Reinforcement Learning Exploration

Shuo Xin [1]

Abstract

This draft provides a summary of my exploration of the RL algorithm. Specifically, a PPO algorithm is implemented to train LLM on a domain specific language (DSL), the EinsteinToolkit code dataset. We trained a 0.5B model with the reward provided by a 1.5B model as a judge grading the generated code, and a BLEU score between the generated code and the reference code. The evaluation is performed on another numerical relativity simulation package, to evaluate whether the model's reasoning ability, and specifically the ability to generate numerical relativity code outside the DSL, is improved. We provide some reflections and preliminary analysis of the results from this exploration.

1. RL Algorithm

Reinforcement learning (RL) could date back to the early development of "Q-Learning"(Watkins & Dayan, 1992). The setup is usually a game environment (e.g. Pacman 1) where you have several *states* s you reside in and *actions* a you can take. The goal is to learn a policy that maximizes the total *reward* over time. In Q-Learning such a policy is represented by a *Q-value* $Q(s, a)$ that estimates the expected reward for taking action a in state s. The Q-value is "learned" using the following update rule, suppose we are in state s, take action a, and get reward r (usually one need to design the reward function), end up in state s' (and the next action is a'):

$$\begin{aligned} Q(s,a) &\leftarrow (1-\eta)Q(s,a) + \eta\,(r + \gamma V(s')) \\ V(s') &= \max_{a' \in \text{Actions}(s')} Q(s', a') \end{aligned} \tag{1}$$

where the hyper parameter η is the learning rate, $V(s')$ is the "future reward" of the next state, and γ is the "discount" factor that controls how much we care about the future rewards V v.s. the immediate reward r. There are also variants where $V(s')$ changes to Q of a random next action instead of the maximum Q of all next actions (e.g. SARSA), which closely resembles the usual Markov Chain Monte Carlo simulations.

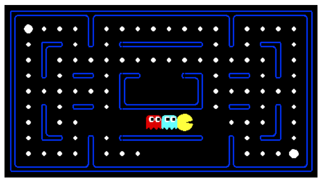


Figure 1. Pacman Gameplay. PC: Chelsea Finn CS 221

The Q-learning defines a policy $\pi : s \to a$ that can be expressed as $\pi(s) = \text{argmax}_a Q(s, a)$. In general if the policy π is parameterized by a set of parameters θ, we arrives at the "policy gradient" (PG) method (Williams, 1992; Wikipedia, 2025a). The policy can also be probabilistic, e.g. following a (learned) distribution $P(a|s;\theta)$. Define the objective J (to maximize) as the total reward (sometimes discounted) expection over time $J(\theta) = E_{\pi_\theta}[\sum_{0<\tau<T} r_\tau]$. Following a "policy gradient theorem", the gradient of the objective J with respect to the parameters θ is given by (Wikipedia, 2025a):

$$\nabla_\theta J(\theta) = E_{\pi_\theta}[\sum_{0<t<T} \nabla_\theta \log P(a_t|s_t;\theta) \sum_{t<\tau<T} r_\tau] \tag{2}$$

In Proximal Policy Optimization (PPO)(Wikipedia, 2025b; Schulman et al., 2017; Ouyang et al., 2022), we are interested in controling the update of our policy π_θ to avoid too large changes, this is estimated by the ratio $\pi_\theta/\pi_{\theta_{\text{old}}}$ (from here on we use π to denote the probabilistic distribution instead of the $s \to a$ mapping). Also, suppose we have a way to estimate the *advantage* A_t (which is supposed to estimate the future reward $\sum_{t<\tau<T} r_\tau$). We can define the "trust region policy optimization" (TRPO) by optimizing the "surrogate" objective

$$E_t\left[\frac{\pi_\theta}{\pi_{\theta_{\text{old}}}} A_t\right] \tag{3}$$

and the PPO would want to avoid too large changes in policy, by clipping the ratio to be within a range $[1-\epsilon, 1+\epsilon]$.

The advantage A_t is also usually re-defined as $R - V_t$ with R the total reward over the entire time and V_t the *value*

[1]Physics Department, Stanford University, Stanford, CA 94305, USA. Correspondence to: Shuo Xin <xinshuo@stanford.edu>.

of the current state. V_t is the model's prediction of the value of the current state and also need to be trained, by a mean-square-error (MSE) loss against the reward.

