Machine Learning of Polyhedral Compilation

Prakshal Nandi

Master of Science

School of Informatics

University of Edinburgh

2023

Abstract

In most compute-intensive applications, a significant amount of time is spent on the execution of the loops. Compilers like clang offer many optimisation options such as loop tiling, unrolling, vectorization or parallel processing. Most optimizations are still particular to the loop they are trying to transform and require specific knowledge in this domain. If the loops follow a specific structure, it is possible to change the execution order of the loop statements to make the program run faster. But finding a profitable schedule at the compile time is a complex task, as there are lots of constraints and dependencies involved.

Fortunately, using the OpenAI Gym interface, PolyGym [1] has converted this problem into an environment of optimizations for Reinforcement Learning algorithms. The loops are converted into polyhedral representation, upon which different transformations can be applied. This thesis explores the feasibility of using a Reinforcement Learning agent to find a profitable schedule. First, we generate valid schedules using agent-chosen actions transforming the polyhedral model. Secondly, we are doing a search in the space of the possible schedule to find the optimal solution using a different RL agent. Finally, with the help of some experiments, we show how different RL algorithms perform, starting from basic Q-Learning algorithm to more advanced and popular deep reinforcement learning algorithms.

Research Ethics Approval

This research does not involve any human participants or personal data. All data are collected based on simulation experiments on benchmark suites; hence no ethical concerns are associated with the project.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Prakshal Nandi*)

Acknowledgements

Any acknowledgements go here.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Reinforcement Learning(RL) has recently been seen as a promising approach for compiler optimization [**?**][**?**]. A Reinforcement Learning agent makes the *observations* within the *environment*, and takes *actions* to receive certain *rewards*. The goal is to maximize the rewards over a period of time. On the other side, to achieve optimizations related to data locality, memory management, communication, and automatic parallelization, the space of the possible transformations available to a compiler is vast. Applying Machine Learning(ML) algorithms to find a good strategy in this space is challenging. Polyhedral Compilation is one type of compiler optimization technique applicable to a particular class of nested loops. It can explore geometric dependency properties of kernel statements in different iterations and apply transformations to improve the schedule of the loop execution. In compiler optimization, machine learning is yet to achieve the level of success in other fields such as Computer Vision or Natural Language Processing. Even though it has been an object of research for more than two decades, machine learning guided techniques are still not widely utilized in industrial compilers.

1.2 Objectives

The aim of the project is to analyze how Reinforcement Learning can help improve Polyhedral Compilation. This analysis will be performed with the help of Deep Neural Networks and other hybrid meta-heuristics. The research hypothesis is that Reinforcement Learning algorithms can be generalized to learn Polyhedral Compilation.

It is not always clear how to represent the actual problem in a Reinforcement Learning environment. However, this part has already been covered in Polygym[**?**]; hence it is not going to be the focus of this research. The main objectives for this project are trying to find answers to these questions:

- Can we implement a Deep Reinforcement Learning algorithm that can help select the best actions related to polyhedral transformations?
- Can we design reward functions that improve the training performance to learn profitable schedules?
- Can we explore different RL algorithms and find the most suitable technique by evaluating their performance?

1.3 Structure

This project proposal is structured as follows: Following the discussion regarding the problem statement, research objective, novelty, and significance of the project in Section 1, Section 2 provides an overview of the background and literature review. Section 3 focuses on project methodology, limitations, risks, and ethical considerations. Section 4 explains how the evaluation of the developed methods will be carried out. Section 5 discusses expected outcomes, and Section 6 outlines the research timeline, milestones, and deliverables.

Chapter 2

Background

2.1 Polyhedral Compilation Model

In Figure 2.1a, a simple two-dimensional loop nest is shown. There are different ways in which this loop can be optimized, including changing the order execution, cache locality, data location, vector instructions, parallel threads and tasks. The execution of this loop next can be executed in the order shown in 2.1c. The individual operation which we execute can be written down as a point in a two-dimensional vector space, Figure 2.1b. In general, there are more than two dimensions. We could generalise this representation as a polyhedron to more concisely represent this set of points. This polyhedron can be described as a set of linear constraints we must adhere to while applying the transformations, 2.1d. Thus, an n-dimensional mathematical model like polytope can be used to represent the iteration space of a nested loop [3]. The iteration space of the loop nest can be described as in the Equation 2.1. The time when these iterations are performed, i.e. the schedule, can be described as in Equation 2.2. If we want to change the behaviour of how the program runs, we only need to change this execution schedule. Notably, the loop interchange can be implemented here by just changing the schedule from *(i,j)* to *(j,i)*.

