgroup size over a baseline. This baseline is the workgroup size which provides the best average case performance across all scenarios and is known to be safe. Such a baseline value represents the *best* possible performance which can be achieved using a single, fixed workgroup size. As when predicting runtimes, this approach performs classification using regressors (Algorithm 2). We train a regressor $R(f(s), w)$ to predict the relative performance of workgroup size w over a baseline parameter for scenario s. The fitness function returns the output of the regressor, so the selected workgroup size $\bar{\Omega}(s)$ is the workgroup size from a pool of candidates which is predicted to provide the best relative performance. This has the same advantageous properties as predicting runtimes, but by training using relative performance, we negate the challenges of predicting dynamic code behaviour.

5. Experimental Setup

To evaluate the performance of the presented autotuning techniques, an exhaustive enumeration of the workgroup size optimisation space for 429 combinations of architecture, program, and dataset was performed.

Table 1 describes the experimental platforms and OpenCL devices used. Each platform was unloaded, frequency governors disabled, and benchmark processes set to the highest priority available to the task scheduler. Datasets and programs were stored in an in-memory file system. All runtimes were recorded with millisecond precision using OpenCL's Profiling API to record the kernel execution time. The workgroup size space was enumerated for each combination of w_r and w_c values in multiples of 2, up to the maximum workgroup size. For each combination of scenario and workgroup size, a minimum of 30 runtimes were recorded.

In addition to the synthetic stencil benchmarks described in Section 3.3, six stencil kernels taken from four reference implementations of standard stencil applications from the fields of image processing, cellular automata, and partial differential equation solvers are used: Canny Edge Detection, Conway's Game of Life, Heat Equation, and Gaussian Blur. Table 2 shows details of the stencil kernels for these reference applications and the synthetic training benchmarks used. Dataset sizes of size 512×512, 1024×1024, 2048×2048, and 4096×4096 were used.

Program behavior is validated by comparing program output against a gold standard output collected by executing each of the real-world benchmarks programs using the baseline workgroup size. The output of real-world benchmarks with other workgroup sizes is compared to this gold standard output to test for correct program execution.

Five different classification algorithms are used to predict oracle workgroup sizes, chosen for their contrasting properties: Naive Bayes, SMO, Logistic Regression, J48 Decision tree, and Random Forest [4]. For regression, a Random Forest with regression trees is used, chosen because of its efficient handling of large feature sets compared to linear models [5]. The autotuning system is implemented in Python as a system daemon. SkelCL stencil programs request workgroup sizes from this daemon, which performs feature extraction and classification.

6. Performance Results

This section describes the performance results of enumerating the workgroup size optimisation space. The effectiveness of autotuning techniques for exploiting this space are examined in Section 7. The experimental results consist of measured runtimes for a set of *test cases*, where a test case τ_i consists of a scenario, workgroup size pair $\tau_i = (s_i, w_i)$, and is associated with a *sample* of observed runtimes of the program. A total of 269813 test cases were evaluated, which represents an exhaustive enumeration of the workgroup size optimisation space for 429 scenarios. For each scenario, runtimes for an average of 629 (max 7260) unique workgroup sizes were

Host	Host Memory	OpenCL Device	Compute units	Frequency	Local Memory	Global Cache	Global Memory
Intel i5-2430M	8 GB	CPU	4	2400 Hz	32 KB	256 KB	7937 MB
Intel i5-4570	8 GB	CPU	4	3200 Hz	32 KB	256 KB	7901 MB
Intel i7-3820	8 GB	CPU	8	1200 Hz	32 KB	256 KB	7944 MB
Intel i7-3820	8 GB	AMD Tahiti 7970	32	1000 Hz	32 KB	16 KB	2959 MB
Intel i7-3820	8 GB	Nvidia GTX 590	1	1215 Hz	48 KB	256 KB	1536 MB
Intel i7-2600K	16 GB	Nvidia GTX 690	8	1019 Hz	48 KB	128 KB	2048 MB
Intel i7-2600	8 GB	Nvidia GTX TITAN	14	980 Hz	48 KB	224 KB	6144 MB

Table 1: Specification of experimental platforms and OpenCL devices.

Name	North	South	East	West	Instruction Count
synthetic-a	1–30	1–30	1–30	1–30	67–137
synthetic-b	1–30	1–30	1–30	1–30	592–706
gaussian	1–10	1–10	1–10	1–10	82–83
gol	1	1	1	1	190
he	1	1	1	1	113
nms	1	1	1	1	224
sobel	1	1	1	1	246
threshold	0	0	0	0	46

Table 2: Stencil kernels, border sizes (north, south, east, and west), and static instruction counts.

measured. The average sample size for each test case is 83 (min 33, total 16917118).

The workgroup size optimisation space is non-linear and complex, as shown in Figure 2, which plots the distribution of optimal workgroup sizes. Across the 429 scenarios, there are 135 distinct optimal workgroup sizes (31.5%). The average speedup of the oracle workgroup size over the worst workgroup size for each scenario is 15.14× (min 1.03×, max 207.72×).

Of the 8504 unique workgroup sizes tested, 11.4% were refused in one or more test cases, with an average of 5.5% test cases leading to refused parameters. There are certain patterns to the refused parameters: for example, workgroup sizes which contain w_c and w_r values which are multiples of eight are less frequently refused, since eight is a common width of SIMD vector operations [6]. However, a refused parameter is an obvious inconvenience to the user, as one would expect that any workgroup size within the specified maximum should generate a working program, if not a performant one.

Experimental results suggest that the problem is vendor — or at least device — specific. Figure 3 shows the ratio of refused test cases, grouped by device. We see many more refused parameters for test cases on Intel CPU devices than any other type, while no workgroup sizes were refused by the AMD GPU. The exact underlying cause for these refused parameters is unknown, but can likely by explained by inconsistencies or errors in specific OpenCL driver implementations. Note that the ratio of refused parameters decreases across the three generations of Nvidia GPUs: GTX 590 (2011), GTX 690 (2012), and GTX TITAN (2013). For now, it is imperative that any autotuning system is capable of

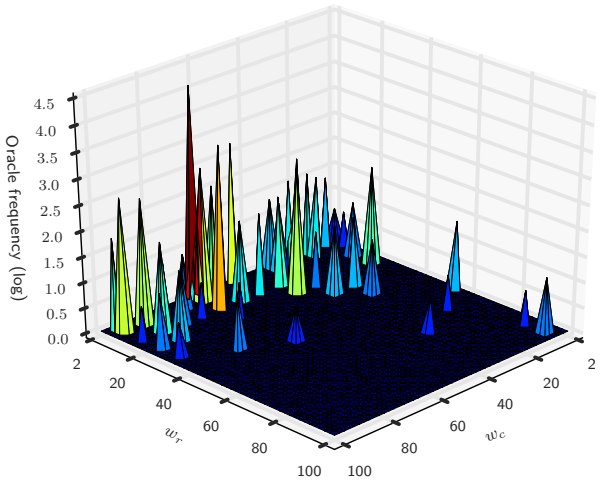


Figure 2: Oracle frequency counts for a subset of the workgroup sizes, $w_c \leq 100, w_r \leq 100$. There are 135 unique oracle workgroup sizes. The most common oracle workgroup size is $w_{(64\times4)}$, optimal for 15% of scenarios.

adapting to these refused parameters by suggesting alternatives when they occur.

The baseline parameter is the workgroup size providing the best overall performance while being legal for all scenarios. Because of refused parameters, only a *single* workgroup size $w_{(4\times4)}$ from the set of experimental results is found to have a legality of 100%, suggesting that an adaptive approach to setting workgroup size is necessary not just for the sake of maximising performance, but also for guaranteeing program execution. The utility of the baseline parameter is that it represents the best performance that can be achieved through static tuning of the workgroup size parameter; however, compared to the oracle workgroup size for each scenario, the baseline parameter achieves only 24% of the optimal performance.

7. Evaluation of Autotuning Methods

In this section we evaluate the effectiveness of the three proposed autotuning techniques for predicting performant workgroup sizes. For each autotuning technique, we partition the experimental data into training and testing sets. Three strategies for partitioning the data are used: the first is a 10-fold cross-validation; the sec-

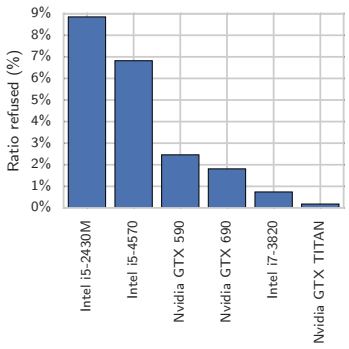


Figure 3: The ratio of test cases with refused workgroup sizes, grouped by OpenCL device ID. No parameters were refused by the AMD device.

ond is to divide the data such that only data collected from synthetic benchmarks are used for training and only data collected from the real-world benchmarks are used for testing; the third strategy is to create leave-one-out partitions for each unique device, kernel, and dataset. For each combination of autotuning technique and testing dataset, we evaluate each of the workgroup sizes predicted for the testing data using the following metrics:

- time (real) — the time taken to make the autotuning prediction. This includes classification time and any communication overheads.
- accuracy (binary) — whether the predicted workgroup size is the true oracle, $w = \Omega(s)$.
- validity (binary) — whether the predicted workgroup size satisfies the workgroup size constraints constraints, $w < W_{\max}(s)$.
- refused (binary) — whether the predicted workgroup size is refused, $w \in W_{refused}(s)$.
- performance (real) — the performance of the predicted workgroup size relative to the oracle for that scenario.
- speedup (real) — the relative performance of the predicted workgroup size relative to the baseline workgroup size $w_{(4\times4)}$.

The *validty* and *refused* metrics measure how often fallback strategies are required to select a legal workgroup size $w \in W_{legal}(s)$. This is only required for the classification approach to autotuning, since the process of selecting workgroup sizes using regressors respects workgroup size constraints.

7.1 Predicting Oracle Workgroup Size

Figure 4 shows the results when classifiers are trained using data from synthetic benchmarks and tested using real-world benchmarks. With the exception of the ZeroR, a dummy classifier which "predicts" only the baseline workgroup size $w_{(4\times4)}$, the other classifiers achieve good speedups over the baseline, ranging from 4.61× to 5.05× when averaged across all test sets. The differences in speedups between classifiers is not significant, with the exception of SimpleLogistic, which performs poorly when trained with synthetic benchmarks and tested against real-world programs. This suggests the model over-fitting to features of the synthetic benchmarks which are not shared by the real-world tests. Of the three fallback handlers, NearestNeighbour provides the best performance, indicating that it successfully exploits structure in the optimisation space. In our evaluation, the largest number of iterations of a fallback handler required before selecting a legal workgroup size was 2.

7.2 Predicting with Regressors

Figure 5 shows a summary of results for autotuning using regressors to predict kernel runtimes (5a) and speedups (5b). Of the two regression techniques, predicting the *speedup* of workgroup sizes is much more successful than predicting the *runtime*. This is most likely caused by the inherent difficulty in predicting the runtime of arbitrary code, where dynamic factors such as flow control and loop bounds are not captured by the instruction counts which are used as features for the machine learning models. The average speedup achieved by predicting runtimes is 4.14×. For predicting speedups, the average is 5.57×, the highest of all of the autotuning techniques.

7.3 Autotuning Overheads

Comparing the classification times of Figures 4 and 5 shows that the prediction overhead of regressors is significantly greater than classifiers. This is because, while a classifier makes a single prediction, the number of predictions required of a regressor grows with the size of $W_{\max}(s)$, since classification with regression requires making predictions for all $w \in \{w | w < W_{\max}(s)\}$. The fastest classifier is J48, due to the it's simplicity — it can be implemented as a sequence of nested `if` and `else` statements.

7.4 Comparison with Human Expert

In the original implementation of the SkelCL stencil pattern [7], Steuwer et al. selected a workgroup size of $w_{(32\times4)}$ in an evaluation of 4 stencil operations on a Tesla S1070 system. In our evaluation of 429 combinations of kernel, architecture, and dataset, we found that this workgroup size is refused by 2.6% of scenarios, making it unsuitable for use as a baseline. However, if we remove the scenarios for which $w_{(32\times4)}$ is *not* a legal workgroup size, we can directly compare the performance against the autotuning predictions.

Figure 6 plots the distributions and Interquartile Range (IQR) of all speedups over the human expert

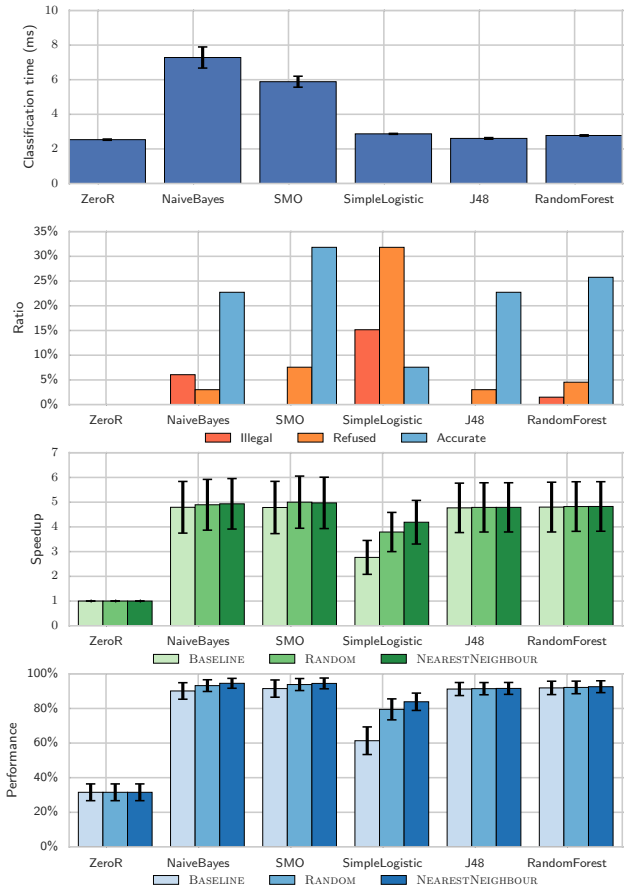


Figure 4: Autotuning performance using classifiers and synthetic benchmarks. Each classifier is trained on data collected from synthetic stencil applications, and tested for prediction quality using data from 6 real-world benchmarks. 95% confidence intervals are shown where appropriate.

parameter for each autotuning technique. The distributions show consistent classification results for the five classification techniques, with the speedup at Q1 for all classifiers being $\geq 1.0\times$. The IQR for all classifiers is < 0.5, but there are outliers with speedups both well below $1.0\times$ and well above $2.0\times$. In contrast, the speedups achieved using regressors to predict runtimes have a lower range, but also a lower median and a larger IQR. Clearly, this approach is the least effective of the evaluated autotuning techniques. Using regressors to predict relative performance is more successful, achieving the highest median speedup of all the techniques ($1.33\times$).