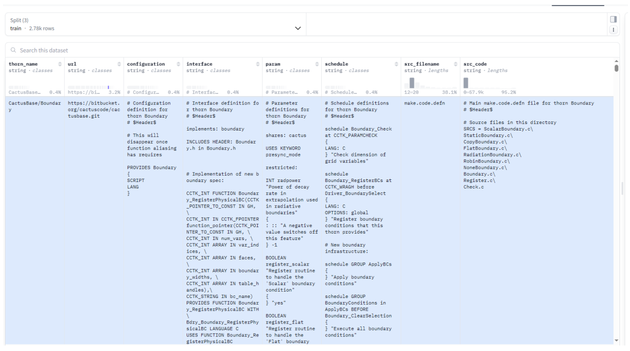
To sum up, in our PPO implementation, we would use MSE loss to train the value network for better estimation of how good the states are, and a policy network (the LLM) is trained by the clipped loss. So in total our loss function is

$$\mathcal{L} = -\min\left(\frac{\pi_\theta}{\pi_{\theta_{\text{old}}}} A_t, \text{clip}(\frac{\pi_\theta}{\pi_{\theta_{\text{old}}}}, 1-\epsilon, 1+\epsilon) A_t\right) + \frac{1}{2}\text{MSE}(V_t, R) \tag{4}$$

To demonstrate the idea I included a minimal code that performs PPO training in a simple binary environment, or a 1D-1cell Pacman. Other codes in the supplemental files implement the slightly larger PPO training of a large language model (LLM), described in the following sections.

2. RL Task

In this work, I explore the application of reinforcement learning to code generation in the specific domain of numerical relativity simulations. Specifically, I focus on the EinsteinToolkit (Gundlach, 1997; EinsteinToolkit, 2025), a well-established codebase that has been maintained for several decades and serves as an excellent dataset for RL training.



Split (3)
train · 2.78k rows

Search this dataset

thorn_name string · classes	url string · classes	configuration string · classes	interface string · classes	param string · classes	schedule string · classes	src_filename string · lengths	src_code string · lengths
CactusBase… 0.4%	https://bi… 3.2%	# Configur… 0.4%	# Interfac… 0.4%	# Paramete… 0.4%	# Schedule… 0.4%	12⇔20 38.1%	0⇔57.9k 95.2%

CactusBase/Boundary	https://bitbucket.org/cactuscode/cactusbase.git	# Configuration definition for thorn Boundary
		# $Header$
		
		# This will disappear once function aliasing has requires
		
		PROVIDES Boundary
		{
		SCRIPT
		LANG
		}
			# Interface definition for thorn Boundary
			# $Header$
			
			implements: boundary
			
			INCLUDES HEADER: Boundary.h in Boundary.h
			
			# Implementation of new boundary spec:
			
			CCTK_INT FUNCTION Boundary_RegisterPhysicalBC(CCTK_POINTER_TO_CONST IN GH, \\
			CCTK_INT IN CCTK_FPOINTER function_pointer(CCTK_POINTER_TO_CONST IN GH, \\
			CCTK_INT IN num_vars, \\
			CCTK_INT ARRAY IN var_indices, \\
			CCTK_INT ARRAY IN faces, \\
			CCTK_INT ARRAY IN boundary_widths, \\
			CCTK_INT ARRAY IN table_handles),\\
			CCTK_STRING IN bc_name)
			PROVIDES FUNCTION Boundary_RegisterPhysicalBC WITH \\
			Bdry_Boundary_RegisterPhysicalBC LANGUAGE C
			USES FUNCTION Boundary_RegisterPhysicalBC
				# Parameter definitions for thorn Boundary
				# $Header$
				
				shares: cactus
				
				USES KEYWORD presync_mode
				
				restricted:
				
				INT radpower "Power of decay rate in extrapolation used in radiative boundaries"
				{
				: :: "A negative value switches off this feature"
				} -1
				
				BOOLEAN register_scalar "Register routine to handle the 'Scalar' boundary condition"
				{
				} "yes"
				
				BOOLEAN register_flat "Register routine to handle the 'Flat' boundary
					# Schedule definitions for thorn Boundary
					# $Header$
					
					schedule Boundary_Check at CCTK_PARAMCHECK
					{
					LANG: C
					} "Check dimension of grid variables"
					
					schedule Boundary_RegisterBCs at CCTK_WRAGH before Driver_BoundarySelect
					{
					LANG: C
					OPTIONS: global
					} "Register boundary conditions that this thorn provides"
					
					# New boundary infrastructure:
					
					schedule GROUP ApplyBCs
					{
					} "Apply boundary conditions"
					
					schedule GROUP BoundaryConditions in ApplyBCs BEFORE Boundary_ClearSelection
					{
					} "Execute all boundary conditions"
						make.code.defn	# Main make.code.defn file for thorn Boundary
							# $Header$
							
							# Source files in this directory
							SRCS = ScalarBoundary.c\\
							StaticBoundary.c\\
							CopyBoundary.c\\
							FlatBoundary.c\\
							RadiationBoundary.c\\
							RobinBoundary.c\\
							NoneBoundary.c\\
							Boundary.c\\
							Register.c\\
							Check.c

Figure 2. A peek into the EinsteinToolkit code dataset.

