# rlscope

*Release 1.0.0*

**James Gleeson**

**Jan 27, 2021**

# CONTENTS:

RL-Scope is a cross-stack profiler for deep reinforcement learning workloads.

For installation instructions, see *Installation*.

For a tutorial on reproducing figures in the RL-Scope paper, see *RL-Scope artifact evaluation*.

For information on running the RL-Scope development docker container, see *Docker development environment*.

CHAPTER

ONE

# INSTALLATION

The following page describes the steps to install rlscope using the standard `pip` python tool so you can use it in your own RL code base. In particular, to install RL-Scope you must enable GPU hardware counter profiling, and install an RL-Scope version that matches the CUDA version used by your DL framework.

**Note:** *Don't* follow these steps if you are trying to reproduce RL-Scope paper artifacts; instead, follow the instructions for running RL-Scope inside a reproducible docker environment: *RL-Scope artifact evaluation*.

## 1.1 1. NVIDIA driver

By default, the `nvidia` kernel module doesn't allow non-root users to access GPU hardware counters. To allow non-root user access, do the following:

1. Paste the following contents into `/etc/modprobe.d/nvidia-profiler.conf`:

```
options nvidia NVreg_RestrictProfilingToAdminUsers=0
```

2. Reboot the machine for the changes to take effect:

```
[host]$ sudo reboot now
```

**Warning:** If you forget to do this, RL-Scope will fail during profiling with an `CUPTI_ERROR_INSUFFICIENT_PRIVILEGES` error when attempting to read GPU hardware counters.

## 1.2 2. Determine the CUDA version used by your DL framework

RL-Scope does not have dependencies on DL frameworks, but it does have dependencies on different CUDA versions.

In order to host multiple CUDA versions, we provide our own wheel file index *instead* of hosting packages on PyPi (NOTE: this is the same approach taken by PyTorch).

DL frameworks like TensorFlow and PyTorch have their own CUDA version dependencies. So, depending on which DL framework version you are using, you must choose to install RL-Scope with a matching CUDA version.

### 1.2.1 TensorFlow

For TensorFlow, the CUDA version it uses is determined by your TensorFlow version. For example TensorFlow v2.4.0 uses CUDA 11.0. You can find a full list here.

### 1.2.2 PyTorch

For PyTorch, multiple CUDA versions are available, but your specific PyTorch installation will only support one CUDA version. You can determine the CUDA version by looking at the version of the installed PyTorch by doing

```
$ pip freeze | grep torch
torch==1.7.1+cu101
```

In this case the installed CUDA version is "101" which corresponds to 10.1.

## 1.3 3. pip installation

Once you've determined your CUDA version, you can use pip to install rlscope. To install RL-Scope version 0.0.1, CUDA 10.1 you can run:

```
$ pip install rlscope==0.0.1+cu101 -f https://uoft-ecosystem.github.io/rlscope/whl
```

More generally, the syntax is:

```
$ pip install rlscope==${RLSCOPE_VERSION}+cu${CUDA_VERSION}
```

Where `RLSCOPE_VERSION` corresponds to a tag on github, and `CUDA_VERSION` corresponds to a CUDA version with "." removed (e.g., 10.1 → 101).

For a full list of available releases and CUDA versions, visit the RL-Scope github releases page.

## 1.4 4. requirements.txt

To add RL-Scope to your requirements.txt file, make sure to add **two** lines to the file:

```
$ cat requirements.txt
-f https://uoft-ecosystem.github.io/rlscope/whl
rlscope==0.0.1+cu101
```

The `-f ...` line ensures that the rlscope package is fetched using our custom wheel index (otherwise, `pip` will fail when it attempts to install from the default PyPi index).

**Warning:** `pip freeze` will *not* remember to add `-f https://uoft-ecosystem.github.io/rlscope/whl`, so avoid generating requirements.txt using its raw output alone.

CHAPTER

TWO

# RL-SCOPE ARTIFACT EVALUATION

This is a tutorial for reproducing figures in the RL-Scope paper. To ease reproducibility, all experiments will run within a Docker development environment.

