Compiler-RL: Reinforcement Learning Approaches to Phase Ordering

CS 7643 Final Project by: Tsz Hang Kiang, Adithya Vasudev, Rishabh Jain, Zheyi Zhang

tkiang3@gatech.edu, avasudev8@gatech.edu, rjain343@gatech.edu, zzhang3058@gatech.edu

Introduction

The order in which a compiler makes passes on the code such as dead code elimination, register allocation, and code in-lining can have great impact on the quality of the resulting emitted assembly. However, finding the optimal order in which to these passes are performed, and the optimal number of times, given a goal such as binary size, execution time, is a hard problem known as phase ordering. There is already research in applying RL techniques to phase ordering, targeting a wide variety of resulting benefits, such as smaller memory foot- print, lower instruction count (and consequently, smaller executable file size), as well as effective speed of executable. The problem can be formalized as learning a function that maps an Intermediate Representation (IR) of a program to a weight for each possible optimization pass and optimizing this function based on a reward. We wish to investigate further using Deep RL to improve upon existing results.

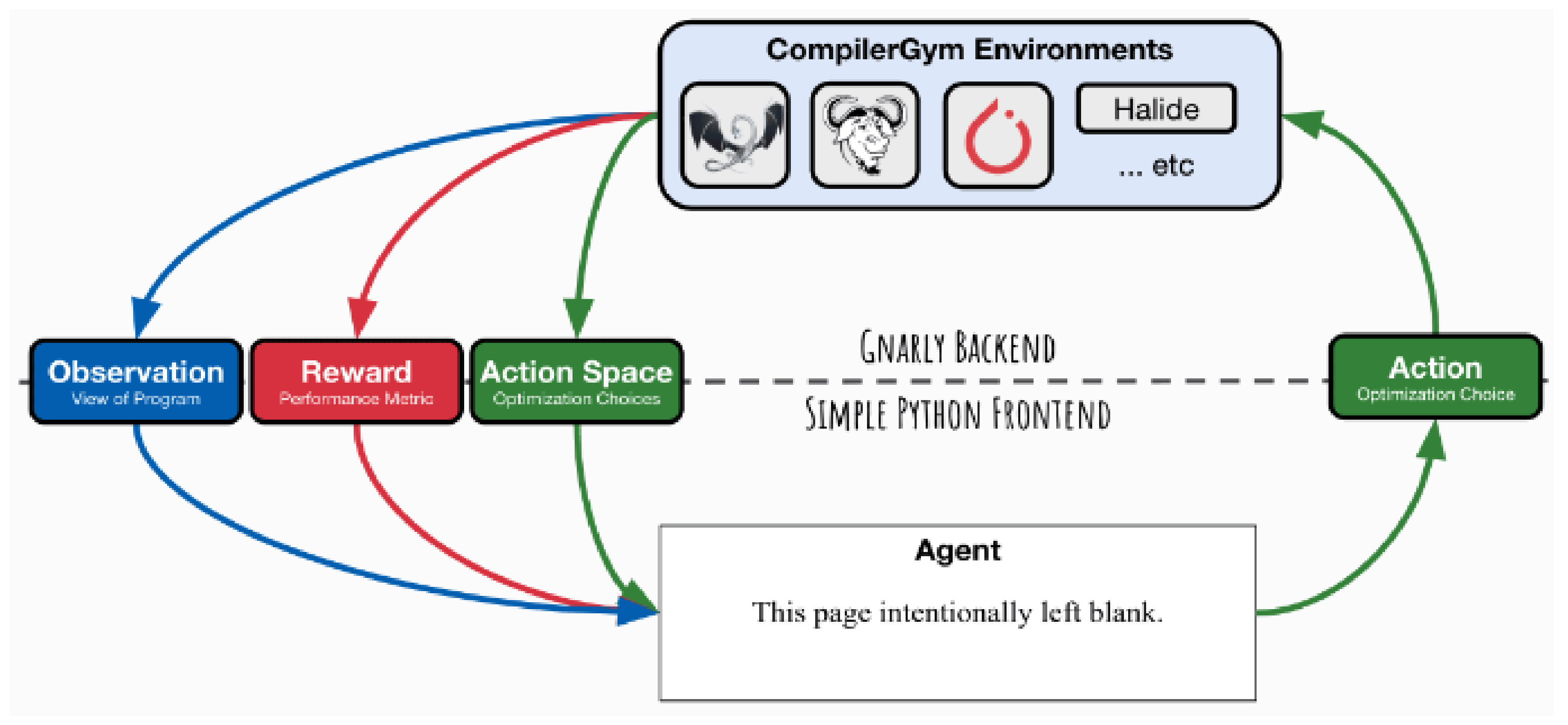

Figure 1: A visual representation of the CompilerGym environment

Objectives

The objective of our project was twofold:

1. To use a multitude of reinforcement learning based approaches (Deep Q-Learning, Proximal Policy Optimization) and examine how they can be used to solve the phase ordering problem in compilation.

2. To examine how we could use a more optimal data structure for representing and storing ProGraML representations of pre-compiled code to aid in the training process and improve accuracy and efficiency of these reinforcement learning based approaches.

To this end, we have proposed a series of machine learning models and evaluated them against a base result of random sampling, using the ProGraML observation space in Meta AI's CompilerGym Environment.