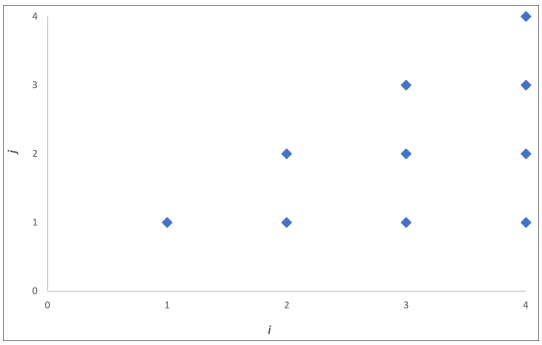
$$I_s = \{S(i,j) \mid 1 <= i <= n \, and \, 1 <= j <= i\} \tag{2.1}$$

$$\theta_s = \{S(i,j) \Rightarrow (i,j)\} \tag{2.2}$$

The optimization through Polyhedral Compilation is achieved in three different stages, moving from code to polyhedron model, applying program transformations

```
for (i = 1; i <= n; i++)
{
    for (j = 1; j <= i; j++)
    {
        S(i, j);
    }
}
```

(a) Polly Loop.



(b) Structure of loop.

```
                                S(4, 4)
                        S(3, 3) S(4, 3)
                S(2, 2) S(3, 2) S(4, 2)
S(1, 2) S(2, 1) S(3, 1) S(4, 1)
```

(c) Schedule of Loop Execution

```
i >= 1,
j >= 1,
j <= i,
i <= n
```

(d) Loop Inequalities

Figure 2.1: Polyhedral Representation of Loop Execution

within the model, and finally regenerating the code from the model. This model gives the framework to transform the loops, where the array references and loop bounds are affine functions of loop iterators and parameters. The polyhedral model is broadly applicable due to the high presence of affine loops in compute-bound applications[**?**]. In the kernel example shown in Figure 1, geometric interpretations such as two-dimensional polygons can be used to visualize dependencies between statements. Based on these dependencies, conclusions can be made regarding the preferred execution time of each statement.

2.2 Deep Reinforcement Learning

The problem of reinforcement learning can become challenging if the number of states possible in the environment is too vast. Deep Learning addresses this challenge by approximating the Q-values of the state and action pairs, using model parameters called weights. The framework of deep learning is made of the neural network, consisting of an input layer, an output layer and one or more intermediate layers. The neurons in the layer have weights associated with them, which we are trying to optimize. The output generated by the network is compared with the target, and loss will be calculated. Different loss functions are available for comparisons, such as mean square error, cross-entropy loss or maximum likelihood loss. They all represent how far is the output of the model with respect to the desired result. Once we know the loss value, the gradients can be calculated considering the effect of each parameter on the output. Using an optimizer, such as SGD, Adam, or RMSprop, we can change the weight values to reduce the loss using the calculated gradients. We are moving towards the desired result by doing this in an iterative manner. Performing all these steps manually could be useful in understanding how deep learning works, but it could be extremely difficult in practical application. Luckily, we have libraries such as PyTorch available, which can calculate the gradients and perform backpropagation (may be more) to develop new network parameters.

2.3 PyTorch

PyTorch implements a dynamic graph approach for gradient calculation, where the library keeps track of the operations performed on the data, and unrolls this operation sequence when we need to calculate the associated gradients. The basic building block of DL is the *tensor*, representing a multi-dimensional array, which is also implemented by this toolkit. PyTorch has methods to create tensors, convert numpy arrays to tensors, and generate tensors with specific data, such as *torch.ones()*. It can also apply different operations on the tensors to concatenate, resize and transpose them. The library also allows for offloading the tensor calculations to GPU by specific the device property associated with it. A tensor may or may not have gradient construction available to it, which can be configured and retrieved using properties such as *requires_grad* and *grad*. By using *backward* operation, we can ask PyTorch to calculate the gradients of all the variables in the network. By leveraging *nn.Module* of PyTorch, we can define the

network architecture, design dropouts, reinitiate the gradients and apply transformations such as *Softmax* (may be more).

2.4 PolyGym

PolyGym provides an environment of Polyhedral optimizations for Reinforcement Learning, adhering to OpenAI Gym interface [**?**], which is a toolkit for solving RL problems. Suppose the actions are chosen to directly represent the transformations, such as loop fusion or tiling. In that case, the model may learn the profitable schedules only for a particular set of loops or domains. Instead, PolyGym takes a more granular approach[1]. It uses the existing formulation of Farkas' lemma from discrete geometry to construct the space of valid schedules. In the next step, a Markov decision process(MDP) can explore this space to find a profitable schedule, independent of the loop being optimized. Hence, this makes it a two-stage technique, the construction of schedule space and its exploration, using two different Markov Decision Processes with different states and actions. This framework, however, still lacks a training agent, although just by using this simple heuristic, PolyGym is able to find transformations that lead to a speed up of 3.39x over LLVM Or and 1.34x over best isl transformation [1]. As shown in Figure **??**, it should be noted that PolyGym does not use any agent, but an agent can use the environment defined by PolyGym for learning.