8. Related Work

Ganapathi et al. demonstrated early attempts at autotuning multicore stencil codes in [8], drawing upon the successes of statistical machine learning techniques in the compiler community. They use Kernel Canonical Correlation Analysis to build correlations between

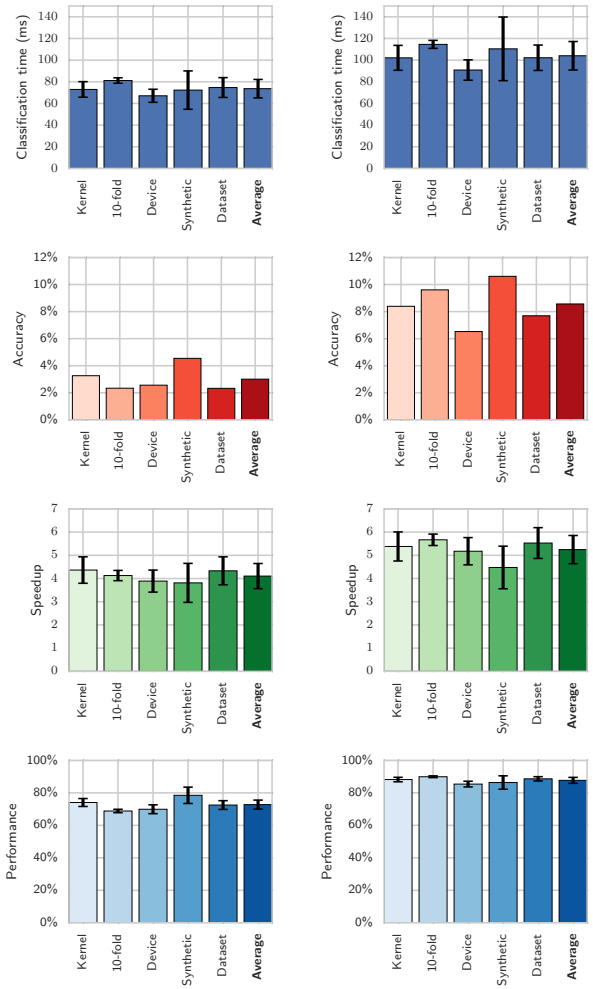


Figure 5: Autotuning performance for each type of test dataset using regressors to predict: (a) kernel runtimes, and (b) relative performance of workgroup sizes.

stencil features and optimisation parameters. Their use of KCCA restricts the scalability of their system, as the complexity of model building grows exponentially with the number of features. A code generator and autotuner for 3D Jacobi stencil codes is presented in [9], although their approach requires a full enumeration of the parameter space for each new program, and has no cross-program learning. Similarly, CLTune [10] is an autotuner which applies iterative search techniques to user-specified OpenCL parameters. The number of parallel mappers and reducers for MapReduce workloads is tuned in [11] using surrogate models rather than machine learning, although the optimisation space is not subject to the level of constraints that OpenCL workgroup size is. A generic OpenCL autotuner is presented in [12] which uses neural networks to predict good configurations of user-specified parameters, although the authors present only a preliminary eval-

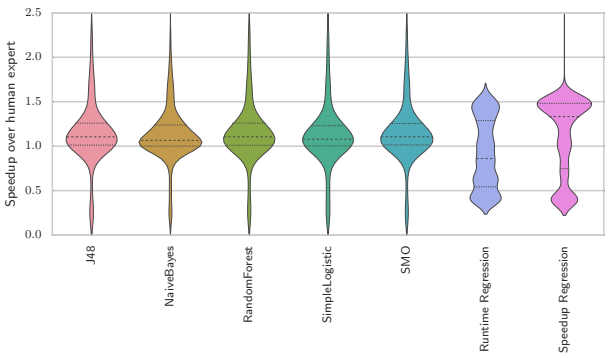


Figure 6: Violin plot of speedups over *human expert*, ignoring cases where the workgroup size selected by human experts is invalid. Classifiers are using NEAREST-NEIGHBOUR fallback handlers. Horizontal dashed lines show the median, Q1, and Q3. Kernel Density Estimates show the distribution of results. The speedup axis is fixed to the range 0–2.5 to highlight the IQRs, which results in some outlier speedups > 2.5 being clipped.

uation using three benchmarks. Both systems require the user to specify parameters on a per-program basis. The autotuner presented in this work, embedded at the skeletal level, requires no user effort for new programs and is transparent to the user. A DSL and CUDA code generator for stencils is presented in [13]. Unlike the SkelCL stencil pattern, the generated stencil codes do not exploit fast local device memory. The automatic generation of synthetic benchmarks using parameterised template substitution is presented in [14]. The authors describe an application of their tool for generating OpenCL stencil kernels for machine learning, but do not report any performance results.

9. Conclusions

We present and compare novel methodologies for autotuning the workgroup size of stencil patterns using the established open source library SkelCL. These techniques achieve up to 94% of the maximum performance, while providing robust fallbacks in the presence of unexpected behaviour in OpenCL driver implementations. Of the three techniques proposed, predicting the relative performances of workgroup sizes using regressors provides the highest median speedup, whilst predicting the oracle workgroup size using decision tree classifiers adds the lowest runtime overhead. This presents a trade-off between classification time and training time that could be explored in future work using a hybrid of the classifier and regressor techniques presented in this paper.

In future work, we will extend the autotuner to accommodate additional OpenCL optimisation parameters and skeleton patterns. Feature selection can be evaluated using Principle Component Analysis, as well exploring the relationship between prediction accuracy and the number of synthetic benchmarks used. A promising avenue for further research is in the transition towards online machine learning which is enabled by using regressors to predict kernel runtimes. This could be combined with the use of adaptive sampling plans to minimise the number of observations required to distinguish bad from good parameter values, such as presented in [15]. Dynamic profiling can be used to increase the prediction accuracy of kernel runtimes by capturing the runtime behaviour of stencil kernels.

Acknowledgments

This work was supported by the UK Engineering and Physical Sciences Research Council under grants EP/L01503X/1 for the University of Edinburgh School of Informatics Centre for Doctoral Training in Pervasive Parallelism (`http://pervasiveparallelism.inf.ed.ac.uk/`), EP/H044752/1 (ALEA), and EP/M015793/1 (DIVIDEND).

References

[1] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schroder. "Sparse matrix solvers on the GPU: conjugate gradients and multigrid". In: *ACM TOG* 22.3 (2003), pp. 917–924.

[2] John E. Stone, David Gohara, and Guochun Shi. "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems". In: *Computing in Science & Engineering* 12.3 (2010), pp. 66–73.

[3] Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. "SkelCL - A Portable Skeleton Library for High-Level GPU Programming". In: *IPDPSW*. IEEE, May 2011, pp. 1176–1182.

[4] Jiawei Han, Micheline Kamber, and Jian Pei. *Data mining: concepts and techniques*. Elsevier, 2011.

[5] Leo Breiman. "Random forest". In: *Machine Learning* 45.1 (2001), pp. 5–32.

[6] Intel Corporation. *OpenCL* Optimization Guide*. 2012. URL: `https://software.intel.com/sites/landingpage/opencl/optimization-guide/index.htm`.

[7] Michel Steuwer, Michael Haidl, Stefan Breuer, and Sergei Gorlatch. "High-level programming of stencil computations on multi-GPU systems using the SkelCL library". In: *Parallel Processing Letters* 24.03 (2014), p. 1441005.

[8] Archana Ganapathi, Kaushik Datta, Armando Fox, and David Patterson. "A Case for Machine Learning to Optimize Multicore Performance". In: *HotPar*. 2009.

[9] Yongpeng Zhang and Frank Mueller. "Auto-generation and Auto-tuning of 3D Stencil Codes on GPU clusters". In: *CGO*. 2012, pp. 155–164.

[10] Cedric Nugteren and Valeriu Codreanu. "CLTune: A Generic Auto-Tuner for OpenCL Kernels". In: *MCSoC*. IEEE, 2015, pp. 195–202.

[11] Travis Johnston, Mohammad Alsulmi, Pietro Cicotti, and Michela Taufer. "Performance Tuning of MapReduce Jobs Using Surrogate-Based Modeling". In: *ICCS* (2015), pp. 49–59.

[12] Thomas L. Falch and Anne C. Elster. "Machine Learning Based Auto-tuning for Enhanced OpenCL Performance Portability". In: *IPDPSW*. IEEE, 2015, pp. 3–8.

[13] Shoaib Kamil, Cy Chan, Leonid Oliker, John Shall, and Samuel Williams. "An auto-tuning framework for parallel multicore stencil computations". In: *IPDPS* (2010).

[14] Alton Chiu, Joseph Garvey, and Tarek S Abdelrahman. "Genesis: A Language for Generating Synthetic Training Programs for Machine Learning". In: *International Conference on Computing Frontiers*. ACM, 2015, p. 8.

[15] Hugh Leather, Michael O'Boyle, and Bruce Worton. "Raced Profiles: Efficient Selection of Competing Compiler Optimizations". In: *ACM Sigplan Notices* 44.7 (2009), pp. 50–59.

Towards Collaborative Performance Tuning of Algorithmic Skeletons

Chris Cummins Pavlos Petoumenos Michel Steuwer Hugh Leather
University of Edinburgh
c.cummins@ed.ac.uk, ppetoume@inf.ed.ac.uk, michel.steuwer@ed.ac.uk, hleather@inf.ed.ac.uk

Abstract

The physical limitations of microprocessor design have forced the industry towards increasingly heterogeneous designs to extract performance. This trend has not been matched with adequate software tools, leading to a growing disparity between the availability of parallelism and the ability for application developers to exploit it.

Algorithmic skeletons simplify parallel programming by providing high-level, reusable patterns of computation. Achieving performant skeleton implementations is a difficult task; skeleton authors must attempt to anticipate and tune for a wide range of architectures and use cases. This results in implementations that target the general case and cannot provide the performance advantages that are gained from tuning low level optimization parameters. Autotuning combined with machine learning offers promising performance benefits in these situations, but the high cost of training and lack of available tools limits the practicality of autotuning for real world programming. We believe that performing autotuning at the level of the skeleton library can overcome these issues.

In this work, we present OmniTune — an extensible and distributed framework for dynamic autotuning of optimization parameters at runtime. OmniTune uses a client-server model with a flexible API to support machine learning enabled autotuning. Training data is shared across a network of cooperating systems, using a collective approach to performance tuning.

We demonstrate the practicality of OmniTune in a case study using the algorithmic skeleton library SkelCL. By automatically tuning the workgroup size of OpenCL Stencil skeleton kernels, we show that that static tuning across a range of GPUs and programs can achieve only 26% of the optimal performance, while OmniTune achieves 92% of this maximum, equating to an average 5.65× speedup. OmniTune achieves this without introducing a significant runtime overhead, and enables portable, cross-device and cross-program tuning.

1. Introduction

General purpose programming with GPUs has been shown to provide huge parallel throughput, but poses a significant programming challenge, requiring application developers to master an unfamiliar programming model (such as provided by CUDA or OpenCL) and architecture (SIMD with a multi-level memory hierarchy). As a result, GPGPU programming is often considered beyond the realm of everyday development. If steps are not taken to increase the accessibility of such parallelism, the gap between potential and utilized performance will continue to widen as hardware core counts increases.

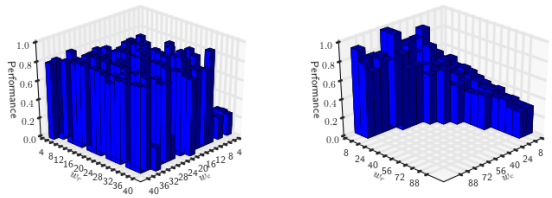
Algorithmic skeletons offer a solution to this this *programmability challenge* by raising the level of abstraction. This simplifies parallel programming, allowing developers to focus on solving problems rather than coordinating parallel resources. Skeleton frameworks provide robust parallel implementations of common patterns of computation which developers parameterise with their application-specific code. This greatly reduces the challenge of parallel programming, allowing users to structure their problem-solving logic sequentially, while offloading the cognitive cost of parallel coordination to the skeleton library author. The rising number of skeleton frameworks supporting graphics hardware illustrates the demand for high level abstractions for GPGPU programming [1, 2]. The challenge is in maintaining portable performance across the breadth of devices in the rapidly developing GPU and heterogeneous architecture landscape.

1.1 The Performance Portability Challenge

There are many factors — or *parameters* — which influence the behavior of parallel programs. For example, setting the number of threads to launch for a particular algorithm. The performance of parallel programs is sensitive to the values of these parameters, and when tuning to maximize performance, one size does not fit all. The suitability of parameter values depends on the program implementation, the target hardware, and the dataset that is operated upon. Iterative compilation and autotuning have been shown to help in these cases by automating the process of tuning parameter values to match individual execution environments [3]. However, there have been few attempts to develop general mechanisms for these techniques, and the time taken to develop ad-hoc autotuning solutions and gather performance data is often prohibitively expensive.