Reinforcement Learning Approaches

GraphConverter

As we were working with the ProGraML observation space, all of our inputs were graphs. These graphs encoded information about the structure and flow of the program and are represented using a MultiDiGraph. GraphConverter is an addendum to this MultiDiGraph that also encodes the multiplicity of edges as part of the metadata of an edge, rather than as multiple individual edges. The benefits of this can be shown through the representation of a batch of identical looping code. We can see the benefits of this through the following snippet of code, its assembly representation, and then its equivalent GraphConverter representation:

```
for i in range(100):
  a(i)

for i in range(100):
  b(i)

for i in range(100):
  c(i)
    .
    .
    .
for i in range(100):
  z(i)
```

```
loop_a:
    call a(i)
    add i i 0
    brneq i 100 loop_a
```

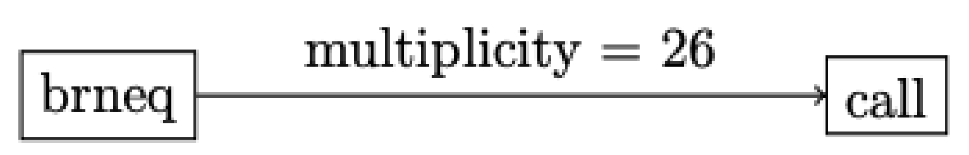

Figure 2: Three representations of a loop

Deep Q-Learning

The first agent we designed was a Deep Q-Learning agent using a Convolutional Graph Neural Network. The network was four layers deep, with 2 graph convolutional layers, with dropout between the two, 2 linear layers with the LeakyReLU activation function, and a batch normalization between the convolutional and linear layers.

The network was evaluated against a truly random agent on 150 different programs, with the Q-Learning agent strictly winning 133 of the 150 total games, for a win rate of **88.6%.** When evaluated on CompilerGym's benchmarks, we achieved a geometric mean codesize reduction of ~0.301x.

Proximal Policy Optimization

The second agent we designed was of a GATv2 (Graph Attention Network) model-based PPO design. The input observation ProGraML graph is first converted into our multiplicity representation, and then embedded, before being passed into two attention layers. We used Ray's RLlib to allow for concurrent state space search. When evaluated on CompilerGym's benchmarks, we achieved a geometric mean code size reduction of ~0.479x.

