CompilerGym: Robust, Performant Compiler Optimization Environments for AI Research

Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, Yuandong Tian, Hugh Leather
Meta, USA
cummins@fb.com

2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) | 978-1-6654-0584-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/CGO53902.2022.9741258

***Abstract*—Interest in applying Artificial Intelligence (AI) techniques to compiler optimizations is increasing rapidly, but compiler research has a high entry barrier. Unlike in other domains, compiler and AI researchers do not have access to the datasets and frameworks that enable fast iteration and development of ideas, and getting started requires a significant engineering investment. What is needed is an easy, reusable experimental infrastructure for real world compiler optimization tasks that can serve as a common benchmark for comparing techniques, and as a platform to accelerate progress in the field.**

We introduce CompilerGym, a set of environments for real world compiler optimization tasks, and a toolkit for exposing new optimization tasks to compiler researchers. CompilerGym enables anyone to experiment on production compiler optimization problems through an easy-to-use package, regardless of their experience with compilers. We build upon the popular OpenAI Gym interface enabling researchers to interact with compilers using Python and a familiar API.

We describe the CompilerGym architecture and implementation, characterize the optimization spaces and computational efficiencies of three included compiler environments, and provide extensive empirical evaluations. Compared to prior works, CompilerGym offers larger datasets and optimization spaces, is 27× more computationally efficient, is fault-tolerant, and capable of detecting reproducibility bugs in the underlying compilers.

In making it easy for anyone to experiment with compilers – irrespective of their background – we aim to accelerate progress in the AI and compiler research domains.

I. Introduction

There is a growing body of work that shows how the performance and portability of compiler optimizations can be improved through autotuning [1], machine learning [2], and reinforcement learning [3], [4], [5]. The goal of these approaches is to supplement or replace the optimization decisions made by hand-crafted heuristics with decisions derived from empirical data. Autotuning makes these decisions by automatically searching over a space of configurations. This is effective, but search may be prohibitively costly for large search spaces, and must be repeated from scratch for each new problem instance. The promise of supervised and reinforcement learning techniques is to reduce or completely eliminate this search cost by inferring optimization decisions from patterns observed in past data.

Despite many strong experimental results showing that these techniques outperform human experts [2], [1], [6], the complexity of experimental infrastructure for compiler research hampers progress in the field. In many other fields there are simple environments, each using standard APIs that machine learning researchers can interact with. From Atari games to physics simulations, a known interface abstracts the problems to the point that AI researchers do not need deep knowledge of the problem to apply their machine learning techniques. CompilerGym provides just that for compilers[1]. AI researchers can solve compiler problems without being compiler experts, and compiler experts can integrate state-of-the-art ML without being AI experts.

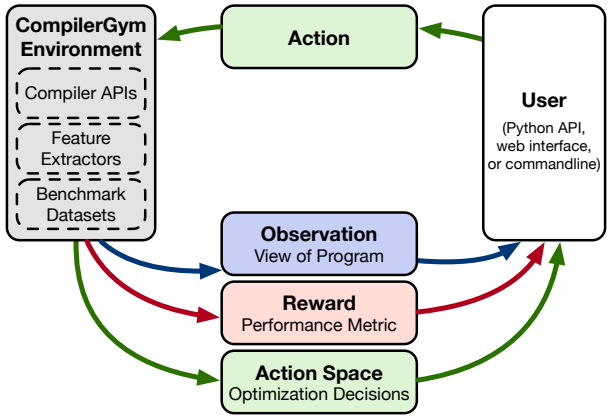

Figure 1: The CompilerGym interaction loop. A CompilerGym environment exposes an observation, reward, and action space. The user's goal is to select the action that will lead to the greatest cumulative reward. This may be through hand-crafted heuristics, search, supervised machine learning, or reinforcement learning. The environment determines when a terminal state has been reached and the interaction loop terminates.

To support this ease of use and performance CompilerGym offers the following key features:

1) **Easy to install.** Precompiled binaries for Linux and macOS can be installed with a single command.
2) **Easy to use.** Builds on the Gym [7] API that is easy to learn and widely used by researchers.
3) **Comprehensive.** Includes a full suite of millions of benchmarks. Provides multiple kinds of pre-computed program representations and appropriate optimization targets and reward functions out of the box.
4) **Reproducible.** Provides validation for correctness of results and public leaderboards to aggregate results.
5) **Accessible.** Includes code-free ways to explore CompilerGym environments, such as an interactive command

[1]CompilerGym is available at: https://compilergym.ai

```python
import compiler_gym
# Create a new environment, selecting the compiler to
# use, the program to compile, the feature vector to
# represent program states, and the optimization target:
env = compiler_gym.make(
    "llvm-v0",
    benchmark="cbench-v1/qsort",
    observation_space="Autophase",
    reward_space="IrInstructionCount",
)
# Start a new compilation session:
observation = env.reset()
# Run a thousand random optimizations. Each step of the
# environment produces a new state observation and reward:
for _ in range(1000):
    observation, reward, done, info = env.step(
        env.action_space.sample()  # User selects action.
    )
    if done:
        env.reset()
# Save output program:
env.write_bitcode("/tmp/output.bc")
```

Listing 1: Example of the CompilerGym environment API. A CompilerGym environment builds on `gym.Env`, formulating a compiler optimization task as a Markov Decision Process, and provides additional compiler-specific functionality such as saving the compiled program to disk.

line shell and a browser-based graphical user interface.

6) **Performant.** Supports the high throughput required for large-scale experiments on massive datasets.
7) **Fault-tolerant.** Detects and gracefully recovers from flaky compiler errors that can occur during autotuning.
8) **Extensible.** Removes the substantial engineering effort required to expose new compiler problems for research and integrate new machine learning techniques.

In this paper, we make the following contributions:

- We introduce CompilerGym, a Python library that formulates compiler optimization problems as easy-to-use Gym [7] environments with a simple API.
- We provide environments for three compiler optimization problems: LLVM phase ordering, GCC flag selection, and CUDA loop nest generation. The environments are designed from the ground up for large-scale experimentation: they are 27× faster than prior works, expose larger search spaces, include millions of programs for training, and support optimizing for both code size and runtime.
- We demonstrate the utility of CompilerGym as a platform for research by evaluating a multitude of autotuning and reinforcement learning techniques. By using a standard interface, CompilerGym seamlessly integrates with third party libraries, offering a substantial reduction in the engineering effort required to create compiler experiments.
- We release a suite of tools to lower the barrier-to-entry to compiler optimization research: the core CompilerGym library and environments, a toolkit for integrating new compiler optimization problems, public leaderboards to aggregate and verify research results, a web interface and API, extensive command line tools, and large offline datasets comprising millions of performance results.

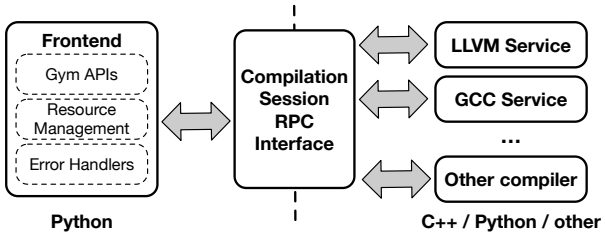

Figure 2: The client-server architecture. The dashed line indicates the boundary between the frontend Python process that the user interacts with and the backend compiler services. The processes communicate over RPC, providing simple distribution, parallelization, and fault tolerance.

II. System Architecture

CompilerGym's architecture comprises two components: a Python frontend that implements the Gym APIs and other user-facing tools, and a backend that provides the integrations with specific compilers.

Frontend: The CompilerGym frontend is a Python library that exposes compiler optimization tasks using the OpenAI Gym [7] environment interface. Figure 1 shows the interaction loop for the Gym environments. This allows researchers to interact with important compiler optimization problems in a familiar language and vocabulary with which many are comfortable. The frontend is described in Section III.

Backend: CompilerGym uses a client-server architecture, shown in Figure 2. This design provides separation of concerns as systems developers can easily add support for new compiler problems by implementing a simple Compilation Session interface that comprises only four methods. The backend is described in Section IV.

III. Frontend API and Tools

This section describes CompilerGym's user-facing tools. We first describe the core formulation of compiler optimization problems as Gym environments, then the API extensions and other features tailored for compiler optimization research.

A. OpenAI Gym Environments

We formulate compiler optimization tasks as Markov Decision Processes (MDPs) and expose them as environments using the popular OpenAI Gym [7] interface. A Gym environment comprises five ingredients:

1) An *Action Space* defines the set of possible actions that can be taken from a given MDP state. In CompilerGym, action spaces can be composed of discrete choices (e.g. selecting an optimization pass from a finite set), continuous choices (e.g. selecting a function inlining threshold), or any combination of the two. The action space can change between states, such as in the case where one optimization precludes another.

