Deep Learning for Compilers

Hugh Leather
University of Edinburgh
Overview

- Machine Learning for Compilers
- Generating Benchmarks
- Deep Learned Heuristics
- Deep Fuzzing Compiler Testing
- Future Work
Chris Cummins
University of Edinburgh

Pavlos Petoumenos
University of Edinburgh

Zheng Wang
Lancaster University

Hugh Leather
University of Edinburgh
Overview

- Machine Learning for Compilers
- Generating Benchmarks
- Deep Learned Heuristics
- Deep Fuzzing Compiler Testing
- Future Work
Compilers are hard

Huge number of variables
NP-hard or worse
Keep changing

Nondeterministic machines
Many components
Keep changing
Compilers are hard

Slow programs

Energy waste

OUT OF DATE
Machine Learning to the Rescue

Collect examples

Learn from examples

Update heuristic

Rerun on change
Summarise the Program

Program

int main(int argc, char** argv) {
    printf("Hello, World!");
    return 0;
}

IR
(AST, CFG, DDG, etc.)

Features

- Number of instructions
- Mean dependency depth
- Trip count
- Loop nest level
Gather Examples

Features

Best parameters
Learn a Model

Features

Best parameters

Supervised Machine Learner

Model
What is a Model?

Optimisation

Decision

Fit a curve to examples
What is a Model?

The curve is the model.
What is a Model?

Prediction is looking up value on curve

Prediction

New Program
Use the Model

New Program Features

Model

Predicted parameters
Overview

- Machine Learning for Compilers
- Generating Benchmarks
- Deep Learned Heuristics
- Deep Fuzzing Compiler Testing
- Future Work
What we want.
What we get.
Learn the Wrong Thing!

- Use a CPU
- Use a GPU
- Target decision boundary
- Learned decision boundary
1. more benchmarks = better models
problem statement

1. more benchmarks = better models
2. there aren’t enough benchmarks

92% of results
problem statement

1. more benchmarks = better models
2. there aren’t enough benchmarks

avg compiler paper = 17
Iris dataset = 150
MNIST dataset = 60,000
ImageNet dataset = 10,000,000
1. more benchmarks = better models
2. there aren’t enough benchmarks
3. benchmarks must be diverse

<table>
<thead>
<tr>
<th></th>
<th>AMD</th>
<th>NPB</th>
<th>NVIDIA</th>
<th>Parboil</th>
<th>Polybench</th>
<th>Rodinia</th>
<th>SHOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD</td>
<td>-</td>
<td>38.0%</td>
<td>74.5%</td>
<td>76.7%</td>
<td>21.7%</td>
<td>45.8%</td>
<td>35.9%</td>
</tr>
<tr>
<td>NPB</td>
<td>22.7%</td>
<td>-</td>
<td>45.3%</td>
<td>36.7%</td>
<td>13.4%</td>
<td>16.1%</td>
<td>23.7%</td>
</tr>
<tr>
<td>NVIDIA</td>
<td>29.9%</td>
<td>37.9%</td>
<td>-</td>
<td>21.8%</td>
<td>78.3%</td>
<td>18.1%</td>
<td>63.2%</td>
</tr>
<tr>
<td>Parboil</td>
<td>89.2%</td>
<td>28.2%</td>
<td>28.2%</td>
<td>-</td>
<td>41.3%</td>
<td>73.0%</td>
<td>33.8%</td>
</tr>
<tr>
<td>Polybench</td>
<td>58.6%</td>
<td>30.8%</td>
<td>45.3%</td>
<td>11.5%</td>
<td>-</td>
<td>43.9%</td>
<td>12.1%</td>
</tr>
<tr>
<td>Rodinia</td>
<td>39.8%</td>
<td>36.4%</td>
<td>29.7%</td>
<td>36.5%</td>
<td>46.1%</td>
<td>-</td>
<td>59.9%</td>
</tr>
<tr>
<td>SHOC</td>
<td>42.9%</td>
<td>71.5%</td>
<td>74.1%</td>
<td>41.4%</td>
<td>35.7%</td>
<td>81.0%</td>
<td>-</td>
</tr>
</tbody>
</table>
1. more benchmarks = better models
2. there aren’t enough benchmarks
3. benchmarks must be diverse
Contributions

Human-like program generator

Model produces code 4.3x faster than state of the art
old approach

- Training programs
- Datasets
- Ad-hoc Drivers
- Training Data
- Predictive Model
our approach

Program Generator → Automatic Driver → Training Data → Predictive Model

Dataset Generator → Automatic Driver
our approach

mine code from web
our approach

model source distr.
our approach

sample lang. model
Infer the common usage of a PL from samples.

