République Algérienne Démocratique et Populaire
الجمهورية الجزائرية الديموقراطية الشعبية
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
وزارة التعليم العالي والبحث العلمي

المدرسة الوطنية للإعلام الآلي
(المعهد الوطني للتكوين في الإعلام الآلي سابقا)
École nationale Supérieure d'Informatique
ex. INI (Institut National de formation en Informatique)

Internship Manuscript

2nd year of the Higher Cycle (2CS)

Option : Intelligent Systems and Data (SID)

Automatic Code Optimization using Reinforcement Learning in the Tiramisu Compiler

Author:
Kamel Brouthen

Supervised By:
Dr. Mohamed Riyadh Baghdadi

Host Organization:
Laboratoire de Méthodes de Conception des Systèmes (LMCS)
New York University Abu Dhabi (NYUAD)

2024 / 2025

Dedication

This work is dedicated to my precious family. To my parents, whose unwavering sacrifices and nurturing of my love for science and curiosity have created a home that cherishes love, sincere work, and dreams. Your endless support, boundless love, and honest belief in me have been the cornerstone of my journey. I aspire to make you proud and honor the values you have instilled in me every step of the way.

To my sister, whose unconditional encouragement and admiration inspire me each day. Your limitless faith in your little brother fuels my determination to excel and brings immense joy and happiness into my life. I dedicate this work to you, hoping to bring smiles and pride to your heart, cherishing every moment we share together.

To the precious and pure soul who stands by my side, inspiring and guiding me with genuine care. I appreciate your presence in my life and dedicate all my efforts to creating happy moments and memories for us to cherish and treasure together. For as much as I am grateful for all satisfaction, meaning, and pride you bring to my life, I dedicate this work to you.

To all my family members who have always loved me and prayed for my best, your collective support has been a constant source of strength and motivation. Your presence in my life enriches every achievement and milestone I reach.

Lastly, to all my friends with whom I share daily smiles, memories, struggles, and achievements. I am grateful for our priceless friendships and wish nothing but success and prosperity for each one of you.

وقل ربي زدني علما

Acknowledgment

All praise and thanks are due to Almighty Allah for granting and blessing me throughout my educational journey. His guidance instilled in me the curiosity to pursue learning and navigate the complexities of science, as well as the patience and perseverance to strive for excellence. I owe everything to His mercy and guidance, and may all my work serve as a step towards seeking knowledge and wisdom solely for Allah.

I would like to express my heartfelt gratitude to my supervisor, Dr. Riyadh Baghdadi, for providing the invaluable opportunity to work on this project. I am honored to have been guided and mentored by his research excellence and continuous support. My sincere thanks also go to the New York University Abu Dhabi team: particularly Djamel Rassem Lamouri, for his valuable guidance and effective onboarding for the project, as well as his ongoing support during the internship, and Aouadj Nassim for his thorough review and contributions to advance the project to larger-scale experimentation. Additionally, I am grateful to Inas Bachiri for her insightful feedback, valuable insights, and remarkable contributions.

I would also like to thank my esteemed university, *École Nationale Supérieure d'Informatique (ESI)*, for providing the precious opportunity to undertake an internship within the *Laboratoire de Méthodes de Conception des Systèmes (LMCS)*. My gratitude extends to all my teachers throughout my academic journey who fostered my growth in computer science, as well as to the jury members reviewing my work.

Furthermore, I am thankful to the School of AI Algiers community for the incredible AI journey I have cherished throughout my university studies, allowing me to learn and grow in the field of Artificial Intelligence.

Finally, I extend my acknowledgment to everyone who provided encouragement and support throughout this work.

Abstract

Efficient code optimization is crucial for high-performance and scalable computationally intensive applications like deep learning and scientific computing. Traditional manual optimization techniques in compilers are time-consuming and require extensive expertise, making them less suitable for the growing complexity of modern applications. Reinforcement learning (RL) has emerged as a promising solution for automatic code optimization, enabling compilers to autonomously learn optimal strategies.

This work presents a novel approach integrating an RL agent with a pretrained autoencoder to embed access matrices within the Abstract Syntax Tree (AST) representations of Tiramisu programs. By effectively compressing high-dimensional AST data, our method addresses significant challenges in training RL agents, such as large data requirements and extensive training times, allowing the agent to focus on the most relevant features for optimization.