2) An *Observation Space* from which observations of the MDP state are drawn. CompilerGym environments support multiple observation types such as numeric feature vectors generated by compiler analyses, control flow graphs, and strings of compiler IR. Each environment exposes multiple observation spaces that can be selected from or composed.

3) A *Reward Space* defines the range of values generated by the reward function, used to provide feedback on the quality of a chosen action, either positive or negative. In CompilerGym, reward spaces can be nondeterministic (e.g. change in program runtime), platform specific (e.g. change in the size of a compiled binary), or entirely deterministic.

4) A *Step* operator applies an action at the current state and responds with a new observation, a reward, and a signal that indicates whether the MDP has reached a terminal state. Not all compiler optimization problems have terminal states.

5) A *Reset* operator resets the environment to an initial state and returns an initial observation.

Listing 1 demonstrates how the core CompilerGym API is used. A `make()` function instantiates a subclass of the `gym.Env` environment that represents a particular compiler optimization task. The Gym interface is self describing: the action space and observation spaces are described by `action_space` and `observation_space` attributes, respectively. This enables CompilerGym environments to be integrated directly with techniques that are compatible with other Gym environments. Listing 2 shows one such integration.

In interacting with an environment the user's goal is to select the sequence of actions that maximizes the cumulative reward. Although Gym is designed primarily for reinforcement learning research, it makes no assumptions about the structure of user code and therefore can be used with a wide range of approaches. For a single environment, the best sequence of actions may be found through search. To generalize a solution that works for unseen environments, a *policy* is learned to map from observation to optimal actions, or a *Q-function* is learned to give expected cumulative rewards for state-action pairs.

B. API Extensions for Compiler Optimization

The advantage of the Gym interface is that it is simple and can be used across a range of domains. We supplement this interface with additional APIs that are specific to compilers.

1) Benchmark Datasets: An instance of a compiler optimization environment requires a program to optimize. We refer to these programs as *benchmarks*, and collections of benchmarks as *datasets*. We designed an API to manage datasets that efficiently scales to millions of benchmarks, and a mechanism for downloading datasets from public servers. This API supports program generators (like CSmith [8]), compiling user-supplied code to use as benchmarks, iterating and looping over sets of benchmarks, and specifying an input dataset and execution environment for running compiled binaries.

2) State Serialization: We provide a mechanism to save and restore environment state that includes the benchmark, action history, and cumulative reward.

3) Validating States: Serialized states can be replayed to validate that results are reproducible. We use this to ensure reproducibility of the underlying compiler infrastructure. For example, we detected a nondeterminism bug in an LLVM optimization pass[2]; we removed this pass from CompilerGym.

[2]LLVM's `-gvn-sink` pass contains an operation that sorts a vector of basic block pointers by address, causing inconsistent output.

```python
import compiler_gym
from ray import tune
from ray.rllib.agents.ppo import PPOTrainer

def make_env(config):
  # Create an LLVM environment using the Autophase
  # observation space and instruction count rewards.
  env = compiler_gym.make("llvm-autophase-ic-v0")
  # Optionally create a time limit for the RL agent.
  env = compiler_gym.wrappers.TimeLimit(env, 45)
  # Loop over the NPB benchmark suite for training.
  dataset = env.datasets["benchmark://npb-v0"]
  env = compiler_gym.wrappers.CycleOverBenchmarks(
      env, dataset.benchmarks()
  )
  return env

tune.register_env("CompilerGym", make_env)
tune.run(PPOTrainer, config={"env": "CompilerGym"})
```

Listing 2: Using RLlib [9] to train a reinforcement learning agent on one of the CompilerGym environments.

4) Validating Semantics: For runnable benchmarks, we provide an additional layer of results validation that automatically applies a differential testing [10] regime to detect correctness errors in the compiled binaries. For the LLVM environments we also integrate LLVM's address, thread, and undefined behavior sanitizers to detect program logic errors.

5) Lazy and Batched Operations: Typically, the observation and reward spaces of a Gym environment are determined at construction time, and each `step()` operation takes a single action and produces a single observation and reward. We extend this method in CompilerGym environments to optionally accept multiple actions, and a list of observation and reward spaces to compute and return. Passing multiple actions enables the backend to more efficiently execute them in a single batch and return a final state and reward, evaluated in Section VII-A. Specifying the observation and reward spaces as arguments to `step()` enables efficient lazy computation of observations or rewards in cases where the values are not needed at every step, or to flexibly change observation and reward space during the liftetime of an environment.

6) Lightweight Deep Copy Operator: CompilerGym environments provide a `fork()` operator that efficiently creates independent deep copies of environment states. This can be used to optimize backtracking or other techniques that require frequently evaluating a common subsequence of actions. For example, a greedy search can be implemented by creating n forks of an environment with an n-dimensional action space, running a single action in each fork, and selecting the one which produced the greatest reward. Backtracking is especially expensive in compilers because most actions have no "undo".

C. Customizing Environment Behavior

The Gym [7] library defines environment wrapper classes to mutate the MDP formulation of a wrapped environment. CompilerGym provides an additional suite of environment wrappers for a broad range of compiler research uses. These include specifying a subset of command line flags to use in an action space, iterating over a suite of benchmarks, and defining

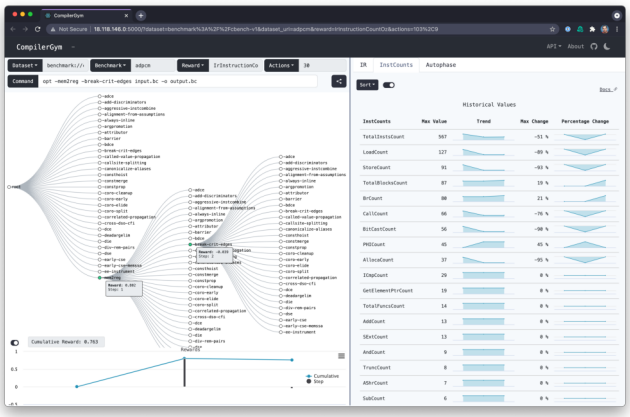

Figure 3: CompilerGym Explorer, a website that enables users to interact with the LLVM phase ordering environment. The left side of the page renders the phase ordering search space as an interactive tree; the right side of the page visualizes the program features and their trends.

derived observation spaces such as using custom compiler analyses on compiler IR. These wrappers can be composed. Listing 2 shows integration with the popular RLlib [9] library using two of these wrappers.

D. Command Line Tools

We include a complete set of command line tools for CompilerGym, including scripts to run parallelized searches, replay and validate results from past runs, and an interactive shell that includes inline documentation and tab completion, enabling users to interact with the compiler optimization environments without writing any code.

E. Web Service and CompilerGym Explorer

We designed a REST API to enable CompilerGym environments to be used over a network, and CompilerGym Explorer[3], a web frontend that makes it easy to navigate compiler optimization spaces, implemented using React. CompilerGym Explorer presents a visualization of the search tree, shown in Figure 3, and asynchronously calls the REST API to update the tree in real time as the user interacts with it.

A key feature of the tool is to visualize not only the current state, but also historical trends of the rewards and observation metrics. This allows users to easily pinpoint interesting actions in a large search tree and trigger new explorations. We expect this to be valuable for feature engineering, debugging the behavior of agents, and as a general educational tool.

F. State Transition Dataset

We designed a relational database schema to log the state transitions of CompilerGym environments for later offline analysis, shown in Figure 4. A `Steps` table records every unique action sequence for a particular benchmark and a hash of the environment state. An `Observations` table stores various representations of each unique state, indexed by state hash.

[3]Available at: https://compilergym.ai/explorer

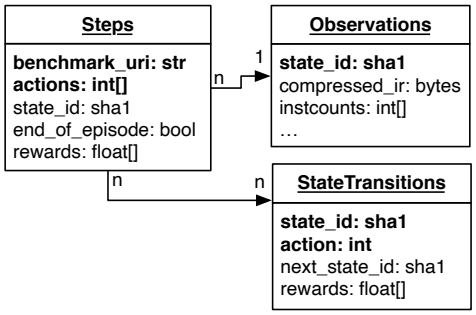

Figure 4: A relational database schema for state transitions in LLVM environments. Fields that comprise unique primary keys are emboldened. We have released an instance of this schema with over 1M unique states that can be used for pre-training, off-policy learning, or general offline analysis.

A `StateTransitions` table encodes the unique transitions between states and the rewards received for each.

We implemented a wrapper class for CompilerGym environments that asynchronously populates the `Steps` and `Observations` tables of a state transition database upon every step of an environment. A post-processing script de-duplicates and populates the `StateTransitions` table.

We have released a large instance of this database (50+GB) which contains over 1M unique LLVM environment states, suitable for a range of offline supervised and unsupervised learning tasks. We evaluate an example usage in Section VII-F.