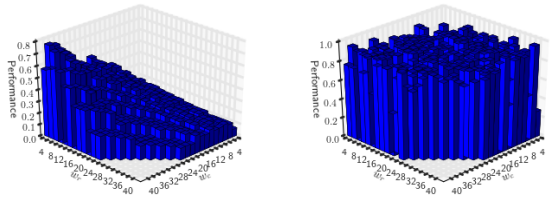
We believe that by embedding autotuning at the skeletal level, it is possible to achieve performance with algorithmic skeletons that is competitive with — and in some cases, exceeds — that of hand tuned parallel implementations which traditionally came at the cost of many man hours of work from expert programmers to develop.

Incorporating autotuning into algorithmic skeleton libraries has two key benefits: first, it minimizes development effort by requiring only a modification to the skeleton implementation rather than to every user program; and second, by targeting a library, it enables a broader and more substantive range of performance data to be gathered than with ad-hoc tuning of individual programs.



(a) (b)

Figure 1: The performance of different workgroup sizes for the same stencil program on two different devices: (a) Intel CPU, (b) NVIDIA GPU. Selecting an appropriate workgroup size depends on the execution device.



(a) (b)

Figure 2: The performance of different workgroup sizes for two different stencil programs on the same execution device. Selecting an appropriate workgroup size depends on the program.

1.2 Contributions

The key contributions of this work are:

- The design and implementation of a generic toolset for autotuning: *OmniTune* is a novel and extensible framework for collaborative autotuning of optimization parameters across the life cycle of programs.
- The integration of OmniTune with an established skeleton library for CPU and multi-GPU parallelism, SkelCL [4]. We extend SkelCL to provide autotuning for the selection of OpenCL workgroup size for Stencil skeletons.
- An empirical evaluation of OmniTune across 7 different architectures, demonstrating that OmniTune achieves 92% of the best possible performance, providing a median speedup of 5.65× over the best possible statically chosen workgroup size.

2. Motivation

In this section we will briefly examine the performance impact of selecting workgroup size for the SkelCL Stencil skeleton. A full explanation of SkelCL and the workgroup size parameter space is given Section 4.

SkelCL uses OpenCL to parallelise skeleton operations across many threads. In OpenCL, multiple threads are grouped into *workgroups*. The shape and size of these groups is known to have a big impact on performance. For the SkelCL stencil skeleton, the selection of workgroup size presents a two dimensional parameter space, consisting of a number of rows and columns ($w_r \times w_c$). Measuring and plotting the runtime of stencil programs using different workgroup sizes allows us to compare the performance of

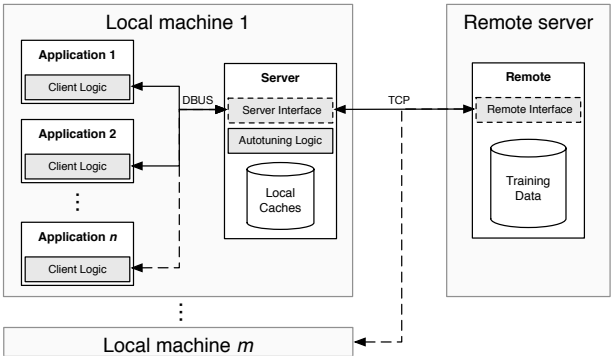


Figure 3: OmniTune system architecture, showing the separate components and the one to many relationship between servers to client applications, and remotes to servers.

different workgroup sizes for different combinations of architecture and program. Figure 1 shows this performance comparison for a single stencil program on two different devices, demonstrating that a good choice of workgroup size is device dependent. The optimization space of the same stencil benchmark on different devices is radically different: not only does the optimal workgroup size change between devices, but the performance of suboptimal workgroup sizes is also dissimilar. The optimization space of 1a has a grid-like structure, with clear performance advantages of workgroup sizes at multiples of 8 for w_c. A developer specifically targeting this device would learn to select workgroup sizes following this pattern. This domain specific knowledge clearly does not transfer to the device shown in 1b.

In Figure 2, we compare the performance of two different stencil programs on the *same* device, showing that workgroup size choice is also program dependent. In each of these four examples, the optimal workgroup size changes, as does the relative performance of suboptimal parameters. The average speedup of the best over the worst workgroup size is 37.0×, and the best average performance that can be achieved using a single fixed workgroup size is only 63% of the maximum.

SkelCL uses a fixed workgroup size by default. Since both the execution device and the user-provided stencil code are not known until runtime, selection of workgroup size should be made dynamically. To the best of our knowledge, there is currently no such generic system which meets our requirements for lightweight runtime machine learning autotuning with distributed training sets, and as a result, a variety of autotuners have been developed ad-hoc and on a per-case basis.

3. The OmniTune Framework

OmniTune is a novel framework for extensible, distributed autotuning of parameter values at runtime using machine learning. It serves as a generic platform for developing autotuning solutions, aiming to reduce both the engineering time required to target new optimization parameters, and the time to deploy on new systems.

It emphasizes collaborative, online learning of optimization spaces. A client-server architecture with clearly delineated separation of concerns minimizes the code footprint in client applications, enabling quick re-purposing for autotuning targets. OmniTune provides a lightweight interface for communication between each of the components, and aims

to strike a balance between offering a fully featured environment for quickly implementing autotuning, while providing enough flexibility to cater to a wide range of use cases. First, we describe the overall structure of OmniTune and the rationale for the design, followed by the interfaces and steps necessary to apply OmniTune.

3.1 System Architecture

Common implementations of autotuning in the literature either embed the autotuning logic within each target application (e.g. [5]), or take a standalone approach in which the autotuner is a program which must be externally invoked by the user to tune a target application (e.g. [6]). Embedding the autotuner within each target application has the advantage of providing "always-on" behavior, but is infeasible for complex systems in which the cost of building machine learning models must be added to each program run. The standalone approach separates the autotuning logic, at the expense of adding one additional step to the build process. The approach taken in OmniTune aims to combine the advantages of both techniques by implementing autotuning *as a service*, in which a standalone autotuning server performs the heavy lifting of managing training data and machine learning models, with a minimal set of lightweight communication logic to be embedded in target applications.

OmniTune is built around a three tier client-server model, shown in Figure 3. The applications which are to be autotuned are the *clients*. These clients communicate with a system-wide *server*, which handles autotuning requests. The server communicates and caches data sourced from a *remote* server, which maintains a global store of all autotuning data. There is a many to one relationship between clients, servers, and remotes, such that a single remote may handle connections to multiple servers, which in turn may accept connections from multiple clients. This design has two primary advantages: the first is that it decouples the autotuning logic from that of the client program, allowing developers to easily repurpose the autotuning framework to target additional optimization parameters without a significant development overhead for the target applications; the second advantage is that this enables collective tuning, in which training data gathered from a range of devices can be accessed and added to by any OmniTune server.

The OmniTune framework is implemented as a set of Python classes which are extended to target specific parameters. The generic implementation of OmniTune's server and remote components consists of 8987 lines of Python and MySQL code. No client logic is provided, since that is use case dependent (See Section 4 for an example implementation for SkelCL). Inter-process communication between client programs and the server uses the D-Bus protocol. D-Bus is cross-platform, and bindings are available for most major programming languages, allowing flexibility for use with a range of clients. Communication between servers and remotes uses TCP/IP (we used an Amazon Web Services database instance for development).

3.2 Autotuning Behavior

The goal of machine learning enabled autotuning is to build models from empirical performance data of past programs to select parameter values for new *unseen* programs. Instead of an iterative process of trial and improvement, parameter values are *predicted*, by building correlations between performance, and *features* (explanatory variables). The data used to build such models is called training data. OmniTune supports autotuning using a separate offline training phase, online training, or a mixture of both. For each autotuning-capable machine, an OmniTune server acts as an intermediary between training data and the client application, and hosts the autotuning logic. On launch, a server requests the latest training data from the remote, which it uses to build the relevant models for performing prediction of optimization parameter values. If additional training data is gathered by the server, this can be uploaded to the remote.

While the data types of the autotuning interface are application-specific (e.g. a binary flag or one or more numeric values), the general pattern is that a client application will request parameter values from an OmniTune server by sending it a set of explanatory variables. The server will then use machine learning models to form a prediction for the optimal parameter values and return these. Crucially, there is a mechanism provided for the client to *refuse* parameter values. This functionality is provided for cases where the predicted parameter values are in some way invalid and do not lead to a valid program.

The server contains a library of machine learning tools to perform parameter prediction, interfacing with the popular datamining software suite Weka[1] using its Java Native Interface. The provided tools include classifiers, regressors, and a selection of meta-learning algorithms.

OmniTune servers may perform additional feature extraction of explanatory variables supplied by incoming client requests. The reason for performing feature extraction on the server as opposed to on the client side is that this allows the results of expensive operations (for example, analyzing source code of target applications) to be cached for use across the lifespan of client applications. The contents of these local caches are periodically and asynchronously synced with the remote to maintain a global store of lookup tables for expensive operations.

3.3 Interfaces

Key design elements of OmniTune are the interfaces exposed by the server and remote components. Figure 4 shows an example communication pattern between the three components of an OmniTune system using these interfaces. In the example, a server first requests training data from the remote. A client application then performs a training phase in which it requests a set of parameters for training, evaluates the performance of the parameters, and then submits a measured value, which the server uses to update the remote. After training, another client program requests a set of parameters for performance, refuses them, and makes a new request.

Client-Server An OmniTune server exposes a public interface over D-Bus with four operations. Client applications invoke these methods to request parameter values, submit new training observations, and refuse suggested parameters:

- REQUEST(x : feature vector) $\rightarrow p$: param
  Given explanatory variables x, request the parameter values p which are expected to provide maximum performance.
- REQUESTTRAINING(x : feature vector) $\rightarrow p$: param
  Given explanatory variables x, allow the server to select parameter values p for evaluating their fitness.

[1] `http://www.cs.waikato.ac.nz/ml/weka/`

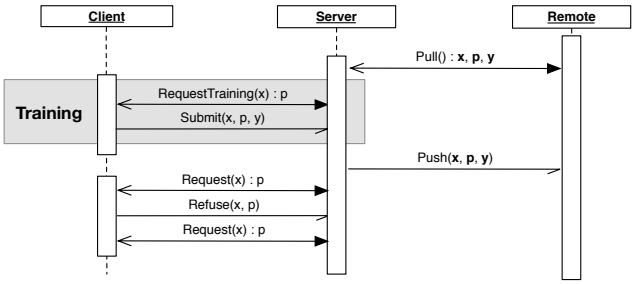


Figure 4: An example communication pattern between OmniTune components, showing an offline training phase.

- SUBMIT(x : feature vector, p : param, y : fitness)
  Submit an observed measurement of fitness y for parameter values p, given explanatory variables x.
- REFUSE(x : feature vector, p : param)
  Refuse parameter values p, given a set of explanatory variables x. Once refused, those parameters are blacklisted and will not be returned by any subsequent calls to REQUEST() or REQUESTTRAINING() for the same explanatory variables x.

Server-Remote The role of the remote is to provide bookkeeping of training data for machine learning. Remotes allow shared access to data from multiple servers using a transactional communication pattern, supported by two methods:

- PUSH($\mathbf{x}$: feature vectors, $\mathbf{p}$: params, $\mathbf{y}$: fitnesses)
  Asynchronously submit training data as three lists: explanatory variables $\mathbf{x}$, parameter values $\mathbf{p}$, and observed outcomes $\mathbf{y}$.
- PULL() $\rightarrow$ ($\mathbf{x}$: feature vectors, $\mathbf{p}$: params, $\mathbf{y}$: fitnesses)
  Request training data as three lists: explanatory variables $\mathbf{x}$, parameter values $\mathbf{p}$, and observed outcomes $\mathbf{y}$.

3.4 Extensibility

To extend OmniTune to target an optimization parameter, a developer extends the server class to implement response handlers for the four public interface operations, and then inserts client code into the target application to call these operations. The implementation of these response handlers and invoking client code determines the type of autotuning methods supported. Figure 5 shows the flow diagram for an example OmniTune implementation. The call to REQUESTTRAINING() is matched with a response call of SUBMIT(), showing the client recording a training observation. In the next Section, we will detail the steps required to apply OmniTune to SkelCL.

4. Integration of OmniTune with SkelCL

In this section we demonstrate the practicality of OmniTune by integrating the framework into an established algorithmic skeleton library. Introduced in [4], SkelCL allows users to easily harness the power of GPUs and CPUs for data parallel computing, offering a set of OpenCL implementations of data parallel skeletons in an object oriented C++ library.

The goal of SkelCL is to enable the transition towards higher-level programming of GPUs, without requiring users to be intimately knowledgeable of the concepts unique to OpenCL programming, such as the memory or execution model. SkelCL has been shown to reduce programming effort

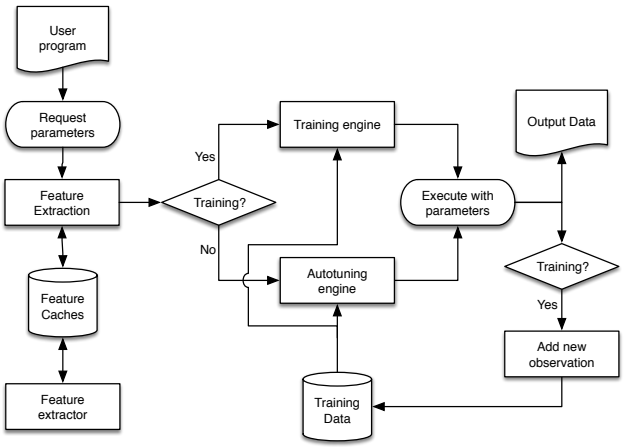


Figure 5: Predicting parameter values and collecting training data with OmniTune.

for developing real applications through the use of robust pattern implementations and automated memory management. Skeletons are parameterised with user functions which are compiled into OpenCL kernels for execution on device hardware. SkelCL supports operations on one or two dimensional arrays of data, with the Vector and Matrix container types transparently handling lazy transfers between host and device memory, and supporting partitioning for multi-GPU execution. SkelCL is freely available and distributed under dual GPL and academic licenses[2].

