Synthesizing Benchmarks for Predictive Modeling

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

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

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

Abstract

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

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

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

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

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

Introduction

Predictive modeling is a well researched method for building optimization heuristics that often exceed human experts and reduces development time [1–11]. Figure 1 shows the process by which these models are trained. A set of training programs are identified which are expected to be representative of the application domain. The programs are compiled and executed with different parameter values for the target heuristic, to determine which are the best values for each training program. Each program is also summarized by a vector of features which describe the information that is expected to be important in predicting the best heuristic parameter values. These training examples of program features and desired heuristic values are used to create a machine learning model which, when given the features from a new, unseen program, can predict good heuristic values for it.

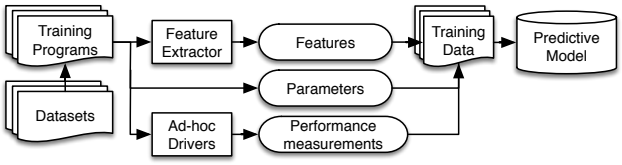


Figure 1. Training a predictive model.

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

There have been efforts to solve this problem using templates. The essence of the approach is to construct a probabilistic grammar with embedded semantic actions that defines a language of possible programs. New programs may be created by sampling the grammar and, through setting probabilities on the grammar productions, the sampling is biased towards producing programs from one part of the space or another. This technique is potentially completely general, since a grammar can theoretically be constructed to match any desired program domain. However, despite being theoretically possible, it is not easy to construct grammars which are both suitably general and also produce programs that are in any

way similar to human written programs. It has been shown to be successful over a highly restricted space of stencil benchmarks with little control flow or program variability [4, 8]. But, it is not clear how much effort it will take, or even if it is possible for human experts to define grammars capable of producing human like programs in more complex domains.

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

The first is the breakthrough effectiveness of deep learning for modeling complex structure in natural languages [12, 13]. As we show, deep learning is capable not just of learning the macro syntactical and semantic structure of programs, but also the nuances of how humans typically write code. It is truly remarkable when one considers that it is given no prior knowledge of the syntax or semantics of the language.

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

We make the following contributions:

- We are the first to apply deep learning over source codes to synthesize compilable, executable benchmarks.
- A novel tool CLgen[1] for general-purpose benchmark synthesis using deep learning. CLgen automatically and rapidly generates thousands of human like programs for use in predictive modeling.
- We use CLgen to automatically improve the performance of a state of the art predictive model by $1.27\times$, and expose limitations in the feature design of the model which, after correcting, further increases performance by $4.30\times$.

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

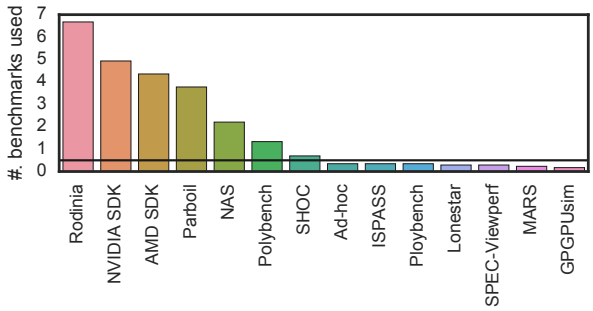


Figure 2. The average number of benchmarks used in GPGPU research papers, organized by origin. In this work we use the seven most popular benchmark suites.

Motivation

In this section we make the argument for synthetic benchmarks. We identified frequently used benchmark suites in a survey of 25 research papers in the field of GPGPU performance tuning from four top tier conferences between 2013–2016: CGO, HiPC, PACT, and PPoPP. We found the average number of benchmarks used in each paper to be 17, and that a small pool of benchmarks suites account for the majority of results, shown in Figure 2. We selected the 7 most frequently used benchmark suites (accounting for 92% of results), and evaluated the performance of the state of the art *Grewe et al.* [14] predictive model across each. The model predicts whether running a given OpenCL kernel on the GPU gives better performance than on the CPU. We describe the full experimental methodology in Section 7.