IV. Backend Runtime and Interface

The CompilerGym backend comprises a CompilationSession interface for integrating compilers and a common client-server runtime that map this interface to the Gym API.

A. The CompilationSession Interface

CompilerGym is designed for seamless compiler integration. The integration centers around implementing a state machine to interact with the compiler called a CompilationSession. A CompilationSession exposes actions and observations using a simple schema and must implement two methods, `apply_action` and `get_observation`, as shown in Figure 5. We provide CompilationSession interfaces for Python and C++. Listing 3 demonstrates an example implementation.

B. Compiler Service Runtime

A common runtime maps implementations of the CompilationSession interface (Listing 3) to the Gym API (Listing 1). This runtime is shared by all compiler integrations and is architected to be performant and scalable. The design is resilient to failures, crashes, infinite loops, and nondeterministic behavior in backend compiler services. All compiler service operations have appropriate timeouts, graceful error handling, or retry loops. Improvements to the runtime can be made without changing compiler integration or user code.

A key design point of the CompilerGym runtime is that the service that provides the compiler integration is isolated in a separate process to the user's Python interpreter. The Python interpreter invokes operations on the compiler service through Remote Procedure Calls (RPCs). The benefits of this are fault

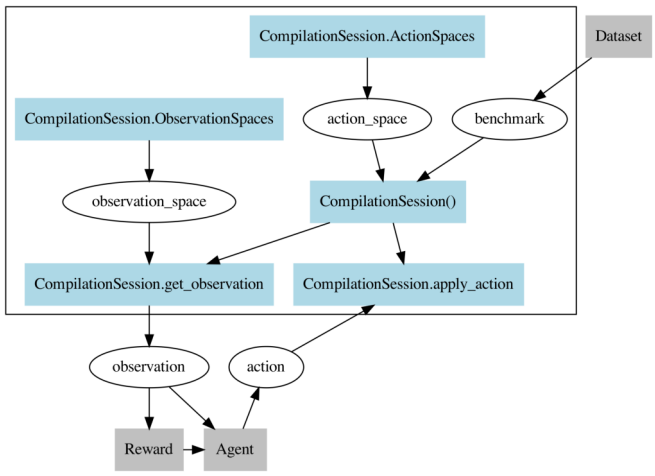

Figure 5: A graphical representation of the CompilationSession integration. To add a new compiler to CompilerGym, users need only define the boxes highlighted in blue. Grey boxes demonstrate the integration of the CompilerSession with a typical reinforcement learning loop.

```
#include "compiler_gym/service/CompilationSession.h"
#include "compiler_gym/service/runtime/Runtime.h"
using namespace compiler_gym;

struct MyCompilationSession: public CompilationSession{
  vector<ActionSpace> getActionSpaces() {...}
  vector<ObservationSpace> getObservationSpaces() {...}

  Status init(
    const ActionSpace& actionSpace,
    const Benchmark& benchmark) {...}
  Status applyAction(
    const Action& action,
    bool& endOfEpisode,
    bool& actionSpaceChanged) {...}
  Status setObservation(
    const ObservationSpace& observationSpace,
    Observation& observation) {...}
};

int main(int argc, char** argv) {
  runtime::createAndRunService<MyCompilationSession>(
    argc, argv, "My compiler service");
}
```

Listing 3: A C++ implementation of the CompilationSession interface to add support for a new compiler. Method bodies are omitted for brevity. There is an equivalent API for Python.

tolerance and recovery in cases where the compiler crashes or terminates abruptly; support for compiling on a different system architecture than the host by running the compiler service on a remote machine; and scalability as the expensive compute work is offloaded, enabling many user threads to interact with separate compiler environments without contention on Python's global interpreter lock.

V. Environments

This section describes three compiler integrations shipped in CompilerGym.

A. LLVM Phase Ordering

LLVM [11] is a modular compiler infrastructure used throughout academia and industry. After parsing an input source program to a language-agnostic Intermediate Representation (IR), the LLVM optimizer applies a configurable pipeline of optimization passes to the IR. The selection and ordering of compiler optimizations – known as *phase ordering* – greatly impacts the quality of the final binary and has been the focus of much research [1], [12].

We include a phase ordering environment in CompilerGym as an example of a challenging, high-dimensional optimization problem in which significant gains can be achieved.

Actions: The action space consists of a discrete choice from 124 optimization passes extracted automatically from LLVM. There is no maximal episode length as episodes can run forever (except in the case of a compiler bug leading to an error), the user must estimate when no further gains can be achieved and no further actions should be taken. For any particular program the optimal phase ordering may omit or repeat actions.

Rewards: We support optimizing for three metrics: *code size*, which is the number of instructions in the IR; *binary size*, which is the size of the `.text` section in the compiled object file; and *runtime*, which is the wall time of the compiled program when run using a specific configuration of inputs on the machine hosting the CompilerGym backend. When used as a reward signal each metric returns the change in value between the previous environment state and the new environment state. Each reward signal can optionally be scaled against the gains achieved by the compiler's default phase orderings, `-Oz` for size reduction and `-O3` for runtime. Code size is platform-independent and determinsitic, binary size is platform-dependent and deterministic, and runtime is both platform-specific and nondeterministic.

Observations: We provide five observation spaces for LLVM ranging from counter-based numeric feature vectors [4] to sequential language models [13] up to graph-based program representations [14]. See Table III for a comparison.

Datasets: We provide millions of programs for evaluation, summarized in Table I. We aggregate C, C++, OpenCL, and Fortran programs from benchmark suites in a variety of domains, open source programs, and synthetic program generators. Accessing these datasets within CompilerGym is as simple as specifying the name of the dataset and optionally the name of a specific benchmark. Presently only cBench [15] and Csmith [8] support optimizing for runtime.

B. GCC Flag Tuning

We include an environment that exposes the optimization space defined by GCC's command line flags. The environment works with any version of GCC from 5 up to and including the current version at time of writing, 11.2. The environment uses Docker images to enable hassle free install and consistency across machines. Alternatively, any local installation of the compiler can be used. This selection is made by simple string specifier of the path or docker image name. The only change that an RL agent needs to make to work with GCC instead of LLVM is to call `env = gym.make("gcc-v0")`, instead of using `"llvm-v0"`.

Table I: LLVM BENCHMARK DATASETS INCLUDED IN COMPILERGYM, COMPARED TO THE NUMBER OF BENCHMARKS USED IN TWO RECENT MACHINE LEARNING WORKS. THE † SYMBOL DENOTES RANDOM PROGRAM GENERATORS WITH 32-BIT SEEDS. EXCLUDING THE PROGRAM GENERATORS, THE TOTAL NUMBER OF BENCHMARKS IS 1,145,499.

Dataset	Number of Benchmarks		
	Autophase [4]	MLGO [3]	CompilerGym
AnghaBench [16]			1,041,333
BLAS [17]			300
cBench [15]			23
CHStone [18]	9		12
CLgen [19]			996
GitHub [14]			49,738
Linux kernel			13,894
MiBench [20]			40
NPB [21]			122
OpenCV			442
POJ-104 [22]			49,816
TensorFlow [23]			1,985
Csmith [8]	100		2^{32}†
llvm-stress [11]			2^{32}†
Proprietary		28,000	

While the LLVM phase ordering action space is unbounded as passes may be executed forever, the number of GCC command line configurations is bounded. GCC's action space consists of all the available optimization flags and parameters that can be specified from the command line. These are automatically extracted from the "help" documentation of whichever GCC version is used. For GCC 11.2.0[4], the optimization space includes 502 options:

- the six `-O<n>` flags, e.g. `-O0, -O3, -Ofast, -Os`.
- 242 flags such as `-fpeel-loops`, each of which may be missing, present, or negated (e.g. `-fno-peel-loops`). Some of these flags may take integer or enumerated arguments which are also included in the space.
- 260 parameterized command line flags such as `--param inline-heuristics-hint-percent=<number>`. The number of options for each of these varies. Most take numbers, a few take enumerated values.

This gives a finite optimization space with a modest size of approximately 10^{4461}. Earlier versions of GCC report their parameter spaces less clearly and so the tool finds smaller spaces when pointed at those. For example, on GCC 5, the optimization space is only 10^{430}.

Actions: We provide two action spaces that can be used interchangeably. The first directly exposes the optimization space via a list of integers, each encoding the choice for one option with a known cardinality. A second action space is intended to make it easy for RL tools that operate on a flat list of categorical actions. For every option with a cardinality of fewer than ten, we provide actions that directly set the choice for that action. For options with greater cardinalities we provide actions that add and subtract 1, 10, 100, and 1000 to the choice integer corresponding to the option. For GCC 11.2.0, this creates a set of 2281 actions that can modify the choices of the current state.