2.5 Literature Review

Previous surveys have explored the relationship between Machine Learning and Compiler Optimizations [**?**][**?**]. Several studies have also begun to examine how RL can be applied to the field of compiler optimizations. In 2014, Emani *et al.* demonstrated a new approach for dynamically mapping parallel programs to varying system resources [**?**]. They used Markov Decision Process for online thread scheduling and an offline model trained based on the system environment and the structure of the code. Similar to PolyGym, CompilerGym extends the Gym interface to provide a reinforcement learning environment to researchers for compiler optimizations [**?**]. A recent study by Mammadli *et al.* presented an excellent example of using Deep Reinforcement Learning for finding effective sequences of phase-ordering, which is a long-standing problem for compiler generators [**?**]. However, finding profitable loop transformations is a much

[1] https://github.com/PolyGym/polygym

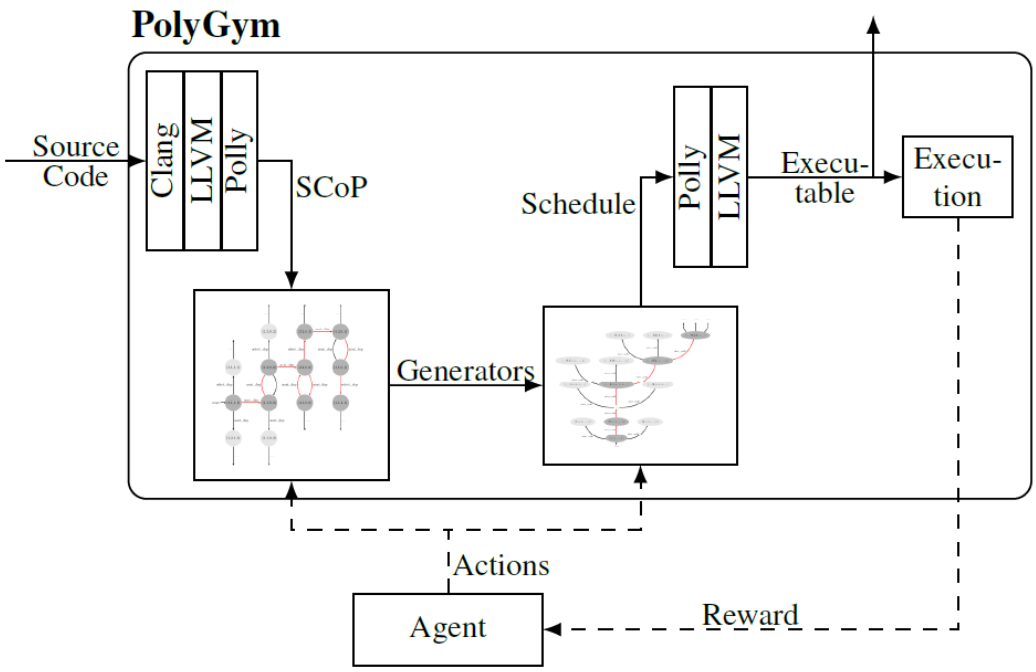


Figure 2.2: Polygym workflow originally demonstrated in [1]

more fine-drawn problem than finding the optimum number of threads or distributing the work on different cores. In an investigation related to current Single Instruction, Multiple Data (SIMD) architectures, Neurovectorizer [**?**] addresses the challenge to find a good vectorization strategy with the help of Deep Reinforcement Learning. MLGO [**?**] attempted to leverage Reinforcement Learning and Evolutionary Algorithms for making decisions about function in-lining for improving code size. However, the evaluation for the code size could be less challenging than the evaluation of the heuristics focusing on the program's execution speed due to the uncertainty involved in the time taken during each execution.

All these methods, however, did not consider polyhedral optimizations. In 1993, Feautrier presented the theoretical basis of the polyhedral formulation for single and multidimensional schedules [**?**][**?**]. After more than twenty years, Pouchet *et al.* proposed an iterative optimization method for systematic exploration of the schedule search space [**?**][**?**]. Building on top of these methods, Ganser *et al.* demonstrated improvements in optimization with the use of genetic algorithms in 2017 [**?**]. One year later, they showed a reduction in benchmarking efforts by using surrogate performance models, where they trained the models based on the program transformation evaluated

in the previous iterative optimizations, at the cost of 15% degradation in speed-up [**?**]. This set of methods followed the iterative approach with benchmarking, making them time-consuming. In contrast, methods such as isl[**?**] and Pluto[**?**] generated the models directly to find a good compilation strategy in less time. However, this approach has certain limitations. The optimization performed is still dependent on the type of loop being analyzed, and its performance can not beat the time taking iterative methods. Hence, the proposed research with Reinforcement Learning fits very well between these two approaches.