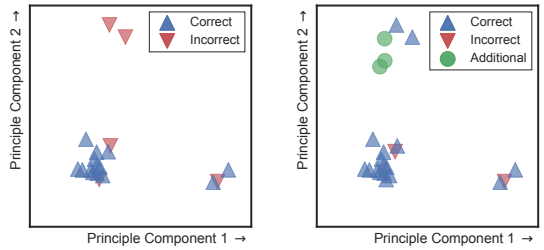
Table 1 summarizes our results. The performance of a model trained on one benchmark suite and used to predict the mapping for another suite is generally very poor. The benchmark suite which provides the best results, NVIDIA SDK, achieves on average only 49% of the optimal performance. The worst case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

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

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

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

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



(a) (b)

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

Figure 3b, we hand-selected benchmarks which are neighbouring in the feature space and retrained the model. The addition of these observations (and the information they provide about that part of the feature space) causes the two outliers to be correctly optimized. We found such outliers in all of the benchmark suites of Table 1.

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

Overview of Our Approach

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

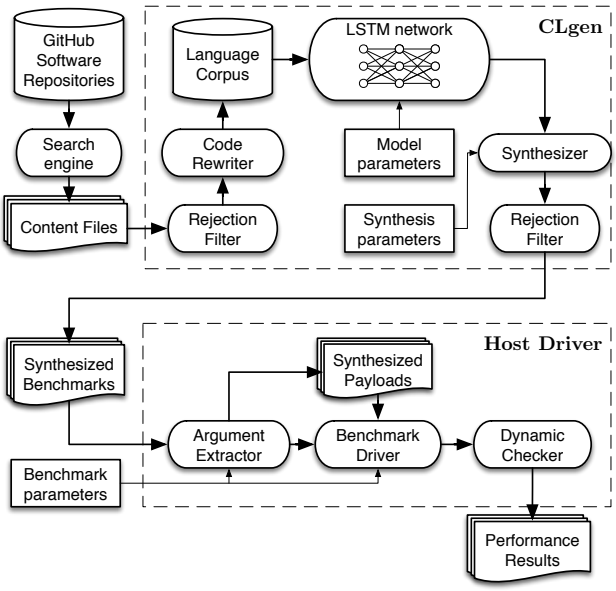


Figure 4. Benchmark synthesis and execution pipeline.

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

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

CLgen: Benchmark Synthesis

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

An OpenCL Language Corpus

Deep learning requires large datasets [15]. For the purpose of modeling a programming language, this means assembling a very large collection of real, hand-written source codes. We assembled OpenCL codes by mining public repositories on the popular code hosting site GitHub.

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

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

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

During initial development it became apparent that isolating the OpenCL device code leads to a higher-than-expected discard rate (that is, seemingly valid OpenCL files being rejected). Through analyzing 148k lines of compilation errors, we discovered a large number of failures caused by undeclared identifiers — a result of isolating device code — 50% of undeclared identifier errors in the GitHub dataset were caused by only 60 unique identifiers. To address this, we developed a *shim header* which contains inferred values for common type definitions (e.g. `FLOAT_T`), and common constants (e.g. `WGSIZE`), shown in Listing 1.

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

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

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

Code Rewriter Programming languages have few of the issues of semantic interpretation present in natural language, though there remains many sources of variance at the syntactic level. For example, the presence and content of comments in code, and the choice of identifying names given to variables. We consider these ambiguities to be *non-functional variance*, and developed a tool to normalize code of these variances so as to make the code more amenable to machine learning. This is a three step process:

1. The source is pre-processed to remove macros, conditional compilation, and source comments.
2. Identifiers are rewritten to have a short but unique name based on their order of appearance, using the sequential series $\{a, b, c, \ldots, aa, ab, ac, \ldots\}$ for variables and $\{A, B, C, \ldots, AA, AB, AC, \ldots\}$ for functions. This process isolates the syntactic structure of the code, and unlike prior work [16], our rewrite method preserves program behavior. Language built-ins (e.g. `get_global_id`, `asin`) are not rewritten.
3. A variant of the Google C++ code style is enforced to ensure consistent use of braces, parentheses, and white space.