4.1 The Stencil Skeleton

Stencils are patterns of computation which operate on uniform grids of data, where the value of each grid element (cell) is updated based on its current value and the value of one or more neighboring elements, called the *border region*. Figure 6 shows the use of a stencil to apply a Gaussian blur to an image. SkelCL provides a 2D stencil skeleton which allows users to provide a function which updates a cell's value, while SkelCL orchestrates the parallel execution of this function across all cells [7].

The border region is described by a *stencil shape*, which defines an $i \times j$ rectangular region around each cell which is used to update the cell value. Stencil shapes may be asymmetrical, and are defined in terms of the number of cells in the border region to the north, east, south, and west of each cell. Given a function f, a stencil shape S, and an $n \times m$ matrix with elements x_{ij}:

$$\text{Stencil}\left(f, S, \begin{bmatrix} x_{11} & \cdots & x_{1m} \\ \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{nm} \end{bmatrix}\right) \rightarrow \begin{bmatrix} z_{11} & \cdots & z_{1m} \\ \vdots & \ddots & \vdots \\ z_{n1} & \cdots & z_{nm} \end{bmatrix} \quad (1)$$

where:

$$z_{ij} = f\left(\begin{bmatrix} x_{i-S_n,j-S_w} & \cdots & x_{i-S_n,j+S_e} \\ \vdots & \ddots & \vdots \\ x_{i+S_s,j-S_w} & \cdots & x_{i+S_s,j+S_e} \end{bmatrix}\right) \quad (2)$$

For border region elements outside the bounds of the matrix, values are substituted from either a predefined padding value, or the value of the nearest element within the matrix, depending on user preference.

[2] http://skelcl.uni-muenster.de

A popular usage of Stencil codes is for iterative problem solving, whereby a stencil operation is repeated over a range of discrete time steps $0 \leq t \leq t_{max}$, and $t \in \mathbb{N}$. An iterative stencil operation g accepts a customizing function f, a Stencil shape S, and a matrix M with initial values M_{init}. The value of an iterative stencil can be defined recursively as:

$$g(f, S, M, t) = \begin{cases} \text{Stencil}\left(f, S, g(f, S, M, t-1)\right), & \text{if } t \geq 1 \\ M_{init}, & \text{otherwise} \end{cases} \quad (3)$$

Examples of iterative stencils include cellular automata and partial differential equation solvers.

In the implementation of the SkelCL stencil skeleton, each element in the matrix is mapped to a unique thread (known as a *work item* in OpenCL) which applies the user-specified function. The work items are then divided into *workgroups* for execution on the target hardware. Each work-item reads the value of its corresponding matrix element and the surrounding elements defined by the border region. Since the border regions of neighboring elements overlap, the value of all elements within a workgroup are copied into a *tile*, allocated as a contiguous region of the fast, but small local memory. As local memory access times are much faster than that of global device memory, this greatly reduces the latency of the border region memory accesses performed by each work item. Changing the size of workgroups thus affects the amount of local memory required for each workgroup, and in turn affects the number of workgroups which may be simultaneously active on the device. While the user defines the data size and type, the shape of the border region, and the function being applied to each element, it is the responsibility of the SkelCL stencil implementation to select an appropriate workgroup size to use.

4.2 Optimization Parameters

SkelCL stencil kernels are parameterised by a workgroup size w, which consists of two integer values to denote the number of rows and columns in a workgroup. The space of optimization parameter values is subject to hard constraints, and these constraints cannot conveniently be statically determined. Contributing factors are architectural limitations, kernel constraints, and parameters which are refused for other reasons. Each OpenCL device imposes a maximum workgroup size which can be statically checked. These are defined by architectural limitations of how code is mapped to the underlying execution hardware. At runtime, once an OpenCL program has been compiled to a kernel, users can query the maximum workgroup size supported by that particular kernel dynamically. This value cannot easily be obtained statically as there is no mechanism to determine the maximum workgroup size for a given source code and device without first compiling it, which in OpenCL does not occur until runtime.

Factors which affect a kernel's maximum workgroup size include the number of registers required for a kernel, and the available number of SIMD execution units for each type of instructions in a kernel. In addition to satisfying the constraints of the device and kernel, not all points in the workgroup size optimization space are guaranteed to provide working programs. A *refused parameter* is a workgroup size which satisfies the kernel and architectural constraints, yet causes a `CL_OUT_OF_RESOURCES` error to be thrown when the kernel is enqueued. Note that in many OpenCL implementations, this error type acts as a generic placeholder and may not necessarily indicate that the underlying cause of the

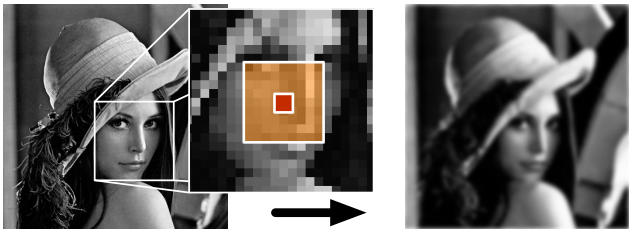


Figure 6: Application of a Gaussian blur stencil operation to an image, with a border region of radius 1. In a Gaussian blur, pixel values are interpolated with neighboring pixels, producing a smoothed effect.

error was due to finite resources constraints. We define a *legal* workgroup size as one which, for a given *scenario* s (a combination of program, device, and dataset), satisfies the architectural and kernel constraints, and is not refused. The subset of all possible workgroup sizes $W_{legal}(s) \subset W$ that are legal for a given scenario s is then:

$$W_{legal}(s) = \{w | w \in W, w < W_{\max}(s)\} - W_{refused}(s) \quad (4)$$

Where $W_{\max}(s)$ can be determined at runtime prior to the kernels execution, but the set $W_{refused}(s)$ can only be determined experimentally.

The *oracle* workgroup size $\Omega(s) \in W_{legal}(s)$ of a scenario s is the w value which provides the lowest mean runtime. The relative performance $p(s, w)$ of a particular workgroup against the maximum available performance for that scenario, within the range $0 \leq p(s, w) \leq 1$, is the ratio of the runtime of a program with workgroup size w over the oracle workgroup size $\Omega(s)$. For a given workgroup size, the average performance $\bar{p}(w)$ across a set of scenarios S can be found using the geometric mean of performance relative to the oracle:

$$\bar{p}(w) = \left(\prod_{s \in S} p(s, w) \right)^{1/|S|} \quad (5)$$

4.3 Machine Learning

The optimization space presented by the workgroup size of OpenCL kernels is large, complex, and non-linear. The challenge is to design a system which, given a set of prior observations of the empirical performance of stencil codes with different workgroup sizes, predict workgroup sizes for *unseen* stencils which will maximize the performance. Successfully applying machine learning requires plentiful training data, the careful selection of explanatory variables, and appropriate machine learning methods. For the purpose of this work we use a *classification* approach, in which a classifier automatically correlates patterns between explanatory variables and the workgroup sizes which provide optimal performance. The classifier used is the popular J48 Decision Tree [8], chosen due to its low classification cost and ability to efficiently handle large dimensionality training data.

For each scenario, a total of 102 explanatory variables are extracted to capture information about the device, program, and dataset. Device variables encode the device type (e.g. CPU or GPU, integrated or external, connection bus), properties about the host (e.g. system memory, maximum clock frequency), and numerous properties about the execution device (e.g. number of compute units, local memory size, global caches). Program variables include instruction densities for each instruction type, the total number of basic

blocks, and the total instruction count. They are extracted using static instruction count passes over an LLVM IR compiled version of the user stencil implementation. Compilation to bitcode is a relatively expensive task, so lookup tables are used to cache repeated uses of the same stencil codes, identified by a checksum of the source code. Dataset variables include the data types (input and output), and dimensions of the input matrix and stencil region.

To collect training data, we run multiple iterations of a stencil program to enumerate the workgroup size optimization space, and use the OpenCL's Profiling API to record stencil kernel execution times in the client application, which are then submitted to the OmniTune server. The RequestTraining(x) server interface returns a workgroup size with a randomly selected even number of rows and columns that obeys the maximum size constraints.

A parameterised template substitution engine is used to generate synthetic stencil applications for gathering performance data. Stencils templates are parameterised with a border region size and *complexity*, a simple metric to broadly dictate the number of operations in a given stencil code.

Once the performance of different workgroup sizes for a scenario is assessed, the set of explanatory variables describing the scenario is paired with the oracle workgroup size. This process is repeated for multiple scenarios to create training data. A classifier learns from this training data to make predictions for new sets of explanatory variables, by predicting a workgroup size from the set of oracle workgroup sizes of the training data.

This approach presents the problem that after training, there is no guarantee that the set of workgroup sizes which may be predicted is within the set of legal workgroup sizes for future scenarios. This may result in a classifier predicting a workgroup size which is not legal for a scenario, $w \notin W_{legal}(s)$, either because it exceeds $W_{\max}(s)$, or because the parameter is refused. If this occurs, a *nearest neighbor* approach is used to select the workgroup size w which is expected to be legal and has the lowest Euclidian distance to the predicted value c. This is achieved by comparing row (r) and column (c) indices:

$$w = \underset{w \in W_{legal(s)}}{\arg\min} \sqrt{(c_r - w_r)^2 + (c_c - w_c)^2} \qquad (6)$$

This process of selecting alternative parameters will iterate until a legal parameter is found.

4.4 Implementation

The OmniTune framework consists of 8987 lines of Python and MySQL code. A further 976 lines are required for the SkelCL frontend to implement the server response handlers and database backend. By design, the client-server model minimizes the impact of number of modifications that are required to enable autotuning in client applications. The only modification required to SkelCL is to replace the hard-coded values for workgroup size with a subroutine to request a workgroup size from the OmniTune server over a D-Bus connection. To use the system, a user must download a copy of SkelCL modified with the OmniTune functionality, and start a local OmniTune server instance. A configuration file is used to determine the domain address and authentication details of the remote server. On first launch, the OmniTune server will fetch the latest training data from the remote.

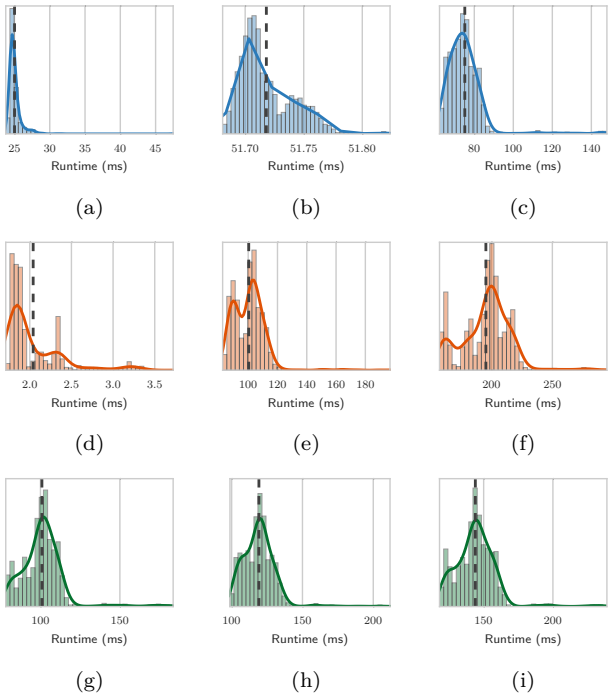


Figure 7: Distribution of runtime samples for test cases from three devices. Each plot contains a 35-bin histogram of 1000 samples, and a fitted kernel density estimate with bandwidth 0.3. The sample mean is shown as a vertical dashed line. The top row are from the Intel i5-4570, the second row from the Nvidia GTX 590, and the third row from the AMD Tahiti 7970. In some of the plots, the distribution of runtimes is bi- or multi-modal, and skewed to the lower end of the runtimes range.

5. Experimental Setup

This section describes an exhaustive enumeration of the workgroup size optimization space for 429 combinations of architecture, program, and dataset. It contains the methodology used to collect empirical performance data on which to base performance comparisons of different workgroup sizes, and the steps necessary to obtain repeatable results.

A full enumeration of the workgroup size optimization spaces was performed across synthetically generated benchmarks and four reference stencil benchmarks: Canny Edge Detection, Conway's Game of Life, Heat Equation, and Gaussian Blur [4]. Performance data was collected from 7 experimental platforms, comprising 4 GPU devices: AMD Tahiti 7970, Nvidia GTX 590, Nvidia GTX 690, Nvidia GTX TITAN; and 3 CPU devices: Intel i5-2430M, Intel i5-4570, i7-3820. Each platform was unloaded, frequency governors disabled, and benchmark processes set to the highest priority available to the task scheduler. Datasets and programs were stored in an in-memory file system. For each program, dataset sizes of size 512×512, 1024×1024, 2048×2048, and 4096×4096 were used. A minimum of 30 samples were recorded for each scenario and workgroup size.

Program behavior is validated by comparing program output against a gold standard output collected by executing each of the real-world benchmarks programs using the *baseline* workgroup size (defined below). The output of real-

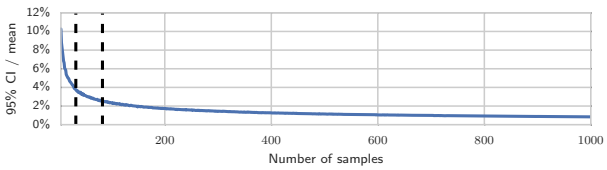


Figure 8: Ratio of 95% confidence interval to mean as a function of sample size. Two dashed lines indicate the confidence intervals at the minimum (3.7%) and mean (2.5%) sample size found in the experimental dataset.

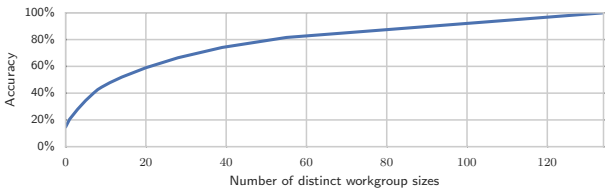


Figure 9: Accuracy compared to the oracle as a function of the number of unique workgroup sizes. The greatest accuracy that can be achieved using a single statically chosen workgroup size is 15%. Achieving 50% oracle accuracy requires a minimum of 14 distinct workgroup sizes.

world benchmarks with other workgroup sizes is compared to this gold standard output to test for correct program execution.

