

ProGraML: Graph-based Deep Learning for Program Optimization and Analysis.

Chris Cummins
Facebook AI Research

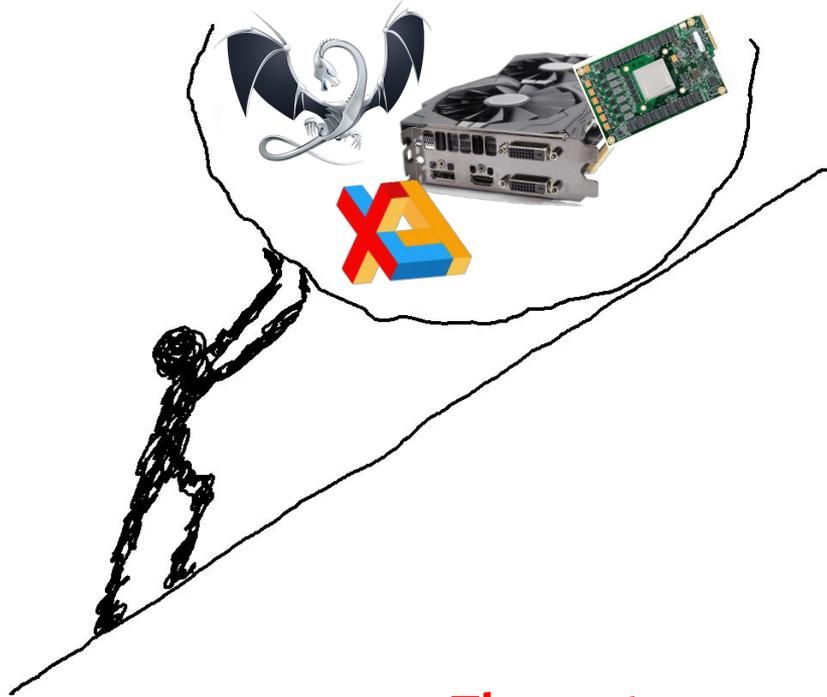
"machine learning for compilers for machine learning"

Compilers



**Machine
Learning**

Tuning optimizing compilers...



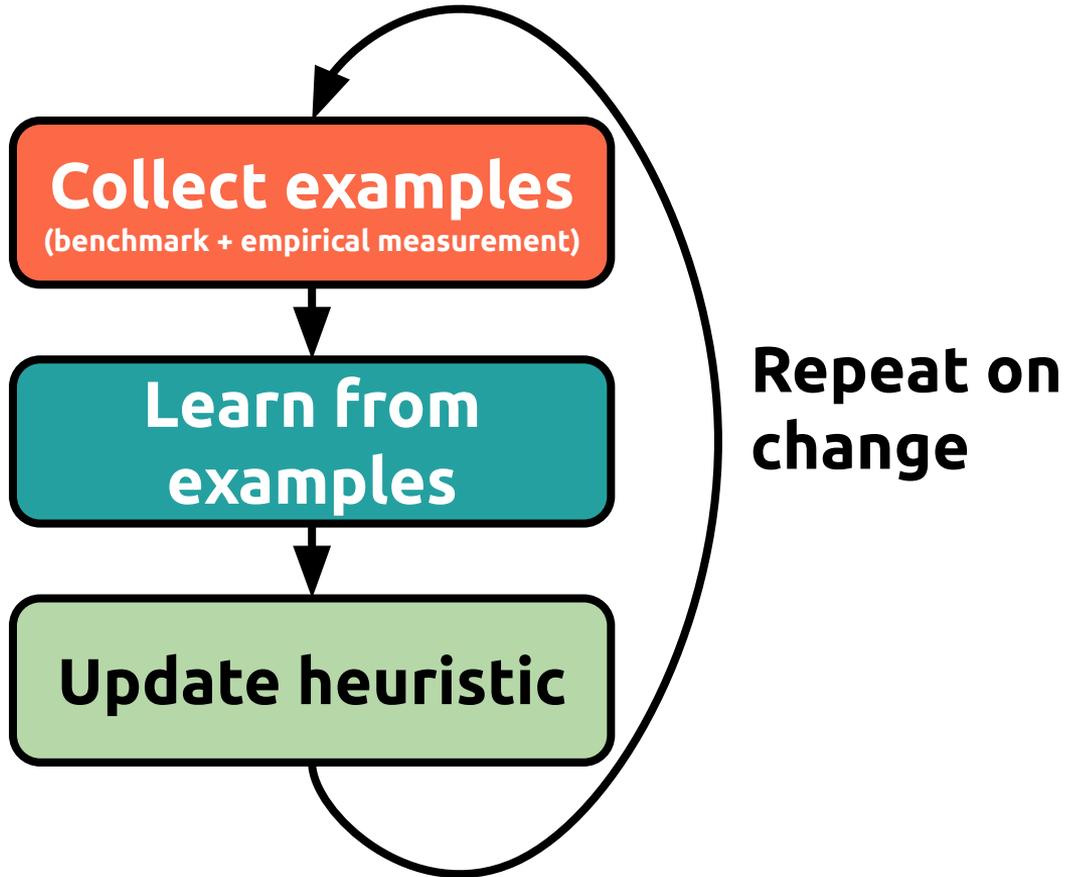
The problem

- 1000s of variables
- Limited by domain expertise
- Compiler / HW keeps changing

The cost

- Bad heuristics
- Wasted energy, \$\$\$
- Widening performance gap

"Build an optimizing compiler, your code will be fast for a day.
Teach a compiler to optimize ... "