The task is defined as follows: given a context that describes required functionality along with relevant background information, interface specifications, and parameters (namely everything except the `src_code` column in Figure 2), generate source code (typically in C, C++, Fortran, or Makefile) that implements the functionality for the Einstein Toolkit.

The reward for such a task can be based on:

- **Semantic Analysis**:
  - BLEU (Papineni et al., 2002) similarity score between the generated implementation and the existing reference implementation
  - Evaluation by an LLM judge who grades the generated implementation against the existing one
  - Human feedback
- **Compilation**: A correct implementation should at least pass compilation
- **Execution**: After compilation, the executable should run without errors
- **Unit Tests**: A valid execution should be followed by correct simulation results. Comparison against existing data in unit tests would give a score for subtle details of the simulation output.

Due to the computational constraints of the Einstein Toolkit: compilation of the entire package takes several hours on an 8-core machine, and even individual modules require several minutes to compile, this exploration focuses primarily on the BLEU similarity score and the LLM judge grading as rewards. In other words, the reward to a generated code would be entirely semantic

$$\text{reward} = w_1 \cdot \text{BLEU\_score} + w_2 \cdot \text{Judge\_score} \tag{5}$$

The compilation, execution, and unit test are used in Section 4 for evaluation. RL training using these computational expensive tasks can be explored for future work.

3. Sampling

The T4 machine has 16GB GPU memory. From observations in later experiments, the generated logits would already occupy 10 GB of memory, while I set a small batch size of 3 and also aggressively shortened the context provided to the model for code generation (by cutting off certain columns as shown in Figure 2). By observation, further reducing the context provided would contain too little information (all relevant codes, Cactus Language description, etc. are cut off). Given this observation I train a 0.5B model (Qwen2.5-Coder-0.5B-Instruct), with a 1.5B model (Qwen2.5-Coder-1.5B-Instruct) as a judge. Some explorations of running inference on the EinsteinToolkit code dataset, as well as BLEU score and LLM judge grading, are provided in Appendix A.

4. Evaluation metrics

Suppose our training is successful and the model works well on the specific EinsteinToolkit code dataset (though in fact the training was not successful, as shown in next section

Figure 6). We would like to see whether RL on that task can help the model's general reasoning ability.

There are several other sets of numerical relativity simulation packages (Most & Philippov, 2022), some based on a more general AMREX framework (Zhang et al., 2019), and they do not necessarily use the same language as the EinsteinToolkit. However, we would expect they are all described by a same underlying machenism. Therefore, we may expect that, a successful and clever way of RL training on the EinsteinToolkit code dataset would also help the model to generate code for other numerical relativity simulation packages, clever in the sense that the model is not simply mimicking the dataset, or even overfitting to a domain specific language.

Here since we are doing evaluation rather than training, we can relax on the computational constraints, and actually evaluate the nontrivial **compilation**, **execution**, and **unit test** of the generated code. We use the GReX code (Most & Philippov, 2022) as an evaluation testbed.

For the GReX evaluation, we still define the task as code generation under some context. Here since the code data is much less than the EinsteinToolkit code dataset, we perform code completion instead of entire code generation. Namely, for a 300+ lines of file, we mask some lines, and ask the model to complete them. Such as:

```
... 374 lines of code ...
vars[0] = vars[0] * myfactor;
vars[1] = vars[1] * myfactor; // masked
 for completion
vars[2] = vars[2] * myfactor; // masked
 for completion
vars[3] = vars[3] * myfactor;
... 15 lines of code ...
```

After masking out the lines, the prompt defined in Appendix B is used to complete the code.

4.1. Compilation

The compilation is performed by the following command:

```
make -j128 > log.txt 2>&1
```

and a successful compilation is defined as the log file containing the word `SUCCESS` in the last line. Some examples are given in Appendix B. This is supposed to be the most strong signal in an early stage of learning. We assign `compilation_score` of 1.0 to a successful compilation, 0.0 to an unsuccessful one and fail any subsequent evaluation.