6. Evaluation

This section evaluates the performance of OmniTune when tasked with selecting workgroup sizes for SkelCL stencil codes. The experimental results consist of measured runtimes for a set of *test cases*, where each test case τ_i consists of a scenario, workgroup size pair $\tau_i = (s_i, w_i)$, and is associated with a *sample* of observed runtimes from multiple runs of the program. A total of 269,813 test cases have been evaluated with an average sample size of 83 (min 33, total 16,917,118). This represents an exhaustive enumeration of the workgroup size optimization space for 429 scenarios, with an average of 629 (max 7,260) unique workgroup sizes for each scenario.

6.1 Runtime Noise

First we examine the noise present in program runtime measurements. The complex interaction between processes competing for the finite resources of a system introduces many sources for such noise. Figure 7 plots the distributions of 1000 runtimes recorded for 9 SkelCL stencil kernels, (a)–(i). The plots show that the distribution of runtimes is not Gaussian; rather, it is sometimes multimodal, and generally skewed to the lower end of the runtime range, with a long "tail" to the right. This fits our intuition that programs have a hard *minimum* runtime enforced by the time taken to execute the instructions of a program, and that noise introduced to the system extends this runtime. For example, preempting an OpenCL process on a CPU so that another process may run may cause the very long tail visible in Figure 7a.

It is important to ensure a sufficiently large sample size when performing optimisations based on empirical performance data. A recommendation of ≥ 30 samples is common in the benchmarking literature [9]. Our experimental results support this recommendation: Figure 8 plots the ratio of 95% confidence interval to the sample mean for different sample sizes, showing a 50% reduction in confidence interval size when increasing the sample size from 10 to 30. In this experimental dataset, the ratio of confidence interval to mean at the smallest sample size (33) is 3.7%, and 2.5% at the mean sample size (83).

6.2 OpenCL Workgroup Size Optimization Space

We can calculate an upper bound for the performance impact of the workgroup size parameter by comparing the average runtimes of the *best* and *worst* workgroup size for a single scenario. Applying this to all scenarios, we find the average speedup upper bound to be 15.14× (min 1.03×, max 207.72×). This demonstrates the importance of tuning stencil workgroup sizes — if chosen incorrectly, the runtime of stencil programs can be extended by up to 207.72×. Note that for 5 of the scenarios, the speedup of the best over worst workgroup sizes is less than 5%. For these scenarios, there is little benefit to autotuning; however, this represents only 1.1% of the tested scenarios. For 50% of the scenarios, the speedup of the best over worst workgroup sizes is greater than 6.19×.

For the purposes of evaluating autotuning, we use three *baselines* to compare program runtimes against. The relative performance of a workgroup size for a particular scenario is compared against runtimes for each of three parameters:

- *Oracle* — The *oracle* workgroup size is the workgroup size which provided the lowest mean runtime for a given scenario. Speedup relative to the oracle is in the range $0 \leq x \leq 1$, so this can be referred to as *performance*.
- *Baseline* — The *baseline* parameter is the workgroup size which provides the best overall performance while being legal for all scenarios. Such a baseline value represents the *best* possible performance which can be achieved using a single, statically chosen workgroup size. By defining $W_{safe} \in W$ as the intersection of legal workgroup sizes, the baseline $\bar{w}$ can be found using:

$$W_{safe} = \cap \{W_{legal}(s) | s \in S\} \tag{7}$$

$$\bar{w} = \underset{w \in W_{safe}}{\arg\max}\, \bar{p}(w) \tag{8}$$

  For our experimental data, we find this value to be $w_{(4\times4)}$.
- *Human expert* — In the original implementation of the SkelCL stencil skeleton [7], Steuwer et al. selected a workgroup size of $w_{(32\times4)}$, based on an evaluation of 4 stencil programs on a Tesla S1070 system.

Across the 429 scenarios tested, there are 135 unique *oracle* workgroup sizes. This demonstrates the difficulty in attempting to statically tune for *optimal* parameter values, since 31.5% of scenarios have different oracle workgroup sizes. Figure 9 shows that a minimum of 14 distinct workgroup sizes are needed to achieve just 50% of the oracle accuracy, although it is important to make the distinction that oracle *accuracy* and *performance* are not equivalent.

We find that the *human expert* selected workgroup size is invalid for 2.6% of scenarios, as it is refused by 11 test cases. By device, these are: 3 on the GTX 690, 6 on the i5-2430M,

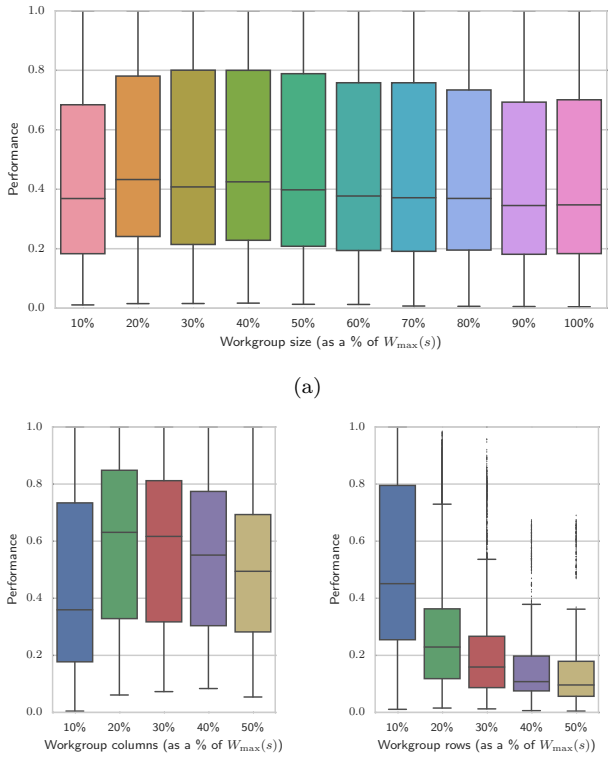


Figure 10: Comparing performance of workgroup sizes relative to the oracle as a function of: (a) maximum legal size, (b) number of columns, and (c) number of rows. Each workgroup size is normalized to the maximum allowed for that scenario, $W_{\max}(s)$. There is no clear correlation between workgroup size and performance.

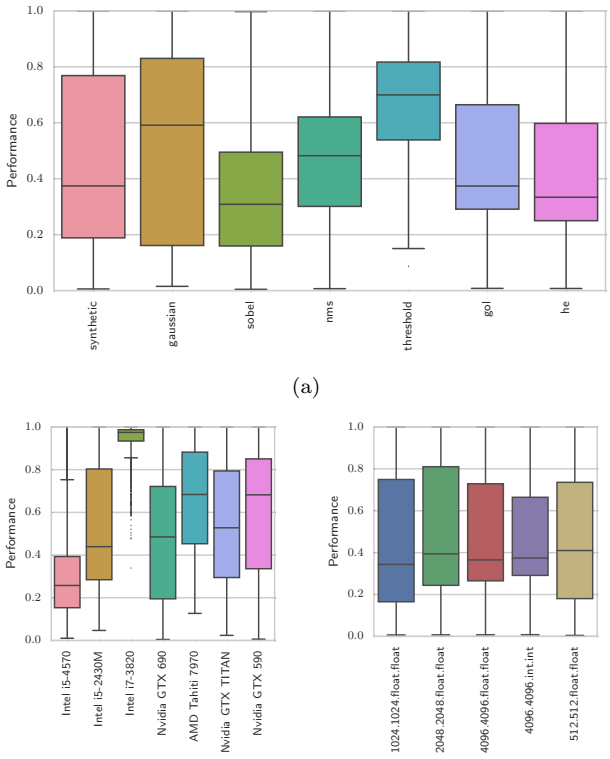


Figure 11: Performance relative to the oracle of workgroup sizes across all test cases, grouped by: (a) kernels, (b) devices, and (c) datasets. The performance impact is not consistent across kernels, devices, or datasets. The Intel i7-3820 has the lowest performance gains from tuning workgroup size.

and 2 on the i5-4570. For the purpose of comparing performance against human experts, we will ignore these test cases, but it demonstrates the utility of autotuning not just for maximizing performance, but ensuring program reliability. For the scenarios where the human expert workgroup size *is* legal, it achieves an impressive geometric mean of 79.2% of the oracle performance. The average speedup of oracle workgroup sizes over the workgroup size selected by a human expert is 1.37× (min 1.0×, max 5.17×).

The utility of the *baseline* workgroup size is that it represents the best performance that can be achieved through static tuning. The baseline workgroup size achieves only 24% of the maximum performance. Figures 10 and 11 show box plots for the performance of all workgroup sizes using different groupings: ratio of maximum workgroup size, kernel, device, and dataset. The plots show the median performance, interquartile range, and outliers. What is evident is both the large range of workgroup size performances (i.e. the high performance upper bounds), and the lack of obvious correlations between any of the groupings and performance.

6.3 Autotuning Workgroup Sizes

To evaluate the performance of machine learning-enabled autotuning of SkelCL stencils, we partition the experimental data into *training* and *test* sets. The training set is used to build the machine learning model. The predicted workgroup size for each entry in the test set is then used to evaluate the autotuning performance. We use 5 different approaches to partitioning the test and training data, which each test different aspects of the system. The first is a k-fold cross validation, a standard machine learning model validation technique in which the set of all data is shuffled and then divided into k equally sized validation sets. Each validation set is used to test a model trained on the remaining data [8]. In our evaluation we use a value of $k = 10$. The second technique is to partition the data such that it consists of data gathered from synthetic benchmarks, and use data collected from real-world benchmarks to test. This tests the utility of training using synthetically generated benchmarks. The third, forth, and fifth approaches involve creating leave-one-out training sets for all data grouped by device, kernel, and dataset, respectively. This tests the ability to successfully apply prior knowledge about other devices, kernels, and datasets, to new unseen cases. For example, of the n devices used to collect performance data, the model is trained on data from $n-1$ devices, and tested against data from the n^{th}. Table 1 summarizes the results of evaluating the autotuner using each of the different validation techniques.

The autotuner achieves good performance, with average speedups over the baseline across all validation sets range between 4.79× and 5.65×. Importantly, the performance when validating across devices, kernels, and datasets, is comparable to the 10-fold validation. This demonstrates that the autotuner is capable of learning *across* these targets. So if

Training Dataset	Performance	Speedup over Baseline	Speedup over Human Expert
10-fold	92%	5.65×	1.26×
Synthetic	92%	4.79×	1.13×
$n-1$ Device	85%	5.23×	1.17×
$n-1$ Kernel	89%	5.43×	1.21×
$n-1$ Dataset	91%	5.63×	1.25×
Average	**90%**	**5.45×**	**1.22×**

Table 1: Performance results using a J48 Decision Tree across different validation sets. Note that the human expert selected workgroup size is invalid for 2.6% of test cases, which we excluded for the purpose of performance comparisons against human expert.

the autotuner is deployed to a system for which it has no prior knowledge, it does not suffer a significant drop in performance. The same is true for an unseen kernel, or dataset type. This, combined with the distributed datasets provided by the OmniTune framework, demonstrates the utility of autotuning at the skeletal level, allowing machine learning to successfully learn predictions across unseen programs, kernels, and datasets.

Classification using decision trees is a lightweight process (they can be implemented using a chain of `if/else` statements). The measured overhead of autotuning is 2.5ms, of which only 0.3ms is required for classification using Weka, although an optimized decision tree implementation could reduce this further. The remaining 2.2ms is required for feature extraction and the inter-process round trip between the OmniTune server and client.

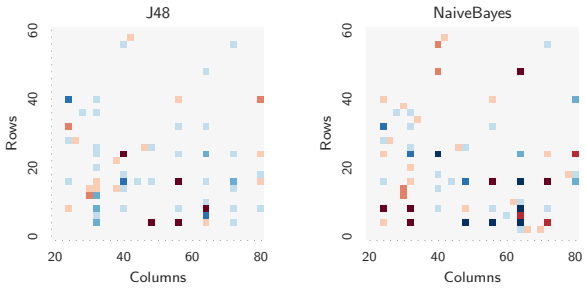
6.4 OmniTune Extensibility

The client-server architecture OmniTune neatly separates the autotuning logic from the target application. This makes adjusting the autotuning methodology a simple process. To demonstrate this, we changed the machine learning algorithm from a J48 decision tree to a Naive Bayes classifier, and duplicated the evaluation. This required only a single line of source code in the OmniTune server extension to be changed. Figure 12 visualizes the differences in autotuning predictions when changing between these two classifiers. While the average performances of the two classifiers is comparable, the distribution of predictions is not. For example, the Naive Bayes classifier predicted the human expert selected workgroup size of $w_{(32\times4)}$ more frequently than it was optimal, while the decision tree predicted it less frequently. Selection of machine learning algorithms has a large impact on the effectiveness of autotuning, and the OmniTune client-server design allows for low cost experimenting with different approaches. In future work we will investigate meta-tuning techniques for selecting autotuning algorithms.

6.5 Summary

In this section we have explored the performance impact of the workgroup size optimization space, and the effectiveness of autotuning using OmniTune to exploit this. By comparing the relative performance of an average of 629 workgroup sizes for each of 429 scenarios, the following conclusions can be drawn:

- The performance gap between the best and workgroup sizes for a particular combination of hardware, software, and dataset is up to 207.72×.



(a) (b)

Figure 12: Heatmaps of autotuner predictions for a subset of the explored optimization space ($w_c < 80, w_r < 80$) using two different classifiers. The shading in each cells indicates if it is predicted less frequently (blue), ore more frequently (red) than it is optimal. Color gradients are normalized across plots.