Huge repository of public knowledge: 🐱

And they have an API :-) 2.8 million lines of OpenCL

$ curl https://api.github.com/search/repositories?q=opencl&sort=stars&order=desc
{
  "total_count": 3155,
  "incomplete_results": false,
  "items": [
    {
      "id": 7296244,
      "name": "lwjgl3",
      "full_name": "LWJGL/lwjgl3",
    }
  ]
}
140821
MODEL

Scrape Github

Homogeniser

1 4 0 8 21...

Tokeniser

Language corpus

Synthesiser

Filter

Driver

Performance Results
/* Copyright (C) 2014, Joe Blogs. */
#define CLAMPING
#define THRESHOLD_MAX 1.0f
float myclamp(float in) {
#if defined CLAMPING
    return in > THRESHOLD_MAX ? THRESHOLD_MAX : in < 0.0f ? 0.0f : in;
#else
    return in;
#endif // CLAMPING
}
__kernel void findAllNodesMergedAabb(__global float* in, __global float* out, int numelems) {
    // Do something really flipping cool
    int id = get_global_id(0);
    if (id < numelems) {
        out[id] = myclamp(in[id]);
    }
}
float myclamp(float in) {
    #ifdef CLAMPING
        return in > THRESHOLD_MAX ? THRESHOLD_MAX : in < 0.0f ? 0.0f : in;
    #else
        return in;
    #endif // CLAMPING
}

__kernel void findAllNodesMergedAabb(__global float* in, __global float* out, int num elems) {
    // Do something really flipping cool
    int id = get_global_id(0);
    if (id < num elems) {
        out[id] = myclamp(in[id]);
    }
}
float A(float a) {
    return a > 1.0f ? 1.0f : a < 0.0f ? 0.0f : a;
}

__kernel void B(__global float* b, __global float* c, int d) {
    int e = get_global_id(0);
    if (e < d) {
        c[e] = A(b[e]);
    }
}
kernel void A(global float* a, const float b) {
    a[get_global_id(0)] *= 3.14 + b;
}

Vocab:

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

<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>
</tbody>
</table>

Encoded: 0 1 2 1 3 4 5 1
kernel void A(global float* a, const float b) {
    a[get_global_id(0)] *= 3.14 + b;
}
```c
kernel void A(global float* a, const float b) {
    a[get_global_id(0)] *= 3.14 + b;
}
```

<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>
</tbody>
</table>

**Vocab:**

**Encoded:** [0, 1, 2, 1, 3, 4, 5, 1, 6, 7]
kernel void A(global float* _a, const float b) {
    a[get_global_id(0)] *= 3.14 + b;
}
kernel void A(global float* \( \textbf{a} \), const float b) {
    a[get_global_id(0)] *= 3.14 + b;
}
kernel void A(global float* a, const float b) {
    a[get_global_id(0)] *= 3.14 + b;
}
Scrape Github

Homogeniser

Tokeniser

Language corpus

1 4 0 8 2 1 ...

Synthesiser

Filter

Driver

Performance Results

MODEL
neural network

Input: 30M token corpus
Learns probability distribution over corpus.
< 500 lines of code, 12 hours training on GPU.
1. Seed the model with the start of a program.
2. Predict tokens until \{ \} brackets balance.

**Decoded:**

kernel void
1. Seed the model with the start of a program.
2. Predict tokens until \{ \} brackets balance.
1. Seed the model with the start of a program.
2. Predict tokens until {} brackets balance.

Decoded:
kernel void A
1. Seed the model with the start of a program.
2. Predict tokens until \{ \} brackets balance.

Decoded: kernel void A

Input: 0 1 2 1 3

Output:

kernel ' ' void A ( global int double float ...
1. Seed the model with the start of a program.
2. Predict tokens until {} brackets balance.

Decoded: kernel void A
1. Seed the model with the start of a program.
2. Predict tokens until { } brackets balance.

Input: 0 1 2 1 3

Output: kernel ' ' void A ( global int double float ...

Decoded: kernel void A( ...
1. Seed the model with the start of a program.
2. Predict tokens until { } brackets balance.

Decoded:

kernel void A(
1. Seed the model with the start of a program.
2. Predict tokens until {} brackets balance.

Decoded: kernel void A(}
1. Seed the model with the start of a program.
2. Predict tokens until \{ \} brackets balance.