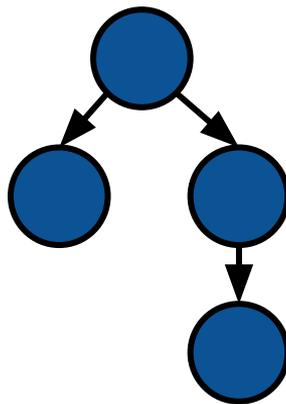
Summarize the program

Program

```
void LinearAlgebraOp<InputScalar,
OutputScalar>::AnalyzeInputs(
    OpKernelContext* context, TensorInputs* inputs,
    TensorShapes* input_matrix_shapes, TensorShape*
batch_shape) {
    int input_rank = -1;
    for (int i = 0; i < NumMatrixInputs(context); ++i) {
        const Tensor& in = context->input(i);
        if (i == 0) {
            input_rank = in.dims();
            OP_REQUIRES(
                context, input_rank >= 2,
                errors::InvalidArgument(
                    "Input tensor ", i,
                    " must have rank >= 2"));
        }
    }
}
```



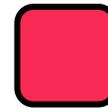
IR (CFG, DFG, AST,...)



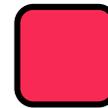
Features



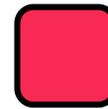
#. instructions



loop nest level



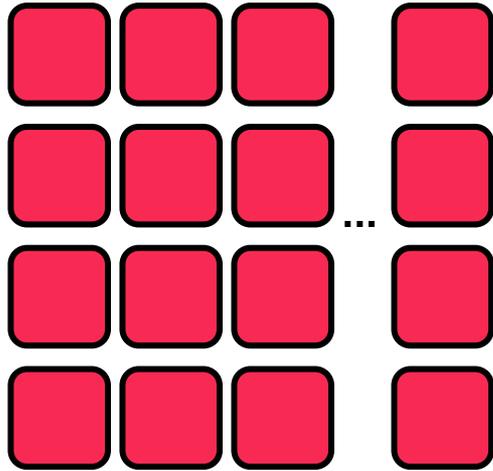
arithmetic density



trip counts

Collect examples

Features

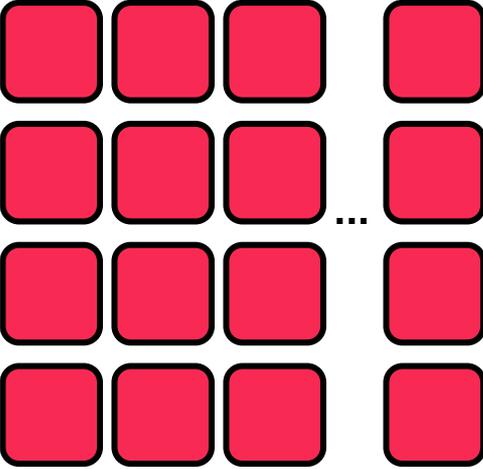


Best Param

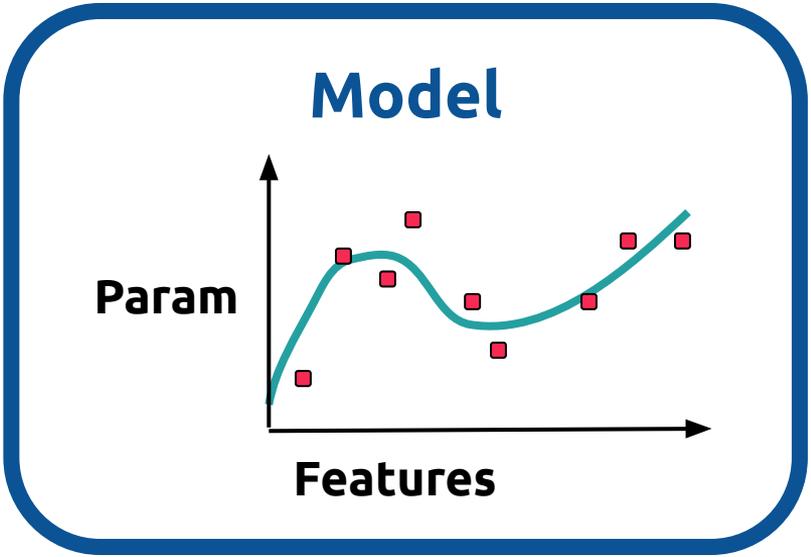
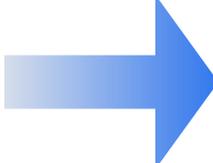


