Pearl: Automatic Code Optimization Using Deep Reinforcement Learning

Djamel Rassem Lamouri
dl5133@nyu.edu
New York University Abu Dhabi
Abu Dhabi, UAE

Iheb Nassim Aouadj
ia2280@nyu.edu
New York University Abu Dhabi
Abu Dhabi, UAE

Smail KOURTA
sk10691@nyu.edu
New York University Abu Dhabi
Abu Dhabi, UAE

Riyadh Baghdadi
baghdadi@nyu.edu
New York University Abu Dhabi
Abu Dhabi, UAE

Abstract

Compilers are crucial in optimizing programs and accelerating their execution, particularly for compute-intensive tasks such as training deep learning models and conducting physics simulations. However, optimizing programs automatically using compilers is not trivial. Recent work has attempted to use reinforcement learning (RL) to solve this problem. It has limitations though. Current methods either do not support the optimization of general loop nests or can only be used to optimize loop nests seen during training. In this paper, we propose Pearl, a novel framework that uses deep reinforcement learning to automate compiler code optimization. It uses an RL agent to select the sequence of code optimizations a compiler should apply to make the input code run faster. This agent can optimize general loop nests (i.e., it is not domain-specific) and can generalize to programs unseen during training. To enable the optimization of general loop nests, we propose a novel representation of the action space that allows the RL agent to select on which part of the loop nest a given code optimization should be applied. One of the main challenges that hinder the development of RL agents for optimizing general loop nests is the fact that this task is data-intensive, with each experiment taking weeks. To avoid this problem and enable fast training of the proposed RL agent, we propose two methods: 1) execution time and legality check memoization; and 2) actor-critic pre-training. We implement our approach in Tiramisu, a state-of-the-art polyhedral compiler designed for accelerating compute-intensive programs. Our approach streamlines the optimization process and offers performance improvements compared to existing methods. To the best of our knowledge, Pearl is the first RL-based system to support general programs composed of loop nests manipulating tensors while still being able to generalize to programs unseen during training. It is also the first to support the class of polyhedral optimizations, a class of advanced loop nest optimizations. We evaluate Pearl on a set of benchmarks, and demonstrate competitive performance improvements over state-of-the-art compilers. Notably, Pearl achieves a geometric mean speedup of 2.02× compared to Tiramisu and 3.36× compared to Pluto.

Keywords

Compiler, Code Optimization, Reinforcement Learning, Polyhedral Model

1 Introduction

Writing fast and efficient code is a challenging task that requires significant expertise. This is especially true for compute-intensive fields such as deep learning and scientific computing. Optimizing compute-intensive programs can significantly reduce execution time, often achieving speedups by orders of magnitude. However, manual code optimization is time-consuming, error-prone, and demands expertise, making automated compiler optimizations increasingly crucial.

ICS '25, June 8–11, 2025, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1537-2/2025/06
https://doi.org/10.1145/3721145.3725766

The most significant part of the execution time of compute-intensive programs is usually spent in loops. That is why loop optimization has received considerable attention in the context of code optimization. State-of-the-art compilers such as Tiramisu [9], Halide [47], TVM [57], and Pluto [11] all focus on applying loop transformations to accelerate the execution of programs. Nevertheless, the search space of these transformations is considerably large, hindering the usage of exhaustive search methods due to their impracticality.

Several techniques have been proposed to solve this challenge, including the leverage of integer linear programming (ILP) to find the best code optimizations [11, 12, 55]. Recently, state-of-the-art optimizing compilers have explored tree-search methods guided by a deep learning cost model, serving as a fitness function to evaluate code optimization candidates during search [1, 8, 57]. To shed light on the size of the search space, let's take the Tiramisu Autoscheduler [8] as an example. Its search space has an estimated number of 10^{170} candidates of loop optimizations [8], where each candidate is a sequence of code optimizations along with their parameters. This is why the Tiramisu Autoscheduler imposes restrictions on how the search space is explored. For example, code optimizations are explored in a fixed order, and many optimizations are explored only once, which is known to be sub-optimal. The Tiramisu Autoscheduler uses beam search to explore the space, limiting its ability to perform global code optimization. In general, tree-search methods tend to restrict the search space to enable efficient exploration. For example, they might explore only a small number of code optimizations, explore code optimizations in a fixed order, and limit the number of times certain code optimizations are explored (e.g., explore them once only).

To avoid the limitations of tree-search methods, recent state-of-the-art compilers have attempted to use reinforcement learning. HalideRL [43], for example, uses PPO [48] to select code optimizations for image processing applications. SuperSonic [30], a framework for automating RL architecture design for code optimization, was also demonstrated by building an RL agent for the Halide compiler. However, neither HalideRL nor SuperSonic's Halide agents are built to generalize over programs unseen during training. In both cases, the RL agent is trained to optimize a set of programs, and then it is deployed to optimize the same programs. This design choice allows fast training of the RL agent since it is trained on a single program only, but it prevents its generality. Additionally, HalideRL has the limitation of being semi-automatic. It requires the user (developer) to specify the list of code optimizations and optimization parameters that will likely help optimize the code. The RL agent then discards the optimizations that are not useful from this list and adjusts the optimization parameters.

Other state-of-the-art systems focus on proposing an environment for automatic code optimization using reinforcement learning but do not propose an RL agent that automatically optimizes code. PolyGym [14], for example, introduced an RL environment to search for polyhedral code optimizations (affine code optimizations). It does not propose an RL agent though. It rather uses a random policy to demonstrate the effectiveness of its environment and action space. Such a random policy has the limitation of being slow as it has to randomly sample thousands of actions from the search space, and for each sampled action, it has to compile and run the optimized program to find the most effective ones. Proposing an RL agent that efficiently search for code optimizations in PolyGym was left for future work.

AutoPhase [29] is another example of using RL for code optimization. It proposes a framework that uses deep reinforcement learning to optimize programs for hardware synthesis (High-Level Synthesis). It only considers the problem of phase reordering (i.e., choosing the best order for the compiler passes). Phase ordering is a sub-problem of the larger problem of automatic code optimization. In automatic code optimization, the goal is to identify which optimizations to apply, in which order, on which part of the code each of them should be applied, and with which parameters. Chameleon [2] is another example of a compiler that uses reinforcement learning to find the best compiler transformations to accelerate the execution time of neural networks during deployment. However, it is not general. It is limited to the acceleration of deep learning models and does not cover the optimization of general loop nests, which limits its applicability. Optimizing programs with loops is a much harder problem since general loops can have diverse and complex structures and code patterns, unlike deep learning operators which comprise a limited set of operators with regular shapes and code patterns.

In this paper, we present Pearl, a deep reinforcement learning-based system for polyhedral code optimization. It supports the optimization of general loop nests and generalizes to programs unseen during training. Pearl avoids the limitations of tree-search methods and uses a deep policy network to predict the sequence of code optimizations to apply. We also propose a novel representation of the action space that enables the RL agent to select on which part of the loop nest a given code optimization should be applied. Our technique explores a large action space that includes six loop transformations with their parameters. Unlike existing work, it supports a set of polyhedral code optimizations, it can generalize to unseen programs, and can be applied to any program that can be expressed as a sequence of loop nests and operates on tensors. Examples of types of computations that our RL agent supports include image processing, deep learning, linear algebra, stencil computations, and tensor operators.

During the development of the RL agent, one needs to train the RL agent repeatedly, to test different features and hypotheses and to fine-tune hyper-prameters. With each training taking weeks, and with the need for tens of experiments, the development of the RL becomes challenging. We also propose a set of techniques to mitigate this challenge. The main idea of our proposed technique is to store any value computed when training the RL agent, if it can be reused in a future training. For example, when the RL agent picks a particular action, it has to check whether it is legal, and if so, it applies it by compiling the optimized program and then gets the reward by running the optimized program. Both of these operations are computationally expensive and constitute a significant part of the RL training time. We store the results of these operations (legality and reward of an action) in a dataset and reuse them in future trainings. In subsequent trainings, when the RL picks an action, we first check whether its legality or reward have already been computed in previous trainings. If so, we retrieve them directly, otherwise we compute them and add them to the dataset. We call this method *execution time and legality check memoization* and it helps significantly in accelerating the training. We also propose another technique where we pre-train the actor-critic network of the RL agent with data collected in previous trainings. We call this technique *actor-critic pre-training* and it helps also in improving the learning in the actor-critic neural networks.