Experimental results demonstrate that certain embedding configurations facilitate faster convergence during training and achieve competitive performance compared to the default representation, while significantly reducing input dimensionality. These findings highlight the potential of embedding strategies in enhancing RL-based code optimization within the Tiramisu compiler.

This research lays the foundation for further exploration of pretraining techniques in automatic code optimization using RL. Despite hardware limitations in our study, the promising results encourage future work with larger datasets and more powerful computational resources, paving the way for more efficient and scalable RL-based code optimization systems.

Keywords — Reinforcement Learning, Autoencoder, Embedding, Automatic Code Optimization, Tiramisu Compiler

---

Contents

List of Figures

List of Algorithms

List of Tables

Abbreviations

ACO	**A**utomatic **C**ode **O**ptimization
API	**A**pplication **P**rogramming **I**nterface
AST	**A**bstract **S**yntax **T**ree
DSL	**D**omain **S**pecific **L**anguage
GAE	**G**eneralized **A**dvantage **E**stimation
GAT	**G**raph **A**ttention **N**etworks
GNN	**G**raph **N**eural **N**etworks
LSTM	**L**ong **S**hort-**T**erm **M**emory
MDP	**M**arkov **D**ecision **P**rocess
ML	**M**achine **L**earning
PCA	**P**rincipal **C**omponent **A**nalysis
PPO	**P**roximal **P**olicy **O**ptimization
RL	**R**einforcement **L**earning
TRPO	**T**rust **R**egion **P**olicy **O**ptimization

General Introduction

In computationally intensive applications like deep learning and scientific computing, efficient code optimization is crucial for achieving high performance and scalability. Traditionally, compilers have depended on manual optimization techniques, requiring extensive expertise and domain knowledge. Although effective, these manual methods are time-consuming and struggle to scale with the growing complexity of modern applications.

Recent advancements have seen a paradigm shift towards machine learning-based solutions for automatic code optimization. Reinforcement learning (RL), in particular, has emerged as a promising approach, enabling compilers to learn optimal optimization strategies through interaction with the code execution environment. This transition from manual to automated optimization reduces the dependency on expert knowledge and adapts dynamically to diverse optimization scenarios.

The Tiramisu compiler provides a flexible framework for scheduling and optimizing computations. It relies on expert-defined scheduling parameters or employs tree search methods that leverage cost models to estimate the speedups of various code optimizations. Recognizing the potential of RL, Tiramisu considers RL-based automatic code optimization as a promising research direction, aiming to enhance its capability to autonomously discover and apply efficient optimization strategies.

However, training an RL agent to effectively optimize code poses significant challenges, including large data requirements, extensive training time, and the necessity for effective code representation. To address these challenges, this manuscript proposes a novel approach that combines an RL agent with a pretrained autoencoder. This combination facilitates the embedding of Tiramisu programs' access matrices within their Abstract Syntax Tree (AST) representations, enabling more efficient and meaningful learning.

The organization of this manuscript is as follows: Chapter 1 provides background information on compilers, with a focus on Tiramisu, as well as an overview of automatic code optimization techniques, reinforcement learning, and autoencoder-based representation learning. Chapter 2 details the design of the proposed system, including the architecture of the RL agent and the pretrained autoencoder, and their integration within the overall framework. Chapter 3 delves into the implementation specifics, covering the tools used, the Tiramisu Python API, the technical aspects of the RL and autoencoder systems, and the training workflow. Finally, Chapter 4 presents the evaluation of the system under various embedding configurations compared to the default representation, offering insights into the effectiveness of the approach and suggesting potential directions for future research.

Chapter 1

Background

1.1 Code Optimization

1.1.1 Compilers

Definition

A compiler is a software program that translates code written in a high-level programming language into a low-level target language (Aho & Lam, 2022), such as machine code, assembly, or bytecode, making it executable by a computer.

Compilation Passes

The compilation process involves a sequence of steps, called compilation passes (Nystrom, 2021), that progressively transform the code from its original form into efficient and executable instructions. These passes include:

- **Lexical Analysis:** Converts raw source code into tokens by identifying keywords, identifiers, and symbols.
- **Syntactic Analysis:** Constructs a parse tree from tokens to validate grammatical structure according to the language's syntax.
- **Semantic Analysis:** Ensures program correctness by checking types and variable scopes within the parse tree.
- **Intermediate Representation Generation:** Transforms the code into an intermediate format that simplifies optimization and target code generation.
- **Code Optimization:** Applies transformations to improve code efficiency and performance while maintaining its semantics.