Learn from examples

Features



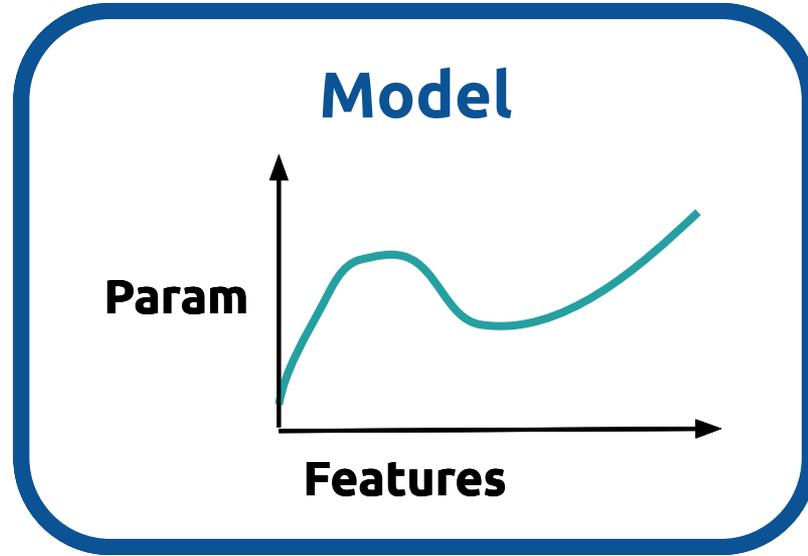
Supervised
Machine
Learner



Best Param

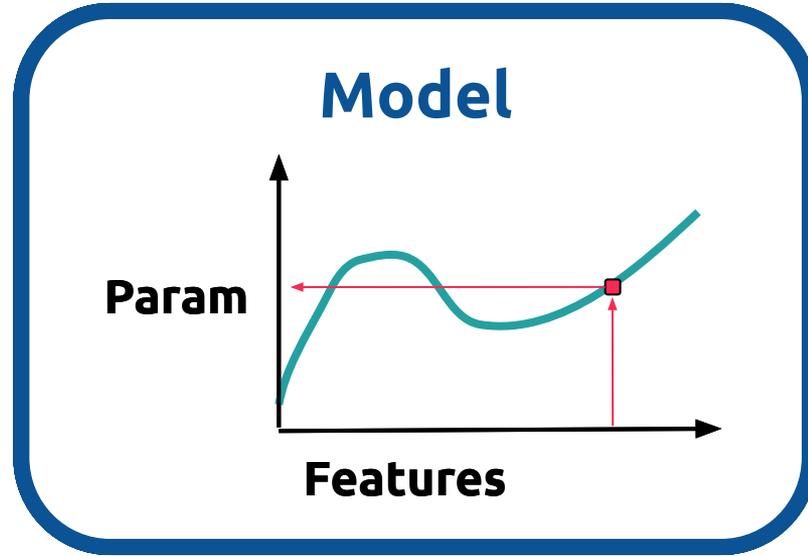
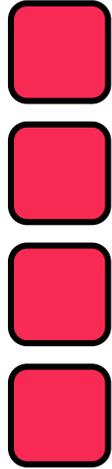


The model is the heuristic

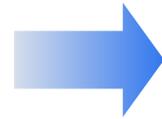


The model is the heuristic

New Program
Features

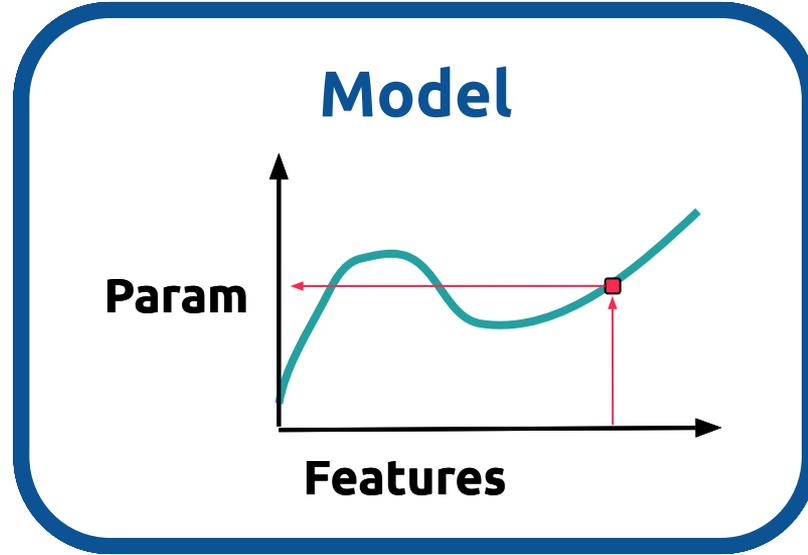
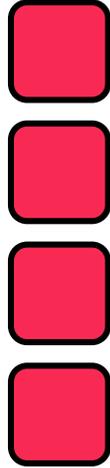


Predicted
param

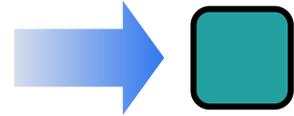


The model is the heuristic

New Program
Features



Predicted
param



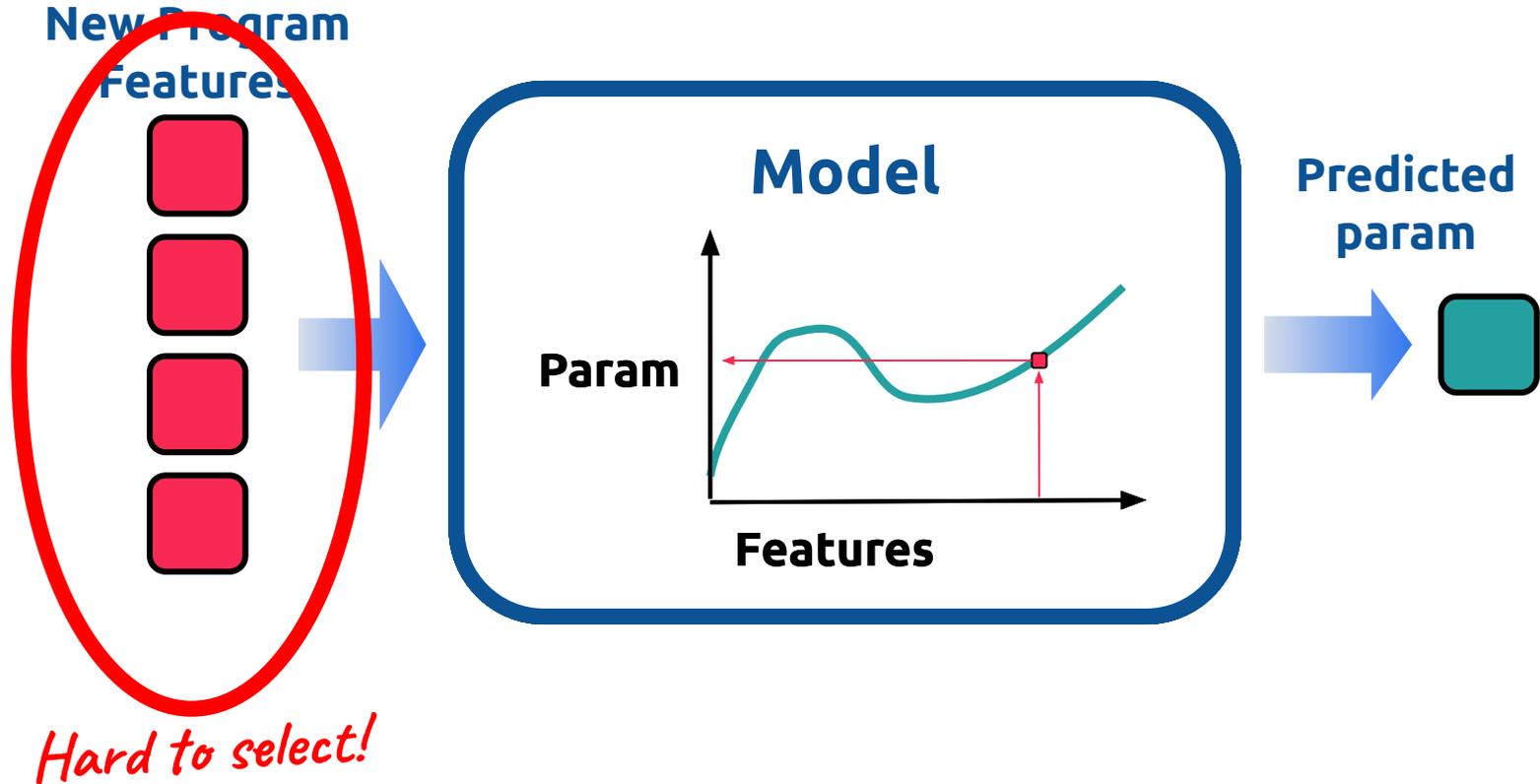
Very successful!