Unlike HalideRL and SuperSonic's Halide agent, Pearl generalizes to programs unseen during training. The agent is trained on randomly generated programs and evaluated on a completely different set of benchmarks. It is also fully automatic. It does not require input from the user. In contrast with PolyGym, which only proposes an environment, we propose both an environment and a deep RL agent for that environment. In contrast to AutoPhase, which selects the best order for compiler passes, our goal is to tackle the more general problem of automatic code optimization, which includes selecting optimizations and their parameters, the order of applying them, and on which part of the code. Unlike Chameleon, our system is not limited to accelerating deep neural networks but is general to any program that can be expressed as a sequence of loop nests and operates on tensors.

We implement the proposed approach in the Tiramisu compiler, a state-of-the-art compiler [9], and evaluate it on a set of benchmarks from the fields of linear algebra, image processing, and scientific computing. We show competitive performance improvements compared to state-of-the-art compilers. Notably, our proposed approach achieves an overall geometric mean speedup of 2.02× compared to Tiramisu and 3.36× compared to Pluto. You can find the full implementation of our method.[1]

In this paper, we make the following contributions:

- We introduce Pearl, a deep reinforcement-learning system for polyhedral loop nest optimization.
- We propose a novel representation of the action space that allows the RL agent to optimize loop nests. It allows the RL agent to choose the most appropriate code optimizations for each part of the loop nest.
- To the best of our knowledge, Pearl is the first RL system that supports the optimization of programs composed of general loop nests and can still generalize to programs unseen during training.
- To the best of our knowledge, Pearl is also the first to propose an RL agent that supports the class of polyhedral optimizations.
- We implement and evaluate Pearl and show that it outperforms state-of-the-art.
- We release our dataset and make our code publicly available to the community.

2 Related Work

In this section, we provide an overview of state-of-the-art methods used by compilers for automatic code optimization (auto-scheduling). First, we present compilers that use a search-based method and a learned performance model for automatic code optimization. We then present early attempts to use reinforcement learning for automatic code optimization. Table 1 shows a summarized comparison between these methods. Finally, we present other methods that do not rely on machine learning but rather use Integer Linear Programming (ILP) for automatic code optimization.

Search-based methods with cost models. This method was widely used due to its efficiency in exploring the large space of possible candidates. A heuristic search method

[1] Code available at https://github.com/Modern-Compilers-Lab/GNN_RL_Pretrain

Table 1: Comparison with RL-based Systems.

Features	Pearl	PolyGym	CompilerGym	HalideRL	AutoPhase	Chameleon	X-RLflow
RL Environment	✓	✓	✓	✓	✓	✓	✓
RL Agent	✓	×	×	✓	✓	✓	✓
Fully Automatic	✓	✓	✓	×	✓	✓	✓
Support Affine Transformations	✓	✓	×	×	×	×	×
Supports General Loop Nests	✓	✓	✓	✓	✓	×	×
Optimize Unseen Programs	✓	-	-	×	✓	✓	✓
Auto-scheduling Framework	✓	✓	✓	✓	×	✓	✓

is guided by a cost model to locate the best sequences of code optimizations. Tree-search methods were successfully used in Tiramisu [8, 26, 35, 38, 40], Halide [1], and ProTuner [25]. Other work uses genetic and evolutionary algorithms to explore the search space [16, 17, 58] but they follow the same approach. RL-based methods surpass search-based methods in two ways. First, search-based methods are slower since they explore a large space of code optimizations and evaluate the best candidates within that space. As an example, Halide [1] evaluates millions of candidates in the search space, while Tiramisu [8] evaluates thousands of candidates. In our proposed system, Pearl, the policy network selects the most appropriate sequence of code optimizations directly, without the need to explore a large space. Second, many search-based methods constrain the order of exploring code optimizations to keep the size of the search space smaller, which leads to sub-optimal results.

RL-based methods. Recent attempts explored the use of reinforcement learning to solve the problem of choosing the right sequence of code transformations. In PolyGym [14] and CompilerGym [18], the authors propose only RL environments without implementing RL agents to optimize code, their main contribution is to show that their action space has potentially good optimizations to explore. They leave the implementation of an RL agent as future work.

Other works such as HalideRL [43], AutoPhase [29] and SuperSonic [30] propose RL agents to optimize code. HalideRL is not fully automatic. The user has to provide an initial set of code transformations. The HalideRL agent then discards transformations that are not useful and keeps only those that are useful. It then selects the best parameters for the useful transformations. In addition, HalideRL does not generalize to programs unseen during training. It is trained on a given program with multiple random data input sizes. Then during deployment, it is used to optimize that same program. This is different from our approach. Our RL agent is designed to generalize to programs unseen during training. We first train our RL agent on a large set of random programs. Once it learns how to optimize them, we then deploy it on new unseen programs and use it to optimize them.

SuperSonic [30] is a meta-optimizer that targets the problem of choosing the best RL algorithm and the best representation of states and actions, while AutoPhase [29] targets the problem of phase ordering, i.e., selecting the best order for compiler passes. It does not target the problem of auto-scheduling which we address in this paper. Phase ordering is only a sub-problem of the larger problem of auto-scheduling. In addition, AutoPhase targets HLS (High-level Synthesis) and does not target CPUs which we focus on.

All of the previous approaches, with the exception of PolyGym, are not designed for the class of affine transformations that we target (a.k.a., polyhedral transformations). Affine transformations allow advanced code optimizations for improving data locality and parallelism extraction and are hard to model due to their complexity and diversity. A summarized comparison between our proposed RL and RL-based compiler frameworks is presented in Table 1.

Graph level optimization for deep learning. Compilers in this category target the application of code transformations in the domain of deep learning. With a higher level of abstraction, these compilers consider graph-level transformations on the deep learning computation graphs. They focus on transformations such as data layout transformations [36], operator fusion [16, 59], and auto-batching [37]. Chameleon [2] uses reinforcement learning to find the best compiler transformations that accelerate the execution time of neural networks during deployment. REGAL [42] targets only model parallelism on computational graphs to minimize execution time and memory peak usage. X-RLflow [28] addresses the tensor graph superoptimisation problem using graph neural networks and reinforcement learning to perform neural network dataflow graph rewriting, which substitutes a subgraph one at a time. Our method is different from these in the sense that it does not operate on the high-level abstraction of computation graphs, but rather operates on a lower level. Because of that, our proposed RL agent is more general since it is not limited to deep learning computation graphs, but rather supports any program that can be expressed as a sequence of loop nests and operates on tensors. Types of computations that our RL agent supports include image processing, deep learning, linear algebra, stencil computations, tensor operators, etc.

Polyhedral Optimization with ILP. The polyhedral model [21] is a mathematical model for representing code and code transformations and is used in state-of-the-art compilers to apply complex code transformations and reason about their correctness [3–7, 9–11, 13, 19, 20, 23, 24, 31, 34, 39, 45, 46, 50, 51, 53, 54, 56]. As a method to solve the code optimization problem, Integer Linear Programming (ILP) was used by [11, 12, 55] to explore the search space and find optimal solutions. The limitation of these ILP-based approaches is the lack of a precise cost model for predicting the performance of code optimizations. This limitation comes from the use of ILP which limits the cost function to a simple linear cost function. Because of this, more recent polyhedral compilers, such as Tiramisu, have switched to the use of search-based approaches along with a deep learning cost model for performance prediction.

3 Background

3.1 Reinforcement Learning

Reinforcement learning is a machine learning paradigm where an agent learns from interacting with an environment to maximize a cumulative reward [49]. In this work, we model the problem of automatic code optimization as a Markov Decision Process represented as a tuple $M = \langle S, A, P, R, \gamma \rangle$. In this model, S is the set of all states while A is the set of all actions. P is the transition probability to a state $s^{'}$ given a the state s and action a where $Ps^{'}|s, a = \mathbb{P}S_t = s^{'}|S_{t-1} = s, A_{t-1} = a$, and R is the reward function that indicates the expected reward for a given state-action pair $Rs, a = \mathbb{E}R_t|S_{t-1} = s, A_{t-1} = a$. The discount rate $\gamma \in 0, 1$ determines the weight of future rewards in the agent's decision-making process [49]. As the agent interacts with the environment, it learns a policy $\pi a|s$ that maximizes the cumulative reward. We are interested in policy-based RL algorithms. We train our agent using Proximal Policy Optimization [48].

