Compiler Optimization using Generative Adversarial Networks

Thomas Howard III
University of Rhode Island

August 10, 2018

Abstract

Modern software development is a multi-stage process. Once source code has been written, it must be compiled into machine-readable code. This compilation process is extremely complex, and though most modern compilers offer a litany of potential optimizations, as humans, it is incredibly difficult to understand what flags may lead to performance gains. Current solutions rely heavily on developer knowledge and iteration, a state which is not ideal for the current pace of development the market demands.

This article proposes a deep reinforcement learning model to solve this problem. It explores what happens when an agent is presented application features, and must decide what flags to use. Program compilation, in this way, closely resembles a contextual-bandit problem; a type of game in which there is only one turn, the goal being a lower runtime. Training an agent in this manner has shown extremely promising results, with only one-hundred games played, an agent is able to predict optimization flags yielding 2.8x speedups on average (65% of the optimal speedup), with speedups of 6x attained in certain benchmarks.

1 Introduction

When compiling high-level code for release, most developers want the fastest runtime, most power efficient operation, and/or smallest memory footprint; yet, few actually know how to get this top-level performance out of modern compilers. In fact, most compilers support hundreds of possible optimization flags that can be rearranged, repeated, left out completely, or any combination of the former. It is because of this extreme complexity that no perfect compiler optimization-sequence exists. Current solutions to this problem use various statistical and machine-learning-based methods to try and derive the best possible options, short-circuiting essentially iterative processes; yet fall short of truly optimal performance, or lack the ability to generalize across multiple architectures and benchmarks.

Trying to eke out performance gains for production releases may seem trivial to some, however, when one considers the impact this may have on embedded devices, it is impossible to consider the impact as anything less than massive. Source code for embedded devices, such as cars, power tools, digital watches, pace makers, etc., is normally compiled a single time, before being deployed to millions of devices. Even with conservative estimates of 40 embedded devices per household, better preforming compilers would result in billions of devices performing faster, saving energy, and requiring less memory.

To truly master the compiler optimization space, one needs to consider learning *tabula rasa*, since no ground truth actually exists about the search space. At it's heart, this problem is comparable to the work Deep Mind has done with the game of Go, a game that exhibits a similarly cost-prohibitively large search-space. Unlike Go, compiler optimization emits no change in state, and must therefore be treated as an extremely large contextual-bandit problem, where the programs are the bandits themselves, and the 'arms' represent potential optimization sequences. The agent then interacts with these 'bandits' in order to learn the relations between program features and compiler optimizations, using deep learning and hind-sight experience replay to build underlying relations directly from the full feature set, without the need for feature reduction.

This paper explores how deep reinforcement learning copes with such a large search-space in a contextual-bandit problem by first implementing it on a smaller scale. Starting with just seven possible flags, arranged in sub-sets of various lengths, without allowing reordering or repetition. The goal of this work is to produce a model capable of examining the features of a program, and yielding the optimal flags out of the provided search-space. For this work, the only metric being optimized is the runtime of a given program, in future works both larger search-spaces, and multi-goal approaches should be explored.

2 Training and Testing

In order to thoroughly evaluate the proposed model, this work employed leave-one-out testing against each program, for each feature set present in the data. The script begins by first loading all cBench programs, along with their static and dynamic features [1]. Once these features are loaded, the script then computes all 128 runtimes for each individual application, although this approach will not be feasible when the search space is expanded, it allowed direct comparisons between agents selections and each programs optimal performance. Once these runtimes are computed, the main testing/training loop is initiated.

The outer loop of the script cycles through each feature set: static, dynamic, and hybrid; while the inner loop cycles through each individual application. Due to the nature of leave-one-out testing, within the inner loop, a unique agent is created as a combination of the program to test against, and the feature set in use. Once this agent is instantiated, it is given 100 'episodes' to explore the relation between program features, compiler flag sequences, and runtimes.

[1] Courtesy of the work done for COREL

During each episode a program is selected at random from the training set, the selected features are then exposed to the agent as 'context', which the agent then uses to predict one, or more, 'actions' it thinks will *minimize* the runtime. These actions, correspond to a distinct subset of compiler flags; these flags are then evaluated by the program, producing a runtime that is saved into the agents memory, along with the context and action selected. Every 20th episode, the agent batch-trains against a random sample of events saved into its memory.

Testing the model is done in a very similar matter to training, with the exception that the agent does not save any data about the testing program, and that multiple actions are actually used, to compute five-shot testing metrics. Once an agent has completed it's 100 episodes of training, the epsilon (chance of doing something random) is set to 0, and the agent is exposed to the appropriate context of the testing program. Five actions are returned and evaluated. The resulting runtimes, along with a baseline of no flags, just -O3, and just -O2, are used to compute important metrics, such as single-shot speedup, five-shot speedup, and optimal speedup. These metrics are then used as the basis for the rest of this work.

3 Results

4 Conclusions

References

[Ashouri] *COBAYN: Compiler autotuning with BAYsian Network*