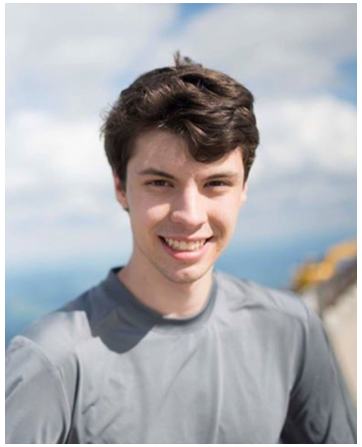
LLVM in the age of LLMs: Machine Learning for IR, Optimization, & More



William S. Moses

University of Illinois, Urbana Champaign
wsmoses@Illinois.edu

AI4Dev Workshop @ SC'23
Nov 13, 2023

This Talk

This Talk

- An introduction to an assortment of interesting and relevant ML and compiler literature leading to why the community is focused on LLM’s

This Talk

- An introduction to an assortment of interesting and relevant ML and compiler literature leading to why the community is focused on LLM’s
- Will provide an intuition for effectively combining compilers + ML through a series of personal case studies

This Talk

- An introduction to an assortment of interesting and relevant ML and compiler literature leading to why the community is focused on LLM’s
- Will provide an intuition for effectively combining compilers + ML through a series of personal case studies
- A biased view of interesting places for the field to go.

This Talk

- An introduction to an assortment of interesting and relevant ML and compiler literature leading to why the community is focused on LLM’s
- Will provide an intuition for effectively combining compilers + ML through a series of personal case studies
- A biased view of interesting places for the field to go.

This Talk is ***Not***

This Talk

- An introduction to an assortment of interesting and relevant ML and compiler literature leading to why the community is focused on LLM's
- Will provide an intuition for effectively combining compilers + ML through a series of personal case studies
- A biased view of interesting places for the field to go.

This Talk is ***<u>Not</u>***

- A comprehensive look at all the related literature (aka not your next paper's related work section)

This Talk

- An introduction to an assortment of interesting and relevant ML and compiler literature leading to why the community is focused on LLM's
- Will provide an intuition for effectively combining compilers + ML through a series of personal case studies
- A biased view of interesting places for the field to go.

This Talk is ***Not***

- A comprehensive look at all the related literature (aka not your next paper's related work section)
- An intro course on ML for HPC

This Talk

- An introduction to an assortment of interesting and relevant ML and compiler literature leading to why the community is focused on LLM's
- Will provide an intuition for effectively combining compilers + ML through a series of personal case studies
- A biased view of interesting places for the field to go.

This Talk is ***<u>Not</u>***

- A comprehensive look at all the related literature (aka not your next paper's related work section)
- An intro course on ML for HPC
- Your PhD thesis

This Talk

- An introduction to an assortment of interesting and relevant ML and compiler literature leading to why the community is focused on LLM's
- Will provide an intuition for effectively combining compilers + ML through a series of personal case studies
- A biased view of interesting places for the field to go.

This Talk is ***Not***

- A comprehensive look at all the related literature (aka not your next paper's related work section)
- An intro course on ML for HPC
- Your PhD thesis
- … Or is it? (https://grad.illinois.edu/admissions/apply)

Default (GPT-3.5)

WM

List me 4 examples of the most impressive AI tools or outcomes

As of my last knowledge update in January 2022, here are four examples of impressive AI tools or outcomes:

1. **AlphaGo by DeepMind (2016):**
2. **GPT-3 by OpenAI (2020):**
3. **DeepMind's AlphaFold (2020):**
4. **IBM Watson for Oncology (2013):**

Default (GPT-3.5)

WM

List me 4 examples of the most impressive AI tools or outcomes

As of my last knowledge update in January 2022, here are four examples of impressive AI tools or outcomes:

1. **AlphaGo by DeepMind (2016):**
2. **GPT-3 by OpenAI (2020):**
3. **DeepMind's AlphaFold (2020):**
4. **IBM Watson for Oncology (2013):**

JS draw_scatterplot.js | draw_scatterplot.py | draw_scatte

```python
import matplotlib.pyplot as plt

def draw_scatterplot(x_values, y_values):
    plt.scatter(x_values, y_values, s=20)
    plt.title("Scatter Plot")
    plt.xlabel("x values")
    plt.ylabel("y values")
    plt.show()
```

Copilot

Beaver hiking up a mountain in the style of Monet (DALLE)

English	Portuguese
I would like a slice of pizza	Gostaria de uma fatia de pizza

DeepL Translator

Why is AI so successful now (and not 20 years ago)?

Why is AI so successful now (and not 20 years ago)?

How do we emulate that success in program optimization?