Chapter 3

Methodology

3.1 Clang/LLVM/Polly

LLVM is a set of optimiser libraries built around an intermediate representation of the code, known as LLVM-IR. Like many other analyzing and optimizing tools, it uses the Clang frontend, a *C* language family compiler, as a library. Polly applies polyhedral transformations on top of this intermediate representation. It first identifies relevant regions of the code suitable for transformations, called Static Control Parts(*SCoP*). It then transforms these regions into a polyhedral representation based on integer sets and maps. On this representation, it applies optimisations, for example, related to data locality and parallelism, finally converting it back into the optimised version of the executable code. As Polly analyses the low-level IR instead of the programming language itself, it is not dependent on any language or platform.

A *SCoP*, generally consisting of loops and conditions, contains part of the program for which control flow and memory access can be determined at the compile time. For a loop to be identified as *SCoP*, the scalar expression for the iteration count should be able to be represented as an affine function. Similarly, the memory accesses for loading and storing should also be possible to be translated into an affine expression. Polly allows the export of this polyhedral representation into a *JSON* based format called *jSCoP*. This allows us to apply external optimization, which can be imported back using the same *jSCoP* format.

Figure 3.3a showcases an example of a *C* loop doing matrix multiplication. Using *clang*, we can identify the *SCoP* parts of the code and convert them into LLVM representation as displayed in Figure 3.3b. The representation consists of statements as basic building blocks. Apart from its name, a *statement* consists of *Domain*, *Schedule*

and *Access*. The *domain* represents a set of different loop iterations executing the statement, as a named integer set. The *schedule* is an integer map which assigns a point to each iteration in a multi-dimensional space following lexicographical order. Changing the schedule will change the order in which the kernel is executed. *Access* is a pair of kind and relation. The kind can be either *read*, *write* or may *write*. The access relation maps the domain with the memory space, which generally can be represented by an affine function. Figure 3.3c explains the dependencies information extracted from the above representation, which can exist between two statements or two iterations of the same statement. When the transformations have been applied, AST can be generated from the LLVM representation as shown in Figure 3.3d and the program can be executed.

3.2 Q-learning

(improve this) The value of a state can be defined as the expectation of the maximum reward which can be gained from that state, considering all the actions that can be taken in that state. To select the optimal actions, we will need to determine the values of each action and select the one with the best possible result. In a way, the value of a state corresponds to the value of the action we can take from this state, giving us the maximum possible outcome.

The state-action value can be represented as mentioned in Equation 3.1 (change this),

$$Q(s,a) = r(s,a) + \gamma max_{a' \in A} Q(s',a') \tag{3.1}$$

A Q-Network can be trained by minimising the loss function at each iteration as shown in Equation 3.2 (change this),

$$L_i(\theta_i) = E_{s,a\sim\rho(\cdot)}[(y_i - Q(s,a;\theta_i))^2] \tag{3.2}$$

where $y_i = E_{s'\sim\varepsilon}[r + \gamma \max_{a'} Q(s',a';\theta_{i-1})|s,a]]$ is the target for iteration i. $\rho(s,a)$ is the probablity distribution over states s and actions a. γ is the discount factor, r is the reward, and $Q(s,a;\theta)$ is the function approximator to estimate the action-value function [**?**].

3.3 Linear Function Approximation

Any form of tabular learning like Q-Learning requires that we visit each and every state multiple times and apply all the available actions to estimate the correct Q-Values. However, this is not a practical approach in most real-world scenarios, especially those involving a vast state space. Also, it takes up a lot of memory to save and update the Q-table containing a large number of states and actions. One workaround to this is to approximate the Q-Functions using machine learning based on the visited states. This will allow us to approximate Q-Value for a state-action pair *(s,a)* even if action a was never applied in the state s. It will also help us to avoid maintaining a large Q table. Linear Function Approximation [2] is a good alternative to Deep Q-Learning here, as it is not as data-hungry as Deep Leaning and requires comparatively less computation.

$$Q(s,a) = f_1(s) * w_1^a + f_2(s) * w_2^a + ... + f_n(s) * w_n^a \tag{3.3}$$

$$w_i^a = w_i^a + \alpha * \delta * f_i(S), \tag{3.4}$$

$$\delta = r + \gamma * max_{a'} Q(s', a') - Q(s,a) \tag{3.5}$$

3.4 Deep Q-Learning

3.5 Agent Representation

For the problems such as this, it is essential that the representation of the state is such that the agent is able to learn. We define two different agents here, one for the construction phase and one for the exploration. This is because the learning of the agent and the representation is completely different in both phases.