**Decoded:**

```
kernel void A(global
```
synthesizer + harness

1. Seed the model with the start of a program.
2. Predict tokens until { } brackets balance.

Input: \[ \cdots \ 2 \ 1 \ 3 \ 4 \ 5 \ \rightarrow \ \text{network} \]

Output:

```
 0
 kernel ' ' void A ( global int double float

Decoded: kernel void A(global}
1. Seed the model with the start of a program.
2. Predict tokens until {} brackets balance.

Decoded:

kernel void A(global
synthesizer + harness

1. Seed the model with the start of a program.
2. Predict tokens until { } brackets balance.
3. Can we parse signature?
   Yes: Generate input data, compile and run it.
   No: Compile it but don’t run it.

Decoded:

```c
kernel void A(global int* a) {
```
synthesizer + harness

1. Seed the model with the start of a program.
2. Predict tokens until { } brackets balance.
3. Can we parse signature?
   Yes: Generate input data, compile and run it.
   No: Compile it but don’t run it.

Decoded:

```
kernel void A(global int* a) { }
```
__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)];
}
__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;
}

Examples
__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;
        /* snip ... */
        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;
}
Does it compile?

70% fail
Does it do anything?

**Dynamic checks**
- has output
- input dependent
- deterministic

Yield 20-25%
Scrape Github → Homogeniser → Tokeniser → Language corpus → Performance Results

Synthesiser → Filter → Driver → Model
__kernel void A(__global float* a, 
   __global float* b, 
   __global float* c, 
   const float d, 
   const int e) {

   int f = get_global_id(0);
   if (f >= e) {
      return;
   }

}
__kernel void A(__global float* a, __global float* b, __global float* c, const float d, const int e) {

    int f = get_global_id(0);
    if (f >= e) {
        return;
    }
}

Payload for size S:

[rand()] * S [rand()] * S [rand()] * S rand() S
Scrape Github → Homogeniser → Tokeniser → Language corpus

Synthesiser → Filter → Driver → PERFORMANCE RESULTS
How well does it work?
Try it

Round 1
Player: 1010, Robot: 938