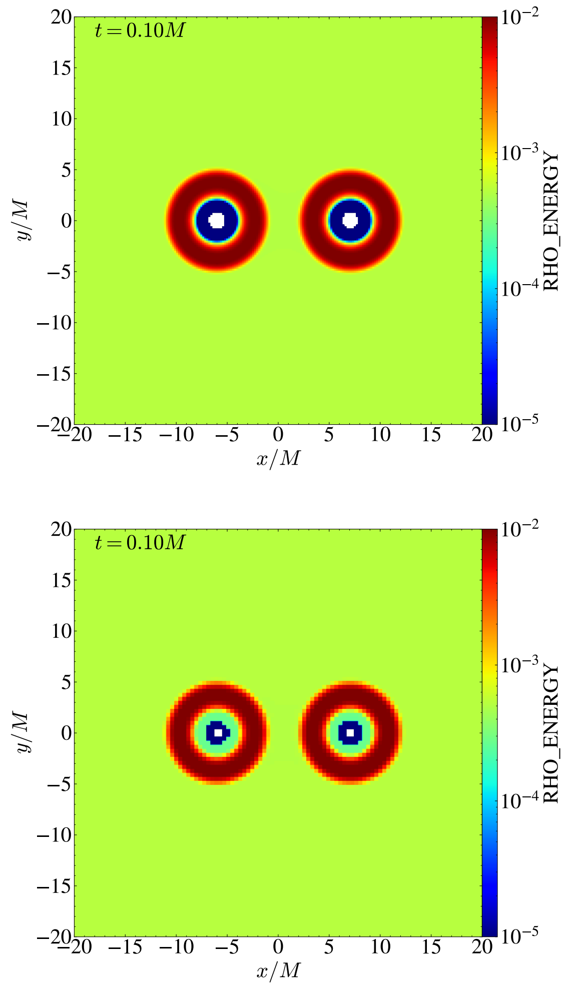


Figure 3. Upper panel: A reference simulation result. Lower panel: A executable, but slightly wrong simulation result, using code completion generated by checkpoint of the model at step 100, this final density profile is graded ~ 0.95 by our metric.

4.2. Execution

A successful execution is defined as running without faults and generated expected number of outputs. In our case, we expect the simulation to generate 5 folders (plt0000[1-5]) containing the snapshot of simulated domains. We assign `execution_score` of 1.0 the folders are found, 0.0 to an unsuccessful one and fail any subsequent evaluation.

4.3. Unit Test

The simulation we ran is exactly a unit test, in the sense that we know the expected numbers in the data we simulated. Here we first check the file headers (at plt0000[1-5]/Header) match. We assign `header_bleu_score` according to the BLEU score between the simulated header and the reference.

The resulted density profile (a 2D array) should look like upper panel of Figure 3. When the simulations pass to this stage, we would compare the difference between the simulated density profile and the reference. Define the score as $\max(0, 1 - \frac{|\rho - \rho_{ref}|}{|\rho_{ref}|})$. And the final evaluation score will be a weighted average of the 4 aforementioned scores.

In principle, all the above evaluation scores should be estimated by sampling the model multiple times and take the average, or running pass@k with a larger k. However, compiling and executing the code requires a longer time. Only pass@1 is tested in this exploration.

5. Experiment and Discussion

After finishing the implementation, I ran a baby experiment that runs for 1 hour, consuming approximately 500 examples in the EinsteinToolkit code dataset. This section sums up some preliminary findings for future improvement and corrections to my implementation.

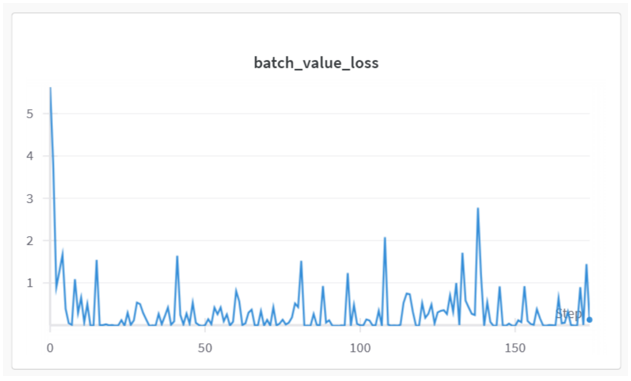


Figure 4. Value loss of the training process. The loss quickly drops in the first few steps, meaning the value net is tune from arbitrary values to a slightly nontrivial guess of the reward. The arbitrary fluctuations after the quick descent mean the model still struggles to learn the value network.