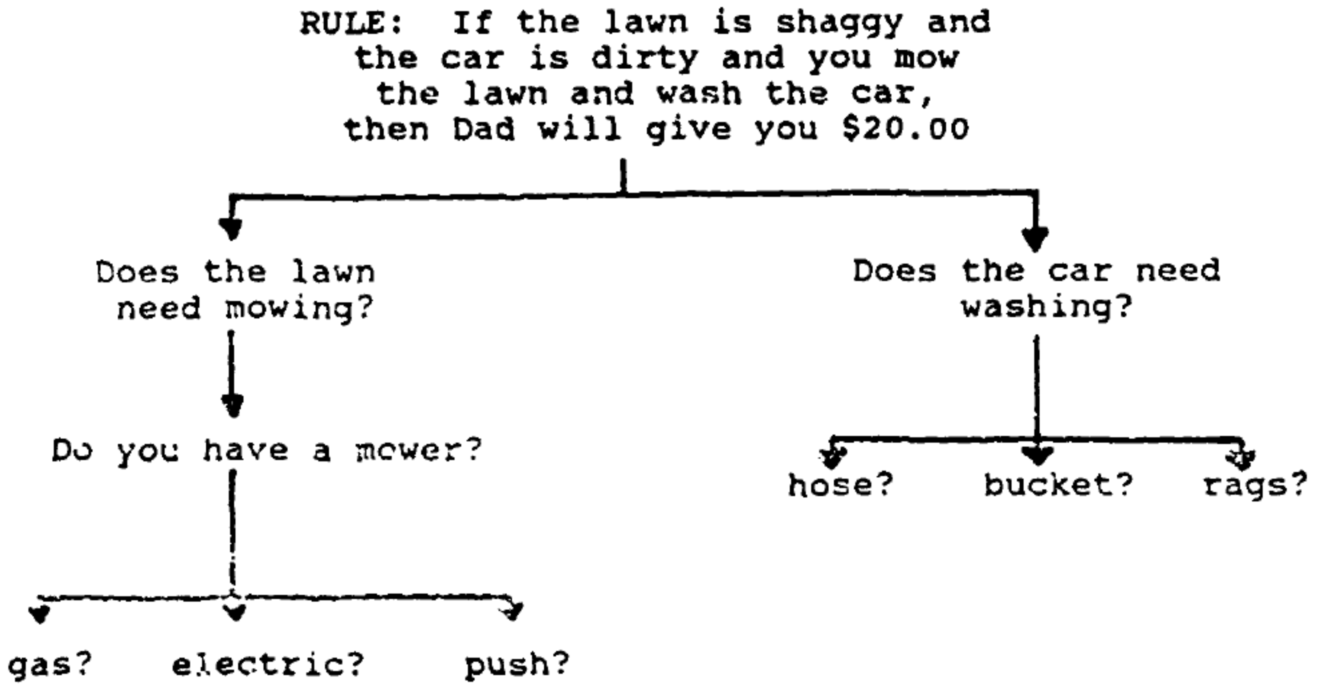
Why is AI so successful now (and not 20 years ago)?

How do we emulate that success in program optimization?

…and push even further?

Early Successful Efforts in AI

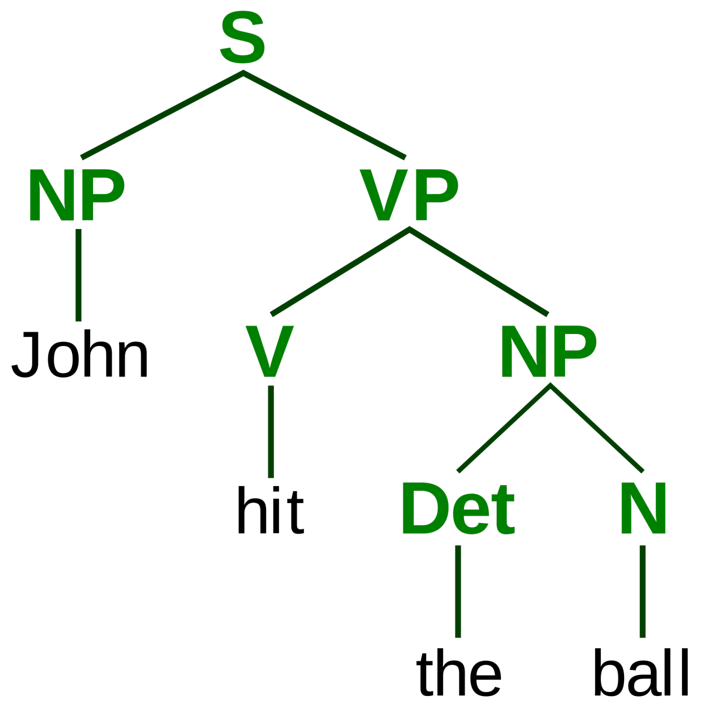
- "Good Old Fashioned AI" (GOFAI) aka Symbolic AI
  - Can we build AI by writing a sufficiently expressive set of rules?
  - Led to creation of LISP programming language, computer time sharing, and many more "non-AI"



Early Successful Efforts in AI

- “Good Old Fashioned AI” (GOFAI) aka Symbolic AI
  - Can we build AI by writing a sufficiently expressive set of rules?
  - Led to creation of LISP programming language, computer time sharing, and many more “non-AI”

- Analyzing language by modelling stages of language (tokenizing, features, etc)
  - Parse Tree