- Not all workgroup sizes are legal, and the space of legal workgroup sizes cannot statically be determined. Adaptive tuning is required to ensure reliable performance.
- Statically tuning workgroup size fails to extract the potential performance across a range of programs, architectures, and datasets. The best statically chosen workgroup size achieves only 26% of the optimal performance.
- Workgroup size prediction using a decision tree achieves an average 90% of the optimal performance.
- Auotuning provides performance portability across programs, devices, and datasets. The performance of predicted workgroup sizes for unseen devices is within 8% of the performance for known devices.

7. Related Work

Early work in autotuning applied iterative search techniques to the space of compiler optimisations [3, 10]. Since then, machine learning techniques have been successfully employed to reduce the cost of iterative compilation [11–13]. However, optimizing GPGPU programs presents different challenges to that of traditional CPU programming. Ryoo et al. demonstrated speedups of up to 432× for matrix multiplication in CUDA through the appropriate use of zero-overhead thread scheduling, memory bandwidth, and thread grouping. The importance of proper exploitation of local shared memory and synchronization costs is explored in [15]. In [5], data locality optimisations are automated using a description of the hardware and a memory-placement-agnostic compiler. Magni, Dubach, and O'Boyle use a machine learning model to predict optimal thread coarsening factors of OpenCL kernels in [16], demonstrating speedups between 1.11× and 1.33×.

Auotuning transformations for stencil codes are explored in [17] using an IR to represent stencils and a CUDA code generator at the backend. However, they do not optimize for the GPU memory hierarchy, using only global memory. In [6], Lutz, Fensch, and Cole demonstrate that optimal swapping strategy for multi-GPU stencils depends on the size of the grid, the number of partitions, and the connection mechanism (e.g. PCI express). Autotuning for algorithmic skeletons is performed using Nearest Neighbor classification and Principle Component Analysis in [18].

OpenTuner is a general purpose toolkit for autotuning which uses ensemble search techniques to reduce the cost of exploring an optimization space, rather than the machine learning approach taken in this work [19]. Since OpenTuner does not *learn* optimization spaces as OmniTune does, performance data is not shared across devices. This means that the search for performant parameter values must be performed by each new device to be autotuned. Our approach combines machine learning with distributed training sets so that new users automatically benefit from the collective tuning experience of other users, which reduces the time to deployment.

A "big data" driven approach to autotuning is presented in [20]. The authors propose the use of "Collective optimization" to leverage training experience across devices, by sharing performance data, datasets and additional metadata about experimental setups. In addition to the mechanism for sharing training datasets, our system provides the capabilities of performing autotuning at runtime using a lightweight inter-process communication interface. Additionally, Collective Mind uses a NoSQL JSON format for storing datasets. The relational schema used in OmniTune offers greater scaling performance and lower storage overhead.

8. Conclusions

As the trend towards increasingly programmable heterogeneous architectures continues, the need for high level, accessible abstractions to manage such parallelism will continue to grow. Autotuning proves to be a valuable aid for achieving these goals, provided that the burden of development and collecting performance data is lifted from the user. The system presented in this paper aims to solve this issue by providing a generic interface for implementing machine learning-enabled autotuning. OmniTune is a novel framework for autotuning which has the benefits of a fast, "always-on" interface for client applications, while being able to synchronize data with global repositories of knowledge which are built up across devices. To demonstrate the utility of this framework, we implemented a frontend for predicting the workgroup size of OpenCL kernels for SkelCL stencil codes. This optimization space is complex, non linear, and critical for the performance of stencil kernels. Selecting the correct workgroup size is difficult — requiring a knowledge of the kernel, dataset, and underlying architecture. The implemented autotuner achieves 92% of the maximum performance, and provides performance portability, even achieving an average of 85% of the maximum performance when deployed on a device for which it has no prior training data. By performing autotuning at the skeletal level, the system is able to exploit underlying similarities between pattern implementations which are not shared in unstructured code. In future work we will explore methods for collaborative exploration of optimization spaces in parallel across multiple cooperating devices, and targeting multiple parameters simultaneously.

Acknowledgments

This work was supported by the UK Engineering and Physical Sciences Research Council under grants EP/L01503X/1 for the University of Edinburgh School of Informatics Centre for Doctoral Training in Pervasive Parallelism (`http://pervasiveparallelism.inf.ed.ac.uk/`), EP/H044752/1 (ALEA), and EP/M015793/1 (DIVIDEND).

References

[1] J Enmyren and CW Kessler. "SkePU: a multi-backend skeleton programming library for multi-GPU systems". In: *HLPP*. ACM, 2010, pp. 5–14.

[2] Ricardo Marques, Hervé Paulino, Fernando Alexandre, and Pedro D. Medeiros. "Algorithmic skeleton framework for the orchestration of GPU computations". In: *Euro-Par 2013 Parallel Processing*. Vol. 8097 LNCS. Springer, 2013, pp. 874–885.

[3] T Kisuki, Niels Bohrweg, and Campus De Beaulieu. "A Feasibility Study in Iterative Compilation". In: *High Performance Computing*. Springer, 1999, pp. 131–132.

[4] Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. "SkelCL - A Portable Skeleton Library for High-Level GPU Programming". In: *IPDPSW*. IEEE, May 2011, pp. 1176–1182.

[5] Guoyang Chen and Bo Wu. "PORPLE: An Extensible Optimizer for Portable Data Placement on GPU". In: *MICRO*. IEEE, 2014, pp. 88–100.

[6] Thibaut Lutz, Christian Fensch, and Murray Cole. "PARTANS: An Autotuning Framework for Stencil Computation on Multi-GPU Systems". In: *TACO* 9.4 (2013), p. 59.

[7] Michel Steuwer, Michael Haidl, Stefan Breuer, and Sergei Gorlatch. "High-level programming of stencil computations on multi-GPU systems using the SkelCL library". In: *Parallel Processing Letters* 24.03 (2014), p. 1441005.

[8] Jiawei Han, Micheline Kamber, and Jian Pei. *Data mining: concepts and techniques*. Elsevier, 2011.

[9] Andy Georges, Dries Buytaert, and Lieven Eeckhout. "Statistically Rigorous Java Performance Evaluation". In: *ACM SIGPLAN*. Vol. 42. 10. New York, NY, USA: ACM, Oct. 2007, p. 57.

[10] F. Bodin, T. Kisuki, P.M.W. Knijnenburg, M.F.P. O'Boyle, and E. Rohou. "Iterative compilation in a non-linear optimisation space". In: *Workshop on Profile Directed Feedback-Compilation, PACT*. 1998.

[11] Felix Agakov, Edwin Bonilla, John Cavazos, Björn Franke, Grigori Fursin, Michael FP O'Boyle, John Thomson, Marc Toussaint, and Christopher KI Williams. "Using Machine Learning to Focus Iterative Optimization". In: *CGO*. IEEE Computer Society, 2006, pp. 295–305.

[12] Mark Stephenson, Martin Martin, and Una-may O Reilly. "Meta Optimization: Improving Compiler Heuristics with Machine Learning". In: *ACM SIGPLAN Notices* 38.5 (2003), pp. 77–90.

[13] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, Francois Bodin, Phil Barnard, Elton Ashton, Edwin Bonilla, John Thomson, Christopher K. I. Williams, and Michael O'Boyle. "Milepost GCC: Machine Learning Enabled Self-tuning Compiler". In: *IJPP* 39.3 (Jan. 2011), pp. 296–327.

[14] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA". In: *PPoPP* (2008), p. 73.

[15] Victor W. Lee, Per Hammarlund, Ronak Singhal, Pradeep Dubey, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, and Srinivas Chennupaty. "Debunking the 100X GPU vs. CPU myth". In: *ACM SIGARCH Computer Architecture News* 38 (2010), p. 451.

[16] Alberto Magni, Christophe Dubach, and Michael O'Boyle. "Automatic optimization of thread-coarsening for graphics processors". In: *PACT*. 2014, pp. 455–466.

[17] Shoaib Kamil, Cy Chan, Leonid Oliker, John Shall, and Samuel Williams. "An auto-tuning framework for parallel multicore stencil computations". In: *IPDPS* (2010).

[18] Alexander Collins, Christian Fensch, Hugh Leather, and Murray Cole. "MaSiF: Machine Learning Guided Auto-tuning of Parallel Skeletons". In: *HiPC* (Dec. 2013), pp. 186–195.

[19] Jason Ansel, Shoaib Kamil, Una-may O Reilly, Saman Amarasinghe, Jason Ansel, and Una-may O Reilly. "OpenTuner: An Extensible Framework for Program Autotuning". In: *PACT*. 2013.

[20] Grigori Fursin, Renato Miceli, Anton Lokhmotov, Michael Gerndt, Marc Baboulin, Allen D Malony, Zbigniew Chamski, Diego Novillo, and Davide Del Vento. "Collective Mind: Towards practical and collaborative auto-tuning". In: *Scientific Programming* 22.4 (2014), pp. 309–329.

Synthesizing Benchmarks for Predictive Modeling

Abstract

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

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

We use our generator for OpenCL programs, CLgen, to automatically synthesize thousands of programs and show that learning over these improves the performance of a state of the art predictive model by 1.27×. In addition, the fine covering of the feature space automatically exposes weaknesses in the feature design which are invisible with the sparse training examples from existing benchmark suites. Correcting these weaknesses further increases performance by 4.30×.

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

1. Introduction

Predictive modeling is a well researched method for building optimization heuristics that often exceed human experts and reduces development time [1–9]. Figure 1 shows the process by which these models are trained. A set of training programs are identified which are expected to be representative of the application domain. The programs are compiled and executed with different parameter values for the target heuristic, to determine which are the best values for each training program. Each program is also summarized by a vector of features which describe the information that is expected to be important in predicting the best heuristic parameter values. These training examples of program features and desired heuristic values are used to create a machine learning model which, when given the features from a new, unseen program, can predict good heuristic values for it.

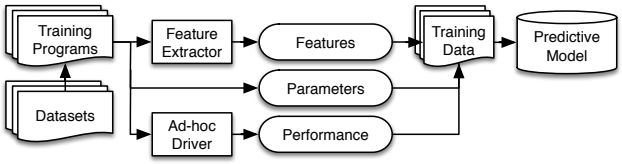


Figure 1: Training a predictive model.

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

There have been efforts to solve this problem using templates. The essence of the approach is to construct a probabilistic grammar with embedded semantic actions that defines a language of possible programs. New programs may be created by sampling the grammar and, through setting probabilities on the grammar productions, the sampling is biased towards producing programs from one part of the space or another. This technique is potentially completely general, since a grammar can theoretically be constructed to match any desired program domain. However, despite being theoretically possible, it is not easy to construct grammars which are both suitably general and also produce programs that are in any 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 [8, 9]. 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 with any desired granularity, 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 [10, 11]. As we show, deep learning is capable not just of learning the macro syntactical and semantic structure of programs, but also the nuances of how humans typically write code. It is truly remarkable when one considers that it is given no prior knowledge of the syntax or semantics of the language.

The second is the increasing popularity of public and open platforms for hosting software projects and source code. This popularity furnishes us with the thousands of programming examples that are necessary to feed into the deep learning. These open source examples are not, sadly, as useful for directly learning the compiler heuristics since they are not presented in a uniform, runnable manner, nor do they typically have extractable test data. Preparing each of the thousands of open source projects to be directly applicable for learning compiler heuristics would be an insurmountable task. In addition to our program generator, CLgen, we also provide an accompanying host driver which generates datasets for, then executes and profiles synthesized programs.

We make the following contributions:

- We are the first to apply deep learning over source codes to synthesize compilable, executable benchmarks.
- A novel tool CLgen for general-purpose benchmark synthesis using deep learning. CLgen automatically 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×.

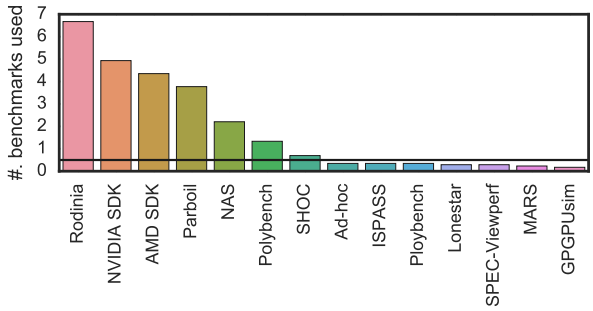


Figure 2: The average number of benchmarks used in GPGPU research papers, organized by origin. In this work we use the seven most popular benchmark suites.

2. Motivation

In this section we make the argument for synthetic benchmarks. We identified frequently used benchmark suites in a survey of 25 research papers in the field of GPGPU performance tuning from top tier conferences between 2013–2016: CGO, HiPC, PACT, and PPoPP. We found the average number of benchmarks used in a paper to be 17, and that a small pool of benchmarks suites account for the majority of results, shown in Figure 2. We selected the 7 most frequently used benchmark suites (used in 92% of results), and evaluated the performance of the state of the art *Grewe et al.* [12] predictive model across each. The model predicts whether running a given OpenCL kernel on the GPU gives better performance than on the CPU. We describe the full experimental methodology in Section 7.

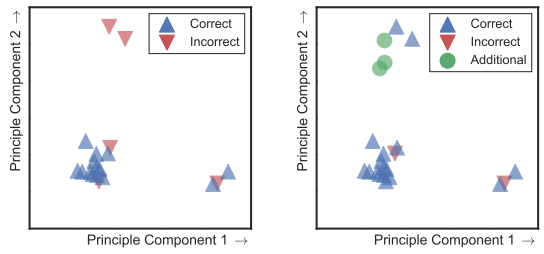
Table 1 summarizes our results. The performance of a model trained on one benchmark suite and used to predict the optimal mapping for another suites is generally very poor. The benchmark suite which provides the best results, NVIDIA SDK, achieves on average only 49% of the optimal performance. The worst case is when training with Parboil to predict the optimal mappings for Polybench, where the model achieves only 11.5% of the optimal performance. It is clear that heuristics learned on one benchmark suite fail to generalize across other suites.