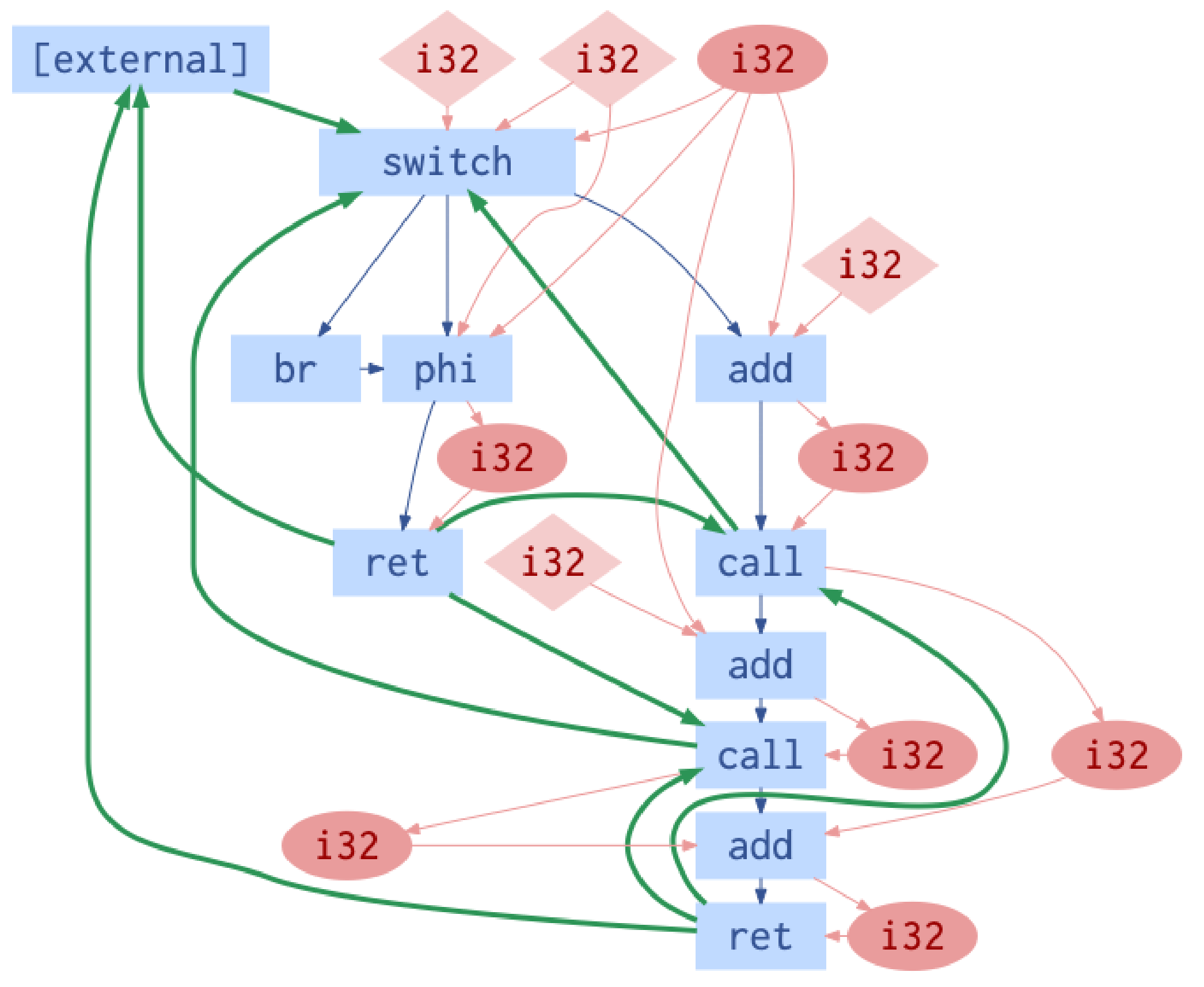

Figure 3: Sample ProGraML Graph

Offline Approaches to Transition Learning

As we were reading through the CompilerGym paper, we noticed that the authors had created a state transition dataset, which stores detailed information such as observations (various representations) of each state and the transitions (which state is reached given an input and a specific compiler action). As an investigation, we wanted to implement a transformer model to try to predict the next state. More specifically, predict the next IR of a program being compiled given an input for current IR and a specific action for the compiler to take. Doing this in a supervised manner can help improve reinforcement learning algorithms and was useful for our own curiosity. We started by basing our implementation off of the example given by the authors, where they attempted to predict the instruction count of a program after two compiler passes based on its ProGraML graph representation. Unfortunately, there was a critical flaw that prevented us from successfully implementing the transformer. The Transitions table in the database is completely empty, and we needed that information to move from state to state. For a future investigation, it should be possible to generate our own test cases, either via storing states and transitions while running our own algorithm in the Gym environment, or via hacking together states and the specific action sequences that led to their generation.