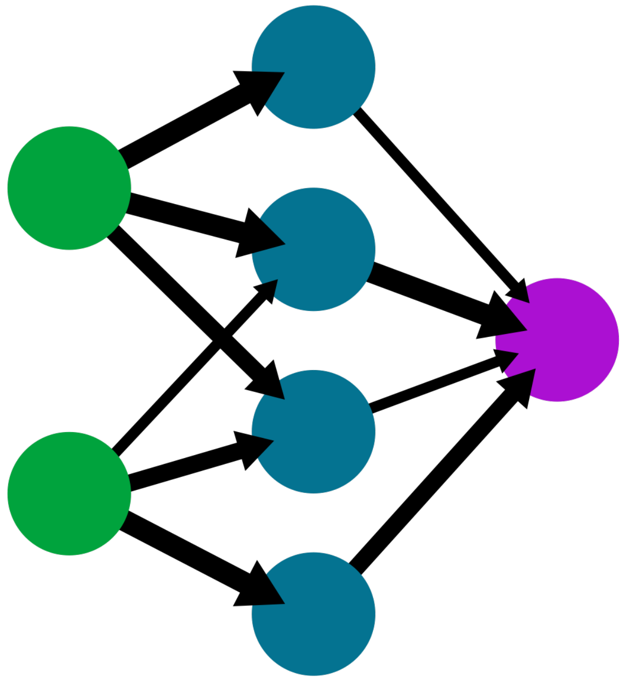
Early Successful Efforts in AI

- "Good Old Fashioned AI" (GOFAI) aka Symbolic AI
  - Can we build AI by writing a sufficiently expressive set of rules?
  - Led to creation of LISP programming language, computer time sharing, and many more "non-AI"
- Analyzing language by modelling stages of language (tokenizing, features, etc)
  - Parse Tree
- Sophisticated filters to identify information in images
  - Canny Edge Detection (1986)

$$* \begin{array}{|c|c|c|}\hline -1 & 0 & +1 \\ \hline -2 & 0 & +2 \\ \hline -1 & 0 & +1 \\ \hline \end{array} \quad + ..$$

Early Successful Efforts in AI

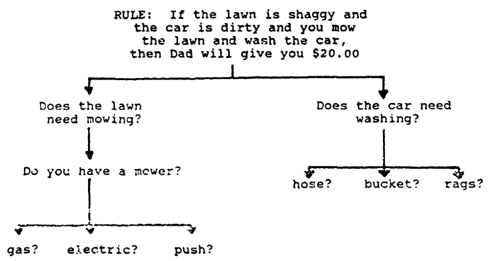
- “Good Old Fashioned AI” (GOFAI) aka Symbolic AI
  - Can we build AI by writing a sufficiently expressive set of rules?
  - Led to creation of LISP programming language, computer time sharing, and many more “non-AI”
- Analyzing language by modelling stages of language (tokenizing, features, etc)
  - Parse Tree
- Sophisticated filters to identify information in images
  - Canny Edge Detection (1986)
- Artificial Neural Networks



Early AI <-> Compiler Optimization

Early AI <-> Compiler Optimization

- GOFAI
  &
  Program
  Transformations

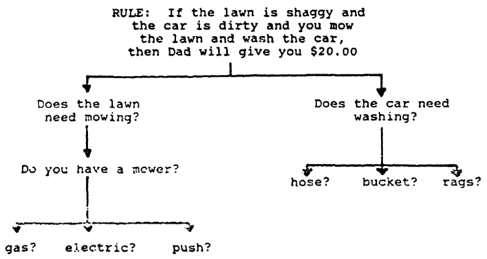


y(i) = A(i,j) * x(j)

```
for (int32_t i0 = 0; i0 < ((A1_dimension + 31) / 32); i0++) {
    for (int32_t i1 = 0; i1 < 32; i1++) {
      int32_t i = i0 * 32 + i1;
      double tjy_val = 0.0;
      for (int32_t jA = A2_pos[i]; jA < A2_pos[(i + 1)]; jA++) {
        int32_t j = A2_crd[jA];
        tjy_val += A_vals[jA] * x_vals[j];
      }
      y_vals[i] = tjy_val;
    }
  }
}
```

Early AI <-> Compiler Optimization

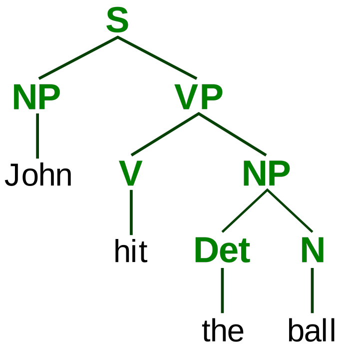
- GOFAI
  &
  Program Transformations

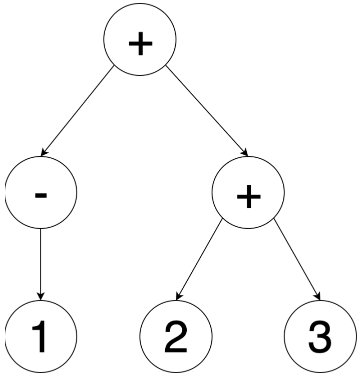



```
y(i) = A(i,j) * x(j)
```

```
for (int32_t i0 = 0; i0 < ((A1_dimension + 31) / 32); i0++) {
    for (int32_t i1 = 0; i1 < 32; i1++) {
      int32_t i = i0 * 32 + i1;
      double tjy_val = 0.0;
      for (int32_t jA = A2_pos[i]; jA < A2_pos[(i + 1)]; jA++) {
        int32_t j = A2_crd[jA];
        tjy_val += A_vals[jA] * x_vals[j];
      }
      y_vals[i] = tjy_val;
    }
  }
}
```