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

Learning OpenCL

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

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

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

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

(a) Example content file

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

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

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

(b) Content file after code rewriting

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

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

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

Synthesizing OpenCL

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

In either mode we generate a *seed* text, and sample the model, character by character, until the end of the compute kernel is reached, or until a predetermined maximum number of characters is reached. Algorithm 1 illustrates this process.

Algorithm 1 Sampling a candidate kernel from a seed text.

Require: LSTM model M, maximum kernel length n.
Ensure: Completed sample string S.

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

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.

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.

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.

Dynamic Checker

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

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

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

(a) Vector operation with branching and synchronization.

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

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

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

(c) Partial reduction over reinterpreted vector type.

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

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

Equality checks for floating point values are performed with an appropriate epsilon to accommodate rounding errors, and a timeout threshold is also used to catch kernels which are non-terminating. Our method is based on random differential testing [20], though we emphasize that this is not a general purpose approach and is tailored specifically for our use case. For example, we anticipate a false positive rate for kernels with subtle sources of non-determinism which more thorough methods may expose [21–23], however we deemed such methods unnecessary for our purpose of performance modeling.

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.

Likeness to Hand-written Code

Judging whether a piece of code has been written by a human is a challenging task for a machine, so we adopt a methodology from machine learning research based on the *Turing Test* [24–26]. We reason that if the output of CLgen is human like code, then a human judge will be unable to distinguish it from hand-written code.

We devised a double blind test in which 15 volunteer OpenCL developers from industry and academia were shown 10 OpenCL kernels each. Participants were tasked with judging whether, for each kernel, they believed it to have been written by hand or by machine. Kernels were randomly selected for each participant from two equal sized pools of synthetically generated and hand-written code from GitHub. We applied the code rewriting process to all kernels to remove comments and ensure uniform identifier naming. The participants were divided into two groups, with 10 of them receiving code generated by CLgen, and 5 of them acting as a control group, receiving code generated by CLSmith [27], a program generator for differential testing[1].

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

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

Raw Code Features		
`comp`	static	#. compute operations
`mem`	static	#. accesses to global memory
`localmem`	static	#. accesses to local memory
`coalesced`	static	#. coalesced memory accesses
`transfer`	dynamic	size of data transfers
`wgsize`	dynamic	#. work-items per kernel

(a) Individual code features

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

(b) Combinations of raw features

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

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

Table 3. List of benchmarks.

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

Table 4. Experimental platforms.

ing outcome as generated programs for testing have multiple "tells", for example, their only input is a single `ulong` pointer. There were no false positives (synthetic code labeled human) for CLSmith, only false negatives (human code labeled synthetic). With CLgen synthesized programs, the average score was 52% (stdev. 17%), and the ratio of errors was even. This suggests that CLgen code is indistinguishable from hand-written programs, with human judges scoring no better than random chance.

Limitations

Our new approach enables the synthesis of more human-like programs than current state of the art program generators, and without the expert guidance required by template based generators, but it has limitations. Our method of seeding the language models with the start of a function means that we cannot support user defined types, or calls to user-defined functions. This means that we only consider scalars and arrays as inputs; while 6 (2.3%) of the benchmark kernels from Table 3 use irregular data types as inputs. We will address this limitation through recursive program synthesis, whereby a call to a user-defined function or unrecognized type will trigger candidate functions and type definitions to be synthesized. Currently we only run single-kernel benchmarks. We will extend the host driver to explore multi-kernel schedules and interleaving of kernel executions. Our host driver generates datasets from uniform random distributions, as do many of the benchmark suites. For cases where non-uniform inputs are required (e.g. profile-directed feedback), an alternate methodology for generating inputs must be adopted.