Huge performance gains to be had. Typically outperforms human expert.

[[Wang et. al. 2018](#)]

**Why aren't our
compilers full of
ML?**

The model is the heuristic



Learning without features

(Cummins et al., PACT 17)
"End-to-end Deep Learning of
Optimization Heuristics"

1. Input

```
kernel void A(global float* a, const float b) {  
    a[get_global_id(0)] *= 3.14 + b;  
}
```

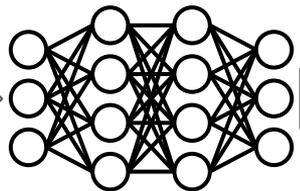
2. Vocab

Token	Index
kernel	0
[space]	1
void	2
A	3
(4
global	5
float	6
*	7
a	8

Token	Index
,	9
const	10
b	11
)	12
{	13
\n	14
[15
get_global_id	16
0	17

181 tokens

3. Encoded



LSTM

Optimization
Decision

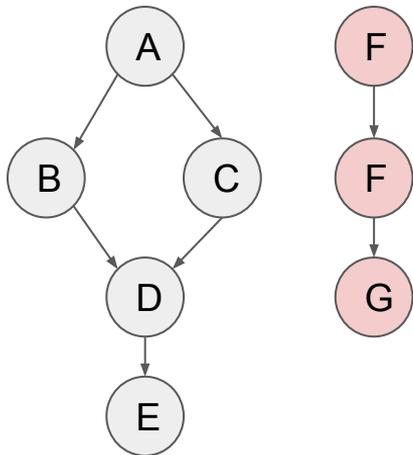


The problem with code representations

Source code is *highly structured*

It isn't a vector of numbers

Feature vectors are easy to fool
(e.g. insert **dead code**).



It isn't a sequence of tokens

Sequential representations fail on
non-linear relations, **long-range** deps.

```
void A(int a) {  
    int b = init();  
    //  
    // ... 1000 lines  
    //  
    //  
    return b - a;  
}
```

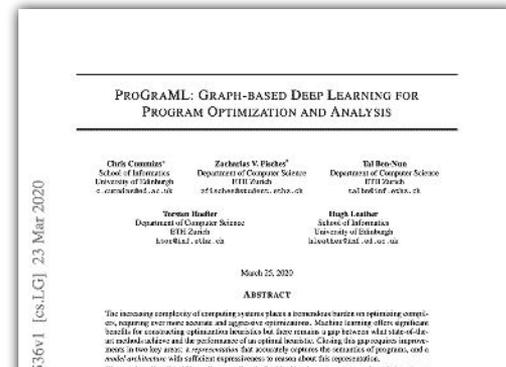
**Can we make ML
think like a
compiler?**

Program Graphs for Machine Learning

General-purpose representation of programs for optimization tasks.

Task independent - capture structured relations fundamental to program reasoning (i.e. data flow analysis)

Language independent - derived from compiler IRs



Building ProGraML: IR

Derive **IR** from input program (here, LLVM)

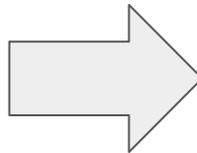
Why IR?