- **Machine Code Generation:** Converts the optimized intermediate representation into executable machine code for the target architecture.

In the scope of this manuscript, we will focus on the **Code Optimization** pass, which will be detailed followingly.

1.1.2 Code Optimization

Definition

Code optimization in compilers aims to enhance the performance of programs by applying different transformations on its code. This process involves various techniques to improve execution speed and reduce resource consumption.

Loop Optimizations

One of the most important code optimization techniques are **Loop Optimizations**. In scientific computing, a significant portion of execution time is often spent within loops (Liu et al., 2019). Consequently, implementing effective loop optimizations can lead to significant performance improvements.

A loop optimization is a transformation that is applied on loops, in order to improve their execution speed and reduce their resource consumption. Some of the most commonly used loop optimizations include:

- **Parallelization:** Splits loop iterations to run in parallel on multiple processors or threads. The concurrent execution reduces execution time.
- **Reversal:** Reverses loop iteration order for better memory access patterns.
- **Unrolling:** Expands loop iterations to reduce loop control overhead and increase instruction-level parallelism.
- **Tiling:** Divides loops into smaller chunks to improve cache usage.
- **Skewing:** Adjusts iteration boundaries to parallelize nested loops. It aligns iterations to facilitate parallel execution and prepares loops for tiling.
- **Interchange:** Reorders nested loops to improve data locality and cache efficiency.

1.1.3 Automatic Code Optimization

Definitions

Schedule: A schedule is a sequence of code transformations, applied in a specific order with set parameter values, that optimizes code at the compiler level, enhancing execution

speed while preserving program semantics.

Autoscheduling: Manually finding an optimal schedule is complex and time-consuming, as it involves selecting, ordering, and tuning transformations. To address this, modern compilers use an Autoscheduler, a module that automates this process and efficiently applies code optimization.

Optimization Elements

Automatic Code Optimization (ACO) typically includes three main components:

- **Search Space:** The search space is the set of all possible transformations and optimizations the compiler can apply to the code.
- **Search Strategy:** The search strategy is an algorithm that explores the search space to construct candidate schedules by selecting transformations, setting their order, and defining parameters. Algorithms vary from exhaustive search to advanced methods like Tree Search and Reinforcement Learning.
- **Evaluation Method:** To identify the best schedule, the compiler evaluates candidates based on the accelerations they yield. To get the acceleration value, we can either run each candidate and get the precise execution times and calculate the acceleration, or use machine-learning-based cost models to predict the acceleration of a candidate without executing it.

Examples

Some of the most famous Autoschedulers in literature are:

- Tiramisu Autoscheduler (Baghdadi et al., 2021)
- Halide Autoscheduler (Adams et al., 2019)
- Milepost GCC (Fursin et al., 2011)
- AutoTVM (Chen et al., 2018)
- DeepTune (Cummins et al., 2017)
- Ithemal (Mendis et al., 2019)

1.1.4 Main Study Case: Tiramisu

Definition

Tiramisu (Baghdadi et al., 2019) is a polyhedral compiler designed to optimize high-performance computing applications. It aims to improve the execution of complex computations, particularly in fields like scientific computing and machine learning.

It provides a high-level Domain Specific Language (DSL) that allows developers to express their computations in a clear and concise manner. By separating the description of the algorithm from the optimization details, Tiramisu enables users to focus on the algorithm's implementation while the compiler takes care of the low-level optimizations. To achieve that, a Tiramisu program contains two parts, one in which we define the algorithm, and the other to define an optimization schedule.

Tiramisu Autoscheduler

Given the variety of optimization strategies available in the Tiramisu Application Programming Interface (API), manually defining the best schedule in the second part of writing a Tiramisu program can be a time-consuming task. Tiramisu's autoscheduler (Baghdadi et al., 2021) addresses this challenge by automatically generating an optimization schedule based on the program's characteristics. The autoscheduler operates through the following components:

- **Search Space:** it encompasses all the available code optimizations that can be applied to a Tiramisu program along with their parameters. This includes various loop transformations, such as tiling, skewing and parallelization.
- **Search Algorithm:** to navigate the search space, the autoscheduler uses tree search algorithms, namely **Monte Carlo Tree Search** and **Beam Search**.
- **Evaluation Method:** the most impactful component in the autoscheduler is its deep learning based cost model. Given features representing the unoptimized code and a schedule, this model predicts the speedup that we would get from applying this schedule on the program, without relying on execution.