Experimental Methodology

Experimental Setup

Predictive Model We reproduce the predictive model from Grewe, Wang, and O'Boyle [14]. The predictive model is used to determine the optimal mapping of a given OpenCL kernel to either a GPU or CPU. It uses supervised learning to construct a decision tree with a combination of static and dynamic kernel features extracted from source code and the OpenCL runtime, detailed in Table 2b.

Benchmarks As in [14], we test our model on the NAS Parallel Benchmarks (NPB) [28]. We use the hand-optimized OpenCL implementation of Seo, Jo, and Lee [29]. In [14] the authors augment the training set of the predictive model with 47 additional kernels taken from 4 GPGPU benchmark suites. To more fully sample the program space, we use a much larger collection of 142 programs, summarized in Table 3. These additional programs are taken from all 7 of the most frequently used benchmark suites identified in Section 2. None of these programs were used to train CLgen. We synthesized 1,000 kernels with CLgen to use as additional benchmarks.

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

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

Methodology

We replicated the methodology of [14]. Each experiment is repeated five times and the average execution time is recorded. The execution time includes both device compute time and the data transfer overheads.

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

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

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

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

Experimental Results

We evaluate the effectiveness of our approach on two heterogeneous systems. We first compare the performance of a state of the art predictive model [14] with and without the addition of synthetic benchmarks, then show how the synthetic benchmarks expose weaknesses in the feature design and how these can be addressed to develop a better model. Finally we compare the ability of CLgen to explore the program feature space against a state of the art program generator [27].

Performance Evaluation

Figure 7 shows speedups of the *Grewe et al.* predictive model over the NAS Parallel Benchmark suite with and without the addition of synthesized benchmarks for training. Speedups are calculated relative to the best single-device mapping for each experimental platform, which is CPU-only for AMD and GPU-only for NVIDIA. The fine grained coverage of the feature space which synthetic benchmarks provide improves performance dramatically for the NAS benchmarks. Across both systems, we achieve an average speedup of $2.42\times$ with the addition of synthetic benchmarks, with prediction improvements over the baseline for 62.5% of benchmarks on AMD and 53.1% on NVIDIA.

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

Extending the Predictive Model

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

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

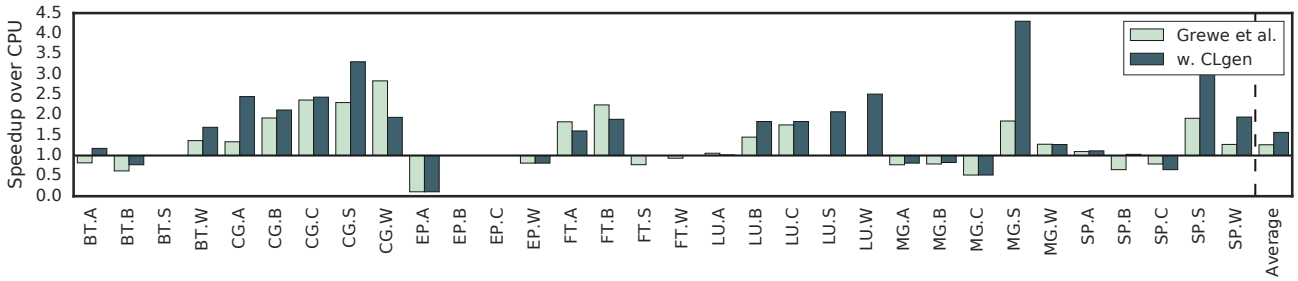
The second case is that some of our generated programs had identical feature values as in the benchmark set, but had different *behavior* (i.e. optimal mappings). Listing 2 shows one example of a CLgen benchmark which is indistinguishable in the feature space to one the of existing benchmarks — the Fast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