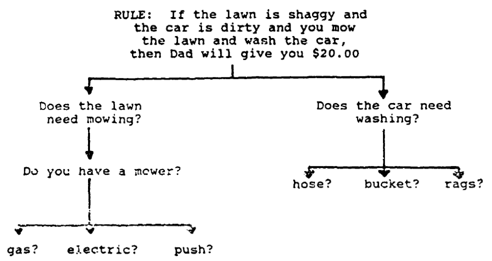
- Early NLP
  &
  Program AST/IR





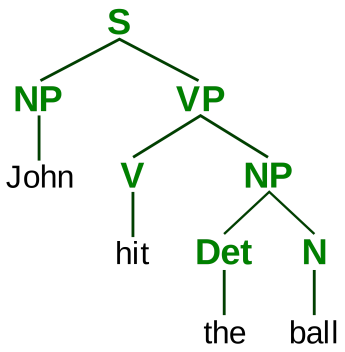
Early AI <-> Compiler Optimization

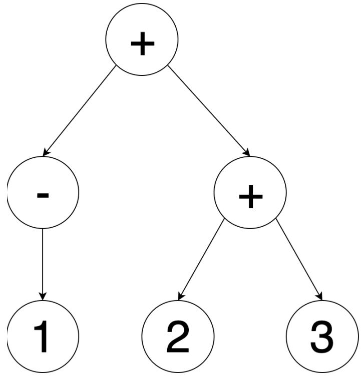
- GOFAI
  &
  Program
  Transformations




```
y(i) = A(i,j) * x(j)
for (int32_t i0 = 0; i0 < ((A1_dimension + 31) / 32); i0++) {
    for (int32_t i1 = 0; i1 < 32; i1++) {
      int32_t i = i0 * 32 + i1;
      double tjy_val = 0.0;
      for (int32_t jA = A2_pos[i]; jA < A2_pos[(i + 1)]; jA++) {
        int32_t j = A2_crd[jA];
        tjy_val += A_vals[jA] * x_vals[j];
      }
      y_vals[i] = tjy_val;
    }
  }
}
```

- Early NLP
  &
  Program AST/IR

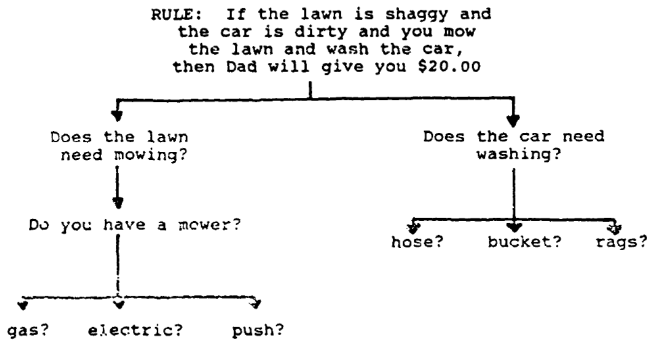




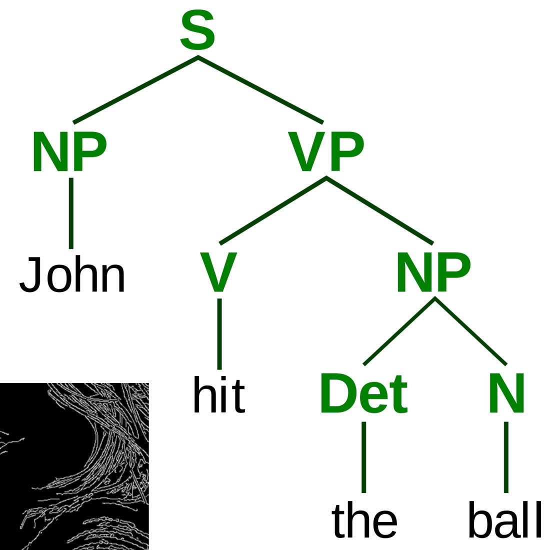
- Coarse Filters
  &
  Program Metrics

```
f1 has 100 instructions
f2 has 10 instructions
f3 has 1000 instructions
```

Early Successful Efforts in AI

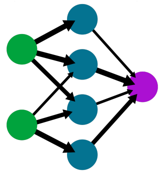


- **“Good Old Fashioned AI” (GOFAI) aka Symbolic AI**
  - **Can we build AI by writing a sufficiently expressive set of rules?**
  - **Led to creation of LISP programming language, computer time sharing, and many more “non-AI”**



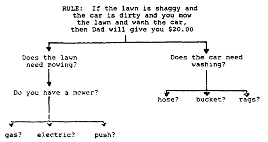
- **Analyzing language by modelling stages of language (tokenizing, features, etc)**
  - **Parse Tree**