The evolution of the value loss (MSE part of Equation (4)) is shown in Fig. 4. Since the value network is not pretrained, the initial loss is large and quickly descends to a smaller value. This can be viewed as a simple check for apparent problems in our implementation: The arbitrarily initialized value network should quickly learn to avoid some apparently bad region in the parameter space. However, after step 10 it simply fluctuate randomly, indicating it's not "learning" to accurately prediction the value of states. Given that, the random fluctuation of policy loss and reward in Figure 7 is also expected since we do not expect the policy to be updated correctly if we do not have a good estimation of state values. Possible fixes to this problem can be further explored by

- Try a better (larger) value network
- Run the experiment for longer
- Crosscheck with another independent implementation
- Relax the context limit of the code data using a larger memory machine
- Relax the policy model (LLM) to a larger one

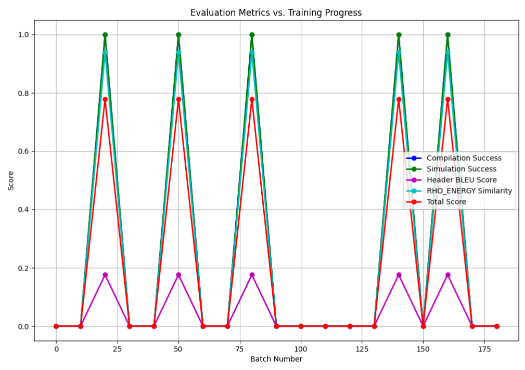


Figure 5. Evaluation score over the training progress. We see random hopping between two values, indicating that randomness overwhelms learning progress for our unaveraged, pass@1 evaluation.

The evolution of the evaluation score as defined in Section 4 is shown in Figure 5, for both the total score and the 4 individual scores. We see the score is simply hopping between two values. Indeed, if we check the output we do see the generated code alternates between two possibilities: a) a line of code that does not compile, b) empty output that pass the compilation but lead to a slightly wrong simulation as show in Figure 3. More evaluation results using slightly different parts for completion are shown in Appendix C.

To conclude, this draft summarizes my preliminary learning of RL training using PPO, and an experiment of doing it on the EinsteinToolkit dataset. I expect to further explore this direction in the near future and see how far we can go with training LLM on scientific computing code bases.

References

EinsteinToolkit. Einstein toolkit. https://einsteintoolkit.org/, 2025.

Gundlach, C. Understanding critical collapse of a scalar field. *Physical Review D*, 55(2):695, 1997.

Most, E. R. and Philippov, A. A. Electromagnetic precursor flares from the late inspiral of neutron star binaries. *Mon. Not. R. Astron. Soc.*, 515(2):2710–2724, 2022. doi: 10.1093/mnras/stac1909.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pp. 311–318, 2002.

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

Watkins, C. J. and Dayan, P. Q-learning. *Machine learning*, 8:279–292, 1992.

Wikipedia. Policy gradient method. https://en.wikipedia.org/wiki/Policy_gradient_method, 2025a.

Wikipedia. Proximal policy optimization. https://en.wikipedia.org/wiki/Proximal_policy_optimization, 2025b.

Williams, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256, 1992.

Zhang, W., Almgren, A., Beckner, V., Bell, J., Blaschke, J., Chan, C., Day, M., Friesen, B., Gott, K., Graves, D., Katz, M. P., Myers, A., Nguyen, T., Nonaka, A., Rosso, M., Williams, S., and Zingale, M. AMReX: a framework for block-structured adaptive mesh refinement. *J. Open Source Software*, 4(37):1370, 2019. doi: 10.21105/joss.01370.

A. Sampling results

In this appendix, I provide some examples of the code generation results using the 0.5B model. The prompt for generating the code is tuned for several rounds by myself to get some code that makes sense. The python code snippet below defines the prompt used. The prompt is generated for each example in the dataset shown in Figure 2 (note the column names).

```
def create_structured_prompt(example, char_limit=10000, ccl_char_limit=1000):
    context = example['context'][:char_limit] if len(example['context']) > char_limit else
     example['context']

    prompt = f"""<|system|>
You are an expert C/C++ developer working on EinsteinToolkit, a codebase for numerical
    relativity simulations.

<|user|>
Create C/C++ code for the file `{example['src_filename']}` in thorn `{example['thorn_name
    ']}`.

## Thorn Information:
- Name: {example['thorn_name']}
- Target file: {example['src_filename']}

## Interface Definition in interface.ccl:
```
{example['interface'][:ccl_char_limit] + "...more..." if len(example['interface']) >
    ccl_char_limit else example['interface']}
```

## Schedule Definition in schedule.ccl:
```
{example['schedule'][:ccl_char_limit] + "...more..." if len(example['schedule']) >
    ccl_char_limit else example['schedule']}
```

## Parameters Definition in param.ccl:
```
{example['param'][:ccl_char_limit] + "...more..." if len(example['param']) >
    ccl_char_limit else example['param']}
```

## Configuration Definition in configuration.ccl:
```
{example['configuration'][:ccl_char_limit] + "...more..." if len(example['configuration'])
     > ccl_char_limit else example['configuration']}
```

## Related Code Context:
```
{context}

Instructions:
Generate only the complete C/C++ source code for `{example['src_filename']}`. Include
    necessary headers, functions, and follow EinsteinToolkit conventions.