3.5.1 Construction Phase

For the construction phase, the action space consists of three actions as defined in Equation 3.6,

$$A_{con} = \{next\_dimension, select\_dependency, next\_dependency\} \tag{3.6}$$

(add more explanation for actions)

The states during this phase represents information of dependecies with respect to the current dimension and it can be described as Equation 3.10

$$S_{con} = (i_{dim}; i_{dep}; n_d; n_a) \tag{3.7}$$

In this representation, i_{dim} represents the current dimension, and i_{dep} represents the current dependency pointer. n_d stores the number of strong dependencies added in current dimensions, and n_a represents the number of total available dependencies across all dimensions which are yet to be added strongly.

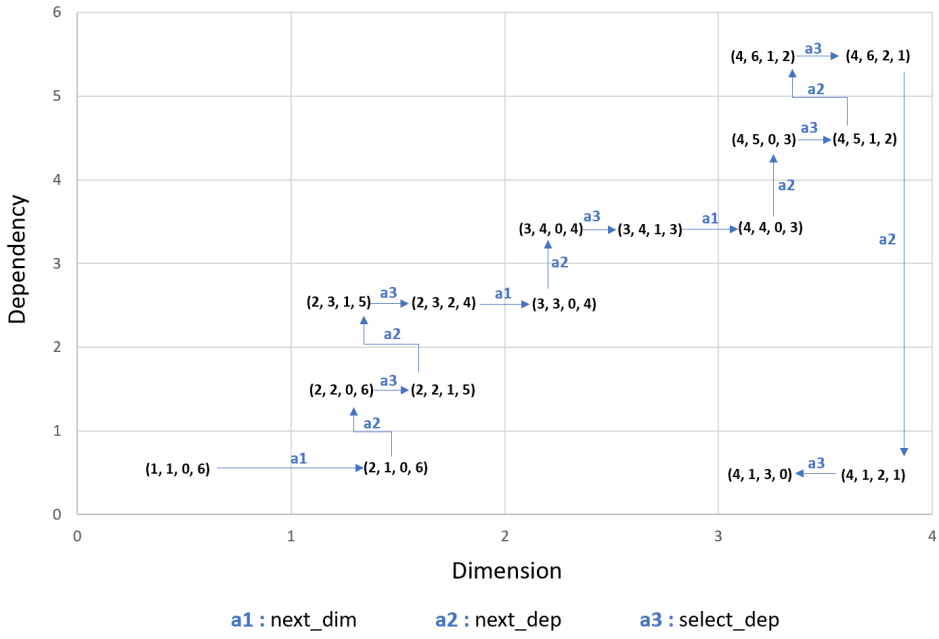


Figure 3.1: Construction Phase State Representation as $(i_{dim}; i_{dep}; n_d; n_a)$.

Using this representation, the agent could learn if it trained and tested on the same kernel. Just within 20 iterations, we achieved an impressive speedup, even better than the best speedup achieved during the training. While this shows that the agent's implementation is correct, more was needed to solve the problem of generalization. While testing this agent on another kernel, the performance was significantly poor.

A critical piece of information we missed was the actual parameters associated with the dependency. A dependency statement can be defined as shown below.

$$\begin{aligned}&[n_i, n_j, n_k] -> Stmt\_for\_body8[i_0, i_1] -> Stmt\_for\_body16[i'_0, i'_1, i_2] : \\ &\quad i'_0 = i_0 \, and \, i'_1 = 0 \, and \, i_2 = i_1 \, and \, n_k > 0 \, and \, 0 <= i_0 < n_i \, and \, 2i_1 <= i_0\end{aligned} \tag{3.8}$$

The dependency defined in 3.8 consists of *domain* variables, *statements* and variable bounds which can be presented as affine functions. To provide all the inequalities a regular structure, we converted all the expressions in the format of less than equal to zero. For example, $0 \leq i_0 < n_i$ can be transformed to two different affine functions $-i_0 \leq 0$ and $i_0 - n_i + 1 \leq 0$. We are allowing dependencies having four variables and four domain parameters. Hence we can have four variables i_0, i_1, i_2, i_3, four prime variables for the second statement i'_0, i'_1, i'_2, i'_3, four domains n_i, n_j, n_k, n_m and one constant. Thus, inequality can be expressed in terms of the vector as shown below.

$$i_0 - n_i + 1 \leq 0 \quad \text{->} \quad [1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ \text{-}1\ 0\ 0\ 0\ 1]$$

If we allow such 8 inequalities in one dependency statement, the number of variables used to represent the statement will be 13 * 2 * 8 = 208.

