Proposal: Solving Phase Ordering problem with Reinforcement Learning

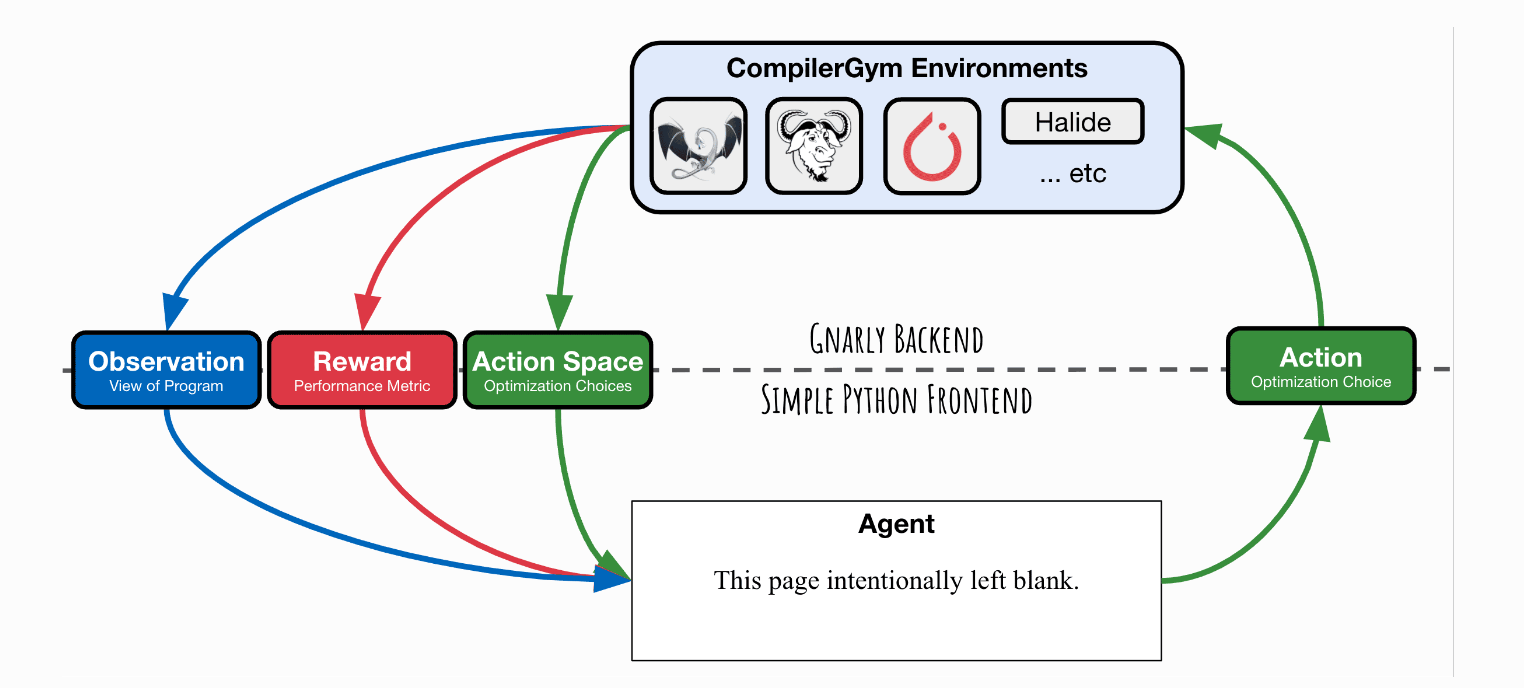
**About**

Phase Ordering is a classical compiler problem. The compilers provide an amount of transformations (LLVM provides at least 130 transformations). How to select which transformations to use and how to order them can highly affect the compilation time and runtime of the compiled programs. However, the optimal transformation sequences are different among programs that are being compiled. Currently, LLVM apply fixed sequences (https://github.com/llvm/llvm-project/blob/main/llvm/lib/Passes/PassBuilderPipelines.cpp#L247) of transformations for all programs.

**Related work**

Recently, most researchers try to use Machine learning to solve the problem. The pipeline for these projects are as follows: The input programs will be profiled and be compressed to vectors. For example, in Autophase[1], 55 features (e.g., Number of Shl insts, Number of BB’s with no Phi nodes) are profiled and combined into vectors. Then, a prediction model will be trained, to predict which transformations should be applied for a given vector. Thus, with the model. For a given program, it will first be profiled, and the profiled vector is used to predict a sequence of transformations. In Autophase. RandomForest are used for prediction, while Cereda, et al [2] use Recommendation system instead.

The whole process can be regarded as Reinforcement Learning: based on currently status (LLVM IR), the agent learns to pick up an action (a LLVM pass), and applies it to get a new status (a new LLVM IR).



There is an open-source project, CompilerGym [3], which provides a framework for implementing RL on Phase Ordering problems. It allows researchers to implement customized agent, to pick up actions for given status (blank box in the above figure).

**Motivation**

All previous projects treat transformations as black boxes. If we involve the information of transformations, we can both speed up the training time and accuracy for the RL process. For example, if there is not loop inside the programs, we should not apply a loop unroll on them. Besides, some passes (e.g., CFGFlatten, Dead Code Elimination) can be applied multiple times. While other passes (e.g., inline, reg2mem) should be applied no more than once. Another example is some passes increasing the code size to decrease the runtime (e.g., LoopUnswitch), these passes should not be applied if we want to compress the code size.

**Plan**

We can first summarize all passes in LLVM, and conclude some useful features (e.g., require loop construction, will enlarge the code size). Then, we can present transformations with these features. Thus, during the RL, instead of treating passes as scalar labels, we can treat them as vectors, which may bring improvements for both the accuracy and latency.

**Reference**

[1] Huang, Qijing, et al. "Autophase: Juggling hls phase orderings in random forests with deep reinforcement learning." arXiv preprint arXiv:2003.00671 (2020).

[2] Cereda, Stefano, et al. "A collaborative filtering approach for the automatic tuning of compiler optimisations." *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. 2020.

[3] Cummins, Chris, et al. "CompilerGym: robust, performant compiler optimization environments for AI research." *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2022.