## 2.1 1. Machine configuration

Generally speaking, RL-Scope works on multiple GPUs models. The only limitation is that you need to use a GPU that supports the newer "CUPTI Profiling API". NVIDIA's documentation states that Volta and later GPU architectures (i.e., devices with compute capability 7.0 and higher) should support this API. If you attempt to use a GPU that is unsupported, the Docker build will fail, since we check for CUPTI Profiling API compatibility using a sample program (see dockerfiles/sh/test_cupti_profiler_api.sh).

The machine we used in the RL-Scope paper had a NVIDIA 2080Ti GPU. We have also reproduced results on an AWS g4dn.xlarge instance which contains a T4 GPU.

### 2.1.1 AWS

As mentioned above, we have reproduced results on an AWS g4dn.xlarge instance which contains a single T4 GPU. Please note that the other AWS instances that have more than one GPU are also fine (e.g., g4dn.12xlarge, p3.8xlarge), and will simply run the experiments faster by using multiple GPUs in parallel.

To make setup as simple as possible, we used NVIDIA's Deep Learning AMI to create an VM instance, which comes preinstalled with Ubuntu 18.04, Docker, and an NVIDIA driver. If you wish to use a different OS image, just make sure you install the NVIDIA driver and Docker.

Regardless of the starting OS image you use, there is still some host setup that is required (which will be discussed in the next section).

## 2.2 2. Running the Docker development environment

In order to run the Docker development environment, you must first perform a one-time configuration of your host system, then use `run_docker.py` to build/run the RL-Scope container. To do this, follow **all** the instructions at *Docker development environment*. Afterwards, you should be running inside the RL-Scope container, which looks like this:


All remaining instructions will run commands inside this container, which we will emphasize with `[container]$`.

## 2.3 3. Building RL-Scope

RL-Scope uses a C++ library to collect CUDA profiling information (`librlscope.so`), and offline analysis of collected traces is performed using a C++ binary (`rls-analyze`)

To build the C++ components, run the following:

```
[container]$ build_rlscope
```

## 2.4 4. Installing experiments

The experiments in RL-Scope consist of taking an existing RL repository and adding RL-Scope annotations to it. In order to clone these repositories and install them using `pip`, run the following:

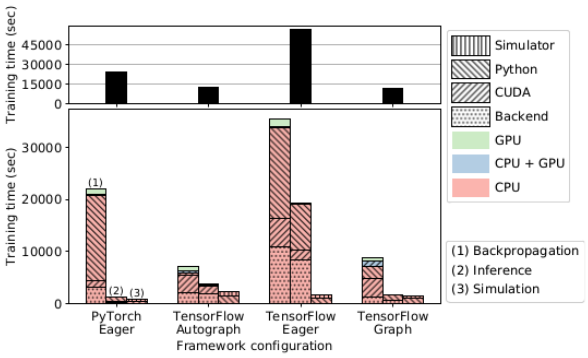
```
[container]$ install_experiments
```

## 2.5 5. Running experiments

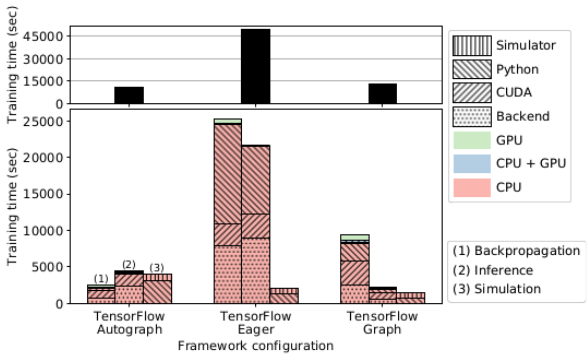
The RL-Scope paper consists of several case studies. Each case study has its own shell script for reproducing figures from that section. The shell script will collect traces from each relevant algorithm/simulator/framework, then generate a figure seen in the paper in a corresponding subfolder `output/artifacts/*` of the RL-Scope repository.