[4]11.2.0 is the latest stable version of GCC at time of writing.

```
for a in 1048576 : L0 [thread]
 for a' in 1 : L1
  for a'' in 1 : L2
   %0[a] <- read()
  for a'' in 1 : L4
   %1[a] <- read()
  for a'' in 1 : L6
   %2[a] <- add(%0, %1)
  for a'' in 1 : L8
   %3[a] <- write(%2)
```

Listing 4: An example loop tree in the loop_tool environment.

Rewards: We provide two deterministic reward signals: the sizes in bytes of the assembly or the object code.

Observations: We provide four observation spaces: a numeric instruction count observation, the Register Transfer Language code at the end of compilation, the assembly code as text, and the object code as a binary.

C. CUDA Loop Nest Code Generation

Manually tuning CUDA code requires sweeping over many parameters. Due to the sheer size of the tunable space, the problem of generating fast CUDA is well suited for automated techniques [24], [25]. As a flexible compilation environment, CompilerGym is well equipped to handle compilers for tuning GPU workloads. We integrated `loop_tool`, a simple dense linear algebra compiler [26]. `loop_tool` takes a minimalist approach to linear algebra representations by decomposing standard BLAS-like routines into a DAG of n-dimensional applications of arithmetic primitives. The DAG is then annotated with three pieces of information about loop ordering: the order in which loops are emitted, the nesting structure of each loop, and the reuse of loops by subsequent operations. This is lowered to a loop tree that can be annotated with which loop should be run in parallel. These four annotations across a slew of point-wise operations represent a large optimization space.

Actions: We map interacting with the loop structure for point-wise additions to a cursor-based discrete action space. At any point the cursor will refer to an individual loop in the loop hierarchy and will have an associated "mode" to control either moving the cursor or modifying the current loop. There is an action "toggle_mode" to swap between these two. When moving the cursor, the actions "up" and "down" will shift the cursor inward and outward respectively. When modifying the current loop, the action "up" will increase its size by one. This is done by changing the size of the parent loop to accommodate the new inner size. Often this induces tail logic, which is handled automatically. Finally, any loop can be changed to be threaded. This will schedule loop execution across CUDA threads which may span multiple warps or even multiple streaming multiprocessors. A second, extended action space allows loops to be split, creating a larger hierarchy.

Rewards: The environment reward signal is a measurement of floating point operations per second (FLOPs) achieved by benchmarking the loop nest in the given state. This is both platform dependent and non-deterministic due to the noise involved in benchmarking.

Observations: There are two observations spaces: action state, which describes the cursor position and mode, and loop tree structure, which is a textual dump of the current state of the `loop_tool` environment, as shown in Listing 4.

VI. Implementation

CompilerGym is implemented in a mixture of Python and C++. The core runtime comprises 12k lines of code. The compiler integrations comprise 6k lines of code for LLVM, 3k for GCC and 0.5k for `loop_tool`. CompilerGym is open source and available under a permissive license.

Binary Releases: Periodic versioned releases are made from a stable branch. We ship pre-compiled release binaries for macOS and Linux (Ubuntu 18.04, Fedora 28, Debian 10 or newer equivalents) that can be installed as Python wheels.

Documentation: Our public facing documentation includes full API references for Python and C++, a getting started guide, FAQ, and code samples demonstrating integration with RLlib [9], implementations of exhaustive, random, and greedy searches, and Q-learning [27] and Actor Critic [28].

Testing: We have a comprehensive unittest suite with 85.8% branch coverage that is run on every code change across a test matrix of all supported operating systems and Python versions. Additionally, a suite of fuzz and stress tests are ran daily by continuous integration services to proactively identify issues.

VII. Evaluation

We evaluate CompilerGym first by comparing the computational efficiency of the environments to prior works. We then show how the simplicity of the CompilerGym APIs enables large-scale autotuning and reinforcement learning experiments to be engineered with remarkably few lines of code.

Experimental Platforms: Results in this section are obtained from shared compute servers equipped with Intel Xeon 8259CL CPUs, NVIDIA GP100 GPUs, and flash storage.

A. Computational Efficiency

A key design goal of CompilerGym is to provide the best performance possible, enabling researchers to train larger models, try more configurations, and get better results in less time. We evaluate the computational efficiency of CompilerGym's LLVM phase ordering environment and compare to two prior works: Autophase [4] and OpenTuner [29].

We use code size rewards signals for all three platforms and the observation space used in [4] for Autophase and CompilerGym; OpenTuner is a black box search framework and so does not provide observation spaces. We measure the computational efficiencies of each environment by measuring the wall times of operations during 1M random trajectories. For CompilerGym, which uses a client-server architecture, we also measure the initial server startup time.

Table II shows the results. CompilerGym achieves a much higher throughput than Autophase while offering the same interface, observation space, and reward signal. This is enabled by CompilerGym's client-server architecture. After initially reading and parsing the bitcode file from disk, the CompilerGym server incrementally applies an individual optimization pass at each step. In contrast, Autophase and OpenTuner must, at each step, read and parse the IR, apply the entire sequence of passes, and then serialize the result. OpenTuner, which was designed for uses where the search time is dominated by compilation time, has the highest environment initialization cost, as it requires several disk operations and the creation of a database. The CompilerGym server maintains a cache of parsed unoptimized bitcodes that enables an amortized $\mathcal{O}(1)$ cost of environment initialization.

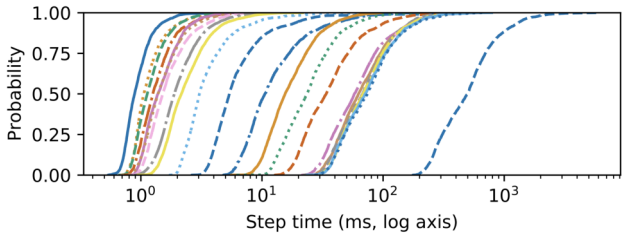

Figure 6: Cumulative density plot of step times for each of the 23 programs in cBench [15]. Each line shows a different program. The difference between the median step times of the fastest program (crc32) and the slowest program (ghostbench) is 560.3×.

The distribution of operation wall times depends on the action being performed and the program being optimized. Figure 6 shows the wide distribution of wall times within the benchmarks of a single dataset.

B. Computational Efficiency of Observation Spaces

This experiment evaluates the computational efficiency of the LLVM environment observation and reward spaces. We recorded 1M wall times of each using random trajectories.

Table III summarizes the results. There is a 192× range in observation space times, demonstrating a tradeoff between observation space computational cost and fidelity; and 4727× range in reward space times, motivating the development of fast approximate proxy rewards and cost models [30], [31].

C. Autotuning LLVM Phase Ordering

We evaluate various autotuning techniques on the LLVM phase ordering task to demonstrate the ease and speed of CompilerGym. We use the following autotuning techniques: Greedy search, which at each step evaluates all possible actions and selects the action which provides the greatest reward, terminating once no positive reward can be achieved by any action; LaMCTS [32], an extension of Monte Carlo Tree Search [33] that partitions the search space on the fly to focus on important search regions [33]; Nevergrad [34] and OpenTuner [29], two black box optimization frameworks that contain ensembles of techniques; and Random Search, which selects actions randomly until a configurable number of steps have elapsed without a positive reward.

We run single threaded versions of each autotuning technique on each benchmark in the cBench [15] suite for one hour. Hyperparameters for all techniques were tuned on a validation set of 50 Csmith [8] benchmarks. We evaluate each technique when optimizing for three different targets: code size, binary size, and runtime. For runtime we use the median of three

Table II: COMPUTATIONAL COSTS OF COMPILERGYM OPERATIONS COMPARED TO PRIOR WORKS WHEN COMPUTING THE SAME ACTIONS, OBSERVATIONS, AND REWARDS. AFTER PAYING A ONE-OFF STARTUP PENALTY, COMPILERGYM IS 27× FASTER THAN AN EQUIVALENT PRIOR WORK. THIS MUCH HIGHER THROUGHPUT ENABLES TRAINING LARGER MODELS WITH LARGER DATASETS. COST DENOTES AVERAGE-CASE TIME COMPLEXITIES *wrt.* n THE SIZE OF THE PROGRAM BEING COMPILED AND m THE NUMBER OF ACTIONS IN THE EPISODE. † DENOTES AMORTIZED COST, ACHIEVED BY CACHING INITIAL ENVIRONMENT STATES. P50 AND P99 DENOTE THE 50TH AND 99TH PERCENTILE OF WALL TIMES, RESPECTIVELY. μ DENOTES THE ARITHMETIC MEAN WALL TIME. WE ALSO MEASURED THE THROUGHPUT OF COMPILERGYM WHEN BATCHES OF ACTIONS ARE PROCESSED IN A SINGLE ENVIRONMENT STEP, DENOTED COMPILERGYM-BATCHED ABOVE. THIS IMPROVES THROUGHPUT BY A FURTHER 2.9× BY REDUCING RPC ROUND TRIPS, BUT LOSES INTERMEDIATE OBSERVATIONS AND REWARDS. MEASUREMENTS TAKEN FROM 1M RANDON TRAJECTORIES, EVENLY DIVIDED ACROSS ALL BENCHMARK DATASETS.

