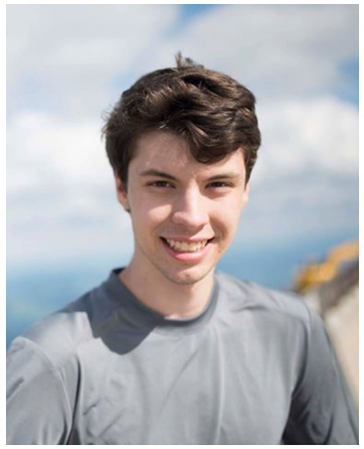
Enabling Transformers to Understand Low-Level Programs

Zifan Guo
MIT

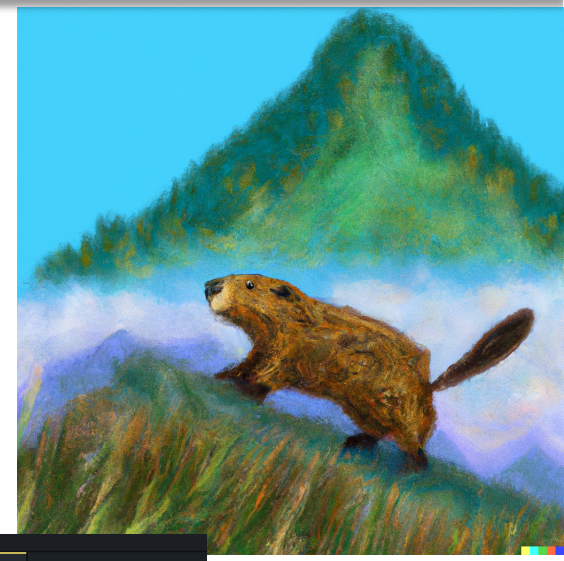


William S. Moses
MIT

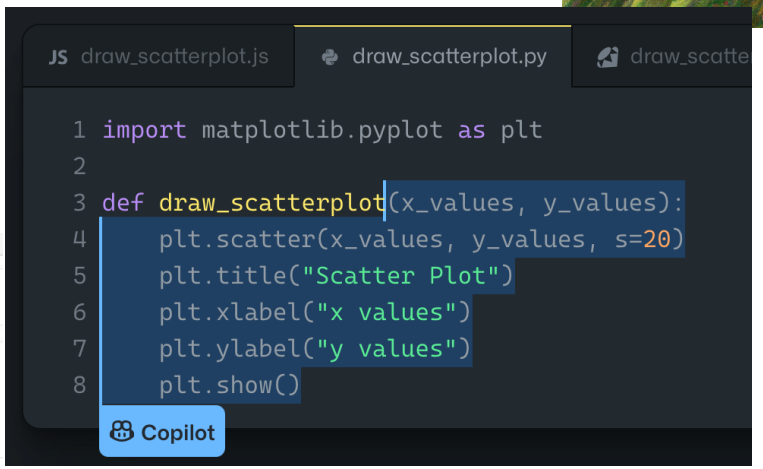
LLVM Dev Meeting – Nov. 9th, 2022

Advances in Modern Machine Learning

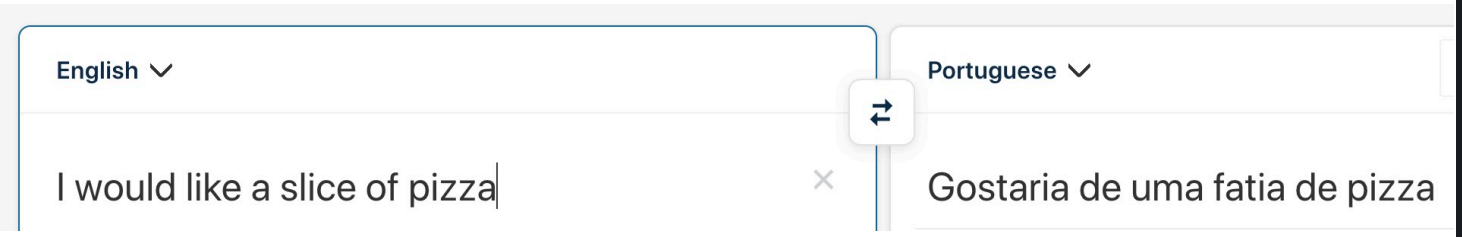
- ML breakthroughs in NLP thanks to Transformers
  - E.g. translation, code generation, image generation
- Larger amount of data + parameters = better outcomes (GPT-3)
- Successful on natural languages & high-level programs
- -> Can we use NLP advances to facilitate program ***optimization***?