3.2 Graph Neural Networks

In our work, we consider undirected attributed graphs GV, E, X where V is the set of nodes, E is the set of edges, and $X \in \mathbb{R}^{|V| \times d}$ is an input matrix representing node features. $x_v \in \mathbb{R}^d$ denotes the features of node $v \in V$. Modern GNNs use the message passing paradigm [22] to update the node features h_v^k in an iterative process where $h_v^0 = x_v$. Message Passing Neural Networks (MPNNs) update h_v^k as follows

$$h_v^{k1} = \phi^k h_v^k, \psi^k h_v^k, \{h_u^k | u \in \mathcal{N}_v\},$$

where $\mathcal{N}_v$ is the set of neighbors of node v, ψ^k is a permutation invariant function that aggregates the representation of the neighborhood of the node v into fixed size vector representation, and ϕ^k is the update function that takes the k^{th} representation of node v with the k^{th} representation of the neighborhood to compute h_v^{k1}. For graph-level prediction, we compute h_G^k using another permutation invariant function that aggregates all node representations at iteration k into a single vector.

$$h_G^k = readout\{h_v^k | v \in V\}$$

We use Graph Attention Networks (GATv2) [15], a famous MPNN model based on the attention mechanism.

3.3 Tiramisu

We implement our method in the Tiramisu compiler [9]. Tiramisu allows the user to express algorithms composed of sequences of loop nests and statements that manipulate scalars and tensors. Such algorithms dominate compute-intensive domains, including image processing, linear algebra, stencil computations, and deep learning.

Tiramisu provides a compiler and a DSL (Domain-Specific Language) embedded in C++ to represent code and its transformations. It provides two APIs: (a) an API for developers to write high-level architecture-independent algorithms, and (b) a second set of functions to describe how to optimize the algorithm. This separation enables the compiler to try a variety of code optimizations on the same algorithm.

In general, every Tiramisu program[2] can be divided into two parts. The first part describes the algorithm as a function with inputs and outputs. Each program comprises a sequence of computations (a computation is a loop nest with a statement in its body). The second part of a Tiramisu program is reserved for specifying how the algorithm is optimized using a specific API.

3.4 Polyhedral Access Relations (Access Matrices)

In this section, we introduce the concept of polyhedral access relations, which are used in the polyhedral model to represent array accesses. These access relations are represented using matrices called access matrices (in other words, access matrices are a matrix representation of the access relations). We pass the access matrices as

[2] Also referred to as Tiramisu function.

input to the RL deep learning models to represent array accesses. In the rest of this section, we will first explain the concept of access relations and then show how an access relation is represented using an access matrix.

Access relations are a set of read, write and may-write access relations that capture memory locations on which statements operate. They map statement execution instances to the array elements that are read or written by those instances.

```
for  (i=1; i<=2; ++i)
    for  (j=1; j<=2; ++j)
S:          A[i,j] = B[i,j];
```

In the previous example, the set of read-access relations is

$$R_s = \{Si, j \to Bi, j\}$$

which means that the statement S in iteration i, j reads the array element Bi, j.

The set of write access relations is

$$W_s = \{Si, j \to Ai, j$$

The read access relation R_s can also be represented using a matrix as follows:

$$R_s \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix}$$

where the matrix

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

is the matrix representation of the access relation in this case. We call it, the access matrix. This access matrix is the representation that we pass as input to the RL deep learning models.

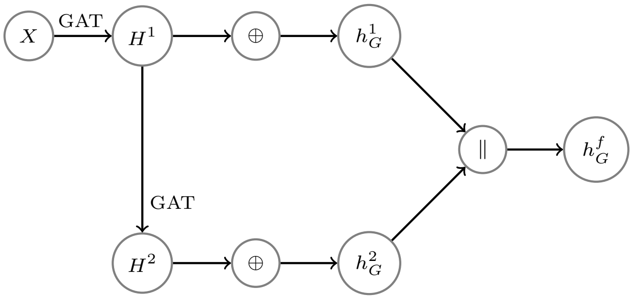


Figure 2: The process of building h_G^f where $\oplus$ produces two vectors, the first vector by summing node features H^k and the second vector by applying element-wise maximization. These two vectors are concatenated to form h_G^k, representing aggregated graph features after k message passing steps. The $\|$ symbol represents the final concatenation of vectors h_G^k to construct h_G^f.

4 Method Description

4.1 States Representation

Our work targets loop transformations since loops take most of a program's execution time. This aligns with the common practice in state-of-the-art compilers with automatic code optimization capabilities [1, 2, 9, 11, 41, 47, 54, 55]. To represent loops written in Tiramisu code, we use an intermediate representation under the form of a tree known as an Abstract Syntax Tree (AST). A node i in the tree can be either an iterator[3] or a computation, and an edge e_{ij} between two nodes i and j signifies that node i is the parent of node j. Children nodes can be iterators or computations[4], while parent nodes can only be iterators. Figure 1 illustrates an example of how the AST is built from a given loop nest[5].

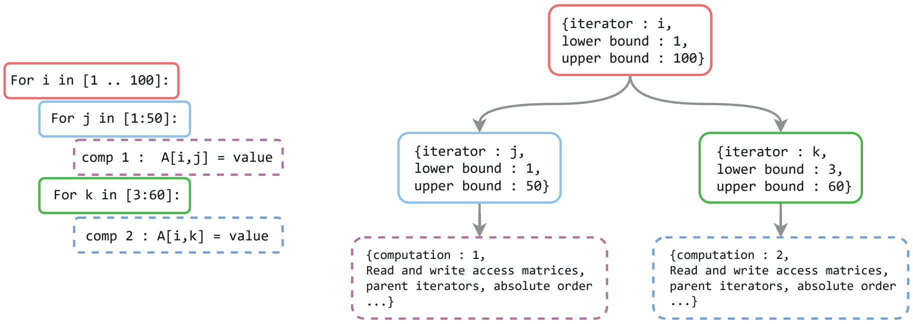
We build the node representation matrix X, and the graph edges E from the AST. A row in the matrix X represents the features of a single node in the AST. For the graph edges (E), we use simple edges with no attributes and keep the connections between the nodes as they originally existed in the AST. We map information from the AST nodes into fixed-size vectors for node features. The entire representation of nodes is detailed in Section 4.1.1.

As illustrated in Figure 2, we pass X through 2-layers of message passing using a GATv2 model and aggregate node representations at each layer using average and max pooling. Finally, we concatenate the aggregated

[3] We use the words iterator and loop interchangeably.

[4] We use the words computation and statement interchangeably.

[5] Loop nest: set of loops where one loop is contained within another.



```
For i in [1 .. 100]:
    For j in [1:50]:
        comp 1 :  A[i,j] = value
    For k in [3:60]:
        comp 2 : A[i,k] = value
```

Figure 1: The construction of the Abstract Syntax Tree from a loop nest in a Tiramisu function. The nodes in the tree with dashed line rectangles represent statements or computations. The nodes with solid line rectangles represent iterators.

representations from each layer of message passing to produce a final feature vector h_G^f that represents the graph. We refer the reader to the background section for a more detailed description of how graph neural networks process their inputs.

4.1.1 Node Feature Representation. There are two types of nodes in the AST: iterators and computations. To make the difference between the two types, we use a vector of the same size to represent both types with different tags and padding. Using the same example as above, we will depict the details of each node representation in Figure 3.

As shown in Figure 3, the first column differentiates the representation of the iterator nodes from the computation nodes, followed by specific features for each type of node. For the iterators, the "Focus Tag" tells the agent that we are targeting this iterator for optimization, so the following action will potentially include that node. The other tags, like Parallelization and Reversal, represent whether or not those actions have transformed the iterator. In addition to the read and write access matrices (following the concept described in section 3.4). We stack those vectors to form the input node representation matrix X.