The last two parameters of the state representation, the number of strong dependencies added in the current dimension and the number of available dependencies to be added, also presented a few challenges. Firstly, they do not contain the information on which dependency was added strongly and which dependency is yet to be added. Secondly, the number of dependencies could be as high as more than 80 in the kernels such as *ludcmp*. Hence, we decided to use one-hot encoding to state the added and available dependencies. For instance, if the maximum number of dependencies considered is 10, the dependencies added in the current dimension can be defined as:

$$[1\ 0\ 0\ 0\ 1\ 1\ 0\ 0\ 0\ 0],$$

showcasing that dependency numbers 1, 5, and 6 are added strongly in the current dimension. Similarly, to define available dependencies, we use vector

$$[0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1],$$

which states that dependencies 2, 9, and 10 are still not added strongly in any dimension and are still available to be added. We have considered the kernels having a maximum number of dependencies as 41 and hence, this adds 82 more variables in our state representation, taking the total variable count, including the current dimension, to 1 + 208 + 82 = 291. We have mixed and matched these representation variables, implementing dropouts, for example, using the number of available dependencies instead of one-hot encoding, to find out which representation gives us the best result. A typical full representation of the state would look as shown below.

Comment - add actual array

array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., -1., 0., 0., 1., 0., 0., 0., 0., 0., 0., 1., 0., 0., -1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., -1., 0., 0., 0., 0., -1., 0., -1., 1., 0., 0., 0., 0., 0., 0., 0., 0., 1., -1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., -1., 1., -1., 0., 0., 0., 0., 0., 0., 0., 0., -1., 1., 0., 0., 0., 0., 0., -1., 0., 0., 1., -1., 2., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])

3.5.2 Exploration Phase

For the exploration phase, the action space consists of three actions as defined in Equation 3.6,

$$A_{con} = \{select\_coeff0, select\_coeff1, select\_coeff2\} \tag{3.9}$$

(add more explanation for actions)

The states during this phase represent information of dependencies with respect to the current dimension, and it can be described as Equation 3.10

$$S_{con} = (i_{dim}; i_{dep}; n_d; n_a) \tag{3.10}$$

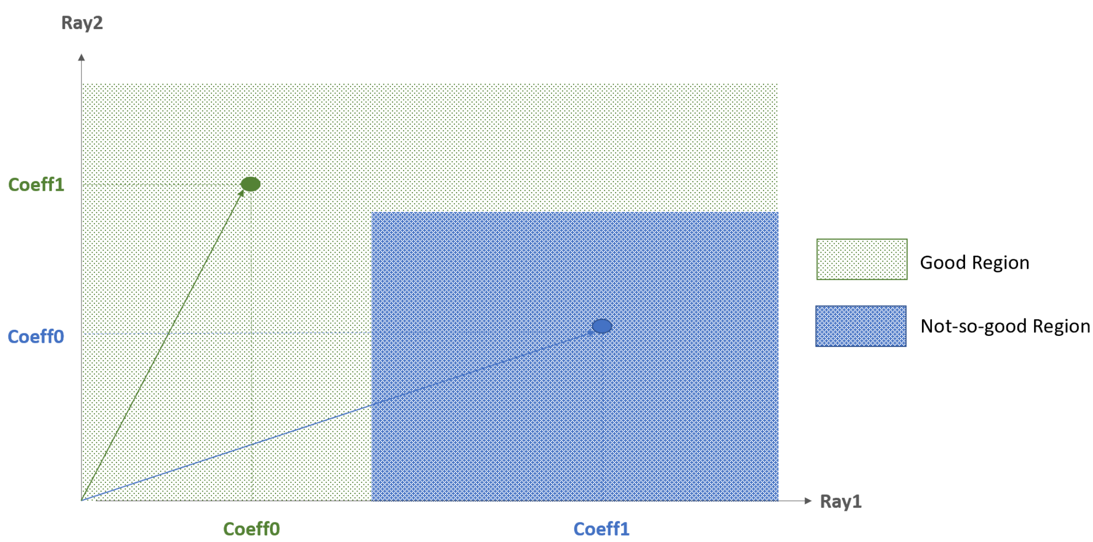


Figure 3.2: Exploration Phase limited to one dimension and two rays for understanding

```c
#include<stdio.h>
int main() {
    int a[10][10], b[10][10], c[10][10], n, i, j, k;
    n =8;

    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            c[i][j] = 0;
            for (k = 0; k < n; k++) {
                c[i][j] += a[i][k] * b[k][j];
            }
        }
    }

    return 0;
}
```