### 2.5.1 RL Framework Comparison

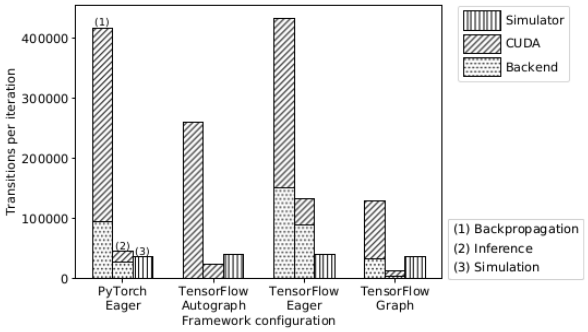
This will reproduce results from the "Case Study: Selecting an RL Framework" section from the RL-Scope paper; In particular, the "RL framework comparison" figures, shown below for reference:



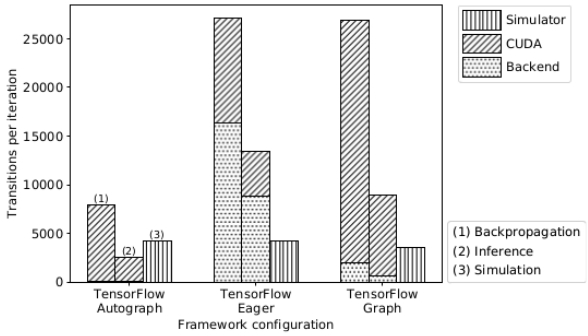

(a) (TD3, Walker2D) - time breakdown




(b) (DDPG, Walker2D) - time breakdown




(c) (TD3, Walker2D) - language transitions




(d) (DDPG, Walker2D) - language transitions

*Figure 4. RL framework comparison:* We used identical RL algorithm (left: TD3, right: DDPG), simulator (Walker2D), and tuned hyperparameters; differences in execution between RL frameworks are strictly due to RL algorithm implementation and ML backend. Differences in *time breakdown* (top) across RL frameworks can be explained by higher number of *language transitions* (bottom) between the Python and the ML backend (*Backend*), and between the ML backend and the accelerator API calls (*CUDA*).

To run the experiment and generate the figures, run:

```
[container]$ experiment_RL_framework_comparison.sh
```

Figures will be output to `output/artifacts/experiment_RL_framework_comparison/*.pdf`.

### 2.5.2 RL Algorithm Comparison

This will reproduce results from the "Case Study: RL Algorithm and Simulator Survey" section from the RL-Scope paper; In particular, the "Simulator choice" figures, shown below for reference:

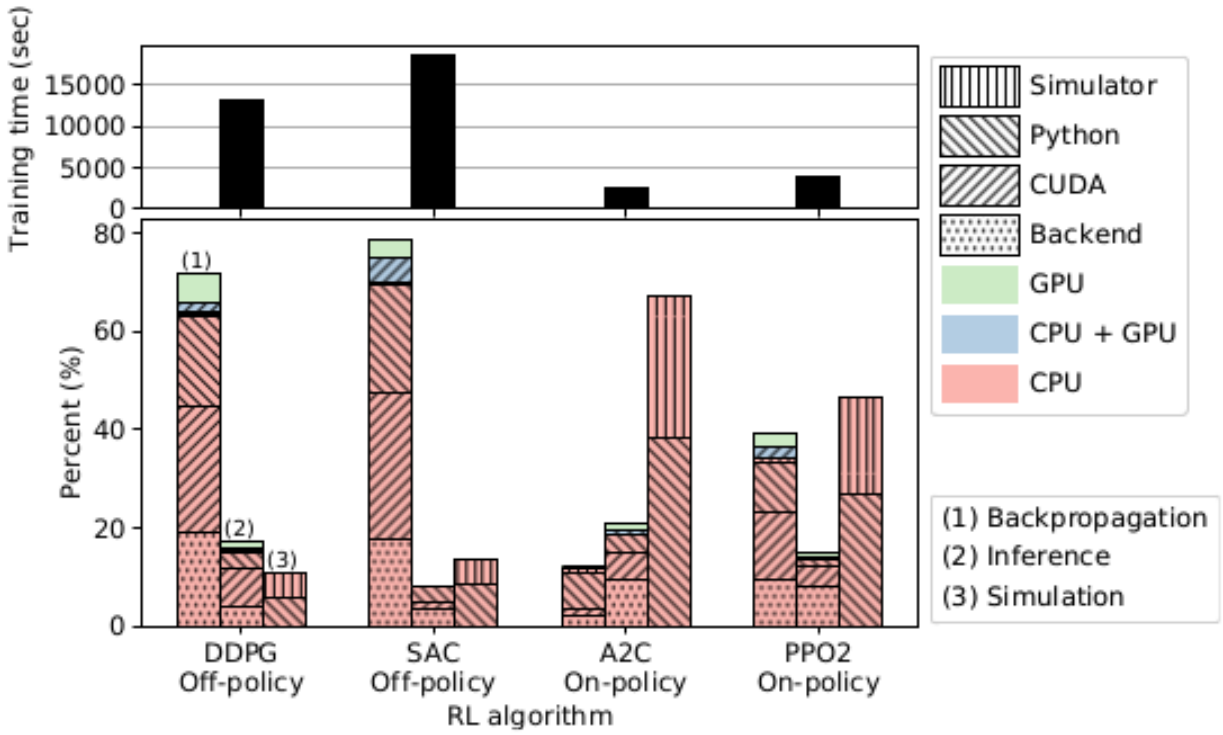



*Figure 5. Algorithm choice:* We used a popular RL environment (Walker2D: robotics task of a walking humanoid) measured how the stages (backpropagation, inference, simulation) of each measured algorithm change with respect to algorithm choice. All tested RL workloads spend about 90% of their runtime purely in the CPU.

To run the experiment and generate the figures, run:

```
[container]$ experiment_algorithm_choice.sh
```

Figures will be output to `output/artifacts/experiment_algorithm_choice/*.pdf`.

### 2.5.3 Simulator Comparison

This will reproduce results from the "Case Study: Simulator Survey" section from the RL-Scope paper; In particular, the "Simulator choice" figures, shown below for reference:

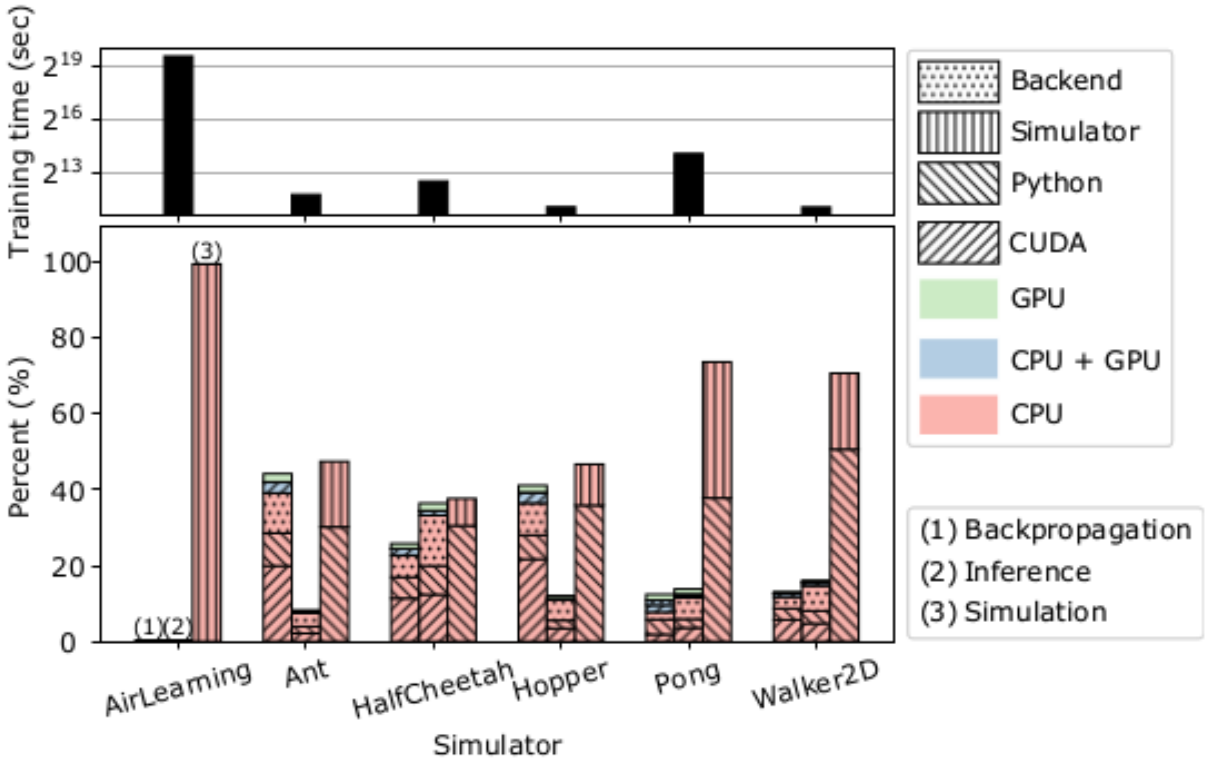



*Figure 7. Simulator choice:* We used a top-performing RL algorithm (PPO) and measured how each stage of the algorithm changes with respect to environment choice. GPU accounts for 5% or less of the runtime across all simulators.