<|assistant|>
Here is the code for `{example['src_filename']}`:
```"""
    return prompt
```

Using the prompt defined above, here are some examples of the code generation results, inferenced from the 0.5B model:

```
LLM Judge loaded: Qwen/Qwen2.5-Coder-1.5B-Instruct on cuda
CombinedReward: BLEU weight=0.600, LLM Judge weight=0.400, LLM_enabled=True
```

```
================================================================
Example 2: CactusBase/Boundary - Check.c
================================================================
generating...
decoding...
Generated code (1781 chars):
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "cctk.h"
#include "cctk_Argumen
...(more code truncated)

Actual code (4203 chars):
/*@@
  @file      Check.c
  @date      19 May 2003
  @author    David Rideout
  @desc
  Check that t
...(more code truncated)

BLEU Score: 0.0418

LLM Judge Score: 0.8000

Combined Reward: 0.3451

================================================================
Example 4: CactusBase/Boundary - Boundary.c
================================================================
generating...
decoding...
Generated code (1750 chars):
#include "cctk.h"
#include "cctk_Arguments.h"

#include <assert.h>
#include <stdlib.h>

#include "Bo
...(more code truncated)

Actual code (43688 chars):
/*@@
  @file      Boundary.c
  @date      Sat Oct 26 22:39:40 CEST 2002
  @author    David Rideout

...(more code truncated)

BLEU Score: 0.0305

LLM Judge Score: 0.8000

Combined Reward: 0.3383

================================================================
Example 6: CactusBase/Boundary - RadiationBoundary.c
================================================================
generating...
decoding...
Generated code (1750 chars):
```

```
#include "cctk.h"
#include "cctk_Arguments.h"

#include <assert.h>
#include <stdlib.h>

#include "Bo
...(more code truncated)

Actual code (54915 chars):
/*@@
  @file      RadiationBoundary.c
  @date      Mon Mar 15 15:09:00 1999
  @author    Miguel Alcu
...(more code truncated)

BLEU Score: 0.0244

LLM Judge Score: 0.8000

Combined Reward: 0.3347
```

B. Evaluation details

The prompt for the GReX code completion is structured as follows:

~~~
    prompt = f"""<|system|>
You are an expert C++ developer. You excel at implementing complex physics algorithms and
    writing efficient, correct code.

<|user|>
I need you to complete {missing_lines_end - missing_lines_start} lines of code in the file
     `{file_name}`.


## Code Context
The code before the missing section:
```cpp
{before_missing}
~~~

The code after the missing section:
```cpp
{after_missing}
```

Please generate only the missing {missing_lines_end - missing_lines_start} lines of code
    that should go between these two parts. Follow the existing coding style and ensure
    the implementation is correct and compatible with the AMREX framework.

<|assistant|>
Here's the {missing_lines_end - missing_lines_start} lines of code for the missing section
    :
```cpp"""
```

A typical example of unsuccessful compilation is