	Service Startup				Environment Initialization				Environment Step			
	Cost	p50	p99	μ	Cost	p50	p99	μ	Cost	p50	p99	μ
Autophase [4]	—	—	—	—	$\mathcal{O}(n)$	22.4ms	388.4ms	53.3ms	$\mathcal{O}(nm)$	71.0ms	2,489.8ms	205.9ms
OpenTuner [29]	—	—	—	—	$\mathcal{O}(n)$	269.6ms	8,515.3ms	777.5ms	$\mathcal{O}(nm)$	50.7ms	1,491.1ms	131.2ms
CompilerGym	$\mathcal{O}(1)$	119.7ms	131.8ms	120.8ms	$\boldsymbol{\mathcal{O}(1)}$†	**2.2ms**	**198.6ms**	**21.3ms**	$\boldsymbol{\mathcal{O}(n)}$	**1.0ms**	**108.6ms**	**7.5ms**
CompilerGym-batched	”	”	”	”	”	”	”	”	”	**0.2ms**	**37.4ms**	**2.6ms**

Table III: COMPUTATIONAL COSTS OF THE OBSERVATION AND REWARD SPACES OF LLVM ENVIRONMENTS FROM 1M WALL TIME MEASUREMENTS, EVENLY DIVIDED ACROSS ALL BENCHMARK DATASETS. P50 AND P99 DENOTE THE 50TH AND 99TH PERCENTILE, RESPECTIVELY, AND μ DENOTES THE ARITHMETIC MEAN.

	Type	p50	p99	μ
LLVM-IR	String	0.9ms	72.1ms	5.9ms
InstCount	70-D int64 vector	0.5ms	6.9ms	0.9ms
Autophase [4]	56-D int64 vector	0.7ms	38.0ms	3.4ms
inst2vec [13]	200-D float vector list	15.8ms	31,847ms	738.1ms
ProGraML [14]	Directed multigraph	104.5ms	14,194ms	821.5ms
Code size	Int64 count	0.4ms	3.6ms	0.4ms
Binary size	Int64 byte count	56.2ms	703.7ms	98.1ms
Runtime	Float wall time	75.9ms	8,406ms	614.4ms

Table IV: LINES OF CODE REQUIRED TO INTEGRATE OR IMPLEMENT AUTOTUNING TECHNIQUES FOR THE LLVM PHASE ORDERING TASK, AND PERFORMANCE ACHIEVED WHEN OPTIMIZING FOR THREE TARGETS ON THE CBENCH SUITE [15] GIVEN A 1HR SEARCH BUDGET. RESULTS ARE COMPARED AGAINST `-Oz` FOR SIZE REDUCTION AND `-O3` FOR RUNTIME.

	Lines of code	Geomean code size reduction	Geomean binary size reduction	Geomean runtime speedup
Greedy Search	10	1.053×	1.267×	1.059×
LaMCTS [32]	35	1.051×	1.273×	1.053×
Nevergrad [34]	41	**1.083×**	**1.318×**	**1.093×**
OpenTuner [29]	165	1.060×	1.102×	0.822×
Random Search	24	1.048×	1.278×	1.078×

measurements to provide the reward signals during search, and the median of 30 measurements for final reported values. Each experiment was repeated 10 times.

The standard interface exposed by CompilerGym makes it simple to integrate with third party autotuning libraries or to develop new autotuning approaches. Table IV shows the number of lines of code required to integrate each search technique, and the performance achieved.

Phase ordering is challenging because the optimization space is unbounded, high-dimensional, and contains sparse rewards. Nevertheless, autotuning – when furnished with a sufficiently generous search budget – outperforms the default compiler heuristics by tailoring the configuration to each benchmark. We note that the optimal configuration differs between all benchmarks and optimization targets.

D. Autotuning GCC Command Line Flags

For GCC we show a different aspect of the CompilerGym. For these experiments we explore the GCC environment's high-dimensional action space using a number of simple search techniques. These experiments are performed using GCC version 11.2.0 in Docker. That version of GCC has 502 optimization settings that can be selected. We evaluate three search techniques on the the CHstone [18] suite:

1) *Random search.* A random list of 502 integers from the allowable range is selected at each step.

2) *Hill climbing search.* At each step a small number of random changes are made to the current choices. If this improves the objective then the current state is accepted and future steps modify from there.

3) *Genetic algorithm (GA).* A population of 100 random choices is maintained. We use the Python library `geneticalgorithm` [35] with its default parameters.

Table V shows the geometric mean of the object code size objective across the benchmarks in CHstone [18], averaged over 3 searches. Each search was allowed 1000 compilations.

E. Autotuning CUDA Loop Nests

The `loop_tool` environment provides an easily accessible interface to being exploring the landscape of GPU optimizations. Tuning a simple space by searching threading and then sizing the inner loop reaches 73.5% of theoretical peak performance on our GP100 test hardware (∼6e10 FLOPs or ∼750GB/s for two 4-byte floating point reads and one write), and parity with PyTorch performance on the same operation across a variety of problem sizes. Figure 7 shows the results for different loop configurations, demonstrating potentially useful hardware and compiler characteristics, notably a drop in performance near 100k threads.

F. Learning a Cost Model using the State Transition Dataset

Auxiliary tasks are commonly used in reinforcement learning to produce better representation learning and help with downstream tasks [36], [37]. This experiment demonstrates using the State Transition Dataset (Section III-F) to learn a cost model of instruction count from a graph representation of program state.

Table V: LINES OF CODE REQUIRED TO INTEGRATE OR IMPLEMENT AUTOTUNING TECHNIQUES FOR THE GCC FLAG TUNING TASK, AND PERFORMANCE ACHIEVED WHEN OPTIMIZING THE CHSTONE SUITE [18], GIVEN A SEARCH BUDGET OF 1000 COMPILATIONS PER BENCHMARK. RESULTS ARE COMPARED AGAINST `-Os`.

	Lines of code	Geomean binary size reduction
Genetic Algorithm [35]	27	**1.27×**
Hill Climbing	14	1.04×
Random Search	9	1.21×

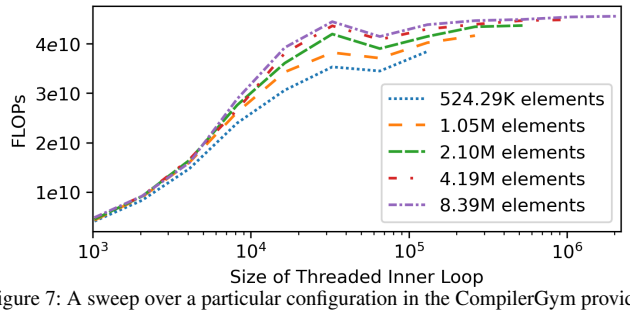

Figure 7: A sweep over a particular configuration in the CompilerGym provided search space for point-wise addition on a GPU using `loop_tool` to generate CUDA.

We implemented a Gated Graph Neural Network [38] in PyTorch [39] and used Mean Squared Error loss to train a regressor to predict the instruction count of a program after two rounds of message passing on the PROGRAML [14] graph representation built into CompilerGym. We trained on 80% of the State Transition Database by iterating over pairs of (graph, instruction count) from the database. We used the remaining 20% of the database as a validation set. Figure 8 shows the convergence of the neural network. The network achieves a relative error of 0.025, while a naive mean prediction scores a relative error of 1.393.

G. Reinforcement Learning for LLVM Phase Ordering

CompilerGym offers seamless integration with third party reinforcement learning frameworks. For example, by changing a single parameter value in Listing 2 we can use any of the 26 reinforcement learning algorithms included in RLlib [9].

Table VI: COMPARISON OF FOUR REINFORCEMENT LEARNING ALGORITHMS, TRAINED FOR 100K EPISODES ON CSMITH [8] PROGRAMS, WHEN EVALUATED ON DATASETS FROM A RANGE OF PROGRAM DOMAINS. THE PROGRAMS USED FOR TESTING ON CSMITH ARE DIFFERENT FROM THOSE USED FOR TRAINING. RESULTS ARE COMPARED TO `-Oz`.

