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

Chris Cummins
Facebook AI Research
“machine learning for compilers for 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 ...

Collect examples
(benchmark + empirical measurement)

Learn from examples

Update heuristic

Repeat on change
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"));
        }
    }
}
Collect examples

Features

Best Param
Learn from examples

Features

Supervised Machine Learner

Best Param

Model

Param

Features
The model is the heuristic.
The model is the heuristic
The model is the heuristic

New Program Features

Model

Features

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

Hard to select!
Learning without features

1. Input

```c
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    |

3. Encoded

0 1 2 1 3 4 5 1

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

```c
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.

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

```llvm
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.
Building ProGraML: Data-flow

Add graph vertices for **constants** (diamonds) and **variables** (oblongs).

Edges are **data-flow**.

Edge position attribute for **operand order**.
Building ProGraML: Call-flow

Edges are call-flow.

Inbound edge to function entry instruction.

Outbound edge from (all) function exit instruction(s).
Building ProGraML: Types

Nodes represent **types**, Edges are **instances**.

Types are **composable**. Edge position per field.

```c
struct S {
    char a;
    char b;
    struct S* c;
};
```
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%         |
| PROGARML             | 2,230           | 98.3% *without types |

\textbf{inst2vec}: combined instruction+operands

\textbf{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

\[ R(h_v^T, h_v^0) = \sigma \left( f(h_v^T, h_v^0) \right) \cdot g(h_v^T) \]

- Per-vertex prediction after \( T \) message-passing steps
### Deep Data Flow

Dataset: 450k LLVM-IRs covering 5 programming languages

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

Dataset: 450k LLVM-IRs covering 5 programming languages

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

**Dominance**
Forwards control-flow
E.g. global code motion

**Data Dependencies**
Forwards data-flow
E.g. instruction selection

**Live-out Variables**
Backwards control- and data-flow
E.g. register allocation

**Global Common Subexpressions**
Instruction/operand sensitive
E.g. GCS Elimination

---

|                  | inst2vec | CDFG | ProGraML |
|------------------|----------|------|----------|
| **Reachability** | 0.012    | 0.998| 0.998    |
| **Dominance**    | 0.004    | 0.999| 1.000    |
| **Data Dependencies** | -       | -    | 0.997    |
| **Live-out Variables** | -      | -    | 0.937    |
| **Global Common Subexpressions** | 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 ($T=60$, $T=200$).
| Category                        | Description                                      | 30 timesteps | 60 timesteps | 200 timesteps |
|--------------------------------|--------------------------------------------------|--------------|--------------|---------------|
| **Reachability**               | Trivial forwards control-flow                    | 0.998        | 0.997        | 0.943         |
|                                | E.g. dead code elimination                       |              |              |               |
| **Dominance**                  | Forwards control-flow                            | 1.000        | 0.991        | 0.123         |
|                                | E.g. global code motion                          |              |              |               |
| **Data Dependencies**          | Forwards data-flow                               | 0.997        | 0.993        | 0.965         |
|                                | E.g. instruction selection                       |              |              |               |
| **Live-out Variables**         | Backwards control- and data-flow                 | 0.937        | 0.939        | 0.625         |
|                                | E.g. register allocation                         |              |              |               |
| **Global Common Subexpressions**| Instruction/operand sensitive                     | 0.996        | 0.967        | 0.959         |
|                                | E.g. GCS Elimination                             |              |              |               |

Dataset: 450k LLVM-IRs covering 5 programming languages

Scaling to larger problems
Scaling to larger problems

Dataset: 450k LLVM-IRs covering 5 programming languages

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

Forwards control-flow
E.g. global code motion

Data Dependencies
Forwards data-flow
E.g. instruction selection

Live-out Variables
Backwards control- and data-flow
E.g. register allocation

Global Common Subexpressions
Instruction/operand sensitive
E.g. GCS Elimination

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.

F1 scores

30 timesteps: 0.998, 0.997, 0.943
60 timesteps: 1.000, 0.991, 0.123
200 timesteps: 0.937, 0.939, 0.625
Downstream tasks

1. Algorithm Classification

C Program

? → sort, bfs, ... → topk

1.35× improvement over state-of-art

2. Heterogeneous Device Mapping

OpenCL Program

? → CPU, GPU

1.20× improvement over state-of-art
Further Reading

Preprint  
https://arxiv.org/abs/2003.10536

In-browser demo  
https://chriscummins.cc/s/program_explorer

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.