1.2 Reinforcement Learning

1.2.1 Definition

Reinforcement Learning (RL) is a branch of Machine Learning (ML) in which agents learn to make sequences of decisions by interacting with an environment, guided by the reward hypothesis (Sutton, 2018). According to this hypothesis, the goal of any problem can be framed as maximizing some cumulative reward, creating a general framework

that applies to a range of applications such as playing games—like chess and Go (Silver et al., 2016)—robotics (Kober et al., 2013), and finance (Deng et al., 2016). Unlike supervised learning, which learns from labeled datasets with static examples, RL learns in an interactive manner through trial and error. This interaction enables the agent to adapt to dynamic environments, making RL a powerful optimization technique for problems that evolve over time, where the optimal solution may depend on sequential decision-making and cumulative learning.

1.2.2 Reinforcement Learning Framework

In RL, the interaction between an agent and its environment is formalized using the concept of a Markov Decision Process (MDP) (Sutton, 2018) as shown in Figure 1.1. An MDP is defined by the tuple $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$. Following is a breakdown for each component and additional related definitions:

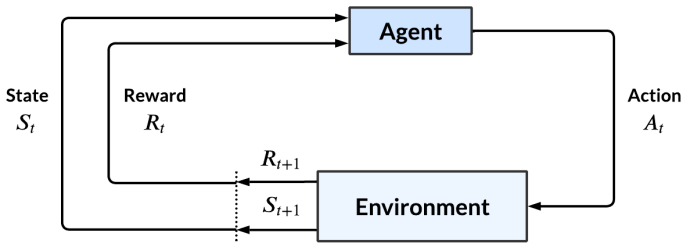

Figure 1.1: The Agent–Environment interaction in RL (Sutton, 2018)

- **Environment**: The environment encompasses everything external to the agent. It provides states and rewards in response to the agent's actions, modeling the dynamics of the system within which the agent operates.
- **Agent**: The agent is the learner or decision-maker that interacts with the environment. Its goal is to learn a policy that maximizes cumulative rewards by selecting appropriate actions based on the current state.
- **State** ($s \in \mathcal{S}$): A state represents the environment's configuration at a specific time step. It is an element of the state space $\mathcal{S}$, which includes all possible states the environment can be in.
- **Action** ($a \in \mathcal{A}$): An action is a decision made by the agent that influences the next state of the environment. Actions are elements of the action space $\mathcal{A}$, representing all possible actions the agent can take.
- **Transition Probability** ($P(s' \mid s, a)$): The transition probability function defines the probability of moving to state s' from state s after taking action a. It captures the stochastic dynamics of the environment and is given by:

$$P(s' \mid s, a) = \Pr\{S_{t+1} = s' \mid S_t = s,\ A_t = a\}.$$

- **Reward** ($R(s,a)$): The reward function assigns a scalar feedback signal to the agent's action in a given state. It quantifies the immediate benefit of taking action a in state s and is defined as:

$$R(s,a) = \mathbb{E}\left[R_{t+1} \mid S_t = s,\ A_t = a\right].$$

- **Return** (G_t): The return is the total accumulated reward from time step t onwards, discounted by a factor $\gamma \in [0,1]$:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}.$$

  The discount factor γ determines the importance of future rewards.

- **Policy** ($\pi(a \mid s)$): A policy defines the agent's behavior by specifying the probability of selecting action a when in state s:

$$\pi(a \mid s) = \Pr\{A_t = a \mid S_t = s\}.$$

  The objective is to find an optimal policy π^* that maximizes the expected cumulative reward.

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\pi}\left[G_t \mid S_t = s\right], \quad \forall s \in \mathcal{S}$$

- **Value Function** ($V^{\pi}(s)$): The value function represents the expected return when starting from state s and following policy π thereafter:

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[G_t \mid S_t = s\right]$$

  where G_t is the return at time t.

- **Action-Value Function** ($Q^{\pi}(s,a)$): The action-value function gives the expected return after taking action a in state s and then following policy π:

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}\left[G_t \mid S_t = s,\ A_t = a\right]$$

- **Entropy** ($S[\pi](s)$): Entropy measures the randomness of the policy at state s, encouraging exploration by promoting a spread over actions:

$$S[\pi](s) = -\sum_{a \in \mathcal{A}} \pi(a \mid s) \log \pi(a \mid s).$$

1.2.3 Policy-Based Optimization

Policy-based optimization methods in RL focus on directly optimizing the policy function without the need to estimate value functions explicitly. These methods parameterize the policy $\pi_\theta(a \mid s)$ with parameters θ and aim to find the optimal parameters that maximize the expected return.