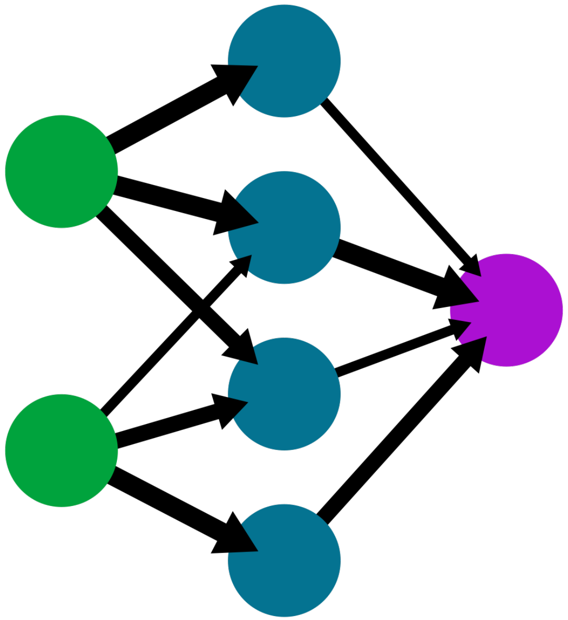
- **Sophisticated filters to identify information in images**
  - **Canny Edge Detection (1986)**



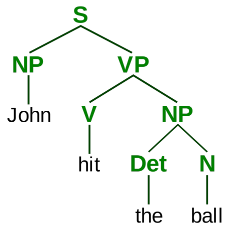
- Artificial Neural Networks

Early Successful Efforts in AI

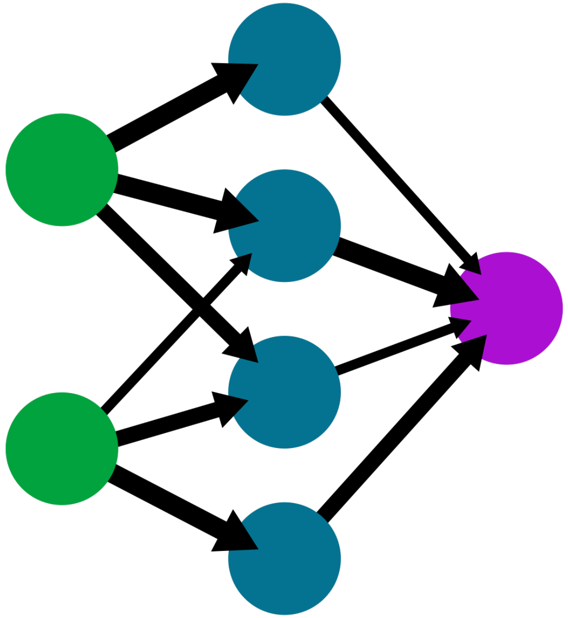




- "Good Old Fashioned AI" (GOFAI) aka Symbolic AI
  - Can we build AI by writing a sufficiently expressive set of rules?
  - Led to creation of LISP programming language, computer time sharing, and many more "non-AI"



- Analyzing language by modelling stages of language (tokenizing, features, etc)
  - Parse Tree



- Sophisticated filters to identify information in images
  - Canny Edge Detection (1986)

- **Artificial Neural Networks**

Early Successful Efforts in AI

- Symbolic AI lost out as it is bottlenecked by the need to manually specify transformations

Early Successful Efforts in AI

- Symbolic AI lost out as it is bottlenecked by the need to manually specify transformations
- Fortunately in compilers we never have to manually specify transformations

```
y(i) = A(i,j) * x(j)
```

```
int compute(taco_tensor_t *y, taco_tensor_t *A, taco_tensor_t *x) {
  int y1_dimension = (int)(y->dimensions[0]);
  double* restrict y_vals = (double*)(y->vals);
  int A1_dimension = (int)(A->dimensions[0]);
  int* restrict A2_pos = (int*)(A->indices[1][0]);
  int* restrict A2_crd = (int*)(A->indices[1][1]);
  double* restrict A_vals = (double*)(A->vals);
  int x1_dimension = (int)(x->dimensions[0]);
  double* restrict x_vals = (double*)(x->vals);

  #pragma omp parallel for schedule(runtime)
  for (int32_t i0 = 0; i0 < ((A1_dimension + 31) / 32); i0++) {
    for (int32_t i1 = 0; i1 < 32; i1++) {
      int32_t i = i0 * 32 + i1;
      if (i >= A1_dimension)
        continue;

      double tjy_val = 0.0;
      for (int32_t jA = A2_pos[i]; jA < A2_pos[(i + 1)]; jA++) {
        int32_t j = A2_crd[jA];
        tjy_val += A_vals[jA] * x_vals[j];
      }
      y_vals[i] = tjy_val;
    }
  }
  return 0;
```