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

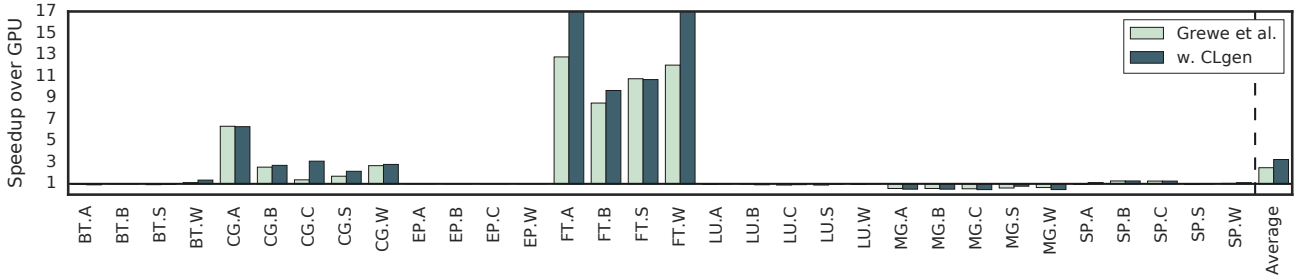
Comparison of Source Features

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

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



(a) AMD Tahiti 7970



(b) NVIDIA GTX 970

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

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

Related Work

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

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

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

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

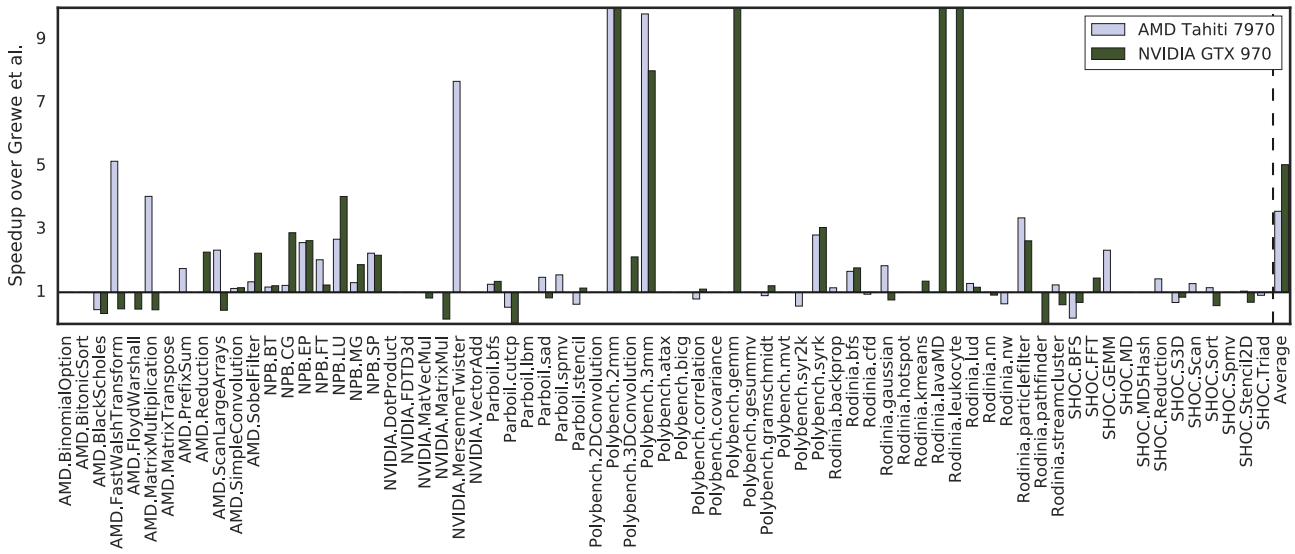


Figure 8. Speedups of predictions using our extended model over *Grewe et al.* on both experimental platforms. Synthetic benchmarks and the additional program features outperform the original predictive model by a factor $3.56\times$ on AMD and $5.04\times$ on NVIDIA.