Language **agnostic**

(e.g. C, C++, OpenCL, Swift,
Haskell, Java for LLVM)

We want to improve
compiler decisions, so
use a **compiler's eye** view.

```
int Fib(int x) {  
  switch (x) {  
    case 0:  
      return 0;  
    case 1:  
      return 1;  
    default:  
      return Fib(x - 1)  
         + Fib(x - 2);  
  }  
}
```



```
define i32 @Fib(i32) #0 {  
  switch i32 %0, label %3 [  
    i32 0, label %9  
    i32 1, label %2  
  ]  
  
; <label>:2:  
br label %9  
  
; <label>:3:  
%4 = add nsw i32 %0, -1  
%5 = tail call i32 @Fib(i32 %4)  
%6 = add nsw i32 %0, -2  
%7 = tail call i32 @Fib(i32 %6)  
%8 = add nsw i32 %7, %5  
ret i32 %8  
  
; <label>:9:  
%10 = phi i32 [ 1, %2 ], [ %0, %1 ]  
ret i32 %10  
}
```

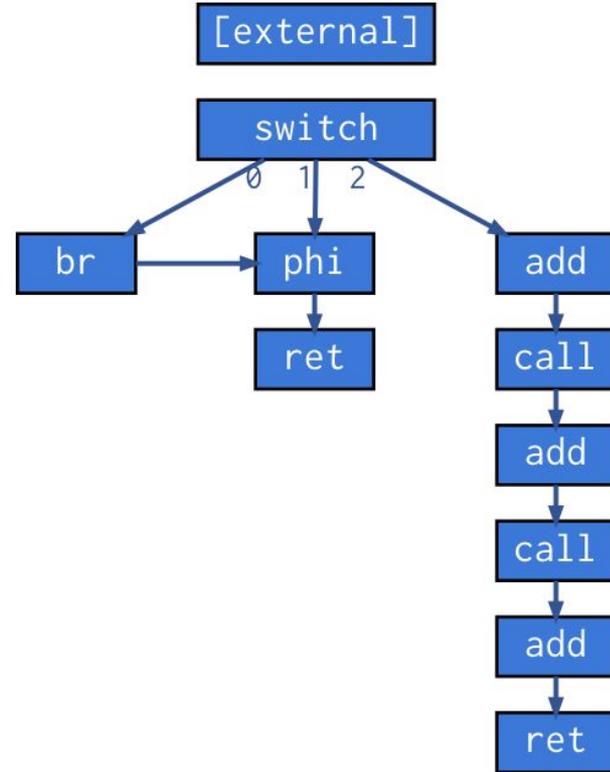
Building ProGraML: Control-flow

Full-flow-graph: represent
each instruction as a vertex.

Vertex label is the **instruction name**.

Edges are **control-flow**.

Edge position attribute for
branching control-flow.



Learning with ProGraML: Node Embeddings

Use vertex labels as embedding keys



Derive vocab from set of unique vertex labels on **training graphs**.

Separate type/instruction nodes leads to **compact vocab**,
excellent coverage on unseen programs compared to prior approaches:

	Vocabulary size	Test coverage
inst2vec [12]	8,565	34.0%
CDFG [14]	75	47.5%
PROGRAML	2,230	98.3% *without types

[inst2vec](#): combined instruction+operands

```
i32 <id> = a<id> <int8>
```

[CDFG](#): uses only instructions for vocab, ignores data

Learning with ProGraML: GGNNs

Message Passing

$$M(h_w^{t-1}, e_{wv}) = W_{\text{type}(e_{wv})} \left(h_w^{t-1} \odot p(e_{wv}) \right) + b_{\text{type}(e_{wv})}$$

6 typed weight matrices for
{forwards, backwards} {control, data, call}
edge types

Position gating to differentiate
control branches and operand order

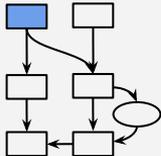
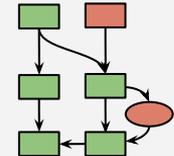
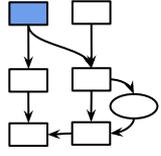
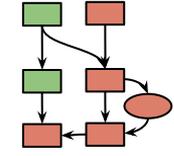
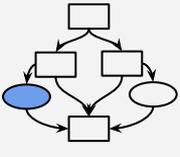
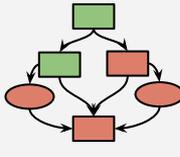
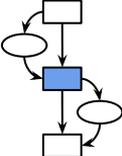
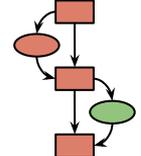
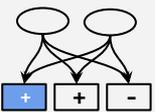
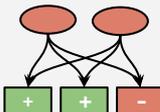
Readout Head

$$R(h_v^T, h_v^0) = \sigma(f(h_v^T, h_v^0)) \cdot g(h_v^T)$$

per-vertex prediction after T
message-passing steps

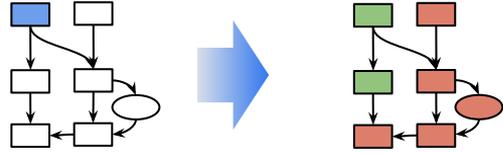
Deep Data Flow

Dataset: 450k LLVM-IRs covering 5 programming languages

				F1 scores		
				inst2vec	CDFG	ProGraML
Reachability Trivial forwards control-flow E.g. dead code elimination				0.012	0.998	0.998
Dominance Forwards control-flow E.g. global code motion				0.004	0.999	1.000
Data Dependencies Forwards data-flow E.g. instruction selection				-	-	0.997
Live-out Variables Backwards control- and data-flow E.g. register allocation				-	-	0.937
Global Common Subexpressions Instruction/operand sensitive E.g. GCS Elimination				0.000	0.009	0.996

Deep Data Flow

Dataset: 450k LLVM-IRs covering 5 programming languages

		F1 scores		
		inst2vec	CDFG	ProGraML
Reachability Trivial forwards control-flow E.g. dead code elimination		0.012	0.998	0.998
Dominance Forwards control-flow E.g. global code motion		0.004	0.999	1.000
Data Dependencies Forwards data-flow E.g. instruction selection		-	-	0.997
Live-out V Backwards control- ar E.g. register		-	-	0.937
Global Common Subexpressions Instruction/operand sensitive E.g. GCS Elimination		0.000	0.009	0.996

inst2vec/CDFG are instruction-level representations, can't reason about variables

Caveat: limited problem size

Data flow analyses iterate until a fixed point is reached.