```
Loading /pscratch/sd/x/xinshuo/amrex/Tools/GNUMake/comps/gnu.mak...
Loading /pscratch/sd/x/xinshuo/amrex/Tools/GNUMake/comps/nvcc.mak...
Loading /pscratch/sd/x/xinshuo/amrex/Tools/GNUMake/sites/Make.nersc...
Compiling id.cpp ...
nvcc -MMD -MP -ccbin=g++-12 -Xcompiler=' -Werror=return-type -gdwarf-4 -O3 -finline-limit
    =43210 -std=c++17  -fopenmp -pthread  -std=c++17' --std=c++17 -Wno-deprecated-gpu-
```

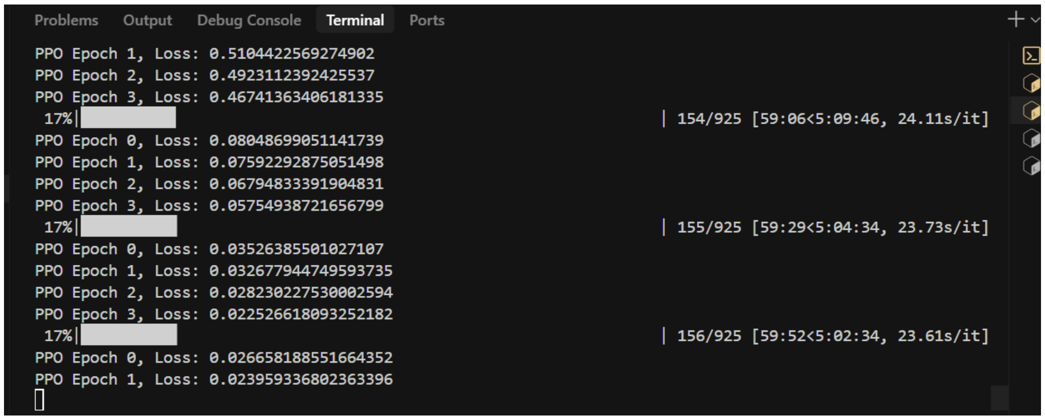
```
    targets -m64 --generate-code arch=compute_80,code=sm_80 -maxrregcount=255 --expt-
    relaxed-constexpr --expt-extended-lambda --forward-unknown-to-host-compiler -Xcudafe
    --diag_suppress=esa_on_defaulted_function_ignored -Xcudafe --diag_suppress=
    implicit_return_from_non_void_function --Werror cross-execution-space-call -lineinfo
    --ptxas-options=-O3 --ptxas-options=-v --use_fast_math  --Werror ext-lambda-captures-
    this --display-error-number --diag-error 20092 -x cu -dc -Xcompiler='-march=core-avx2
    -mtune=core-avx2' -Xcompiler='-march=core-avx2 -mtune=core-avx2'   -
    DAMREX_TINY_PROFILING -DBL_USE_MPI -DAMREX_USE_MPI -DBL_USE_OMP -DAMREX_USE_OMP -
    DAMREX_USE_CUDA -DAMREX_USE_GPU -DBL_COALESCE_FABS -DAMREX_GPU_MAX_THREADS=256 -
    DAMREX_USE_GPU_RDC -DBL_SPACEDIM=2 -DAMREX_SPACEDIM=2 -DBL_FORT_USE_UNDERSCORE -
    DAMREX_FORT_USE_UNDERSCORE -DBL_Linux -DAMREX_Linux -DAMREX_PARTICLES -DNDEBUG -
    Itmp_build_dir/s/2d.gnu.TPROF.MPI.OMP.CUDA.EXE -I. -I/pscratch/sd/x/xinshuo/GReX/
    antelope/ -I/pscratch/sd/x/xinshuo/GReX/lib/tensortemplates/src/ -I/pscratch/sd/x/
    xinshuo/GReX/Source/ -I./ -I/opt/cray/pe/fftw/3.3.10.6/x86_milan/include -I. -I/
    pscratch/sd/x/xinshuo/GReX/Source -I/pscratch/sd/x/xinshuo/GReX/Source/Src_nd -I/
    pscratch/sd/x/xinshuo/GReX/Source/Src_2d -I/pscratch/sd/x/xinshuo/GReX/Source/hydro -I
    /pscratch/sd/x/xinshuo/GReX/Source/Z4 -I/pscratch/sd/x/xinshuo/amrex/Src/Base -I/
    pscratch/sd/x/xinshuo/amrex/Src/Base/Parser -I/pscratch/sd/x/xinshuo/amrex/Src/
    Boundary -I/pscratch/sd/x/xinshuo/amrex/Src/AmrCore -I/pscratch/sd/x/xinshuo/amrex/Src
    /Amr -I/pscratch/sd/x/xinshuo/amrex/Src/Particle -isystem /opt/nvidia/hpc_sdk/
    Linux_x86_64/24.5/cuda/12.4/include -I/opt/cray/pe/mpich/8.1.30/ofi/intel/2022.1/
    include -I/opt/cray/pe/libsci/24.07.0/INTEL/2023.2/x86_64/include -I/opt/cray/pe/fftw
    /3.3.10.8/x86_milan/include -I/opt/cray/pe/dsmml/0.3.0/dsmml//include -I/opt/nvidia/
    hpc_sdk/Linux_x86_64/24.5/cuda/12.4/nvvm/include -I/opt/nvidia/hpc_sdk/Linux_x86_64
    /24.5/cuda/12.4/extras/CUPTI/include -I/opt/nvidia/hpc_sdk/Linux_x86_64/24.5/cuda
    /12.4/extras/Debugger/include  -c id.cpp -o tmp_build_dir/o/2d.gnu.TPROF.MPI.OMP.CUDA.
    EXE/id.o
id.cpp(377): error: identifier "GReX_InitParams" is undefined
  GReX_InitParams();
  ^


1 error detected in the compilation of "id.cpp".
make: *** [/pscratch/sd/x/xinshuo/amrex/Tools/GNUMake/Make.rules:262: tmp_build_dir/o/2d.
    gnu.TPROF.MPI.OMP.CUDA.EXE/id.o] Error 2
```

While a successful compilation usually ends with