To run the experiment and generate the figures, run:

```
[container]$ experiment_simulator_choice.sh
```

Figures will be output to `output/artifacts/experiment_simulator_choice/*.pdf`.

NOTE: Your reproduced graph will have a slightly different breakdown for Pong than seen above from the RL-Scope paper; in particular the simulation time will be closer to HalfCheetah. This is likely due to a difference in library version for the `atari-py` backend simulator used by Pong.

CHAPTER

THREE

# DOCKER DEVELOPMENT ENVIRONMENT

In order to run the Docker development environment, you must perform a one-time configuration of your host system. In particular:

1. *Install docker-compose*: install docker and `docker-compose`.
2. *NVIDIA driver*: allow non-root users to access GPU hardware counters.
3. *Docker default runtime*: make GPUs available to all containers by default.

After you've configured your host system, you can launch the RL-Scope docker container:

4. *Running the Docker development environment*: build and run the container.

## 3.1 1. Install docker-compose

If your host does not yet have docker installed yet, follow the instructions on DockerHub for Ubuntu.

Make sure you are part of the `docker` UNIX group:

```
[host]$ sudo usermod -aG docker $USER
```

NOTE: if you weren't already part of the `docker` group, you will need to logout/login for changes to take effect.

Next, we need to install `docker-compose`. To install `docker-compose` into `/usr/local/bin/docker-compose`, do the following:

```
[host]$ DOCKER_COMPOSE_INSTALL_VERSION=1.27.4
[host]$ sudo curl -L "https://github.com/docker/compose/releases/download/$
↪{DOCKER_COMPOSE_INSTALL_VERSION}/docker-compose-$(uname -s)-$(uname -m)" -
↪o /usr/local/bin/docker-compose
[host]$ sudo chmod ugo+rx /usr/local/bin/docker-compose
```

## 3.2 2. NVIDIA driver

By default, the `nvidia` kernel module doesn't allow non-root users to access GPU hardware counters. To allow non-root user access, do the following:

1. Paste the following contents into `/etc/modprobe.d/nvidia-profiler.conf`:

   ```
   options nvidia NVreg_RestrictProfilingToAdminUsers=0
   ```

2. Reboot the machine for the changes to take effect:

```
[host]$ sudo reboot now
```

**Warning:** If you forget to do this, RL-Scope will fail during profiling with an `CUPTI_ERROR_INSUFFICIENT_PRIVILEGES` error when attempting to read GPU hardware counters.

## 3.3 3. Docker default runtime

By default, GPUs are inaccessible during image builds and within containers launched by `docker-compose`. To fix this, we can make `--runtime=nvidia` the default for all containers on the host. To do this, do the following:

1. Stop docker and any running containers:

```
[host]$ sudo service docker stop
```

2. Paste the following contents into `/etc/docker/daemon.json`:

```
{
  "default-runtime": "nvidia",
  "runtimes": {
    "nvidia": {
      "path": "/usr/bin/nvidia-container-runtime",
      "runtimeArgs": []
    }
  }
}
```

3. Restart docker:

```
[host]$ sudo service docker start
```

## 3.4 4. Running the Docker development environment

The `run_docker.py` python script is used for building and running the docker development environment. In order to run this script on the host, you need to install some minimal "deployment" `pip` dependencies (`requirements.deploy.txt`).

First, on the **host** run the following (replacing `[rlscope-root]` with the directory of your RL-Scope repository):

```
# Install python3/virtualenv on host
[host]$ sudo apt install python3-pip python3-virtualenv

# Create python3 virtualenv on host
[host]$ cd [rlscope-root]
[host]$ python3 -m virtualenv -p /usr/bin/python3 ./venv
[host]$ source ./venv/bin/activate
[host (venv)]$ pip install -r requirements.deploy.txt

# Build and run RL-Scope the docker development environment
[host (venv)]$ cd [rlscope-root]
[host (venv)]$ python run_docker.py
```

After the container is built, it will run and you should be greeted with the welcome banner:


If you wish to restart the container in the future, you can do:

```
[host]$ cd [rlscope-root]
[host]$ source ./venv/bin/activate
[host (venv)]$ python run_docker.py
```

CHAPTER

FOUR

# UNIT TESTS

## 4.1 Running unit tests

RL-Scope has both python and C++ unit tests, which can be run either separately or all together.