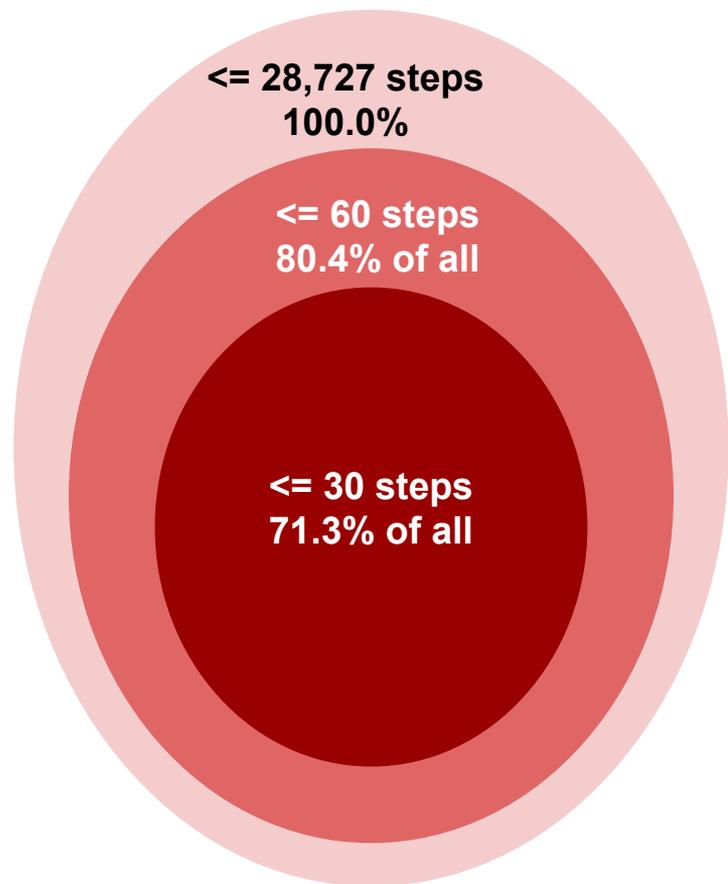
GGNNs iterate for a fixed number of timesteps T .

For each example in the train/test sets, we count the number of steps required for an iterative analysis to solve.

We then filter the train/test set to include only examples which the iterative analysis required $\leq T$ steps to solve.

Previous slide was $T=30$, excluding 28.7% of examples.

Next slide shows performance models, trained on $T=30$, with different inference steps ($T=60$, $T=200$).



Scaling to larger problems

Dataset: 450k LLVM-IRs covering 5 programming languages

F1 scores

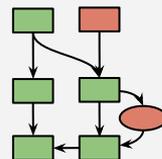
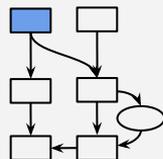
30
timesteps

60 timesteps

200
timesteps

Reachability

Trivial forwards control-flow
E.g. dead code elimination



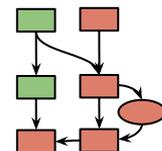
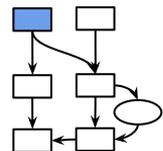
0.998

0.997

0.943

Dominance

Forwards control-flow
E.g. global code motion



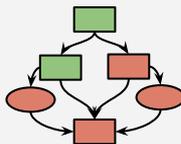
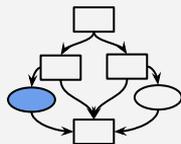
1.000

0.991

0.123

Data Dependencies

Forwards data-flow
E.g. instruction selection



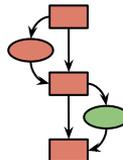
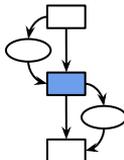
0.997

0.993

0.965

Live-out Variables

Backwards control- and data-flow
E.g. register allocation



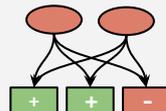
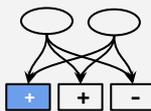
0.937