(a) Original Loop in C.

```
Statements {
    Stmt_for_body3
        Domain :=
            { Stmt_for_body3[i0, i1] : 0 <= i0 <= 7
            and 0 <= i1 <= 7 };
        Schedule :=
            { Stmt_for_body3[i0, i1] -> [i0, i1, 0, 0] };
        MustWriteAccess :=  [Reduction Type: NONE] [Scalar: 0]
            { Stmt_for_body3[i0, i1] -> MemRef_c[i0, i1] };
    Stmt_for_body8
        Domain :=
            { Stmt_for_body8[i0, i1, i2] : 0 <= i0 <= 7
            and 0 <= i1 <= 7 and 0 <= i2 <= 7 };
        Schedule :=
            { Stmt_for_body8[i0, i1, i2] -> [i0, i1, 1, i2] };
        ReadAccess :=       [Reduction Type: NONE] [Scalar: 0]
            { Stmt_for_body8[i0, i1, i2] -> MemRef_a[i0, i2] };
        ReadAccess :=       [Reduction Type: NONE] [Scalar: 0]
            { Stmt_for_body8[i0, i1, i2] -> MemRef_b[i2, i1] };
        ReadAccess :=       [Reduction Type: +] [Scalar: 0]
            { Stmt_for_body8[i0, i1, i2] -> MemRef_c[i0, i1] };
        MustWriteAccess :=  [Reduction Type: +] [Scalar: 0]
            { Stmt_for_body8[i0, i1, i2] -> MemRef_c[i0, i1] };
}
```

(b) Polyhedral Representation of the identified SCoP.

```
RAW dependences:
        { Stmt_for_body3[i0, i1] -> Stmt_for_body8[i0, i1, o2] :
        0 <= i0 <= 7 and 0 <= i1 <= 7 and 0 <= o2 <= 7 }
WAR dependences:
        {  }
WAW dependences:
        { Stmt_for_body3[i0, i1] -> Stmt_for_body8[i0, i1, o2] :
        0 <= i0 <= 7 and 0 <= i1 <= 7 and 0 <= o2 <= 7 }
Reduction dependences:
        { Stmt_for_body8[i0, i1, i2] -> Stmt_for_body8[i0, i1, 1 + i2] :
        0 <= i0 <= 7 and 0 <= i1 <= 7 and 0 <= i2 <= 6 }
Transitive closure of reduction dependences:
        { Stmt_for_body8[i0, i1, i2] -> Stmt_for_body8[i0, i1, o2] :
        0 <= i0 <= 7 and 0 <= i1 <= 7 and
        ((i2 >= 0 and i2 < o2 <= 7) or (i2 <= 7 and 0 <= o2 < i2)) }
```

(c) Dependency Analysis.

```
:: isl ast :: main :: %for.cond1.preheader---%for.end26

if (1)

    for (int c0 = 0; c0 <= 7; c0 += 1)
      for (int c1 = 0; c1 <= 7; c1 += 1) {
        Stmt_for_body3(c0, c1);
        for (int c2 = 0; c2 <= 7; c2 += 1)
          Stmt_for_body8(c0, c1, c2);
      }

else
    {  /* original code */ }
```

(d) AST Generation from IR.

Figure 3.3: Generation of SCoP and AST using LLVM Polly

Chapter 4

Evaluation

4.1 Evaluation Setup

4.2 Epsilon-Greedy Approach

During the training, the agent has two choices. Either keep choosing the new actions, which is called Exploration or choose the actions giving the most reward, termed Exploitation. Exploration is good for the long run as the agent gains some new knowledge. While exploitation helps gain immediate reward, even though it might be sub-optimal. The agent can follow only one approach at a time, presenting an exploration-exploitation trade-off. In the implementation, a parameter *epsilon* was kept to introduce the randomness in the training. We choose a random number from 0 to 1, if the number is less than *epsilon*, then the agent explores, or else it exploits, Figure **??**.

Initially, the agent has little information, which will be a good approach to exploring more. Hence, the value of *epsilon* should be higher. As the training progresses, this value should be reduced so that the agent does not spend time on actions which will not give us good rewards. However, the value should be decreased periodically. Otherwise, more exploitation in the initial phase can cause the agent to get stuck in sub-optimal space. We are reducing the *epsilon* value based on the iteration of the training as described in Equation 4.1. Notably, if ε value is zero, the agent always exploits the knowledge it has. On the other side, if the ε value is one, the agent will always choose random actions.