4.2 Actions and AST branches

In this work, we consider the following loop transformations: 1) loop parallelization (to exploit multicore parallelism); 2) loop unrolling (which unrolls loop iterations to exploit instruction level parallelism); 3) loop tiling (to improve data locality); 4) loop skewing (which allows the extraction of outer parallelism and improves data locality); 5) loop interchange (which changes the order of loops within a loop nest to enable better data locality and parallelism); 6) loop reversal (which reverses the order of the loop iterations to enable better data locality and parallelism). Each of these transformations is applied to particular loops within a loop nest and many transformations have parameters. For example, loop tiling when applied on three loops, has the tuple $T0, T1, T2$ as a parameter where $T0, T1, T2$ are the tile sizes for each one of the three loops being tiled. They can take a value equal to the power of 2 and is between 2 and 256. The size of the search space covered by these transformations and their parameters is in the range of 10^{170} [1, 8].

At each time step t, the RL agent must determine the iterators impacted by the action a_t, where an action is defined by the tuple $\mathcal{T}, \mathcal{I}, \mathcal{C}, \mathcal{F}$, where $\mathcal{T}$ = {Parallelization, Unrolling, Tiling, Skewing, Interchange, Reversal, Next} is the set of loop transformations types in addition to "Next" which does not transform the loops but used to switch between branches, this action will be described in the next paragraph. The sets $\mathcal{I}$ and $\mathcal{C}$ represent the iterators and computations affected by a_t, respectively. Additionally, $\mathcal{F}$ is the set of transformation-specific parameters, such as tile sizes for Tiling.

A given loop transformation is usually applied on a particular iterator within the loop nest. One might create an action space where each tuple (loop-transformation, AST-branch[6], iterator) becomes an action. This is not possible though, because the action space required in this case would be vast. This is mainly because ASTs of programs can take different forms with many branches and depths. To solve this challenge, the agent traverses the AST progressively, branch by branch. It starts at the leftmost branch of the tree as shown in Figure 4, the agent chooses actions that target the iterators inside that branch and after selecting the "Next" action, the agent will target the next branch going from left to right to traverse loops by their order of appearing in the

[6] A branch is a path from the root to a leaf in the AST

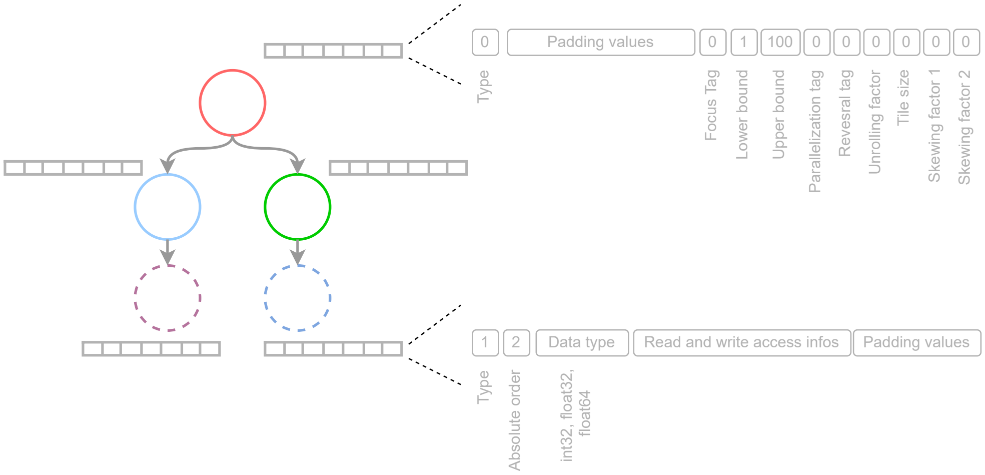


Figure 3: The construction of nodes features.

program. The episode ends when the agent is targeting the rightmost branch and the action "Next" is chosen.

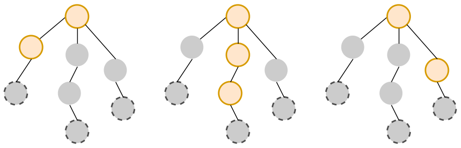


Figure 4: The graph at the left has 2 nodes colored in orange that represent the initial targeted branch. After applying loop transformations on the selected iterators, the agent uses the "Next" action to switch to the middle branch, as illustrated by the middle graph. The graph on the right represents the last branch. Choosing "Next" from that state will put an end to the episode and the optimization process.

4.2.1 Detailed Actions Space. The agent's action space consists of 56 actions, each one represents a loop transformation with its parameters. Figure 5 illustrates the output of the policy and the structure of our action space.

- I(i, j) Interchange of loop levels i and j in the targeted branch.
- R(i) Reversal of the loop level i in the targeted branch.
- S(i, j) Skewing of loop levels i and j in the targeted branch.
- P(i) Parallelization of loop level i in the targeted branch.
- T(i, j, x, y) Tiling of loop levels i and j with tile sizes x and y respectively.
- U(x) Unrolling of the innermost loop level with an unrolling factor x.
- Next: targets the next branch in the AST.

4.3 Rewards

The typical performance evaluation in code optimization is defined by the speedup[7] gained after applying a transformation. The final speedup τ_f is calculated by multiplying intermediate speedups τ_i, $\tau_f = {}_{i=1}^{n} \tau_i$. In reinforcement learning, an agent's goal is to maximize a sum of rewards. To adapt the product of speedups and use it as the agent's reward signal, we use the *log* function to transform the product into a sum ($\log {}_{i=1}^{n} \tau_i = {}_{i=1}^{n} \log \tau_i$) given that the logarithmic function is monotonically increasing. The agent, thus, receives a reward $r_t = \log a_t$ instead of a_t. Using the logarithm also scales down high values that can increase variance and impact the training's stability.

4.4 Actor Critic Network

After extracting the graph-level features h_G^{final}, we feed it as input to a feed forward network that is divided into a policy to predict the action probabilities, and a state-value approximator. Figure 6 illustrates the overall architecture of the model. The presented architecture was selected based on a series of experimental evaluations (section 6.4).

The GAT layers have 4 attention heads, each with a hidden size of 128. We use a linear function for each layer to project its expanded hidden size multiplied by the

[7] Speedup: original execution time divided by the execution time of transformed code.

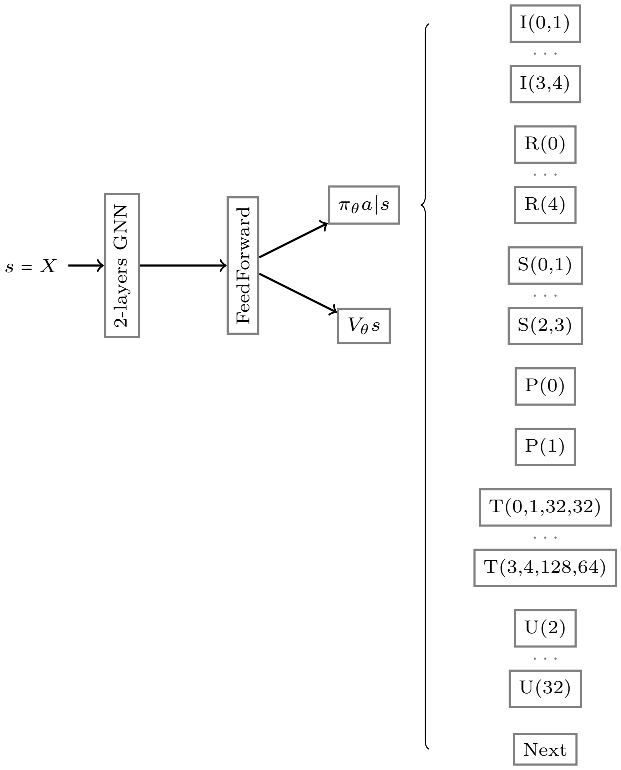


Figure 5: The action space

number of heads into its original size. After concatenating and getting h_G^{final}, we pass it through a 2-layer MLP and use the scaled exponential linear unit (SELU) [33] as the activation function. We then separate the network into two heads: a policy head and a state-value head. Both heads have the same architecture, with only the output layer size different. Both are a 3-layer MLP with sizes (128, 128, 56) for the policy head where 56 is the number of actions and (128, 128, 1) for the state-value head.