0.939

0.625

Global Common Subexpressions

Instruction/operand sensitive
E.g. GCS Elimination



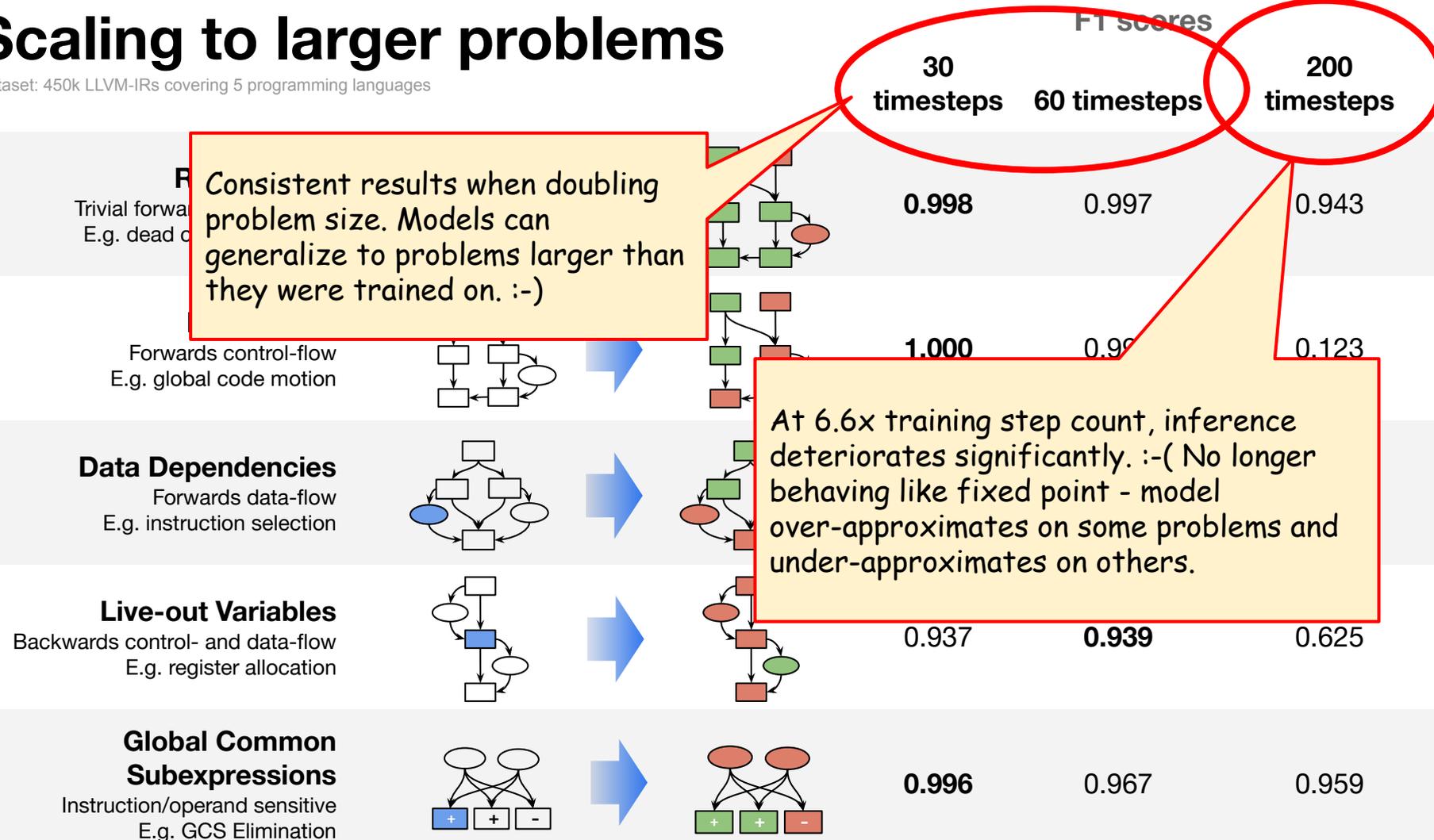
0.996

0.967

0.959

Scaling to larger problems

Dataset: 450k LLVM-IRs covering 5 programming languages

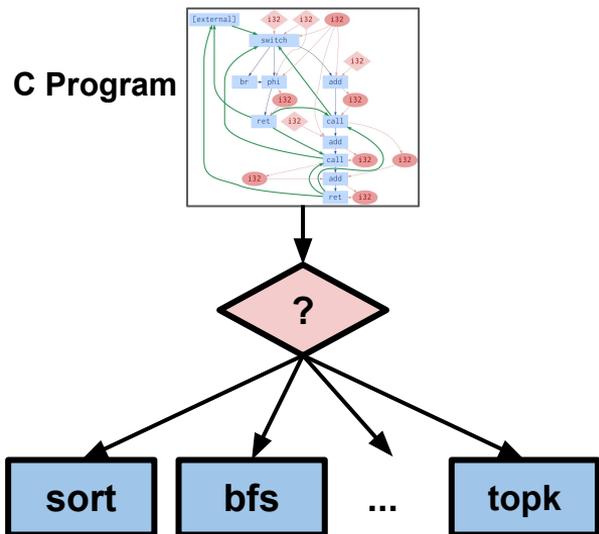


Consistent results when doubling problem size. Models can generalize to problems larger than they were trained on. :-)

At 6.6x training step count, inference deteriorates significantly. :((No longer behaving like fixed point - model over-approximates on some problems and under-approximates on others.)

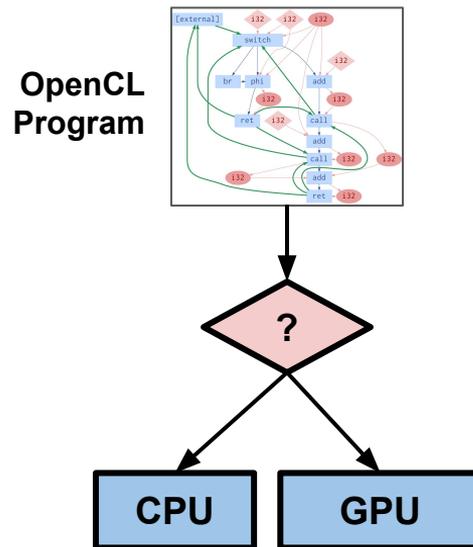
Downstream tasks

1. Algorithm Classification



1.35× improvement over state-of-art

2. Heterogeneous Device Mapping



1.20× improvement over state-of-art

Further Reading

arXiv:2003.10536v1 [cs.LG] 23 Mar 2020

PROGRAML: GRAPH-BASED DEEP LEARNING FOR PROGRAM OPTIMIZATION AND ANALYSIS

Chris Cummins¹
School of Informatics
University of Edinburgh
c.cummins@ed.ac.uk

Zacharias V. Zachariades²
Department of Computer Science
ETH Zurich
zachariades@ethz.ch

Tal Ben-Nun³
Department of Computer Science
ETH Zurich
talb@inf.ethz.ch

Thomas Heuser¹
Department of Computer Science
ETH Zurich
thuser@inf.ethz.ch

High Leasher¹
School of Informatics
University of Edinburgh
h.leasher@ed.ac.uk

March 25, 2020

ABSTRACT

The increasing complexity of computer systems places a tremendous burden on optimizing compilers, requiring ever more accurate and aggressive optimization. Machine learning offers significant benefits for generating optimization heuristics for these systems as pay-beneits and state-of-the-art methods achieve and the performance of an optimal heuristic. Choosing the appropriate improvements, on a two-by-two axis, of representations that accurately capture the semantics of programs, and a model selection strategy, with sufficient expressiveness, to reason about this optimization.