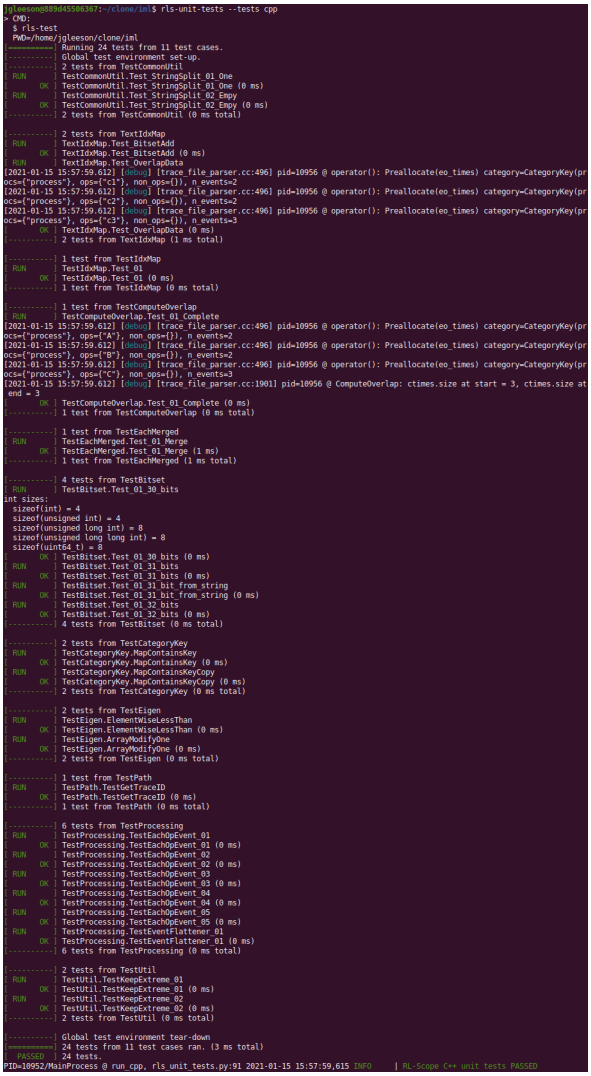
To run all unit tests (i.e., both python and C++):

```
[container]$ rls-unit-tests
```

To run only C++ unit tests:

```
[container]$ rls-unit-tests --tests cpp
```

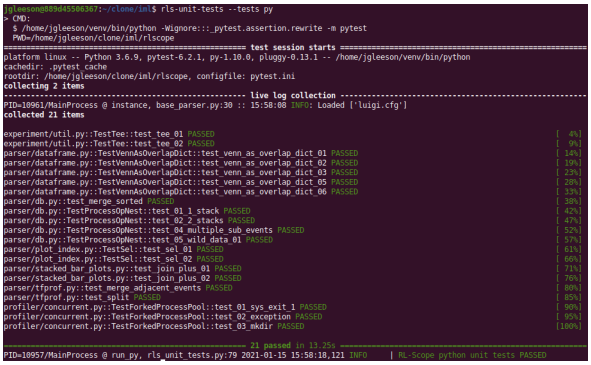
Output should look like:




To run only python unit tests:

```
[container]$ rls-unit-tests --tests py
```

Output should look like:



## 4.2 Python unit tests

Python unit tests made are using the pytest testing framework. Unit tests are written in the same module as the function they are testing. To locate unit tests, search for `def test_` in a file.

## 4.3 C++ unit tests

C++ unit tests are made using the gtest testing framework. Unit tests are any/all files rooted under `test` whose filename matches `test_*.[cc|cpp]`. All unit tests are compiled into the `rls-test` binary.

CHAPTER

FIVE

# SOURCE DOCUMENTATION

- genindex
- modindex
- search
- modules

CHAPTER

SIX

# DOCUMENTATION TODOS

# INDEX

## M



## R