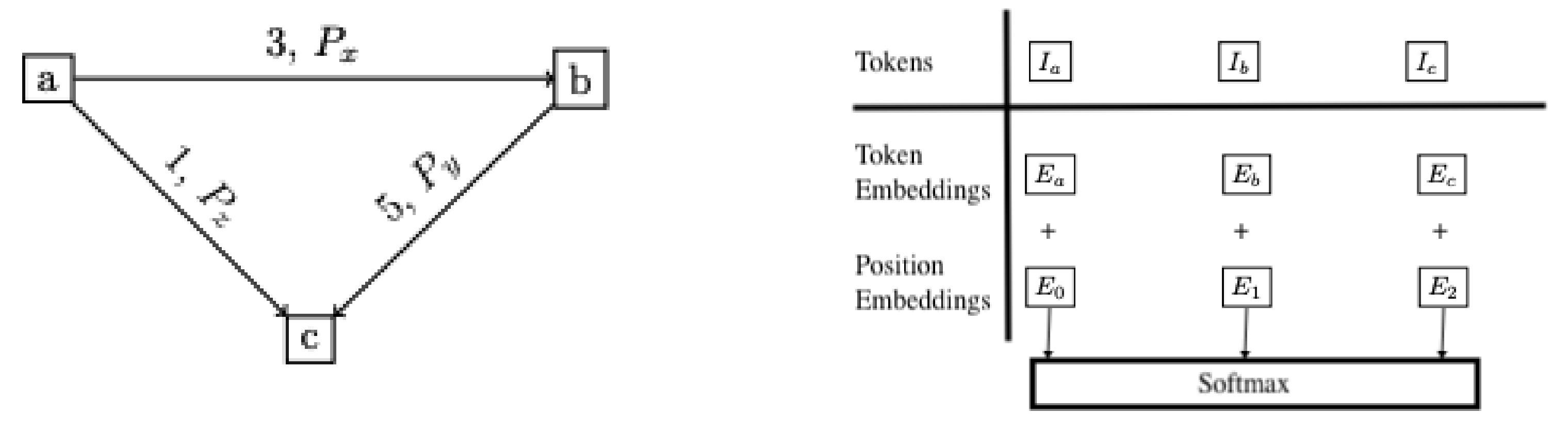

Figure 4: Transformer Embeddings for a Transition

Selected References

C. Cummins *et al*., "CompilerGym: Robust, Performant Compiler Optimization Environments for AI Research," *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2022, pp. 92-105, doi: 10.1109/CGO53902.2022.9741258.

P. Almasan, J. Suárez-Varela, K. Rusek, P. Barlet-Ros, and A. Cabellos-Aparicio, "Deep reinforcement learning meets graph neural networks: Exploring a routing optimization use case," *Computer Communications*, vol. 196, pp. 184–194, 2022.

Cummins, C., Fisches, Z. V., Ben-Nun, T., Hoefler, T., & Leather, H. (2020). Programl: Graph-based deep learning for program optimization and analysis. arXiv preprint arXiv:2003.10536.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Scarselli, Franco, Gori, Marco, Tsoi, Ah Chung, Hagenbuchner, Markus, and Monfardini, Gabriele. The graph neural network model. IEEE Transactions on Neural Networks, 20(1):61–80, 2009.

DHRL: A Graph-Based Approach for Long-Horizon and Sparse Hierarchical Reinforcement Learning: https://arxiv.org/abs/2210.05150