We reimplement ProGraML, a Program Graph for Machine Learning—a novel graph-based program representation using a low-level, language-agnostic, and portable format, and machine learning models capable of performing complex, domain-specific tasks over these graphs. The ProGraML representation is a directed, attributed multigraph that captures control, data, and call relations, and maintains instructions and operand types and registers. Machine Learning Neural Networks process information through this structured representation, enabling domain programs, per-instruction, and per-operand classification tasks. ProGraML, as a compact, unambiguous representation with support currently for LLVM and XLA IRs.

ProGraML provides a graph-processor program representation that applies learnable models to perform the types of program analysis that are fundamental to optimization. To this end, we include the performance of our approach on a wide range of real-world compiler analysis tasks: control flow, reaching, dominated nodes, data dependencies, variable liveness, and control reduction—state-of-the-art. On a benchmark dataset of 200k LLVM IRs, this covering six source programming languages, ProGraML achieves an average F1 score, significantly outperforming the state-of-the-art approaches. We then apply our approach to two high-level tasks—language-agnostic decompiling and program classification—setting a new state-of-the-art performance in both.

1 Introduction

The landscape of computing ecosystems is becoming increasingly complex: multi-core and many-core processors, heterogeneous systems, distributed and cloud platforms. Meanwhile computing performance and energy benefits from equating the those in these the capabilities of ever-progressive. In such an environment, high-quality optimization heuristics are not just desirable, they are required. Despite this, good optimization heuristics are hard to come by.

¹Work done while at Google

Preprint

<https://arxiv.org/abs/2003.10536>

ProGraML [Download](#)

2020.05.13

Graph [Download](#) [Shareable URL](#)

Entire Program

```
node {  
  text: "croots"  
}  
}  
node {  
  text: "switch"  
  features {  
    feature {  
      key: "full_text"  
      value {  
        bytes_list {  
          value: "switch 132 00, label 43 [v  
        ]  
      }  
    }  
  }  
}  
}  
node {  
  text: "br"  
  block: 1  
  features {  
    feature {  
      key: "full_text"  
      value {  
        bytes_list {  
          value: "br label 40"  
        }  
      }  
    }  
  }  
}  
node {  
  text: "add"  
  block: 2  
  features {  
    feature {  
      key: "full_text"  
      value {  
        bytes_list {  
          value: "add + add new 132 00, -1"  
        }  
      }  
    }  
  }  
}  
node {  
  text: "call"  
}
```

In-browser demo

https://chriscummins.cc/s/program_explorer

ChrisCummins ProGraML

Code Issues Pull requests Actions Projects 2 Security 4 Insights Settings

Branch: master [Go to file](#) [Add file](#) [Clone](#)

ChrisCummins committed 8 days ago

- deeplearning
- labml
- notebooks
- program
- third_party
- tools
- bazelrc
- bettercodehub.yml
- env
- gigignore
- travis.yml
- BUILD
- CONTRIBUTING.md
- DEPS.txt
- INSTALL.md
- LICENSE
- README.md
- WORKSPACE
- requirements.txt
- version.txt

deeplearning! Make a couple of targets public. 22 days ago

labml! Make labml/cppcrypto library public. 2 days ago

notebooks! Add install instructions to notebooks. 2 months ago

program! Add missing newline in error message. 2 days ago

third_party! Check in pybind11_base sources directly. 11 days ago

tools! tools/base/Env: Add a link to bug tracker. 9 days ago

bazelrc! Small comment wording improvements. 2 months ago

bettercodehub.yml! Complete ProGraML import. 2 months ago

env! Document implicit dependency on GNU coreutils. 3 months ago

gigignore! Don't track CLAN project files. 3 months ago

travis.yml! Complete ProGraML import. 10 days ago

BUILD! Remove cruft in top-level BUILD file. 9 days ago

CONTRIBUTING.md! Minor typos. 4 months ago

DEPS.txt! Add comment for python-dev requirement. 5 months ago

INSTALL.md! Two small corrections to INSTALL. 2 months ago

LICENSE! Complete ProGraML import. 10 days ago

README.md! program! Describe graph construction in README. 2 months ago

WORKSPACE! program! Update LLVM version to 10.0.0. 2 days ago

requirements.txt! labml/app: Add a -pybind hook to app launch. 21 days ago

version.txt! tools/format: Add runtime checks to formatters. 4 days ago

README.md

ProGraML: Program Graphs for Machine Learning

Build [Success](#) Better code [4/10](#) Issues [New](#) [20/20](#) Rep size [48.8 MB](#) Current activity [0/0/0/0](#)

ProGraML is a representation for programs as input to a machine learning model.

Key features are:

- Expressiveness: We represent programs as graphs, capturing all of the control, data, and call relations. Each node in the graph represents an instruction, variable, or constant, and edges are positional such that non-commutative operations can be differentiated.
- Portability: ProGraML is derived from compiler IRs, making it independent of the source language (e.g. we

Source code + datasets

<https://github.com/ChrisCummins/ProGraML>

Apache 2.0

Conclusions

Reasoning about programs requires the right combination of representation + model.

ProGraML: combines control-, data-, call-, and type-graphs to model programs at IR level.

When processed with GGNNs, significantly outperforms prior approaches.

Interesting challenges

1. Processing **arbitrary sized** graphs.

Idea: Structure the MPNN like an iterative DF solver, self-terminating.

2. Handling **unbounded vocabularies**, e.g. compound types or MLIR dialects.

Idea: decompose types into tree structure in graph.

3. Representing **literal values**.

Requires new vocabulary encoding.