Early Successful Efforts in AI

- Symbolic AI lost out as it is bottlenecked by the need to manually specify transformations
- Fortunately in compilers we never have to manually specify transformations

```
y(i) = A(i,j) * x(j)
```

```
int compute(taco_tensor_t *y, taco_tensor_t *A, taco_tensor_t *x) {
  int y1_dimension = (int)(y->dimensions[0]);
  double* restrict y_vals = (double*)(y->vals);
  int A1_dimension = (int)(A->dimensions[0]);
  int* restrict A2_pos = (int*)(A->indices[1][0]);
  int* restrict A2_crd = (int*)(A->indices[1][1]);
  double* restrict A_vals = (double*)(A->vals);
  int x1_dimension = (int)(x->dimensions[0]);
  double* restrict x_vals = (double*)(x->vals);

  #pragma omp parallel for schedule(runtime)
  for (int32_t i0 = 0; i0 < ((A1_dimension + 31) / 32); i0++) {
    for (int32_t i1 = 0; i1 < 32; i1++) {
      int32_t i = i0 * 32 + i1;
      if (i >= A1_dimension)
        continue;

      double tjy_val = 0.0;
      for (int32_t jA = A2_pos[i]; jA < A2_pos[(i + 1)]; jA++) {
        int32_t j = A2_crd[jA];
        tjy_val += A_vals[jA] * x_vals[j];
      }
      y_vals[i] = tjy_val;
    }
  }
  return 0;
}
```

```
int compute(taco_tensor_t *y, taco_tensor_t *A, taco_tensor_t *x) {
  int y1_dimension = (int)(y->dimensions[0]);
  double* restrict y_vals = (double*)(y->vals);
  int A1_dimension = (int)(A->dimensions[0]);
  int* restrict A1_pos = (int*)(A->indices[0][0]);
  int* restrict A1_crd = (int*)(A->indices[0][1]);
  int* restrict A2_pos = (int*)(A->indices[1][0]);
  int* restrict A2_crd = (int*)(A->indices[1][1]);
  double* restrict A_vals = (double*)(A->vals);
  int x1_dimension = (int)(x->dimensions[0]);
  double* restrict x_vals = (double*)(x->vals);

  #pragma omp parallel for schedule(static)
  for (int32_t py = 0; py < y1_dimension; py++) {
    y_vals[py] = 0.0;
  }

  #pragma omp parallel for schedule(runtime)
  for (int32_t i0 = 0; i0 < ((A1_dimension + 31) / 32); i0++) {
    int32_t pA1_begin = i0 * 32;
    int32_t iA = taco_binarySearchAfter(A1_crd, A1_pos[0], A1_pos[1],
pA1_begin);
    int32_t pA1_end = A1_pos[1];
    int32_t iA0 = A1_crd[iA];
    int32_t i = A1_crd[iA];
    int32_t i1 = i - i0 * 32;
    int32_t i1_end = 32;

    while (iA < pA1_end && i1 < i1_end) {
      iA0 = A1_crd[iA];
      i = A1_crd[iA];
      if (iA0 == i) {
        double tjy_val = 0.0;
        for (int32_t jA = A2_pos[iA]; jA < A2_pos[(iA + 1)]; jA++) {
          int32_t j = A2_crd[jA];
          tjy_val += A_vals[jA] * x_vals[j];
        }
        y_vals[i] = tjy_val;
      }
      iA += (int32_t)(iA0 == i);
      iA0 = A1_crd[iA];
      i = A1_crd[iA];
      i1 = i - i0 * 32;
    }
  }
  return 0;
}
```

```
int compute(taco_tensor_t *y, taco_tensor_t *A, taco_tensor_t *x) {
  int y1_dimension = (int)(y->dimensions[0]);
  double* restrict y_vals = (double*)(y->vals);
  int A1_dimension = (int)(A->dimensions[0]);
  int* restrict A1_pos = (int*)(A->indices[0][0]);
  int* restrict A1_crd = (int*)(A->indices[0][1]);
  int* restrict A2_pos = (int*)(A->indices[1][0]);
  int* restrict A2_crd = (int*)(A->indices[1][1]);
  double* restrict A_vals = (double*)(A->vals);
  int* restrict x1_pos = (int*)(x->indices[0][0]);
  int* restrict x1_crd = (int*)(x->indices[0][1]);
  double* restrict x_vals = (double*)(x->vals);

  #pragma omp parallel for schedule(static)
  for (int32_t py = 0; py < y1_dimension; py++) {
    y_vals[py] = 0.0;
  }

  #pragma omp parallel for schedule(runtime)
  for (int32_t i0 = 0; i0 < ((A1_dimension + 31) / 32); i0++) {
    int32_t pA1_begin = i0 * 32;
    int32_t iA = taco_binarySearchAfter(A1_crd, A1_pos[0], A1_pos[1],
pA1_begin);
    int32_t pA1_end = A1_pos[1];
    int32_t iA0 = A1_crd[iA];
    int32_t i = A1_crd[iA];
    int32_t i1 = i - i0 * 32;
    int32_t i1_end = 32;

    while (iA < pA1_end && i1 < i1_end) {
      iA0 = A1_crd[iA];
      i = A1_crd[iA];
      if (iA0 == i) {
        double tjy_val = 0.0;
        int32_t jA = A2_pos[iA];
        int32_t pA2_end = A2_pos[(iA + 1)];
        int32_t jx = x1_pos[0];
        int32_t px1_end = x1_pos[1];

        while (jA < pA2_end && jx < px1_end) {
          int32_t jA0 = A2_crd[jA];
          int32_t jx0 = x1_crd[jx];
          int32_t j = TACO_MIN(jA0,jx0);
          if (jA0 == j && jx0 == j) {
            tjy_val += A_vals[jA] * x_vals[jx];
          }
          jA += (int32_t)(jA0 == j);
          jx += (int32_t)(jx0 == j);
        }
        y_vals[i] = tjy_val;
      }
      iA += (int32_t)(iA0 == i);
      iA0 = A1_crd[iA];
      i = A1_crd[iA];
      i1 = i - i0 * 32;
    }
  }
  return 0;
}
```