This problem is caused both by the limited number of benchmarks contained in each suite, and the distribution of benchmarks within the feature space. Figure 3 shows the feature space of the Parboil benchmark suite, showing whether, for each benchmark, the model was able to correctly predict the appropriate optimization. We used Principle Component Analysis to reduce the multi-dimensional feature space to aid visualization.

As we see in Figure 3a, there is a dense cluster of neighboring benchmarks, a smaller cluster of three

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

Table 1: Performance relative to the optimal of the *Grewe et al.* predictive model across different benchmark suites on an AMD GPU. The columns show the suite used for training; the rows show the suite used for testing.



(a) (b)

Figure 3: A two dimensional projection of the *Grewe et al.* predictive model over Parboil benchmarks on an NVIDIA GPU. Two outliers in (a) are incorrectly predicted due to the lack of nearby observations. The additional neighboring observations in (b) corrects this.

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 3b, we train the predictive model with additional benchmarks which are neighboring in the feature space. The addition of these observations (and the information they provide about that part of the feature space) causes the two outliers to be correctly optimized. We found such outliers in all of the benchmark suites of Table 1.

These results highlight the significant affect 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. Our novel approach, described in the next section, solves this problem by generating an unbounded number of programs to cover the feature space at any desired granularity.

3. Overview of Our Approach

In this paper we present CLgen, a tool for synthesizing OpenCL benchmarks, and an accompanying host driver for executing synthetic benchmarks for gathering performance data for predictive modeling. While we demonstrate our approach using OpenCL, it is language agnostic. Our tool CLgen learns the semantics and structure from over a million lines of hand-written code from GitHub, and synthesizes programs through a process of iterative model sampling. We then use a host driver to execute the synthesized programs to gather performance data for use in predictive modeling. Figure 4 provides an overview of the program synthesis and execution pipeline. Our approach extends the state of the art by providing a general-purpose solution for benchmark synthesis, leading to better and more accurate predictive models.

In the course of evaluating our technique against prior work we discovered that it is also useful for evaluating the quality of features. Since we are able to cover the space so much more finely than the prior work, which only used standard benchmark suites, we are able to find multiple programs with identical feature values but different best heuristic values. This indicates that the features are not sufficiently discriminative and should be extended with more information to allow those programs to be separated. We go on to show that doing this significantly increases the performance of the learned heuristics. We expect that our technique will be valuable for feature designers.

4. CLgen: Benchmark Synthesis

CLgen is an undirected, general-purpose program synthesizer for OpenCL. It adopts and augments recent advanced techniques from deep learning to learn over massive codebases. In contrast to existing grammar and template based approaches, CLgen is entirely probabilistic. It *learns* to program using neural networks which model the semantics and usage of a huge corpus of code fragments in the target programming language. This section describes the assembly of an OpenCL language corpus, the application of deep learning over this corpus, and the process of synthesizing programs.

4.1 An OpenCL Language Corpus

Deep learning requires large datasets [13]. For the purpose of modeling a programming language, this means

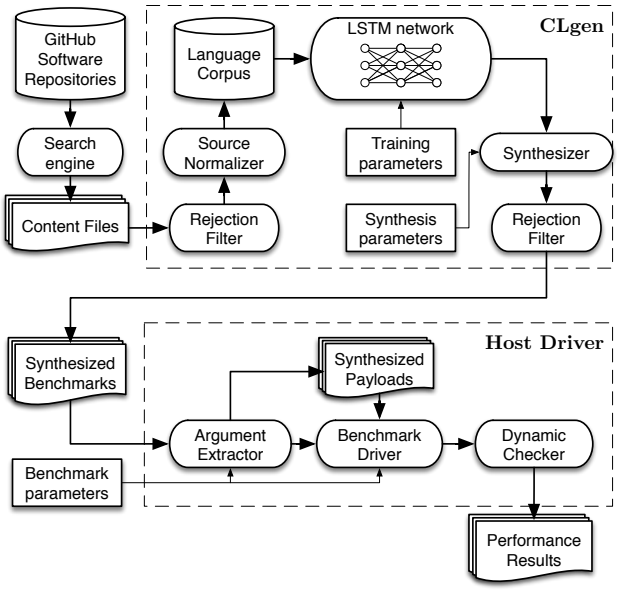


Figure 4: Benchmark synthesis and execution pipeline.

assembling a very large collection of real, hand-written source codes. We assembled OpenCL codes by mining public repositories on the popular code hosting site GitHub.

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

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

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

During initial development it became apparent that isolating the OpenCL device code leads to a higher-than-expected discard rate (that is, seemingly valid OpenCL files being rejected). Through analyzing 148k lines of compilation errors, we discovered a large number of failures caused by undeclared identifiers — a result of isolating device code — 50% of undeclared identifier errors in the GitHub dataset were caused by only 60

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

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

unique identifiers. To address this, we developed a *shim header* which contains inferred values for common type definitions (e.g. `FLOAT_T`), and common constants (e.g. `WGSIZE`), shown in Listing 1.

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

Code Rewriting 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 [14], our rewrite method preserves program behavior. Language builtins (e.g. `get_global_id`, `asin`) are not rewritten.
3. A variant of the Google C++ code style is enforced to ensure consistent use of braces, parentheses, and white space.

An example of the code rewriting process is shown in Figure 5. A side effect of this process is a reduction in code size, largely due to the removal of comments and excess white space. The final language corpus contains 1.3 million lines of transformed OpenCL, consisting of 9487 kernel functions. Identifier rewriting reduces the bag-of-words vocabulary size by 84%.

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

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

(a) Example content file

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

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

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

(b) Content file after code rewriting

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

4.2 Learning OpenCL

Generating valid, executable program code is an ambitious and challenging goal for 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 [15, 16] 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 [17], and an output layer providing normalized probability values from a 1-of-K coded vocabulary.

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

4.3 Synthesizing OpenCL

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

In either mode we generate a *seed* text, and sample the model, character by character, until the end of the compute kernel is reached, or until a predetermined maximum number of characters is reached. The same rejection filter described in Section 4.1 then either accepts or rejects the sample as a candidate synthetic benchmark. Listing 6 shows three examples of unique compute kernels generated in this manner from an argument specification of three single-precision floating-point arrays and a read-only signed integer. We evaluate the quality of synthesized code in Section 6.

5. Benchmark Execution

We developed a host driver to gather performance data from synthesized CLgen code. The driver accepts as input an OpenCL kernel, generates *payloads* of user-configurable sizes, and executes the kernel using the generated payloads, providing dynamic checking of kernel behavior.

5.1 Generating Payloads

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

5.2 Dynamic Checker

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

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

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

(a) Vector operation with branching and synchronization.

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

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

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

(c) Partial reduction over reinterpreted vector type.

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

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

Equality checks for floating point values are performed with an appropriate epsilon to accommodate rounding errors, and a timeout threshold is also used to catch kernels which are non-terminating. Our method is based on random differential testing [18], 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 [19–21], however we deemed such methods unnecessary for our purpose of performance modeling.

6. Evaluation of Synthetic Programs

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

6.1 Likeness to hand-written code

Judging whether a source code was 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* [22–24]. 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[1]. 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 [25], a program generator for differential testing.

[1] An online version of this test is available at *[URL redacted for double-blind review]*

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 2: *Grewe et al.* model features.

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. With CLgen synthesized programs, the average score was 52% (stdev. 17%). This demonstrates that CLgen code is indistinguishable from hand-written programs, with human judges scoring no better than random chance.

6.2 Limitations

Our new approach enables the synthesis of more human-like programs than current state of the art program generators, and without the expert guidance required by template based generators, but it has limitations. Currently we only run single-kernel benchmarks, and 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. The first limitation will be overcome by extending the host driver to explore multi-kernel schedules and interleaving of kernel executions. The second limitation can be addressed through recursive program synthesis, whereby a call to a user-defined function or type will trigger candidate functions and types to be synthesized.

7. Experimental Methodology

7.1 Experimental Setup

Predictive Model We reproduce the predictive model from Grewe, Wang, and O'Boyle [4, 12]. The predictive model is used to determine the optimal mapping of a given OpenCL kernel to either a GPU or CPU. It uses supervised learning to construct a decision tree with a combination of static and dynamic kernel features extracted from source code and the OpenCL runtime, detailed in Table 2b.

	Version	#. benchmarks	#. kernels
NPB (SNU [27])	1.0.3	7	114
Rodinia [28]	3.1	14	31
NVIDIA SDK	4.2	6	12
AMD SDK	3.0	12	16
Parboil [29]	0.2	6	8
PolyBench [30]	1.0	14	27
SHOC [31]	1.1.5	12	48
Total	-	71	256

Table 3: List of benchmarks.

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

Table 4: Experimental platforms.

Benchmarks As in [12], we test our model on the NAS Parallel Benchmarks (NPB) [26]. We use the hand-optimized OpenCL implementation of Seo, Jo, and Lee [27]. In [12] the authors augment the training set of the predictive model with 47 additional kernels taken from 4 GPGPU benchmark suites. To more fully sample the program space, we use a much larger collection of 142 programs, summarized in Table 3. These additional programs are taken from all 7 of the most frequently used benchmark suites identified in Section 2. We synthesized 1,000 kernels with CLgen to use as additional benchmarks.

Platforms We evaluate our approach on two 64-bit CPU-GPU systems, detailed in Table 4. One system has an AMD GPU and uses OpenSUSE 12.3; the other is equipped with an NVIDIA GPU and uses Ubuntu 16.04. Both platforms were unloaded.

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

7.2 Methodology

We replicated the methodology of [12]. 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-validatation* to evaluate predictive models. For each benchmark, a model is trained on data from all other benchmarks and used

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

Listing 2: Using the *Grewe et al.* features, this CLgen program is indistinguishable from AMD's Fast Walsh–Hadamard transform benchmark, but has very different runtime behavior. The addition of a branching feature fixes this.

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 prediction on synthetic benchmarks.

8. Experimental Results

We evaluate the effectiveness of our approach on two heterogeneous systems. We first compare the performance of a state of the art predictive model [12] with and without the addition of synthetic benchmarks, then show how the synthetic benchmarks expose weaknesses in the model's 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 [25].

8.1 Performance Evaluation

Figure 7 shows speedups of the *Grewe et al.* predictive model over the NAS Parallel Benchmark suite with and without the addition of synthesized benchmarks for training. Speedups are calculated relative to the best single-device mapping for each experimental platform, which is CPU-only for AMD and GPU-only for NVIDIA. The fine grained coverage of the feature space which synthetic benchmarks provide improves performance dramatically for the NAS benchmarks. Across both systems, we achieve an average speedup of 2.42× 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, a benchmark which suffers greatly under a single-device mapping. However, the performance on AMD for the same benchmark slightly degrades after adding the synthetic benchmarks, which we address in the next section.

8.2 Extending the Predictive Model

Feature designers are bound to select as features only properties which are significant for the sparse benchmarks they test on, which can limit a model's ability to generalize over a wider range of programs. We found this to be the case with the *Grewe et al.* model. The addition of automatically generated programs exposed two distinct cases where the model failed to generalize as a result of overspecializing to the NPB suite.

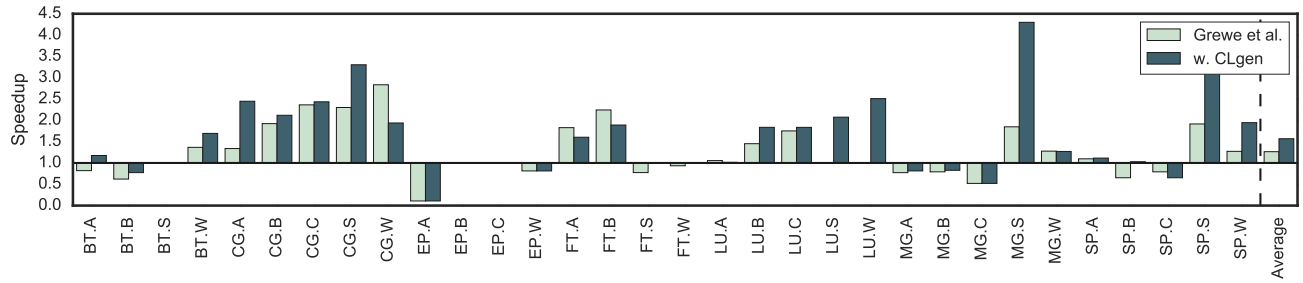
The first case is that `F3` is sparse on many programs. This a result of the NPB implementation's heavy exploitation of local memory buffers and the method by which they combined features (we speculate this was a necessary dimensionality reduction in the presence of sparse training programs). To counter this we extended the model to use the raw features in addition to the combined features.

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

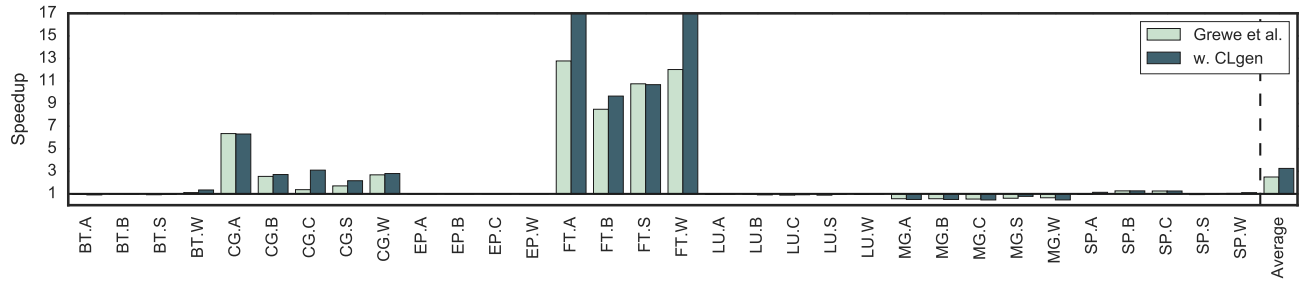
Figure 8 shows speedups of our extended model across all seven of the benchmark suites used in Section 2. Model performance, even on this tenfold increase of benchmarks, is good. There are three benchmarks on which the model performs poorly: `MatrixMul`, `cutcp`, and `pathfinder`. Each of those programs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

8.3 Comparison of Source Features

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



(a) AMD Tahiti 7970



(b) NVIDIA GTX 970

Figure 7: Speedup of programs using *Grewe et al.* predictive model with and without synthetic benchmarks. The predictive model outperforms the best device-only mapping by a factor of 1.26× on AMD and 2.50× on NVIDIA. The addition of synthetic benchmarks improves the performance to 1.57× on AMD and 3.26× on NVIDIA.

ate programs which are close in the feature space to the benchmarks which we are testing on.

