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
Summarize the program

Program

```cpp
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();
    } else {
      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
The model is the heuristic
The model is the heuristic

New Program Features

Model

Features

Param

Predicted param
The model is the heuristic

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

New Program Features

Model

Features

Param

Hard to select!

Predicted param
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

<table>
<thead>
<tr>
<th>Token</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>kernel</td>
<td>0</td>
</tr>
<tr>
<td>[space]</td>
<td>1</td>
</tr>
<tr>
<td>void</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>(</td>
<td>4</td>
</tr>
<tr>
<td>global</td>
<td>5</td>
</tr>
<tr>
<td>float</td>
<td>6</td>
</tr>
<tr>
<td>*</td>
<td>7</td>
</tr>
<tr>
<td>a</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Token</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>,</td>
<td>9</td>
</tr>
<tr>
<td>const</td>
<td>10</td>
</tr>
<tr>
<td>b</td>
<td>11</td>
</tr>
<tr>
<td>)</td>
<td>12</td>
</tr>
<tr>
<td>{</td>
<td>13</td>
</tr>
<tr>
<td>\n</td>
<td>14</td>
</tr>
<tr>
<td>[</td>
<td>15</td>
</tr>
<tr>
<td>get_global_id</td>
<td>16</td>
</tr>
<tr>
<td>0</td>
<td>17</td>
</tr>
</tbody>
</table>

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);
    }
}
```

```assembly
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.
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:

<table>
<thead>
<tr>
<th>Method</th>
<th>Vocabulary size</th>
<th>Test coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>inst2vec [12]</td>
<td>8,565</td>
<td>34.0%</td>
</tr>
<tr>
<td>CDFG [14]</td>
<td>75</td>
<td>47.5%</td>
</tr>
</tbody>
</table>
| ProGraML     | 2,230           | **98.3%** *without types*

**inst2vec**: combined instruction+operands

**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

**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

<table>
<thead>
<tr>
<th></th>
<th>inst2vec</th>
<th>CDFG</th>
<th>ProGraML</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reachability</td>
<td>0.012</td>
<td>0.998</td>
<td>0.998</td>
</tr>
<tr>
<td>Dominance</td>
<td>0.004</td>
<td>0.999</td>
<td>1.000</td>
</tr>
<tr>
<td>Data Dependencies</td>
<td>-</td>
<td>-</td>
<td>0.997</td>
</tr>
<tr>
<td>Live-out Variables</td>
<td>-</td>
<td>-</td>
<td>0.937</td>
</tr>
<tr>
<td>Global Common Subexpressions</td>
<td>0.000</td>
<td>0.009</td>
<td>0.996</td>
</tr>
</tbody>
</table>
# Deep Data Flow

Dataset: 450k LLVM-IRs covering 5 programming languages

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

*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*

<table>
<thead>
<tr>
<th></th>
<th>Reachability</th>
<th>Dominance</th>
<th>Data Dependencies</th>
<th>Live-out Variables</th>
<th>Global Common Subexpressions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Trivial forwards control-flow</strong></td>
<td>0.998</td>
<td>1.000</td>
<td>0.997</td>
<td>0.937</td>
<td>0.996</td>
</tr>
<tr>
<td><strong>E.g. dead code elimination</strong></td>
<td>0.997</td>
<td>0.991</td>
<td>0.993</td>
<td>0.939</td>
<td>0.967</td>
</tr>
<tr>
<td><strong>E.g. global code motion</strong></td>
<td>0.943</td>
<td>0.123</td>
<td>0.965</td>
<td>0.625</td>
<td>0.959</td>
</tr>
<tr>
<td><strong>Forwards control-flow</strong></td>
<td>0.998</td>
<td>1.000</td>
<td>0.997</td>
<td>0.937</td>
<td>0.996</td>
</tr>
<tr>
<td><strong>E.g. global code motion</strong></td>
<td>0.997</td>
<td>0.993</td>
<td>0.993</td>
<td>0.939</td>
<td>0.967</td>
</tr>
<tr>
<td><strong>Forwards data-flow</strong></td>
<td>0.943</td>
<td>0.123</td>
<td>0.965</td>
<td>0.625</td>
<td>0.959</td>
</tr>
<tr>
<td><strong>E.g. instruction selection</strong></td>
<td>0.997</td>
<td>1.000</td>
<td>0.997</td>
<td>0.937</td>
<td>0.996</td>
</tr>
<tr>
<td><strong>Backwards control- and data-flow</strong></td>
<td>0.943</td>
<td>0.123</td>
<td>0.965</td>
<td>0.625</td>
<td>0.959</td>
</tr>
<tr>
<td><strong>E.g. register allocation</strong></td>
<td>0.997</td>
<td>0.993</td>
<td>0.993</td>
<td>0.939</td>
<td>0.967</td>
</tr>
</tbody>
</table>

**Instruction/operand sensitive**

**E.g. GCS Elimination**
Scaling to larger problems

Dataset: 450k LLVM-IRs covering 5 programming languages

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

F1 scores
- 30 timesteps: 0.998
- 60 timesteps: 0.997
- 200 timesteps: 0.943

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

+ + +

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

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.