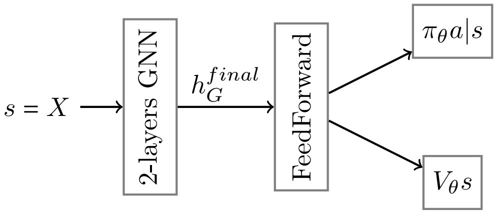


Figure 6: The architecture of our actor-critic agent. The backbone of this model processes the graph input and produces a vector h_G^{final} summarizing the graph's characteristics. Followed by feedforward layers, it is then divided into the policy and value heads.

4.5 Training the Agent

To train our agent, we use PPO [48]. When our agent applies an action, the Tiramisu environment tests the legality[8] of the action before executing it. We use classical polyhedral dependence analysis and legality checking [20, 52] to guarantee the correctness of transformations. As mentioned, we transform speedups using the logarithmic function with a base of 4 to increase the training stability. We assign a speedup of 1 for illegal actions and do not apply them on the original program; we leave it up to the agent to learn what valuable transformations to apply.

4.6 Training Dataset

To train the RL agent we used 2,500 randomly generated tiramisu programs. We used the same methodology defined by Baghdadi et al. [8] to generate these programs. The agent explored a considerable number of schedules (sequences of code optimizations) for each program. Overall, the total number of unique schedules that the agent was trained on has reached $\sim$ 45,000.

[8] In code optimization, a transformation is deemed legal if its application does not violate the program's data dependencies.

5 Accelerating the Training of the RL Agent

During training, when the RL agent picks an action (a code optimization), it first checks the legality of that action (using classical polyhedral legality check provided in the Tiramisu compiler[9]). If the action is legal, the agent then computes its reward. To do that, it applies the code optimization and compiles the optimized program and runs it on the target hardware and then computes the execution time of the optimized program and use that to compute the reward. These three steps: checking legality, compiling code and running code on the target hardware are time consuming. For example, checking the legality of code optimizations takes about 30% of the training time in our RL system, while compiling and running code takes the majority of the remaining time. Because these steps are time consuming, a single training of the RL agent takes weeks. During the development phase of the RL system, one needs to train the RL system many times, to test different features, ideas, and to fine-tune the hyper-parameters. Since every training takes weeks, the development of the RL system becomes challenging. In this section, we propose methods to mitigate this issue.

5.1 Execution Time and Legality Check Memoization

The main idea for this method is to store the result of the legality check and execution times for the actions explored by the agent during a given training, and reuse them in future trainings.

We construct a dataset, where we store the result of the legality check and execution time obtained for each action chosen by the RL agent. The dataset has the following components:

(1) **Programs:** the list of all the randomly generated programs used to train the RL agent.
(2) **Schedules:** the list of schedules explored by the RL agent for each program (a schedule is a sequence of code optimizations).
(3) **Schedule legality:** for each pair of (program, schedule) in the dataset, we store the legality of the schedule, as a boolean value.
(4) **Execution times:** for each pair of (program, schedule) in the dataset, we store the execution time of the *program* when optimized using the *schedule*. The reward can be easily derived from this execution time.

In subsequent trainings, when the RL agent picks an action, it first queries the dataset to check if the corresponding program and schedule exist. If found, the legality check and execution times stored in the dataset are retrieved, bypassing the need for the legality check, compilation, and execution. This significantly reduces redundant computations. If the legality check and execution times are not found, the agent performs the legality check, compiles and executes the optimized program and updates the dataset, enriching it for future use. Initially, when the dataset is empty, the training is slow. Once an initial training is performed, future trainings are likely to be faster. This is particularly true because RL agents, when they converge, tend to pick the same actions, and therefore the probability of picking an action that has been explored in previous trainings is high.

5.2 Actor-Critic Pre-training

The goal of this technique is to enable better learning in the actor-critic neural networks. That would translate in either faster convergence, or to a convergence to a higher average reward. We improve the learning in the actor-critic neural networks by initializing the weights of the actor-critic neural networks through actor-critic pre-training (i.e., by pre-training the actor-critic neural networks).

The GNN layers, the feed forward, and the value network $V_\theta s$ (mentioned in Figure 6) are trained, before training the RL agent, on a surrogate task: predicting the execution times of unoptimized programs from their graph representation. The weight of these layers are then used as an initialization for the corresponding layers in the RL agent. Predicting the execution times of programs aligns well with the goals of pre-training the actor-critic agent because it is a complex task that provides a substantial amount of knowledge about the differences between programs. While a more targeted pre-training, such as using data in the format *(program, schedule, speedup)*, could potentially be better, the lack of sufficient data and the significant time required to generate a large dataset directed our choice toward execution time prediction. Our goal was not to create a highly accurate execution time prediction model but to provide the agent with reasonable initial weights. We believe that the current approach is sufficient to achieve that. We plan to extend our work in the future to pre-train on data of the format *(program, schedule, speedup)*.

Note that even if we train the RL agent with higher quality data, we do not have guarantees on faster convergence. This is mainly because even if we pre-train the actor-critic with such data, the RL agent would still need to explore actions initially with a certain degree of randomness, due to the use the epsilon greedy algorithm [49] in training the RL agent, a common method for balancing exploration and exploitation in training reinforcement learning agents. In epsilon greedy, an entropy coefficient is set to a high value initially and then decays over time to reach zero. Higher values of the entropy coefficient force the RL agent to explore actions more initially, and over time during training, the entropy becomes smaller, allowing the RL agent to converge. Because the agent has to explore actions with a degree of randomness in its initial phase of training (while the entropy coefficient is high), it will continue exploring (with a degree of randomness) even if it converges earlier, making the training last for longer.

We pre-trained the layers (GNN layers, feedforward layers and the value network) on 26,000 data points. The model input is the graph representation of a program, and the output is the execution time of the program (unoptimized program). We use the MSE (Mean Squared Error) Loss and train for 1500 epochs with a 10^{-4} learning rate. The resulting weights are then used to initialize the GNN layers, feedforward layers and the value network layers.

6 Experiments and Evaluation

6.1 Experimental Setup

To train our reinforcement learning agent, we used a cluster where each node is a 28-core Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz, 4 GB of RAM per core. The OS installed on the nodes is CentOS Linux version 8. We used distributed learning to train the model using 4 nodes of the cluster in all the upcoming evaluations.

6.2 Training Details

To train the GNN with PPO we used the following set of hyperparameters as specified in Table 2.

Table 2: Parameters of training the agent.

Parameters (PPO)	Value
ϵ_{clip}	0.3
γ	0.99
λ	0.95
Value coefficient	2
Entropy coefficient (decaying)	$10^{-1} \rightarrow 0$
Batch size	512
Num epochs	5
Mini-Batch size	64
Learning rate	10^{-4}
Parameters (GNN)	
Type	GATv2 [15]
Num layers	2
Num of attention heads	4
Hidden layers size	128

6.3 Evaluation on a Benchmark Suite

In this section, we evaluate how effective is our RL agent in optimizing real-world benchmarks. For this evaluation, we use the same benchmark suite used by Baghdadi et al. [8], a set of benchmarks including image processing, deep learning, and linear algebra programs. For each benchmark, we execute the schedule obtained by the RL agent 30 times before taking the minimum to reduce the effect of noise on the results. The following are descriptions of the benchmarks used:

- **blur:** an image processing filter to blur images.
- **cvtcolor:** an image processing filter for converting the colors of an input image from RGB to grayscale.
- **doitgen:** a kernel from the multiresolution adaptive numerical scientific simulation [44].
- **heat2d:** Heat equation over 2D data domain.
- **heat3d:** Heat equation over 3D data domain.
- **jacobi2d:** a jacobi-style stencil computation over 2D data with 5-point stencil pattern.
- **mvt:** matrix-vector multiplication composed with another matrix-vector multiplication but with a transposed matrix.
- **seidel-2d:** two dimensional Seidel stencil computation.