Test Dataset	Geomean code size reduction			
	A2C [40]	APEX [41]	IMPALA [42]	PPO [43]
AnghaBench [16]	0.951×	0.659×	**0.958×**	0.776×
BLAS [17]	0.928×	**0.934×**	0.861×	0.906×
cBench [15]	0.804×	0.698×	0.814×	**0.964×**
CHStone [18]	0.823×	0.704×	0.707×	**1.014×**
CLgen [19]	**0.950×**	0.687×	0.916×	0.843×
Csmith [8]	1.023×	0.692×	1.144×	**1.245×**
GitHub [14]	0.975×	**0.987×**	0.976×	0.984×
Linux kernel	0.987×	**0.998×**	0.983×	0.995×
llvm-stress [11]	**0.838×**	0.493×	0.736×	0.097×
MiBench [20]	0.996×	0.996×	0.996×	**1.000×**
NPB [21]	**0.961×**	0.816×	0.958×	0.923×
OpenCV	0.976×	0.969×	**0.986×**	0.945×
POJ-104 [22]	0.778×	0.651×	**0.805×**	0.801×
TensorFlow [23]	0.976×	**0.976×**	0.966×	0.933×

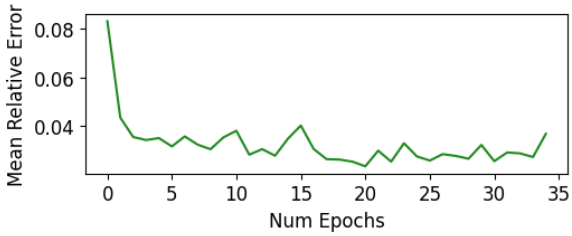

Figure 8: Predicting program instruction using a Graph Neural Networks trained on CompilerGym's State Transition Dataset. This shows performance on a holdout validation set as a function of the number of training epochs.

We use CompilerGym to replicate the LLVM phase ordering environment used in [4]. Specifically: we fix episode lengths to 45 steps, use the same observation space comprising a feature vector concatenated with a histogram of the agent's previous actions, and we use a subset of the full action space[5]. We note that each of these modifications to the base LLVM environment can be achieved using the wrapper classes built into CompilerGym (Section III-C). Our environment differs from [4] in that we use a code size reward signal rather than simulated cycle counts.

We train three different reinforcement learning algorithms for 100k episodes and periodically evaluate performance on a holdout validation set. We use Csmith to generate both training and validation sets, as in [4].

Table VI shows the performance of the trained agents when evaluated on a random 50 programs from each of the datasets available out of the box in CompilerGym. 3 of the 4 algorithms achieve positive results when generalizing to programs within the same domain (Csmith), but only PPO [43] is able to achieve a positive score on two of the 13 other datasets. This highlights the challenge of generalization across program domains.

H. Effect of Training Set on RL

The generalization of reinforcement learning agents across domains is the subject of active research [44], [45], [46]. As demonstrated in the previous experiment, the performance of agents trained on one dataset can differ wildly on datasets from other domains. We evaluate the effect of training set on generalization by training a PPO [43] agent on different training sets and then evaluating their generalization performance on test sets from different domains. All other experimental parameters are as per the previous experiment.

Table VII shows the results. As can be seen, each algorithm performs best when generalizing to benchmarks from within the same dataset, suggesting the importance of training on benchmarks across a wide range of program domains.

I. Effect of Program Representation on Learning

Representation learning and feature engineering is an area of much research [47], [48], [2]. CompilerGym environments

[5]We use 42 actions (out of 124 total) rather than the 45 actions used in [4] as three of the actions have been removed in recent versions of LLVM.

Table VII: COMPILERGYM INCLUDES MILLIONS OF PROGRAMS TO TRAIN ON THAT CAN BE SELECTED BY SIMPLY SPECIFYING THE NAME OF THE DATASET(S) TO USE. HERE WE CROSS-VALIDATE THE GENERALIZATION PERFORMANCE OF A PPO [43] AGENT BY VARYING THE TRAINING AND TEST SETS. THE ROW INDICATES THE DATASET USED FOR TRAINING, THE COLUMN INDICATES THE DATASET USED FOR TESTING. THE VALUES ARE GEOMEAN CODE SIZE REDUCTION RELATIVE TO `-Oz`.

		Training Set		
		Csmith [8]	Github [14]	TensorFlow [23]
Test Set	Csmith [8]	**1.245×**	0.567×	0.723×
	Github [14]	0.984×	**0.981×**	0.995×
	TensorFlow [23]	0.932×	0.950×	**0.998×**

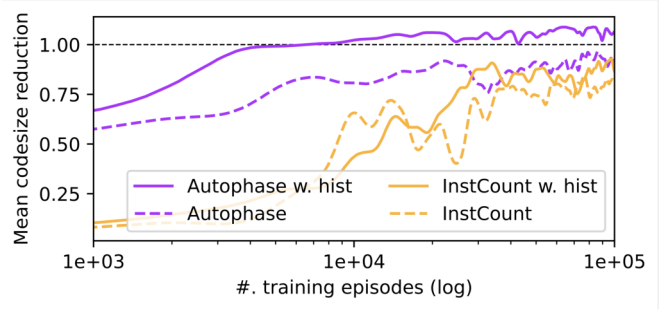

Figure 9: The convergence rate and final performance of agents depends on the observation space. CompilerGym includes several observation spaces out of the box. By changing one line of code we trained PPO [43] agents on different observation spaces. This plot shows the performance on a holdout validation set as a function of the number of training episodes. We applied a Gaussian filter ($\sigma = 5$) to aid in visualizing the trends.

provide multiple state representations for each environment. We evaluate the performance of two different program representations, and their performance when concatenated with a histogram of the agent's previous actions, as used in [4]. We use the same experimental setup as in the prior sections.

The results are shown in Figure 9. In both cases stronger performance is achieved when coupling the program representation with a histogram of the agent's previous actions. The Autophase representation encodes more attributes of the structure of programs than InstCount and achieves greater performance. We believe that representation learning is one of the most exciting areas for future research, and CompilerGym provides the supporting infrastructure for this research.

VIII. RELATED WORK

We present a suite of tools for compiler optimization research. Other compiler research tools include OpenTuner [29] and YaCoS [49], autotuning frameworks that include an ensemble of techniques for compiler optimizations; cTuning [50], a framework for distributing autotuning results; TenSet [51] and LS-CAT [52], large-scale performance datasets suitable for offline learning; and ComPy-Learn [53], a library of program representations for LLVM. CompilerGym has a broader set of features than these prior works, providing several compiler problems, program representations, optimization targets, and offline datasets all in a single package.

There is a growing body of research that applies AI techniques to compilers optimizations [2]. Many approaches have been proposed to phase ordering, including collaborative filtering [54], design space exploration [55], and Bayesian Networks [56]. Even removing passes from standard optimization pipelines has been shown to sometimes improve performance [57]. Autophase [4] and CORL [58] use reinforcement learning to tackle the LLVM phase ordering problem. Both works identify generalization across programs as a key challenge. Our work aims to accelerate progress on this problem by combining several observation spaces with millions of training programs to serve as a platform for research.

Other reinforcement learning compiler works include MLGO [3] which learns a policy for a function inling heuristic, NeuroVectorizer [5] which formulates instruction vectorization as single-step environments, and PolyGym [59] which targets Polyhedral loop transformations. Compared to these works, the search spaces in CompilerGym environments are far larger.

CompilerGym is not limited to reinforcement learning. Prior work has cast compiler optimization tasks as supervised learning problems using classification to select optimization decisions [60], [2] or regression to learn cost models [61], [30], [31]. CompilerGym is as an ideal platform for gathering the data to train and evaluate these approaches, including both offline datasets and the infrastructure to generate new ones.

IX. CONCLUSIONS

We aim to lower the barrier-to-entry to compiler optimization research. We present CompilerGym, a suite of tools that removes the significant engineering investment required try out new ideas on production compiler problems.

APPENDIX

A. Abstract

Our artifact comprises source code and pre-compiled binaries for CompilerGym, and instructions to reproduce the nine experiments in the paper.

B. Artifact Check-List (Meta-information)

- **Run-time environment:** Ubuntu 18.04, Fedora 28, Debian 10 or newer, or macOS. Python 3.8.
- **Experiments:** Computational efficiency benchmarks; autotuning for LLVM phase ordering, GCC command line flags, and CUDA loop nests; supervised graph learning; and reinforcement learning for LLVM phase ordering.
- **How much disk space required (approximately)?:** 20GB.
- **How much time is needed to prepare workflow (approximately)?:** 15 minutes.
- **How much time is needed to complete experiments (approximately)?:** 3 hours.
- **Publicly available?:** Yes. https://compilergym.ai

C. Description

1) How Delivered: The artifact is available from: https://doi.org/10.5281/zenodo.5784251.

2) Hardware Dependencies: We recommend a minimum of 64GB RAM. We used a commodity desktop machine equipped with a AMD Ryzen 9 3900X CPU and NVIDIA 2080 GPU to generate the compute time estimates in this document. A CUDA-equipped GPU is required for one of the experiments (Section VII-E). The following experiments contain runtime measurements and so will produce different vakyes on your system: Sections VII-A, VII-B, VII-C, and VII-E. The remainder of the experiments are hardware-agnostic.