```c
__kernel void A(__global int* a, __global int* b, __global int* c, int d) {
    int e = get_global_id(0);
    if (e >= d) {
        return;
    } else {
        a[e] = a[e];
    }
    b[d] = e;
}
```
71 programs, 1,000 synthetic benchmarks. 4.30x faster
Overview

- Machine Learning for Compilers
- Generating Benchmarks
- Deep Learned Heuristics
- Deep Fuzzing Compiler Testing
- Future Work
Machine learning in compilers

- Training Programs
- Feature Extractor
- Feature Vectors
- Best Decisions
- Training Data
- Optimization Heuristic

1. hard to get right
2. time consuming
3. repetitious

the human bit!
Ways to fail

irrelevant
- e.g. not capturing the right information

incomplete
- e.g. missing critical information

unsuitable
- e.g. wrong combination of features / model
What we have

Training Programs → Driver → Best Decisions → Feature Vectors → Training Data → Predictive Model
What we need

Training Programs → Driver → Best Decisions → Training Data → Predictive Model
Contributions

Heuristics without features

Beats expert approach

Learning across heuristics
I. Dynamic Inputs

Heuristic Model
Language Model

Code in
Homogeniser
Tokeniser

(_concat with lang. model)

Dynamic Inputs

Correct output
Portable Mapping of Data Parallel Programs to OpenCL for Heterogeneous Systems

Dominik Grewe
Zheng Wang
Michael P. O'Boyle
School of Informatics, University of Edinburgh

dominik.grewe, zh.wang@ed.ac.uk, m.Obolie@ed.ac.uk

Abstract
General purpose GPU-based systems are highly attractive as they give potentially massive performance at low cost. Re-
ducing such potential is challenging due to the complexity of
programming some of these systems. A key factor in the
complexity is automatically generated OpenCL code from
data-parallel OpenMP programs for GPUs. Such an approach
brings together historical bests of a clearly high level
language (OpenMP) and an enabling standard (OpenCL)
for heterogeneous multi-cars. A key feature of our solu-
tion is that it is a memory mapping transformation, optimal
for a state of the art, for porting a high level language
program to another high level language. The solution
provides a methodology to achieve performance at
ease. A key feature of our solution is that it allows
performance at little cost. Reaching such potential, how-
ever, is challenging due to the complexity of programming
some of these systems. A key factor in the complexity is
automatically generated OpenCL code from data-parallel
OpenMP programs for GPUs. Such an approach brings
together the benefits of a clearly high level
language (OpenMP) and an enabling standard (OpenCL)
for heterogeneous multi-cars. A key feature of our solu-
tion is that it is a memory mapping transformation, optimal
for a state of the art, for porting a high level language
program to another high level language. The solution
provides a methodology to achieve performance at
ease. A key feature of our solution is that it allows

Heterogeneous Mapping

Prior Art

CGO’13
Grewe et. al

Thread Coarsening

PACT’14
Magni et. al

Automatic Optimization of Thread-Coarsening for Graphics Processors

Alberto Magni
University of Edinburgh
E-mail: amagni@ed.ac.uk

Christoph Dubach
University of Edinburgh
E-mail: christoph.dubach@ed.ac.uk

Michael O'Boyle
University of Edinburgh
E-mail: m.Obolie@ed.ac.uk

ABSTRACT
Optimizing the code generated by compilers is a fundamental problem in software development. Although
standard approaches to compiler optimization have been shown to be effective, modern compilers have
faced diminishing returns on performance improvements. This paper presents a new technique for
optimizing code on GPUs. The technique involves coarsening the number of threads in a parallel loop,
which results in a significant decrease in the number of local memory accesses. The coarsening
process, which requires knowledge of hardware behavior, and may be repeated until the performance is
optimized. The process is particularly well-suited for GPUs, which achieve rapid
acceleration. The technique is useful for optimizing
standard applications and for optimizing
novel applications, such as those
that are specifically targeted
at GPUs. The technique offers
significant performance
improvements for a variety of
applications, including those that
are critical to the performance of
modern systems.
Heterogeneous Mapping

Thread Coarsening

Prior Art

Binary classification
{CPU, GPU}

One-of-six classification
{1, 2, 4, 8, 16, 32}

Decision Space

Model

Decision Tree

Cascading Neural Networks

CGO'13

PACT'14
Heterogeneous Mapping

4 features
Combined from 7 raw values.
Instruction counts / ratios.

Thread Coarsening

7 features
Principal Components of 34 raw values.
Instruction counts / ratios / relative deltas.

Prior Art

2 papers!
Heterogeneous Mapping

2x CPU-GPU architectures

7 Benchmark Suites

Prior Art

Thread Coarsening

4x GPU architectures

3 Benchmark Suites

Hardware

Training Programs

CGO'13

PACT'14
Our Approach

Heterogeneous Mapping

1. Use the same model design for both
2. No tweaking of parameters
3. Minimum change - 3 line diff

Thread Coarsening

int main(int argc ...)

int main(int argc ...)

1

2

3
Neural Networks

Heterogeneous Mapping

Inputs
Embedding
LSTM_1
LSTM_2
Concat.
Normal.
DNN_1
DNN_2

{CPU, GPU}

code
wgsize
dsize

(a)

Thread Coarsening

code

{1, 2, 4, 8, 16, 32}

(b)
How well does it work?
14% and 5% improvements over state-of-the-art

- **Heterogeneous Mapping**
  - State-of-the-art: 2.09x
  - DeepTune: 2.38x

- **Thread Coarsening**
  - State-of-the-art: 1.01x
  - DeepTune: 1.06x

256 benchmarks vs. 17 benchmarks
Heterogeneous Mapping

Thread Coarsening

Transfer Learning

general → specialized

Embedding Language Model Heuristic Model

Embedding Language Model Heuristic Model

initialize with values
14% and 5% improvements over state-of-the-art

State-of-the-art

DeepTune

Heterogeneous Mapping

Speedup

2.09x

2.38x

Thread Coarsening

Speedup

1.01x

1.06x
14% and 11% improvements over state-of-the-art
Overview

- Machine Learning for Compilers
- Generating Benchmarks
- Deep Learned Heuristics
- Deep Fuzzing Compiler Testing

Future Work
compilers break

Compiler crash
Rewrite code around bug

Semantics change
Security risk
Regression suites

- Slow
- Late
- Expensive
- Incomplete
fuzzing a compiler

circa [McKeenan98]
differential testing compilers

- clang6.0
- gcc5.5
- clang3.6

text:

```
int main(int argc, char** argv) { ...
```

output:

- clang6.0 → a.out → $ ./a.out → 42
- gcc5.5 → a.out → $ ./a.out → 42
- clang3.6 → a.out → $ ./a.out → -14522312

Majority rules

circa [McKeenan98]
differential testing compilers

Also works for build failures

circa [McKeenan98]

int main(int argc, char** argv) { ...

}
an ideal fuzzer

1. **Cheap**
   Easy to implement and extend
   (Languages and features grow quickly)

2. **Interpretable Testcases**
   Necessary for triage
   (i.e. 45 lines or less [Sun2016])

3. **Plausible Output**
   Representative of handwritten code
   (So that bugs get fixed)
state-of-the-art: CLSmith

https://github.com/ChrisLidbury/CLSmith

Random grammar enumeration.

Extensive static analyses support subset of OpenCL features.

Targets compiler middle ends.

Incredibly effective!

100s of bugs to date.