We compare the speedups we get using our proposed system to those produced by the Tiramisu autoscheduler [8] (Tiramisu's search-based automatic code optimization). The speedup of an optimized program is defined as follows:

$$speedup = \frac{exec\_time\_unoptimized\_program}{exec\_time\_optimized\_program} \quad (1)$$

Speedups higher than 1 indicate that the optimized program is faster than the original one. The baseline of computing the speedups in these experiments is the execution time of the unoptimized program.

We also compare the speedups of our proposed system to Pluto, a state-of-the-art polyhedral compiler that does not use machine learning (it uses Integer Linear Programming). Pluto, being a polyhedral compiler like Tiramisu, supports a large space of complex code transformations (we used Pluto with the *–parallel –tile* options to enable parallelism and tiling). We also compare to HalideRL, a state-of-the-art compiler that uses reinforcement learning for automatic code optimization, and that we consider to be the closest to our work. HalideRL trains the RL on each one of the benchmarks. We perform the training on our target machine (i.e., on our cluster nodes) and leave it until it converges. We do not compare with PolyGym in this experiment because PolyGym focuses mainly on proposing an RL environment and does not propose a deep RL agent for the environment.

The performance of our proposed system, compared to the state-of-the-art, is presented in Table 3. Our agent predicts code optimizations that lead to a geometric mean speedup reaching 3.16× over unoptimized code. It also has a geometric mean speedup higher than the Tiramisu autoscheduler, Pluto and HalideRL, with a geometric mean speedup of 2.02× over the Tiramisu autoscheduler. HalideRL crashed for 3 benchmarks (*blur*, *doitgen*, and *jacobi2d*), and therefore we did not report speedups for those.

Our system is the only one where the agent found only useful code optimizations. In other words, it did not choose optimizations that led to a slowdown compared to the original unoptimized program (all the speedups are >= 1).

Table 3: Summary of results regarding execution time speedup achieved by each method. The baseline of computing the speedups is the original execution time of the functions without any transformation applied.

Benchmark	Tiramisu Autoscheduler	Pluto	HalideRL	Pearl
blur	0.27	1.01	/	**4.27**
cvtcolor	**1.12**	0.90	0.14	1
doitgen	2.66	0.74	/	**11.37**
heat2d	1.86	0.98	1.15	**2.39**
heat3d	0.82	1.01	3.10^{-3}	**2.36**
jacobi2d	**1.66**	1	/	1
mvt	4.14	0.97	0.27	**6.1**
seidel2d	4.24	0.99	5.57	**6.03**
geo mean	1.56	0.94	0.23	**3.16**

High speedups in benchmarks such as *doitgen*, *mvt*, *seidel2d*, *blur*, *heat2d*, and *heat3d* are due to the application of parallelization and tiling which improves data locality. The agent refrained from parallelizing code in cases where parallelization leads to a decrease in performance (if the overhead of parallelization is higher than its benefit). This was the case for *cvtcolor*, for example, where the outer loop is the color channel and has only 3 iterations. Parallelizing such a loop leads to high overhead with little benefit. The Tiramisu autoscheduler could obtain better speedups than our agent in this case because it applied another transformation (loop interchange) that interchanged one of the inner loops (which represents the image height) to become the outer loop and then parallelized that loop. Since the new outer loop has a high number of iterations, parallelization was beneficial and therefore the Tiramisu autoscheduler obtained a higher speedup.

In summary, the evaluation shows that our proposed agent was able to outperform three state-of-the-art compilers. The first uses a search-based method (Tiramisu autoscheduler), the second uses integer linear programming (Pluto), while the third uses RL (HalideRL). Among these three, the Tiramisu autoscheduler achieved the best speedups, but our proposed RL ouperformed the Tiramisu autoscheduler by 2.02× , highlighting the benefit of an RL-based approach.

6.4 Model Architecture Design Choices

6.4.1 Evaluating Different GNN Models. Alongside the GAT architecture that we use in this work, we evaluated other prominent GNN models such as Graph Convolutional Networks (GCN) [32] and GraphSAGE [27]. Figure 7 depicts the comparison between training the three agents with PPO.

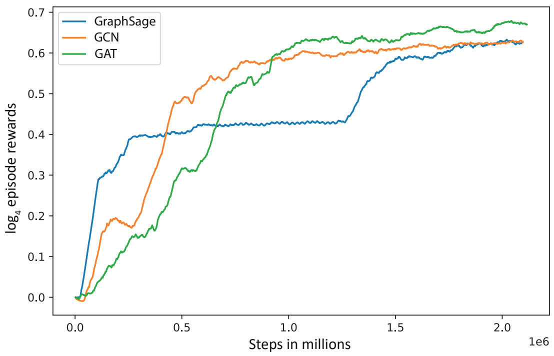


Figure 7: The performance of three agents using different GNN layers, the y-axis represents the average log_4 of the episodes speedups collected in a single PPO iteration. The x-axis represents the number of actions taken in total.

For the three agents, we use the same architecture described in 4.4 except for the GNN layer type we want to test. We notice a slightly better performance of the GAT agent over GCN and GraphSage. Note that we did this experiment early in the lifetime of the project, on a subset of our training dataset. We believe that the results would generalize to the whole dataset though.

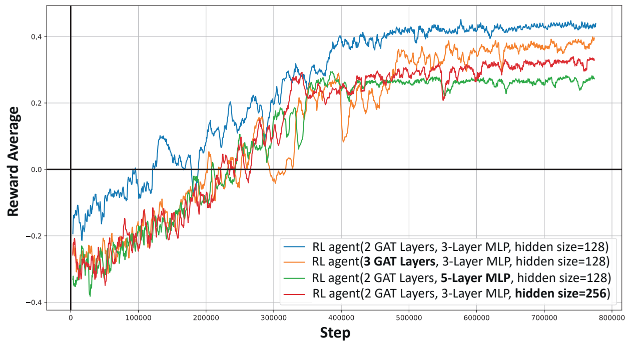


Figure 8: Reward Averages Across Different Experiments over Training Steps

6.4.2 Number of GAT and MLP Layers. In this section, we evaluate different variants of our proposed model. Our focus in these experiments was on the number of layers of GATs, MLPs, and the size of the hidden layers, as these affect the ability of the model to learn complex patterns and relations in the data. The best-performing agent uses 2 GAT layers, 3 MLP layers and a size of the hidden layers equal to 128. We tried out different other configurations that presented in Figure 8.

6.5 Evaluating the Execution Time and Legality Check Memoization Method

To analyze the efficiency of the memoization technique, we evaluated the training time of our proposed RL agent with and without memoization. Figure 9 shows a significant reduction in convergence time for the agent that uses memoization. It converges to the best average reward in 45 hours.

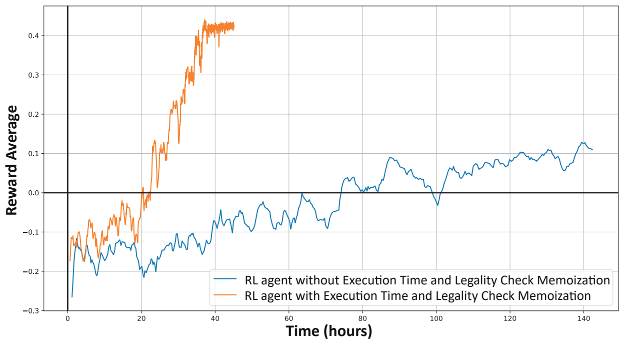


Figure 9: Reward Averages of our Agents With and Without Execution Time and Legality Check Memoization

To further evaluate the technique of memoization, we recorded the total number of hits while training our RL agent. A hit in this case indicates that a schedule (with its legality and execution time) is already present in the dataset. Figure 10 shows how the total number of hits increases as the training progresses, which in turn indicates that the stored values are indeed being used during the RL training.