3) Software Dependencies: All experiments require Python 3.8. We recommend using anaconda to prevent interference with your system Python. Docker is required for one of the experiments (Section VII-D).

D. Installation

(Optional) Use conda[6] to create a clean Python environment:

```
conda create -y -n compiler_gym python=3.8
conda activate compiler_gym
```

Confirm that your environment is using Python 3.8:

```
python --version
```

Next, install the CompilerGym library using:

```
python -m pip install -U compiler_gym
```

Either download the artifact archive linked above, or clone the repository to get the most recent version using:

```
git clone \
    https://github.com/facebookresearch/CompilerGym.git
```

Change to the examples directory and install the dependencies using:

```
cd CompilerGym/examples
python setup.py install
```

The remainder of this document requires that the user remains in the `CompilerGym/examples` directory.

E. Experiments

This section describes how to reproduce all of the experiments in the paper. The workflow, evaluation, and customization steps of each experiment is described in turn.

(Optional) Getting Started with CompilerGym

Before running the experiments below we recommend navigating to https://compilergym.ai/getting_started and clicking the "Open in Colab" button to launch an interactive webpage that describes the main CompilerGym concepts and APIs. It takes approximately 20 minutes to complete.

Sections VII-A and VII-B: Computational Efficiency

Workflow: The following script benchmarks the computation times of the three environment operations (startup, initialization, and step), and each of the eight observation spaces:

```
python -m op_benchmarks run --n=200
```

[6]If not already available, see the documentation for installation instructions: https://docs.conda.io/projects/conda/en/latest/user-guide/install/index.html

where `--n` is the number of runtimes to collect for each type of operation. Collecting `--n=200` measurements will take about 10 minutes. In the paper we use `--n=1000000`.

Evaluation: Aggregate measurements and compare the runtimes to Table II and Table III using:

```
python -m op_benchmarks info
```

Section VII-C: Autotuning LLVM Phase Ordering

Workflow: Run a single autotuning experiment using:

```
export AUTOTUNER=nevergrad; TARGET=codesize; \
python -m llvm_autotuning.tune -m \
    experiment="${AUTOTUNER}-${TARGET}" \
    autotuner="${AUTOTUNER}" \
    autotuner.optimization_target="${TARGET}" \
    num_replicas=1 \
    autotuner.search_time_seconds=300
```

where `AUTOTUNER` is set to one of `nevergrad`, `random`, `greedy`, or `opentuner`; and `TARGET` is set to one of `codesize`, `binsize`, or `runtime`. Run all 12 combinations of autotuner and target to reproduce the full suite of experiments in Section VII-C.

The above command will take about 6 minutes to complete. Compared to the experiment in the paper each autotuner is run only once and with a search budget of 5 minutes instead of one hour. To replicate the full experiment from Section VII-C, taking about 12 hours, run:

```
export AUTOTUNER=nevergrad; TARGET=codesize; \
python -m llvm_autotuning.tune -m \
    experiment="${AUTOTUNER}-${TARGET}" \
    autotuner="${AUTOTUNER}" \
    autotuner.optimization_target="${TARGET}" \
    num_replicas=10 \
    autotuner.search_time_seconds=3600
```

Evaluation: Aggregate and compare to Table IV using:

```
python -m llvm_autotuning.info
```

Customization: The experiment is configurable through several command line arguments described in `llvm_autotuning/README.md`. Example customizations include `autotuner.search_time_seconds` to change the search budget, and `autotuner.algorithm_config.episode_length` to set the number of steps in Nevergrad environment episodes.

Section VII-D: Autotuning GCC Command Line Flags

Workflow: Run a small-scale autotuning experiment using:

```
python -m gcc_autotuning.tune \
    --gcc_search_budget=100 \
    --pop_size=20 \
    --gcc_search_repetitions=1 \
    --gcc_benchmark=benchmark://chstone-v0/aes,benchmark://
        chstone-v0/blowfish,benchmark://chstone-v0/dfmul,
        benchmark://chstone-v0/dfsin,benchmark://chstone-v0
        /sha
```

Note the final `--gcc_benchmark` argument above should entered as a single line without spaces. This requires docker. If docker is not installed, use `--gcc_bin=/path/to/gcc` to specify the path to a GCC installation. The above command takes about 12 minutes to complete and uses fewer benchmarks and a smaller search budget than in the paper. To replicate the full suite setup from Section VII-D, taking about 6 hours, run:

```
python -m gcc_autotuning.tune
```

Evaluation: Aggregate and compare to Table V using:

```
python -m gcc_autotuning.info
```

Customization: The experiment is configurable through a number of command line arguments described in `gcc_autotuning/README.md`. Example customizations include `--gcc_search_budget` to change the search budget, `--gcc_bin` to use a different GCC binary, and `--gcc_benchmark` to change the list of programs used.

Section VII-E: Autotuning CUDA Loop Nests

Workflow: Run a sweep on a CUDA-capable GPU using:

```
python -m loop_tool_sweep --device=cuda --k=128
```

This will take approximately 3 minutes to complete. In Section VII-E this experiment is repeated using the following values for `--k`: 512, 1024, 2048, 4096, and 8192.

Evaluation: Compare values logged to stdout to Figure 7.

Customization: Use `--k` and `--vectorize` to specify the dataset and vectorization sizes, respectively, and `--device=cpu` to run on CPU.

Section VII-F: Learning a Cost Model

Workflow: Train a graph neural network cost model using:

```
python -m gnn_cost_model.train \
    --device=cpu --batch_size=4 \
    --num_epoch=10 --dataset_size=100
```

The above command takes around 10 minutes to complete and uses 100 graphs from a small 3 GB state transition dataset. To replicate the full scale experiment of Section VII-F, run:

```
python -m gnn_cost_model.train --num_epoch=35 \
    --db=https://dl.fbaipublicfiles.com/compiler_gym/
         state_transition_dataset/2021-11-15-cbench.tar.bz2
    --db_sha256=36
         dddceca405126a1249c640cba5b678d4a1db3d9298ad7e5f1d
        a2fa4eccfd20
```

This requires a minimum of 60 GB of RAM and disk space.

Evaluation: Compare loss logged to stdout to Figure 8.

Customization: The training script can be configured through several command line arguments. List them using:

```
python -m gnn_cost_model.train --help
```

Sections VII-G, VII-H, and VII-I: Reinforcement Learning

Workflow: Train a PPO [43] agent for 1000 episodes using:

```
python -m llvm_rl.train -m \
    experiment=algo \
    agent=ppo \
    num_replicas=1 \
    training.episodes=1000
```

This will take approximately 20 minutes to complete. In the paper, agents are trained for `training.episodes=100000` and repeated 10 times (`num_replicas=10`).

To train using four different algorithms and evaluating on all test sets (Section VII-G), run:

```
python -m llvm_rl.train -m \
    experiment=algo \
    agent=a2c,apex,impala,ppo testing=all
```

To cross-validate the effect of different training and test sets (Section VII-H), run:

```
python -m llvm_rl.train -m \
    experiment=training-set \
    training=csmith,github,tensorflow \
    testing=csmith-github-tensorflow
```

To evaluate the effect of different observation spaces on agents (Section VII-I), run:

```
python -m llvm_rl.train -m \
    experiment=observation-spaces \
    training.episodes=100000 \
    environment="autophase,autophase-with-history, \
        instcount,instcount-with-history"
```

Evaluation: Run the following command to provide an overview of the training progress of each of the agents:

```
python -m llvm_rl.info train
```

This will print the training time and geometric mean rewards achieved on the training and validation sets. To summarize agent performance on the holdout test set(s), run:

```
python -m llvm_rl.info test
```

Customization: There are a large number of configuration options available for the experiment, configured through command line arguments. These are described in `llvm_rl/README.md`. Example customizations include `agent.type` to specify the reinforcement learning algorithm to use, `environment.max_episode_steps` to fix the number of steps in each episode, `environment.reward_space` to specify the objective function to optimize for, and `training.episodes` to specify the number of episodes to train for.

F. Uninstallation

After completing the evaluation of this artifact, you can delete the conda environment using:

```
conda deactivate
conda remove --all --name compiler_gym
```

This will remove the Python package and any installed dependencies. The following directories can be removed to tidy up CompilerGym's runtime files and experimental results:

```
rm -rf ~/logs/compiler_gym
rm -rf ~/.local/share/compiler_gym
rm -rf ~/.cache/compiler_gym
```

Finally, you can delete the directory where you cloned the artifact repository.

References

[1] Amir H Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. A Survey on Compiler Autotuning using Machine Learning. *CSUR*, 51(5), 2018.