```
    /usr/lib64/gcc/x86_64-suse-linux/12/../../../../x86_64-suse-linux/bin/ld: warning:
    libgfortran.so.4, needed by /usr/lib64/gcc/x86_64-suse-linux/12/../../../../lib64/
    liblapack.so, may conflict with libgfortran.so.5
    Built for target ===> x86-milan <===
    SUCCESS
```

C. Experiment details



```
PPO Epoch 1, Loss: 0.5104422569274902
PPO Epoch 2, Loss: 0.4923112392425537
PPO Epoch 3, Loss: 0.46741363406181335
 17%|                                                  | 154/925 [59:06<5:09:46, 24.11s/it]
PPO Epoch 0, Loss: 0.08048699051141739
PPO Epoch 1, Loss: 0.07592292875051498
PPO Epoch 2, Loss: 0.06794833391904831
PPO Epoch 3, Loss: 0.05754938721656799
 17%|                                                  | 155/925 [59:29<5:04:34, 23.73s/it]
PPO Epoch 0, Loss: 0.03526385501027107
PPO Epoch 1, Loss: 0.032677944749593735
PPO Epoch 2, Loss: 0.028230227530002594
PPO Epoch 3, Loss: 0.022526618093252182
 17%|                                                  | 156/925 [59:52<5:02:34, 23.61s/it]
PPO Epoch 0, Loss: 0.026658188551664352
PPO Epoch 1, Loss: 0.023959336802363396
```

Figure 6. PPO Training progress of the 0.5B model, with 1.5B model as a judge providing the reward, on the EinsteinToolkit code dataset.

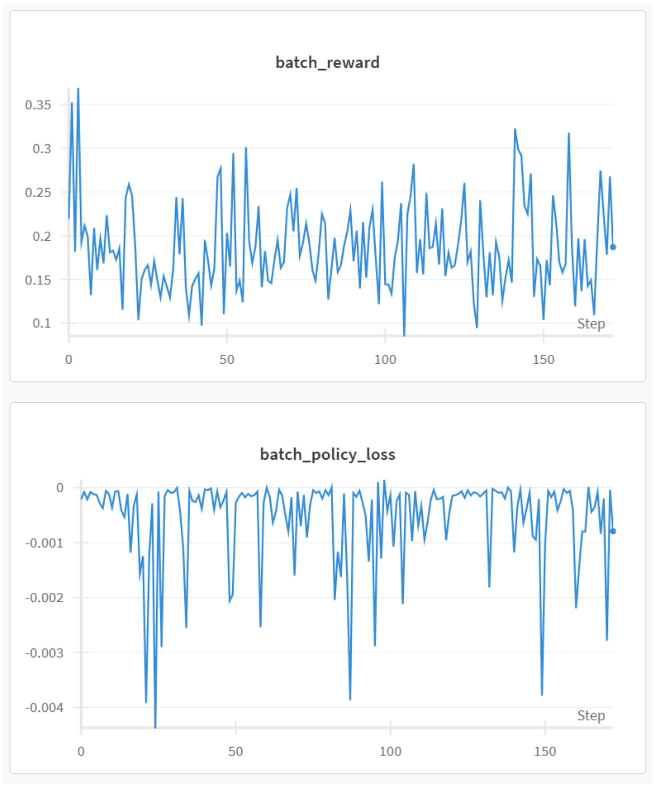


Figure 7. Policy loss and the reward of the training process. Both values fluctuate wildly, and the policy loss is not overall decreasing.

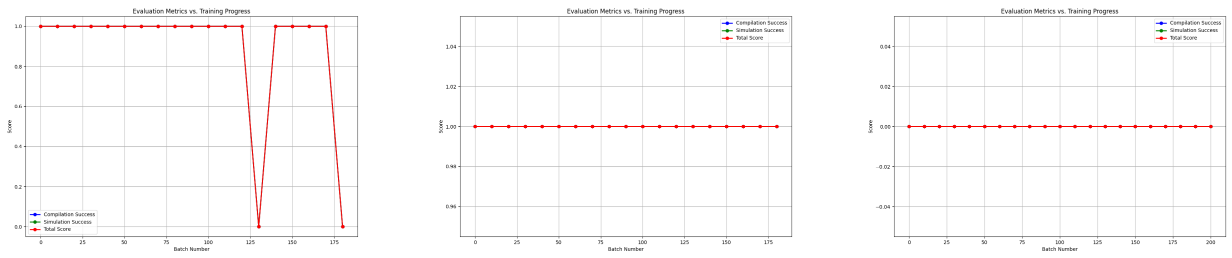


Figure 8. Left: Evaluation where the model is tasked with completing line 68-73, the output hops between complete agreement and compilation failure. Middle: tasked with completing line 133-135, always successful. Right: tasked with completing line 225-230, always failure.