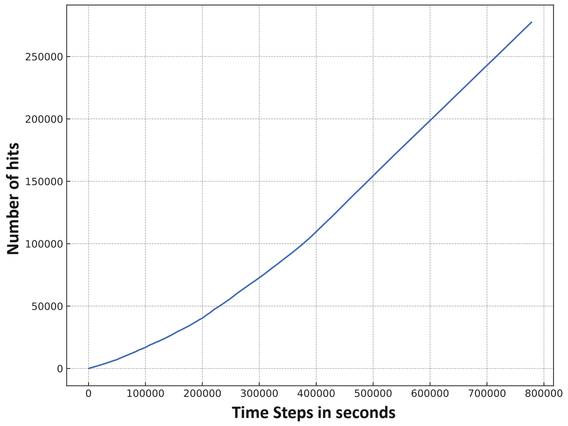


Figure 10: Number of hits during the training of the RL agent.

6.6 Evaluating the Actor-Critic Pre-training

To evaluate the actor-critic pre-training technique, we trained two agents: the first, without pre-training, while the second uses the pre-training technique mentioned in Section 5.2. In these experiments, we use the memoization technique as we have already demonstrated its effectiveness. We recorded the average reward during training, the training time, and then evaluated the trained RL agents on the benchmark suite used in section 6.3.

Figure 11 shows the average reward of the two RL agents. The RL agent that used pre-training managed to obtain a much higher average reward, compared to the agent that did not. This difference in the average reward is significant though, since the reward is the *log* of the speedup, and therefore a small difference in the average reward translates to much larger difference in speedups. Table 4 shows an evaluation of the two RL agents on the benchmark. The RL that used the pre-training method obtained significantly better speedups on the benchmarks with nearly the same training times (Figure 11). The agent without pre-training failed to optimize the *Heat2d* benchmark. In the case of *seidel2d* it unrolled the second loop instead of parallelizing the outermost loop and then applying tiling. We clearly observe that this agent did not learn the importance of parallelizing outermost loops, among other patterns.

In this experiment, we did not notice faster convergence of the RL agent because of the high initial entropy used in this training. This high entropy is important for the agent to learn useful code optimizations though, and lower entropy values would lead to a lower average reward. The actor-critic pre-training method was useful in allowing the actor-critic neural networks to better learn though, and this translates in obtaining a better average reward.

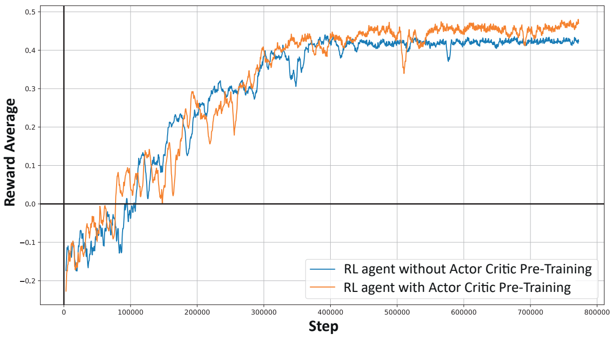


Figure 11: Reward Averages of our Agents With and Without Actor-Critic Pre-training

Table 4: Speedups achieved by Pearl with and without Actor-Critic Pre-training

Benchmark	Pearl	Pearl
	with Pre-Training	without Pre-Training
blur	**4.27**	4.17
cvtcolor	**1**	1
doitgen	11.37	**12.09**
heat2d	**2.39**	1
heat3d	2.36	**2.39**
jacobi2d	**1**	1
mvt	**6.1**	5.98
seidel2d	**6.03**	0.65
geo mean	**3.16**	2.15

6.7 Search Space Exploration Time Comparison

Our RL system not only has better performance compared to the existing Tiramisu auto-scheduler, but it also does so significantly faster, which is one of the key contributions of this work. Reinforcement learning has the advantage of learning a policy network (actor) that directly predicts the sequence of code optimizations to apply to obtain the best performance. This is in contrast to current state-of-the-art methods that use tree-search methods to explore the search space using a tree-search method (e.g., beam-search). Tree-search methods are time-consuming compared to a policy network learned through reinforcement learning. On the reported benchmarks, the RL system identified the sequence of cost optimizations to use in *33.36 milliseconds* on average, which is 563.67× faster than the Tiramisu scheduler [8].

7 Discussion and Future Work

In its current state, our proposed system supports six types of loop transformations. These transformations, along with their parameters and the choices of which loops they apply to, constitute a large search space reaching 10^{170} candidates [1, 8]. Although this search space is already large, we plan to support a wider set of loops and data layout transformations. We also plan to explore the effect of applying RL on raw text data instead of encoding the states.

8 Conclusion

This paper introduces a deep-reinforcement learning-based autoscheduler for the Tiramisu compiler. We train an RL agent on a dataset of synthetic Tiramisu programs. During training, the policy network of the agent converges into a heuristic that we use to infer schedules for unseen programs. By evaluating our proposed system on standard benchmarks, we show its competitiveness with state-of-the-art autoschedulers. Compared to Tiramisu, our RL-based agent achieves an overall geometric mean speedup of 2.02× .

9 Acknowledgment

This research has been partly supported by the Center for Artificial Intelligence and Robotics (CAIR) at New York University Abu Dhabi, funded by Tamkeen under the NYUAD Research Institute Award CG010. The research was carried out on the High-Performance Computing resources at New York University Abu Dhabi.

References

[1] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. 2019. Learning to Optimize Halide with Tree Search and Random Programs. *ACM Trans. Graph.* 38, 4, Article 121 (jul 2019), 12 pages. https://doi.org/10.1145/3306346.3322967

[2] Byung Hoon Ahn, Prannoy Pilligundla, Amir Yazdanbakhsh, and Hadi Esmaeilzadeh. 2020. Chameleon: Adaptive Code Optimization for Expedited Deep Neural Network Compilation.

arXiv:2001.08743

[3] Mohamed Riyadh Baghdadi. 2015. *Improving tiling, reducing compilation time, and extending the scope of polyhedral compilation.* Ph. D. Dissertation. Paris 6.

[4] Riyadh Baghdadi, Albert Cohen, Cedric Bastoul, Louis-Noel Pouchet, and Lawrence Rauchwerger. 2011. The Potential of Synergistic Static, Dynamic and Speculative Loop Nest Optimizations for Automatic Parallelization. arXiv:1111.6756 [cs.DC]

[5] Riyadh Baghdadi, Albert Cohen, Tobias Grosser, Sven Verdoolaege, Javed Absar, Sven Van Haastregt, Alexey Kravets, Anton Lokhmotov, and Alastair Donaldson. 2015. *PENCIL Language Specification.* Ph. D. Dissertation. INRIA.

[6] Riyadh Baghdadi, Albert Cohen, Serge Guelton, Sven Verdoolaege, Jun Inoue, Tobias Grosser, Georgia Kouveli, Alexey Kravets, Anton Lokhmotov, Cedric Nugteren, et al. 2013. PENCIL: Towards a platform-neutral compute intermediate language for DSLs. *arXiv preprint arXiv:1302.5586* (2013).

[7] Riyadh Baghdadi, Abdelkader Nadir Debbagh, Kamel Abdous, Fatima Zohra Benhamida, Alex Renda, Jonathan Elliott Frankle, Michael Carbin, and Saman Amarasinghe. 2020. TIRAMISU: A Polyhedral Compiler for Dense and Sparse Deep Learning. arXiv:2005.04091 [cs.DC]

[8] Riyadh Baghdadi, Massinissa Merouani, Mohamed-Hicham Leghettas, Kamel Abdous, Taha Arbaoui, Karima Benatchba, et al. 2021. A deep learning based cost model for automatic code optimization. *Proceedings of Machine Learning and Systems* 3 (2021), 181–193.

[9] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO).* IEEE, 193–205.

[10] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Patricia Suriana, Shoaib Kamil, and Saman P Amarasinghe. 2018. Tiramisu: A code optimization framework for high performance systems. *arXiv preprint arXiv:1804.10694* (2018).

[11] Uday Bondhugula, Albert Hartono, J Ramanujam, and P Sadayappan. 2008. Pluto: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008).* Citeseer.

[12] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 101–113.

[13] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI.* 101–113.

[14] Alexander Brauckmann, Andrés Goens, and Jeronimo Castrillon. 2021. A reinforcement learning environment for polyhedral optimizations. *arXiv preprint arXiv:2104.13732* (2021).