[2] Hugh Leather and Chris Cummins. Machine Learning in Compilers: Past, Present and Future. In *FDL*, 2020.

[3] Mircea Trofin, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski, and David Li. MLGO: a Machine Learning Guided Compiler Optimizations Framework. *arXiv:2101.04808*, 2021.

[4] Ameer Haj-Ali, Qijing Huang, William Moses, John Xiang, John Wawrzynek, Krste Asanovic, and Ion Stoica. Autophase: Juggling hls phase orderings in random forests with deep reinforcement learning. In *MLSys*, 2020.

[5] Ameer Haj-Ali, Nesreen K Ahmed, Ted Willke, Yakun Sophia Shao, Krste Asanovic, and Ion Stoica. Neurovectorizer: End-to-end vectorization with deep reinforcement learning. In *CGO*, 2020.

[6] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A Survey of Machine Learning for Big Code and Naturalness. *CSUR*, 51(4), 2018.

[7] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *arXiv:1606.01540*, 2016.

[8] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *PLDI*, 2011.

[9] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. RLlib: Abstractions for Distributed Reinforcement Learning. In *ICML*, 2018.

[10] William M McKeeman. Differential Testing for Software. *Digital Technical Journal*, 10(1), 1998.

[11] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*, 2004.

[12] Yang Chen, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Liang Peng, Olivier Temam, and Chengyong Wu. Evaluating Iterative Optimization Across 1000 Datasets. In *PLDI*, 2010.

[13] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. Neural Code Comprehension: A Learnable Representation of Code Semantics. In *NeurIPS*, 2018.

[14] Chris Cummins, Zacharias Fisches, Tal Ben-Nun, Torsten Hoefler, Michael O'Boyle, and Hugh Leather. ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations. In *ICML*, 2021.

[15] Grigori Fursin, John Cavazos, Michael O'Boyle, and Olivier Temam. MiDataSets: Creating the conditions for a more realistic evaluation of iterative optimization. In *HiPEAC*, 2007.

[16] Anderson Faustino da Silva, Bruno Conde Kind, José Wesley de Souza Magalhães, Jerônimo Nunes Rocha, Breno Campos Ferreira Guimaraes, and Fernando Magno Quinão Pereira. AnghaBench: A Suite with One Million Compilable C Benchmarks for Code-Size Reduction. In *CGO*, 2021.

[17] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *TOMS*, 5(3), 1979.

[18] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii. CHStone: A Benchmark Program Suite for Practical C-based High-Level Synthesis. In *ISCAS*, 2008.

[19] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. Synthesizing Benchmarks for Predictive Modeling. In *CGO*, 2017.

[20] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *WWC*, 2001.

[21] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS Parallel Benchmarks 2.0. Technical report, Technical Report NAS-95-020, NASA Ames Research Center, 1995.

[22] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional Neural Networks Over Tree Structures for Programming Language Processing. In *AAAI*, 2016.

[23] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, 2016.

[24] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *OSDI*, 2020.

[25] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *PLDI*, 2013.

[26] Bram Wasti. loop_tool. https://github.com/facebookresearch/loop_tool, 2021.

[27] Christopher JCH Watkins and Peter Dayan. Q-Learning. *Machine learning*, 8(3-4), 1992.

[28] Vijay R Konda and John N Tsitsiklis. Actor-Critic Algorithms. In *NeurIPS*, 2000.

[29] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. OpenTuner: An Extensible Framework for Program Autotuning. In *PACT*, 2014.

[30] Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks. In *ICML*, 2019.

[31] Benoit Steiner, Chris Cummins, Horace He, and Hugh Leather. Value Learning for Throughput Optimization of Deep Learning Workloads. In *MLSys*, 2021.

[32] Linnan Wang, Rodrigo Fonseca, and Yuandong Tian. Learning Search Space Partition for Black-Box Optimization using Monte Carlo Tree Search. In *NeurIPS*, 2020.

[33] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-Carlo Tree Search: A New Framework for Game AI. *AIIDE*, 8, 2008.

[34] Jeremy Rapin and Olivier Teytaud. Nevergrad - A Gradient-Free Optimization Platform. https://github.com/facebookresearch/nevergrad, 2018.

[35] Ryan Solgi. geneticalgorithm. https://pypi.org/project/geneticalgorithm/, 2020.

[36] Marc G. Bellemare, Will Dabney, Robert Dadashi, Adrien Ali Taïga, Pablo Samuel Castro, Nicolas Le Roux, Dale Schuurmans, Tor Lattimore, and Clare Lyle. A Geometric Perspective on Optimal Representations for Reinforcement Learning. *CoRR*, abs/1901.11530, 2019.

[37] Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z. Leibo, David Silver, and Koray Kavukcuoglu. Reinforcement Learning with Unsupervised Auxiliary Tasks. *CoRR*, abs/1611.05397, 2016.

[38] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated Graph Sequence Neural Networks. *arXiv:1511.05493*, 2015.

[39] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An Imperative Style, High-performance Deep Learning Library. *arXiv:1912.01703*, 2019.

[40] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. In *ICML*, 2016.

[41] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado Van Hasselt, and David Silver. Distributed Prioritized Experience Replay. In *ICML*, 2018.

[42] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures. In *ICML*, 2018.

[43] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *arXiv:1707.06347*, 2017.

[44] Karl Cobbe, Oleg Klimov, Chris Hesse, Taehoon Kim, and John Schulman. Quantifying Generalization in Reinforcement Learning. In *ICML*, 2019.

[45] Kaixin Wang, Bingyi Kang, Jie Shao, and Jiashi Feng. Improving Generalization in Reinforcement Learning with Mixture Regularization. In *NeurIPS*, 2020.

[46] Yaniv Ovadia, Emily Fertig, Jie Ren, Zachary Nado, David Sculley, Sebastian Nowozin, Joshua V Dillon, Balaji Lakshminarayanan, and Jasper Snoek. Can You Trust Your Model's Uncertainty? Evaluating Predictive Uncertainty under Dataset Shift. In *NeurIPS*, 2019.

[47] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and Evaluating Contextual Embedding of Source Code. In *ICML*, 2020.

[48] Hong Jin Kang, Tegawendé F Bissyandé, and David Lo. Assessing the Generalizability of code2vec Token Embeddings. In *ASE*, 2019.

[49] André Felipe Zanella, Anderson Faustino da Silva, and Fernando Magno Quintão. YaCoS: a Complete Infrastructure to the Design and Exploration of Code Optimization Sequences. In *SBLP*, 2019.

[50] Grigori Fursin. Collective Tuning Initiative: Automating and Accelerating Development and Optimization of Computing Systems. In *GCC Developers' Summit*, 2009.

[51] Lianmin Zheng, Ruochen Liu, Ameer Haj Ali, Junru Shao, Tianqi Chen, Joseph E Gonzalez, and Ion Stoica. TenSet: A Large-scale Program Performance Dataset for Learned Tensor Compilers. In *NeurIPS*, 2021.

[52] Lars Bjertnes, Jacob O Tørring, and Anne C Elster. LS-CAT: A Large-Scale CUDA AutoTuning Dataset. *arXiv:2103.14409*, 2021.

[53] Alexander Brauckmann, Andrés Goens, and Jeronimo Castrillon. ComPy-Learn: A toolbox for exploring machine learning representations for compilers. In *FDL*, 2020.

[54] Stefano Cereda, Gianluca Palermo, Paolo Cremonesi, and Stefano Doni. A Collaborative Filtering Approach for the Automatic Tuning of Compiler Optimisations. In *LCTES*, 2020.

[55] Ricardo Nobre, Luiz GA Martins, and João MP Cardoso. A graph-based iterative compiler pass selection and phase ordering approach. In *LCTES*, 2016.

[56] Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, Eunjung Park, John Cavazos, and Cristina Silvano. COBAYN: Compiler Autotuning Framework Using Bayesian Networks. *TACO*, 13(2), 2016.

[57] Kyriakos Georgiou, Craig Blackmore, Samuel Xavier-de Souza, and Kerstin Eder. Less is more: Exploiting the standard compiler optimization levels for better performance and energy consumption. In *SCOPES*, 2018.

[58] Rahim Mammadli, Ali Jannesari, and Felix Wolf. Static Neural Compiler Optimization via Deep Reinforcement Learning. In *LLVM-HPC*, 2020.

[59] Alexander Brauckmann, Andrés Goens, and Jeronimo Castrillon. A Reinforcement Learning Environment for Polyhedral Optimizations. *arXiv:2104.13732*, 2021.

[60] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. End-to-end Deep Learning of Optimization Heuristics. In *PACT*, 2017.

[61] Rahim Mammadli, Marija Selakovic, Felix Wolf, and Michael Pradel. Learning to Make Compiler Optimizations More Effective. In *MAPS*, 2021.