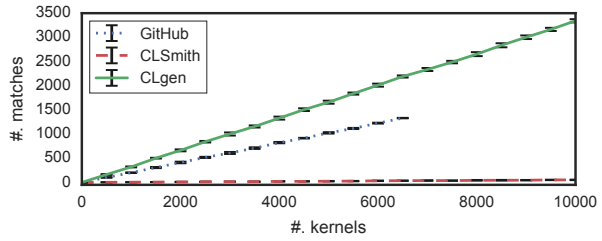


Figure 9. The number of kernels from GitHub, CLSmith, and CLgen with static code features matching the benchmarks. CLgen generates kernels that are closer in the feature space than CLSmith, and can continue to do so long after we have exhausted the extent of the GitHub dataset. Error bars show standard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

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

Conclusion

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

Acknowledgments

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[28] D. H. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. "The NAS Parallel Benchmarks". In: *IJHPCA* 5.3 (1991), pp. 63–73.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Artifact description

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.

Description

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/`

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.

Installation

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

Remote Access

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

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

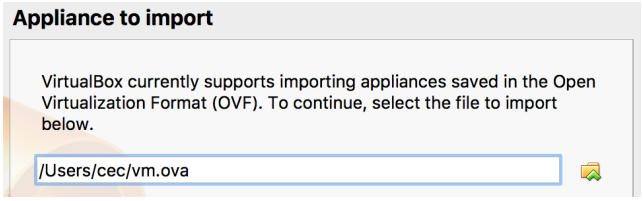
High system load may lead to inconsistent performance results; this may occur if multiple reviewers are accessing the server simultaneously.

Virtual Machine

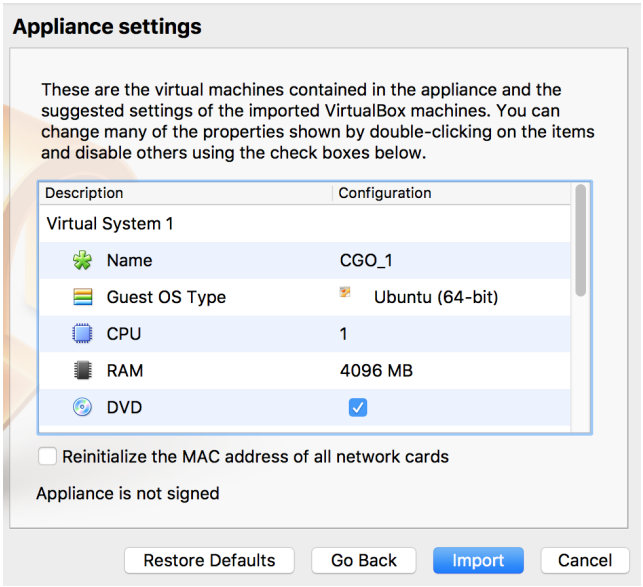
Copy our pre-configured 5.21 GB VirtualBox image using:

```
$ 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.

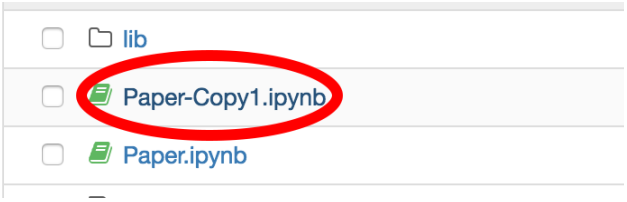
Local Install

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.

Experiment Workflow

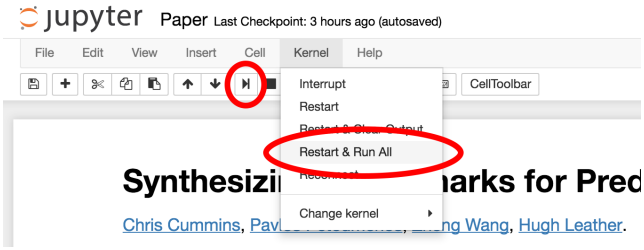
1. Access the Jupyter notebook server using one of the three options described in Section A.3.
2. From the Jupyter server page, tick the checkbox next to one of the two notebooks: `AE.ipynb` for minimal artifact reproduction or `Paper.ipynb` for a comprehensive interactive paper.
3. Click the button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



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.

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	0.77	0.22
0.45	0.37	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	0.85	0.23
0.44	0.21	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.

Notes

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