[15] Shaked Brody, Uri Alon, and Eran Yahav. 2022. How Attentive are Graph Attention Networks? arXiv:2105.14491 [cs.LG]

[16] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. *arXiv preprint arXiv:1802.04799* (2018).

[17] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2019. Learning to Optimize Tensor Programs. arXiv:1805.08166

[18] Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, et al. 2022. Compilergym: Robust, performant compiler optimization environments for ai research. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO).* IEEE, 92–105.

[19] Alain Darte and Guillaume Huard. 2005. New Complexity Results on Array Contraction and Related Problems. *J. VLSI Signal Process. Syst.* 40, 1 (May 2005), 35–55. https://doi.org/10.1007/s11265-005-4937-3

[20] P. Feautrier. 1988. Array expansion. In *Proceedings of the 2nd international conference on Supercomputing.* ACM, St. Malo, France, 429–441. https://doi.org/10.1145/55364.55406

[21] Paul Feautrier and Christian Lengauer. 2011. *Polyhedron Model.* Springer US, Boston, MA, 1581–1592. https://doi.org/10.1007/978-0-387-09766-4_502

[22] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. 2017. Neural message passing for quantum chemistry. In *International conference on machine learning.* PMLR, 1263–1272.

[23] Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. 2014. Hybrid Hexagonal/Classical Tiling for GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Orlando, FL, USA) *(CGO '14).* ACM, New York, NY, USA, Article 66, 10 pages.

[24] Tobias Grosser, Armin Groslinger, and Christian Lengauer. 2012. Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Processing Letters* 22, 4 (2012). http://dblp.uni-trier.de/db/journals/ppl/ppl22.html#GrosserGL12

[25] Ameer Haj-Ali, Hasan Genc, Qijing Huang, William Moses, John Wawrzynek, Krste Asanović, and Ion Stoica. 2020. Protuner: tuning programs with monte carlo tree search. *arXiv preprint arXiv:2005.13685* (2020).

[26] Yacine Hakimi, Riyadh Baghdadi, and Yacine Challal. 2023. A hybrid machine learning model for code optimization. *International Journal of Parallel Programming* 51, 6 (2023), 309–331.

[27] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

[28] Guoliang He, Sean Parker, and Eiko Yoneki. 2023. X-RLflow: Graph Reinforcement Learning for Neural Network Subgraphs Transformation. arXiv:2304.14698 [cs.LG] https://arxiv.org/abs/2304.14698

[29] Qijing Huang, Ameer Haj-Ali, William Moses, John Xiang, Ion Stoica, Krste Asanovic, and John Wawrzynek. 2020. Autophase: Juggling hls phase orderings in random forests with deep reinforcement learning. *arXiv preprint arXiv:2003.00671* (2020).

[30] Wang Huanting, Tang Zhanyong, Zhang Cheng, Zhao Jiaqi, Cummins Chris, Leather Hugh, and Wang Zheng. 2022. Automating Reinforcement Learning Architecture Design for Code Optimization. In *Proceedings of the 31st ACM*

SIGPLAN International Conference on Compiler Construction (Seoul, South Korea) *(CC 2022)*. Association for Computing Machinery, New York, NY, USA, 129–143. https://doi.org/10.1145/3497776.3517769

[31] F. Irigoin and R. Triolet. 1988. Supernode Partitioning. In *(POPL'88)*. San Diego, CA, 319–328.

[32] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[33] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. 2017. Self-normalizing neural networks. *Advances in neural information processing systems* 30 (2017).

[34] Vincent Lefebvre and Paul Feautrier. 1998. Automatic storage management for parallel programs. *Parallel Comput.* 24 (1998), 649–671. https://doi.org/10.1016/S0167-8191(98)00029-5

[35] Chunting Liu and Riyadh Baghdadi. 2025. Data-Efficient Performance Modeling via Pre-training. In *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction*. 48–59.

[36] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. 2019. Optimizing {CNN} model inference on {CPUs}. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 1025–1040.

[37] Moshe Looks, Marcello Herreshoff, DeLesley Hutchins, and Peter Norvig. 2017. Deep Learning with Dynamic Computation Graphs. arXiv:1702.02181

[38] Massinissa Merouani, Khaled Afif Boudaoud, Iheb Nassim Aouadj, Nassim Tchoulak, Islem Kara Bernou, Hamza Benyamina, Fatima Benbouzid-Si Tayeb, Karima Benatchba, Hugh Leather, and Riyadh Baghdadi. 2024. LOOPer: A Learned Automatic Code Optimizer For Polyhedral Compilers. *arXiv preprint arXiv:2403.11522* (2024).

[39] Massinissa Merouani, Mohamed-Hicham Leghettas, Riyadh Baghdadi, Taha Arbaoui, and Karima Benatchba. 2020. *A deep learning based cost model for automatic code optimization in tiramisu.* Ph. D. Dissertation. PhD thesis, 10 2020.

[40] Lina Mezdour, Khadidja Kadem, Massinissa Merouani, Amina Selma Haichour, Saman Amarasinghe, and Riyadh Baghdadi. 2023. A deep learning model for loop interchange. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*. 50–60.

[41] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically Scheduling Halide Image Processing Pipelines. *ACM Trans. Graph.* 35, 4, Article 83 (jul 2016), 11 pages. https://doi.org/10.1145/2897824.2925952

[42] Aditya Paliwal, Felix Gimeno, Vinod Nair, Yujia Li, Miles Lubin, Pushmeet Kohli, and Oriol Vinyals. 2020. Reinforced Genetic Algorithm Learning for Optimizing Computation Graphs. arXiv:1905.02494

[43] Marcelo Pecenin, André Murbach Maidl, and Daniel Weingaertner. 2019. Optimization of halide image processing schedules with reinforcement learning. In *Anais do XX Simpósio em Sistemas Computacionais de Alto Desempenho*. SBC, 37–48.

[44] Louis-Noël Pouchet et al. 2012. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/pouchet/-software/polybench* 437 (2012), 1–1.

[45] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. 2011. Loop Transformations: Convexity, Pruning and Optimization. In *38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'11)*. ACM Press, Austin, TX, 549–562.

[46] F. Quilleré and S. Rajopadhye. 2000. Optimizing Memory Usage in the Polyhedral Model. *ACM Trans. on Programming Languages and Systems* 22, 5 (Sept. 2000), 773–815.

[47] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *SIGPLAN Not.* 48, 6 (jun 2013), 519–530. https://doi.org/10.1145/2499370.2462176

[48] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[49] R.S. Sutton and A.G. Barto. 2018. *Reinforcement Learning, second edition: An Introduction.* MIT Press. https://books.google.dz/books?id=sWV0DwAAQBAJ

[50] William Thies, Frédéric Vivien, Jeffrey Sheldon, and Saman Amarasinghe. 2001. A unified framework for schedule and storage optimization. In *Proc. of the 2001 PLDI Conf.*

[51] Konrad Trifunovic, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjodin, and Ramakrishna Upadrasta. 2010. GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation.

[52] Nicolas Vasilache, Cedric Bastoul, Albert Cohen, and Sylvain Girbal. 2006. Violated Dependence Analysis. In *Proceedings of the 20th Annual International Conference on Supercomputing* (Cairns, Queensland, Australia) *(ICS '06)*. Association for Computing Machinery, New York, NY, USA, 335–344. https://doi.org/10.1145/1183401.1183448

[53] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zach DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *CoRR* abs/1802.04730 (2018).

[54] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730* (2018).

[55] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, 4, Article 54 (jan 2013), 23 pages. https://doi.org/10.1145/2400682.2400713

[56] Michael E Wolf and Monica S Lam. 1991. A loop transformation theory and an algorithm to maximize parallelism. *IEEE transactions on parallel and distributed systems* 2, 4 (1991), 452–471.

[57] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating {High-Performance} tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*. 863–879.

[58] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2023. Ansor: Generating High-Performance Tensor Programs for Deep Learning. arXiv:2006.06762

[59] Zhen Zheng, Pengzhan Zhao, Guoping Long, Feiwen Zhu, Kai Zhu, Wenyi Zhao, Lansong Diao, Jun Yang, and Wei Lin. 2021. FusionStitching: Boosting Memory Intensive Computations for Deep Learning Workloads. arXiv:2009.10924