((add pseudo code for epsilon implementation?))

$$\varepsilon = 1.0 - (min\,(1.0,\, t/(0.9 * T))) * 0.95 \tag{4.1}$$

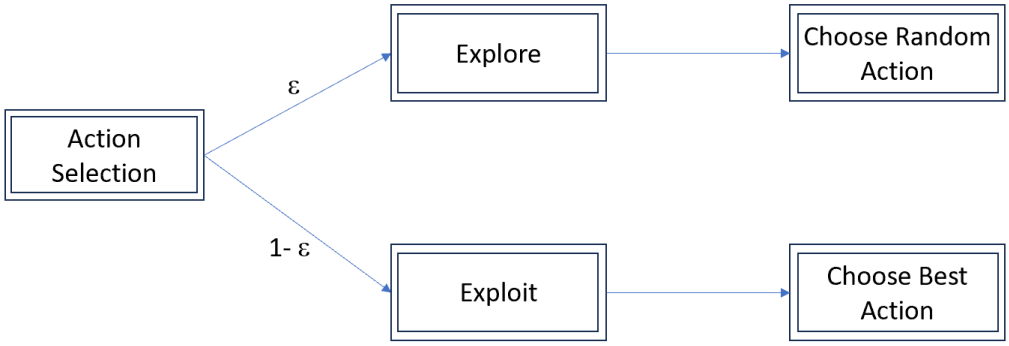


Figure 4.1: Exploration vs Exploitation

,

where t is the current iteration, and T is the total number of iterations for the training.

4.3 Experience Replay Buffer

Instead of learning as soon as we take a step in the environment, storing these transitions in a memory buffer and using them later on for learning will be useful. This is known as learning based on Experience Replay in Reinforcement learning. This approach has many advantages. Firstly, we need a good way to distribute the reward to all the state-actions pairs visited during the scheduled construction and exploration. However, we can do this only after we know the speed-up of the new schedule generated. Hence, if we have the transition history, we update the values for those state-action pairs later on. Secondly, if we are planning to train on each kernel n times, we can pick one kernel train on that kernel for n times, and move to the next kernel. However, this approach might introduce some correlation in the learning. If we store the transition history, we can randomly select the samples to break the correlation.

The base unit of this replay buffer is a transition, which can be described as per Equation 4.2

$$T = (s, a, r, s, d), \tag{4.2}$$

where T is Transition, s is the current state in which the action is taken, a is the

action performed, *r* is the reward received while performing action *a* in states *s*, *s'* is the next state in the transition and *d* is the binary value to indicate if the transition is completed, which is if the schedule is generated.

4.4 Leave-one-out Evaluation

We have conflicting priorities while distributing training and testing sets. Our training set should be large enough for the agent to learn correctly. On the other side, our testing set should be large enough to measure our agent's performance accurately. One approach to finding a middle way is to perform cross-validation, where we train and test our algorithm on the same data multiple times. We first train the agent on the training set and measure the performance on the testing set. On the next iteration, we flip the sets to train the agent on the set which was used for testing and test it on the remaining set. In general, there are more than two partitions, each time, we take a different partition for testing. In the end, we take an average of the performance of each instance. In this case, for n number of kernel examples, we tested the agent on one loop kernel after training it on n-1 kernels. Each time leave one kernel out and train the agent on other kernels, which is similar to the Leave-One-Out evaluation method. This way, our agent is exposed to the most learning, although it has increased the overall computation significantly. This method is shown in Figure **??** with the help of five different kernels, however, there were more than five kernels during actual evaluation.

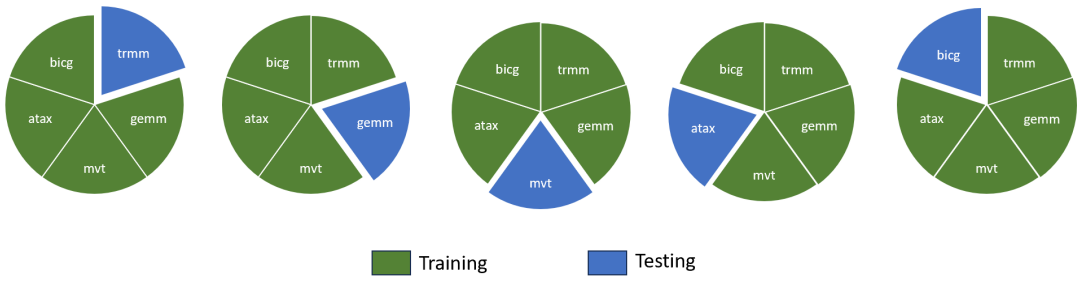


Figure 4.2: Leave-One-Out Evaluation

Chapter 5

Results and Discussions

Chapter 6

Conclusions

6.1 Unresolved Issues

6.2 Future Work

Bibliography

[1] A. Brauckmann, A. Goens, and J. Castrillon. Polygym: Polyhedral optimizations as an environment for reinforcement learning. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 17–29.

[2] Matthieu Geist and Olivier Pietquin. An algorithmic survey of parametric value function approximation. *IEEE Transactions on Neural Networks and Learning Systems*, 24, 06 2013.

[3] Zino Benaissa Tobias Grosser, Johannes Doerfert. Analyzing and optimizing your loops with polly. EuroLLVM 2016, 17. March, Barcelona.

Appendix A

First appendix

A.1 First section