…

Extending the reasoning of compilers

- Manually specified transformations are the bread and butter of compilers (we tend to call the ”optimization passes”)
- Very GOFAI style-symbolic reasoning of “we can prove program A is faster than program B”
  - Often extend this to say if we *think* A is faster than B
- Compilers are rampant with manually specified heuristics of when we think A is faster than B … and often get it wrong (or at least for some people)

…
- Hundreds of arbitrarily chosen flags, orderings, etc.
- A change to optimization pass ordering led to a 50% performance reduction for NVPTX (https://bugs.llvm.org/show_bug.cgi?id=52037)

Automated transformations within compilers

- Much work early work focused on using ML to automate the “coarse reasoning” of input programs, and select parameters for the fastest program
- Advantage: all output programs are correct

- Tensor Comprehensions (2017) used genetic algorithms to pick schedules (achieved 90+% of peak)
- End-to-end Deep Learning of Optimization Heuristics (2017) used LSTM networks to predict

- Yet, you still need to manually specify the structure of the output programs

Future of Optimization + ML

1. Automated Transformations

2.

3.

(Deep) Reinforcement Learning

- The significant area of AI research in the mid-late 2010's
- Achieved state of the art skill at various games including Go, StarCraft, Atari games, etc
- Given a state **`s`** and set of possible actions **`actions(s)`**, pick the optimal action **`a`** that maximizes a (often end-game) reward.

(Deep) Reinforcement Learning

- The significant area of AI research in the mid-late 2010's

- Achieved state of the art skill at various games including Go, StarCraft, Atari games, etc

- Given a state **`s`** and set of possible actions **`actions(s)`**, pick the optimal action **`a`** that maximizes a (often end-game) reward.

- Idea: we can model each GOFAI-style compiler optimization as one of these actions!

Reinforcement Learning for Compilers

- AutoPhase (2019) used deep RL to predict optimal LLVM compiler pass orderings
  - Achieved 28% boost beyond O3, with promising generality results
  - At the time, no good way to represent the program in a way that can be analyzed by the network
  - Demonstrates the significant data and compute problem within RL: bottlenecked by the iterations through the reward/action simulator

- Learning to optimize halide with tree search and random programs (2019) used beam search (e.g. greedily take the top k)
- ProTuner (2020) used plain old Monte Carlo Tree Search (MCTS) to search for schedules

- MLGO (2021) use policy gradients and evolution strategies to optimize for size (7% reduction beyond Oz)

Future of Optimization + ML

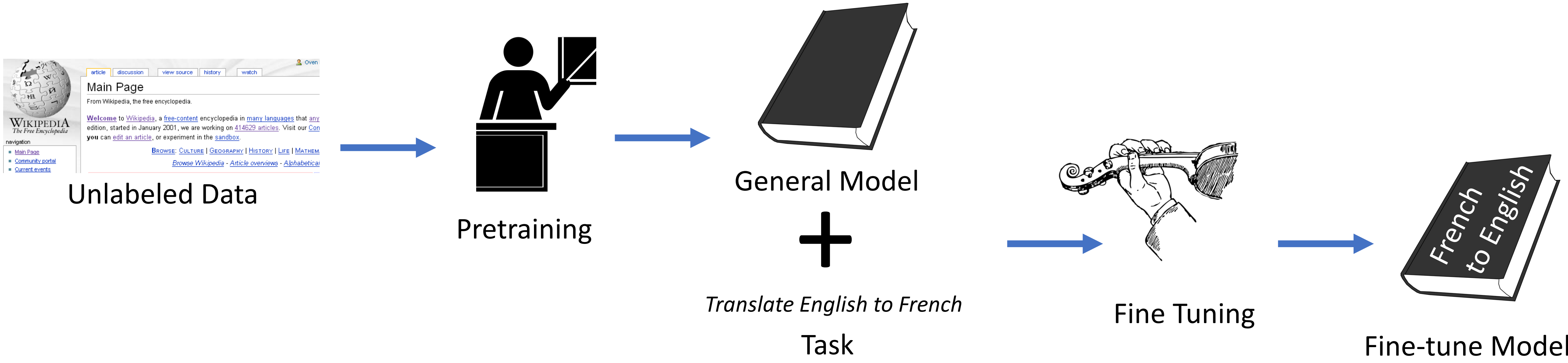
1. Automated Transformations

2. Neural Program Representation

3.

Unsupervised Learning + Transformers

- Earlier approaches were bottlenecked by the amount of labeled data
- Train on a large corpus of unlabeled data (all of the internet) & fine-tune on a small dataset (some sample phrases in two languages)
- Transformers enable efficient contextual access without serializing inputs
- This is the secret sauce behind modern LLMs (like GPT).



How well can Transformers compile code?