To quantify this effect we use the static code features of Table 2a, plus the branching feature discussed in the previous subsection, to measure the number of CLgen kernels generated with the same feature values as those of the benchmarks we examined in the previous subsections. We examine only static code features to allow comparison with the GitHub kernels for which we have no automated method to execute them and extract runtime features, and CLSmith generated programs.

Figure 9 plots the number of matches as a function of the number of kernels. Out of 10,000 unique CLgen kernels, more than a third have static feature values matching those of the benchmarks, providing on average 14 CLgen kernels for each benchmark. This confirms our original intuition: CLgen kernels, by emulating the way real humans write OpenCL programs, are concentrated in the same area of the feature space as real programs. Moreover, the number of CLgen kernels we generate is unbounded, allowing us to continually refine the exploration of the feature space, while the number of kernels available on GitHub is finite. CLSmith rarely produces code similar to real-world OpenCL programs, with only 0.53% of the generated kernels have matching feature values with benchmark kernels. We conclude that the unique contribution of CLgen is its ability to generate many thousands of programs *that are appropriate for predictive modeling*.

9. Related Work

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

GENESIS [32] 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 [33, 34]. 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

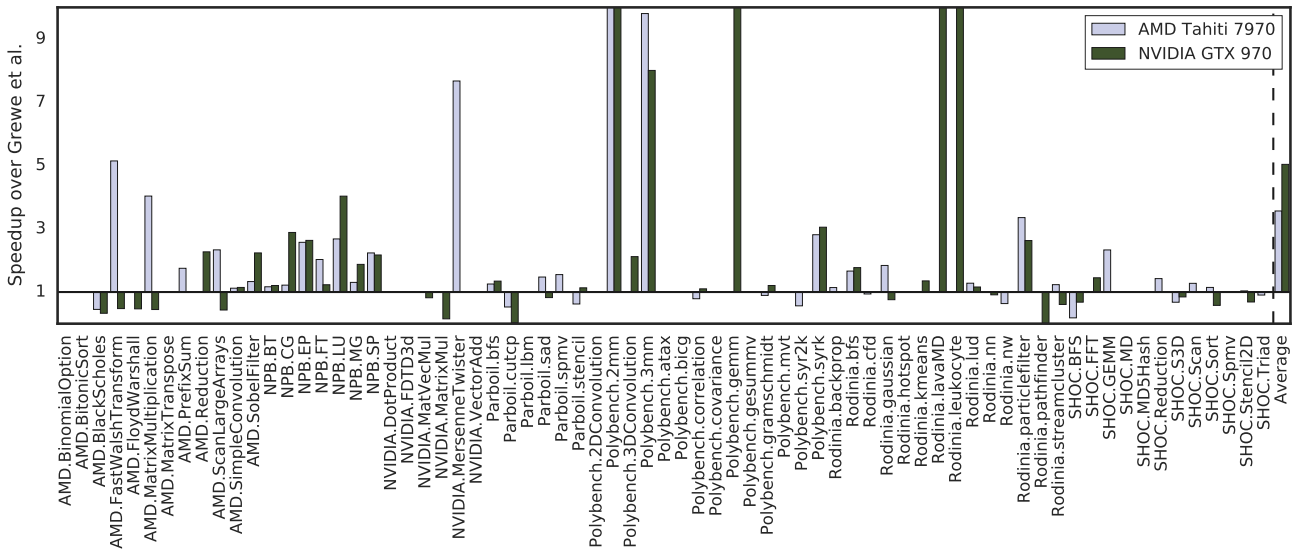


Figure 8: Speedups of predictions using our extended model over *Grewe et al.* on both experimental platforms. Synthetic benchmarks and the additional program features outperform the original predictive model by a factor 3.56× on AMD and 5.04× on NVIDIA.

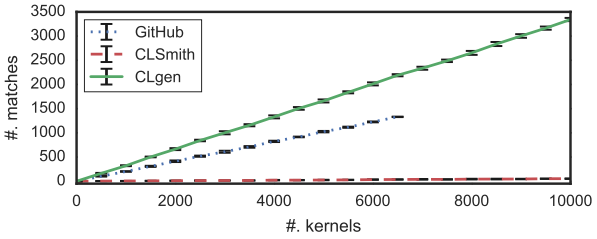


Figure 9: The number of kernels from GitHub, CLSmith, and CLgen with static code features matching the benchmarks. CLgen generates kernels that are closer in the feature space than CLSmith, and can continue to do so long after we have exhausted the extent of the GitHub dataset.

been developed for C [35] and OpenCL [25]. A mutation-based approach for the Java Virtual Machine is demonstrated in [36]. Goal-directed program generators have been used for a variety of domains, including generating linear transforms [37], MapReduce programs [38], and data structure implementations [39].

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 [40]. JSNice leverages probabilistic graphical models to predict program properties such as identifier names for Javascript [41].

There is an increasing interest in mining source code repositories at large scale [14, 42, 43]. Previous studies have involved data mining of GitHub to analyze software engineering practices [44–47], for example code generation [48], code summarization [49], comment generation [50], and code completion [51]. However, no work so far has exploited mined source code for benchmark generation. This work is the first to do so.

10. Conclusion

The quality of predictive models is bound by the quantity and quality of programs used for training, yet there is typically only a few dozen common benchmarks available for experiments. We present a novel tool which is the first of it's kind — an entirely probabilistic program generator capable of generating an unbounded number of human like programs. Our approach applies deep learning over a huge corpus of publicly available code from GitHub to automatically infer the semantics and practical usage of a programming language. Our tool generates programs which to trained eyes are indistinguishable from hand-written code. We tested our approach using a state of the art predictive model, improving its performance by a factor of 1.27×. We found that synthetic benchmarks exposed weaknesses in the feature set which, when corrected, further improved the performance by 4.30×. In future work we will extend CLgen to synthesize benchmarks in multiple programming languages, and investigate methods for performing an automatic directed search of the program space.

Acknowledgments

Our thanks to the volunteers at Codeplay Software Ltd and the University of Edinburgh for participating in the qualitative evaluation. This work was supported by the UK Engineering and Physical Sciences Research Council under grants EP/L01503X/1 (CDT in Pervasive Parallelism), EP/L000055/1 (ALEA), EP/M01567X/1 (SANDeRs), EP/M015823/1, and EP/M015793/1 (DIVIDEND). The code and data for this paper is available at `http://chriscummins.cc/cgo17`.

References

[1] Zheng Wang and Michael F.P. O'Boyle. "Mapping Parallelism to Multi-cores: A Machine Learning Based Approach". In: *PPoPP*. 2009.

[2] Zheng Wang, Georgios Tournavitis, Björn Franke, and Michael F. P. O'boyle. "Integrating Profile-driven Parallelism Detection and Machine-learning-based Mapping". In: *ACM Trans. Archit. Code Optim.* (2014).

[3] Zheng Wang and Michael F.P. O'Boyle. "Partitioning streaming parallelism for multi-cores: a machine learning based approach". In: *PACT*. 2010.

[4] Haichuan Wang and David Padua. "Optimizing R VM: Allocation Removal and Path Length Reduction via Interpreter-level Specialization". In: *CGO*. 2014.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[26] D. Bailey et al. "The NAS Parallel Benchmarks". In: *IJHPCA* (1991).

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

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

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

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

[31] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. "The Scalable HeterOgeneous Computing (SHOC) Benchmark Suite". In: *GPGPU* July 2016 (2010), pp. 63–74.

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

[33] J. D. Garvey and T. S. Abdelrahman. "Automatic Performance Tuning of Stencil Computations on GPUs". In: *ICPP* (2015), pp. 300–309.

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

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

[36] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao. "Coverage-Directed Differential Testing of JVM Implementations". In: *PLDI*. 2016, pp. 85–99.

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

[38] C. Smith. "MapReduce Program Synthesis". In: *PLDI*. 2016.

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

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

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

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

[43] E. Kalliamvakou, L. Singer, G. Gousios, D. M. German, K. Blincoe, and D. Damian. "The Promises and Perils of Mining GitHub". In: *MSR*. 2009, pp. 1–10.

[44] Y. Wu, J. Kropczynski, P. C. Shih, and J. M. Carroll. "Exploring the Ecosystem of Software Developers on GitHub and Other Platforms". In: *CSCW*. 2014, pp. 265–268.

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

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

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

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

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

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

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

A. Artifact description

A.1 Abstract

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

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

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

A.2 Description

A.2.1 Check-list (artifact meta information)

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

A.2.2 How delivered

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

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

A.3 Installation

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

A.3.1 Remote Access

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

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

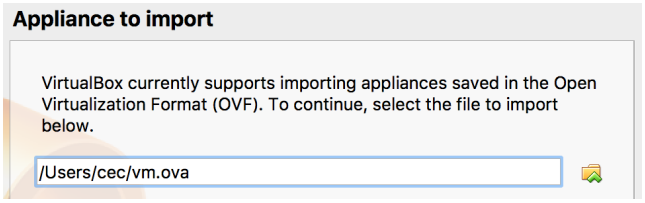
High system load may lead to inconsistent performance results; this may occur if multiple reviewers are accessing the server simultaneously.

A.3.2 Virtual Machine

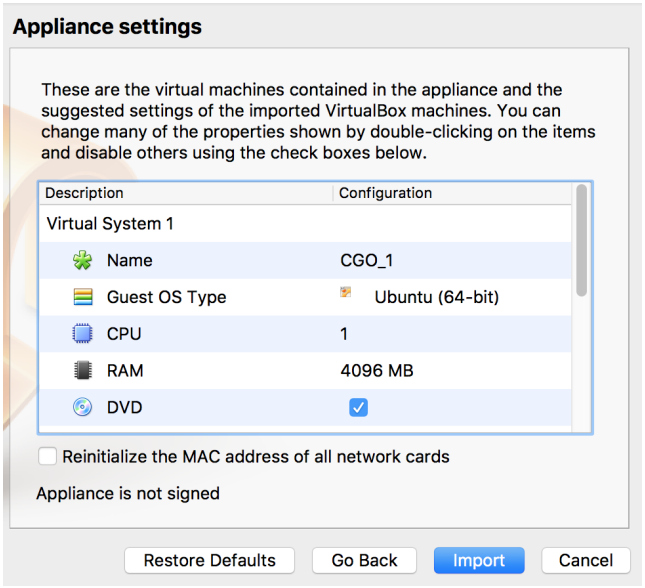
Copy our pre-configured 5.21 GB VirtualBox image using:

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

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



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



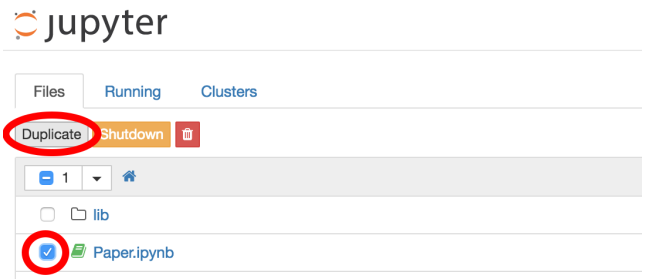
Start the machine and log in using username and password `cgo`. Once at the shell, run `launch`. This will start the Jupyter notebook server and print its address. You can access the notebooks at this address using the browser of the host device. Please note that the VirtualBox image does not have OpenCL,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

A.3.3 Local Install

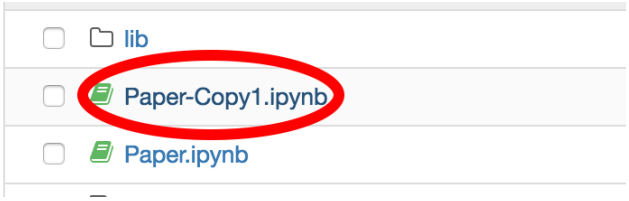
See `http://chriscummins.cc/cgo17/` for instructions. Note that we only support Ubuntu 16.04 or OS X, and sudo privileges are required to install the necessary requirements. Other Linux distributions may work but will require extra steps to install the correct package versions.

A.4 Experiment workflow

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

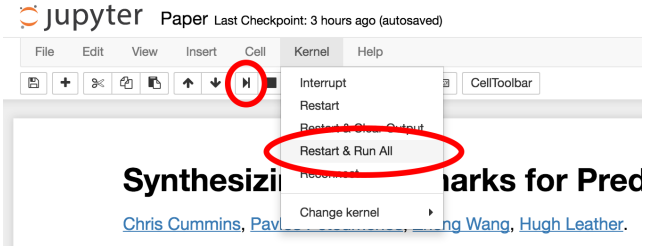


4. Click on the name of the newly created copy, e.g. `Paper-Copy1.ipynb` or `AE-Copy3.ipynb`.



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

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



A.5 Evaluation and expected result

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

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

A.6 Experiment customization

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

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

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

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

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

	AMD	NPB
AMD	1.00	0.38
NPB	0.23	1.00
	0.75	0.45
	0.77	0.37
	0.22	0.13
	0.46	0.16
	0.36	0.24

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

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

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

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

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

	AMD	NPB
AMD	1.00	0.21
NPB	0.65	1.00
	0.56	0.44
	0.85	0.21
	0.23	0.18
	0.39	0.45
	0.13	0.20

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

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

A.7 Notes

For more information about CLgen, visit:

`http://chriscummins.cc/clgen/`

Please report bugs and issues at:

`https://github.com/ChrisCummins/clgen/issues`