DALL-E



GitHub Copilot



DeepL Translator

How Are Optimizations Applied Presently

```
float sq(float x) {
  return x * x;
}
```

High Level Program

Lower

Optimization

Low Level Program

CodeGen

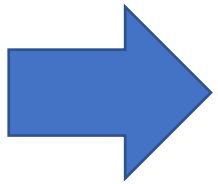
Executable

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);
  }
}
```



```
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
```

Loop invariant code motion (LICM)

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

Previous Approaches

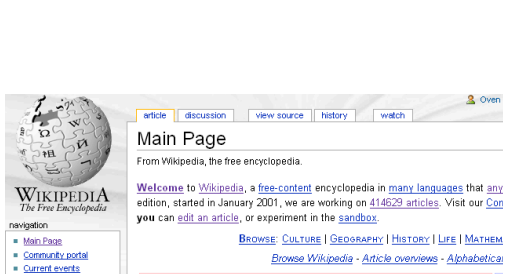
- Focus on smaller objectives=> training on specific objectives like optimizing inlining-for-size or register-allocation (MLGO)
- More supervised ML than unsupervised
- Unsupervised approaches are just starting to be applied to code and tend to be applied to either high level programs or very specific objectives (e.g. superoptimization)

Our Paper: Enabling Transformers to Understand Low-Level Programs (IEEE HPEC ’22)

- ***Goal: Determine effectiveness of end-to-end optimization / generation of low level programs***
- Whole program analysis with Transformers
- Leverage autogenerated and unlabeled training data from compiler (clang)
- Build novel low-level specific optimizations

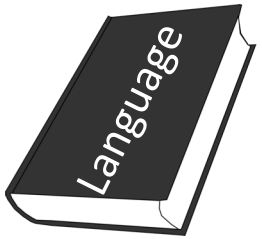
- Paper Link: https://ieeexplore.ieee.org/abstract/document/9926313

Transformers (Vaswani et al. 2017)



Unlabeled Data

Pretraining

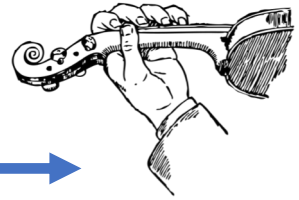


General Model

+

Translate English to French

Task



Fine Tuning

Fine-tune Model

- Train on a large corpus of unlabeled data & fine-tune on a small dataset
- Cross-lingual model of both high-level and low-level programs => provide information that compensates each other
- Better inform us where and how to apply compiler optimization considering whole program context

Case study: Translating C to LLVM-IR

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

```
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
}
```

- Preprocessing
  - Use TransCoder tokenizer for C tokenizer and build our own LLVM-IR tokenizer based on Clang
  - Performed Byte-Pair Encoding (BPE)
- Transfer Learning:
  - Pretrained first with Masked Language Modeling (MLM) on all data
  - Fine-tuned with Machine Translation instead of Back Translation on functions only

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
```

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

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

Summary

- We explore how effectively ML can analyze/optimize low-level programs
- Case study: translate C to unoptimized and optimized LLVM IR
- A 49.57% verbatim match and BLEU score of 87.68 against Clang -O0 38.73% verbatim match and BLEU score of 77.03 against Clang -O1
- Preprocessing transformations (e.g. prefix notation, type sugaring) improves BLEU score from 72.48 to 87.68
- Tradeoff between ability to be compiled and accuracy
- Opens up possibilities for whole program optimization

Acknowledgement

- Thanks to Susan Tan (Princeton), Yebin Chon (Princeton), and Johannes Doerfert (LLNL) for thoughtful discussions on this work.
- This research was supported in part by the MIT PRIMES program, grant number 6946149.
- William S. Moses was supported in part by a DOE Computational Sciences Graduate Fellowship DE-SC0019323.
- This research was supported in part by Los Alamos National Laboratories grant 531711.
- Research was sponsored by the United States Air Force Research Laboratory and the United States Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

Summary

- We explore how effectively ML can analyze/optimize low-level programs
- Case study: translate C to unoptimized and optimized LLVM IR
- A 49.57% verbatim match and BLEU score of 87.68 against Clang -O0 38.73% verbatim match and BLEU score of 77.03 against Clang -O1
- Preprocessing transformations (e.g. prefix notation, type sugaring) improves BLEU score from 72.48 to 87.68
- Tradeoff between ability to be compiled and accuracy
- Opens up possibilities for whole program optimization