- ***Goal: Determine effectiveness of end-to-end optimization / generation of low level programs***
- Enabling Transformers to Understand Low-Level Programs (IEEE HPEC ’22) https://ieeexplore.ieee.org/abstract/document/9926313
- Whole program analysis and optimization with transformers
- Leverage autogenerated and unlabeled training data from compiler (clang)
- Build novel LLVM-specific specific optimizations for better training

Why ML on Low-Level Code Is Hard

```
//Compute magnitude in O(n)
double mag(double[] x) ) { … }

//Compute norm in O(n^2)
void norm(double[] out, double[] in) {

  for (int i=0; i<n; i++) {
    out[i] = in[i] / mag(in);
  }
}
```

Loop invariant code motion (LICM)

```
declare double @mag(double* readonly) argmemonly
```

```
define void @norm(double* noalias %0, double* noalias %1)
  br label %loop
```

```
%5 = phi i64 [ 0, %2 ], [ %11, %4 ]
%6 = getelementptr inbounds double, double* %1, i64 %5
%7 = load double, double* %6, align 8, !tbaa !4
%8 = call double @mag(double* noundef %1) #2
%9 = fdiv double %7, %8
%10 = getelementptr inbounds double, double* %0, i64 %5
store double %9, double* %10, align 8, !tbaa !4
%11 = add nuw nsw i64 %5, 1
%12 = icmp eq i64 %11, 100
br i1 %12, label %end, label %loop
```

```
ret void
```

- More verbose and precise semantics
- -> Ensures that optimizations can be performed (moving mag outside loop requires mag to be readonly)

Case study: Translating C to (Optimized) LLVM

```c
double relu3(double x) {
  double result;
  if (x > 0)
    result = pow(x, 3);
  else
    result = 0;
  return result;
}
```

```llvm
define double @relu3(double %0) {
  %2 = fcmp ogt double %0, 0
  br %2, label %3 , label %5
3:
  %4 = tail call double @pow (%0, double
  3)
  br label %5
5:
  %6 = phi [%4, %3], [0, %1]
  ret %6
}
```

Data & Results

- Csmith (randomly generated compilable C programs) (Yang et al., 2011)
- Project CodeNet (web scrape of competitive programming online judging websites) (Puri et al., 2021)
- AnghaBench (1 million selected and cleaned compilable GitHub C programs) (de Silva et al., 2021)

Model evaluation result on the 3 datasets

	Csmith	Project CodeNet	AnghaBench
Training Accuracy	90.73%	93.66%	99.03%
Reference Match	N/A	5.76	13.33%
BLEU Score (0~100)	43.39	51.01	69.21

Preprocessing Modification & Optimizations

- Expanding preprocessing directives with clang –E such as pasting the definition of imported libraries, compile-time constants, and more.

```
#ifdef AARCH64
#define size_t int64_t
#else
#define size_t int32_t
#endif
size_t getsize();
```

```
int64_t getsize();
```

- Reduce redundancies in program grammar while making sure to faithfully restore the original

```
%4 = load i32**, i32*** %2
```

```
%4 = load i32** %2
```

Preprocessing Modification, cont.

- Prefix Notation
  - A * B + C / D => + * A B / C D
  - Prefix notation previously shown effective for mathematics (Griffith & Kalita, 2019)

```
{ [4 x i8], i32, { i8, i32 }}
```

```
STRUCT 5 ARR 3 4 x i8 i32 STRUCT 2 i8 i32
```

- Writing out definitions of global variables so they can be recoverable on the function level, which makes the programs more complex

```
%struct.1 = type { i32, i32, i64 }
...
%2 = alloca %struct.1, i64 %1
```

```
%struct.1 = type { i32, i32, i64 }
...
%2 = alloca { i32, i32, i64 }, i64 %1
```

Ablation Analysis

Ablation studies of model evaluation result on AnghaBench dataset

	Original	Cleaned	Prefix	Prefix & Global	-O1
Training Accuracy	99.03%	97.84%	99.60%	99.36%	97.87%
Reference Match	13.33%	21.15%	**49.57%**	38.61%	38.73%
BLEU Score (0~100)	69.21	72.48	**87.68**	82.55	77.03
Compilation Acc.	14.97%	N/A	N/A	**43.07%**	N/A

- The various cleanup simplifies LLVM IR programs and boosts accuracy, while the expansion of global variables ensures compilation but reduces accuracy

Future of Optimization + ML

1. Automated Transformations
2. Neural Program Representation
3. Unlabeled Data

Future of Optimization + ML

1. Automated Transformations

2. Neural Program Representation

3. Unlabeled Data [1]

[1] ComPile: A Large IR Dataset from Production Sources (2023),

Aiden Grossman, Ludger Paehler, Konstantinos Parasyris, Tal Ben-Nun, Jacob Hegna, William Moses, Jose M Monsalve Diaz, Mircea Trofin, Johannes Doerfert

Problems in AI (including for Code)

- Language models are designed to predict tokens that are seem likely to appear at the next location
- This means that when they respond to queries, they aren't really ``solving'' the problem in the conventional sense, and may output answers that seem reasonable, but make no sense (hallucinations).
- This is exacerbated for symbolic reasoning tasks like math, or programming …the very tasks the first AI systems were designed to do

Future of Optimization + ML

1. Automated Transformations
2. Neural ***and Symbolic*** Program Representation
3. Unlabeled Data

Summary

- AI and optimization have a long, and very intertwined history
- There have been many interesting optimization + ML studies, many of which have followed corresponding trends in AI
- However, we’re still quite far from the “dream” compiler that auto optimizes all of your code perfectly well
- To get there we will need to look not just at the current trends, but also the history of these fields
- There’s a lot of opportunity for work here, please reach out if interested!