```
#include "CLSmith.h"

struct S0 {
    int32_t g_4[4][10];
    ...
};

kernel void A(global ulong *r) {
    int i, j, k;
    struct S0 c_1856;
    struct S0* p_1855 = &c_1856;
    c_1856 = c_1857;
    func_1(p_1855);
    barrier(CLK_LOCAL_MEM_FENCE |
            CLK_GLOBAL_MEM_FENCE);
    for (i = 0; i < 4; i++)
        for (j = 0; j < 10; j++)
            ...>g_4[i][j], "p_1855->g_4[i][j]",
            print_hash_value);
    result[get_linear_global_id()] =
        crc64_context ^
```
state-of-the-art: CLSmith
https://github.com/ChrisLidbury/CLSmith

1. **Cheap ✗ nope!**
   Years to develop! 50k lines of C++.
   Each PL feature engineered by hand.

2. **Interpretable Testcases ✗ nope!**
   Avg. 1200 lines (excluding headers).
   Requires reduction: ~4 hours / test.

3. **Plausible Output ✗ nope!**
   Unusual and restricted combinations of PL features.
   87 dials control “shape” of output - hand tuned.
contributions

Automatic inference of fuzzers from examples.

102x less code than state-of-art.

Similar bug finding power, simpler test cases.
MODEL

Scrape Github

Homogeniser

140821...

Tokeniser

Language corpus

Synthesiser

Filter

Driver
how well does it work?
testing campaign

- 10 OpenCL compilers
- 3 GPUs, 5 CPUs, Xeon Phi, Emulator
- Test with optimizations on / off
  Treat as separate testbeds
- 48 hours per testbed
Errors in every compiler!

results overview

Compiler crash: 7,040
Build Timeout: 860
Build Failure: 51
Program Crash: 252
Wrong Output: 69
67 bug reports to date...

... crashes during parsing / compilation

```c
void A() {void* a; uint4 b=0; b=(b>b)?a:a }
```

Affects: Intel OpenCL SDK 1.2.0.25

```c
kernel void A(global int* a) {
    int b = get_global_id(0);
    a[b] = (6 * 32) + 4 * (32 / 32) + a;
}
```

Affects: Beignet 1.3

“Bad code” finds bugs in error handling
67 bug reports to date...

... crashes during type checking

```c
kernel void A() {
    __builtin_astype(d, uint4);
}
```

Affects: 6 / 10 compilers we tested

**Unexpected outcome:** Learning from handwritten code leads to bugs found in compiler builtins!
67 bug reports to date...

... errors in optimizers

```c
kernel void A(global double* a, global double* b, 
              global double* c, int d, int e) {

double f;
    int g = get_global_id(0);
if (g < e - d - 1)
    c[g] = (((e) / d) % 5) % (e + d);
}
```

Affects: Intel OpenCL SDK 1.2.0.25

CLSmith doesn’t allow thread-dependent control flow.
Overview

- Machine Learning for Compilers
- Generating Benchmarks
- Deep Learned Heuristics
- Deep Fuzzing Compiler Testing

Future Work
Deep Compilation

Front end

Code gen

.xform

.xform

.xform

.xform

.EXE
Deep Reinforced Super Optimisation

Super optimisation
- Brute force search for optimal code
- Excellent results
- Slow
- Need smart search

- Use reinforcement learning
- DNN chooses actions
- Actions are xform or change focus
- Stop when predicts no improvement
Deep Data Flow

Learn analyses for heuristics not correctness

DNN struggle with data flow

LSTM cannot analyse even reachability on CFG
But can learn if given traces
Can we extend to abstract interpretation?

LSTMs at program points
Transfer is LSTM processing instruction
Meet merges LSTM states
Results are LSTM outputs
Automatic Bug Triage

Fuzzers make thousands of bug cases too quickly

Non buggy programs

Learn language models to discriminate

Buggy programs

DNN Difference encodes bug recognition

Record activation paths

Cluster by activation paths
Deep Active Learning

Most points uninteresting

Good ones do just as well

Active learning directly selects useful points

Desired features

Drivable language model

Matching program

FEATURE "Y"

FEATURE "X"

FEATURE "X"
Conclusion

- Deep learning = better compilers
- Deep learning = lower cost
- Fun stuff still to do