The core idea is to adjust the policy parameters in the direction that increases the expected cumulative reward, using gradient ascent techniques. This approach relies on the **Policy Gradient Theorem** (Sutton et al., 1999), which provides a formal way to compute the gradient of the expected return with respect to the policy parameters.

To optimize the expected return $J(\theta)$ with respect to the policy parameters θ, we define:

$$J(\theta) = \mathbb{E}_{\pi_\theta}\left[G_t\right],$$

The **Policy Gradient Theorem** states that the gradient of the expected return is:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a_t \mid s_t)\, Q^{\pi_\theta}(s_t, a_t)\right],$$

In practice, the gradient is estimated using sampled trajectories:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t^i \mid s_t^i)\, G_t^i,$$

where N is the number of trajectories, and G_t^i is the return from time step t in trajectory i.

By following the gradient of the expected return, the agent can iteratively improve its policy to maximize long-term rewards. This direct optimization of the policy is particularly useful in environments with continuous or high-dimensional action spaces.

Several effective policy gradient methods have been developed, such as **REINFORCE** (Zhang et al., 2021), **Trust Region Policy Optimization (TRPO)** (Schulman, 2015), and **Proximal Policy Optimization (PPO)** (Schulman et al., 2017). These algorithms build upon the policy gradient framework and introduce techniques to improve stability and efficiency in learning. For example, REINFORCE uses Monte Carlo estimates of the return, while TRPO and PPO introduce constraints or surrogate objectives to ensure reliable policy updates.

1.2.4 Proximal Policy Optimization

PPO is a policy gradient method designed to improve the stability and efficiency of reinforcement learning algorithms (Schulman et al., 2017). PPO achieves this by modifying the objective function to prevent large updates to the policy, ensuring that each update is within a safe, *"proximal"* region relative to the current policy.

Components of the PPO Loss Function

- **Probability Ratio $r_t(\theta)$:**

PPO measures the change between the new policy π_θ and the old policy $\pi_{\theta_{\text{old}}}$ using the probability ratio:

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)},$$

- **Advantage Function Estimator $\hat{A}_t$:**

  The advantage function $\hat{A}_t$ estimates how much better an action is compared to the average, guiding the policy update:

  $$\hat{A}_t = Q^{\pi_\theta}(s_t, a_t) - V^{\pi_\theta}(s_t),$$

  In practice, $\hat{A}_t$ is often computed using methods like Generalized Advantage Estimation (GAE) (Schulman et al., 2015) for more accurate and stable estimates.

- **Clipping Mechanism with Hyperparameter ε:**

  The hyperparameter ε controls the range within which the policy is allowed to change during an update. It is a small positive value (e.g., $\varepsilon = 0.2$) that defines the clipping range in the objective function.

- **Clipped Surrogate Objective Function $L^{\mathbf{CLIP}}(\theta)$:**

  The PPO objective function incorporates the probability ratio, advantage estimator, and clipping mechanism:

  $$L^{\text{CLIP}}(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta)\hat{A}_t,\ \text{clip}\left(r_t(\theta),\ 1-\varepsilon,\ 1+\varepsilon\right)\hat{A}_t \right) \right].$$

  By maximizing $L^{\text{CLIP}}(\theta)$, PPO effectively updates the policy to improve performance without risking large, destabilizing changes.

- **Value Function Loss $L^{\mathbf{VF}}(\theta)$:**

  To ensure accurate estimation of state values, PPO includes a value function loss term. This is typically computed as the mean squared error between the predicted values $V^{\pi_\theta}(s_t)$ and the observed returns R_t:

  $$L^{\text{VF}}(\theta) = \mathbb{E}_t \left[(V^{\pi_\theta}(s_t) - R_t)^2 \right].$$

- **Entropy Bonus $L^{\mathbf{S}}(\theta)$:**

  To encourage exploration, PPO incorporates an entropy bonus. This term promotes policy entropy, preventing the policy from becoming too deterministic too quickly:

  $$L^{\text{S}}(\theta) = \mathbb{E}_t \left[S[\pi_\theta](s_t) \right].$$

The total loss L is formulated as:

$$L = L^{\text{CLIP}}(\theta) - c_1 L^{\text{VF}}(\theta) + c_2 L^{\text{S}}(\theta),$$

where c_1 and c_2 are coefficients that balance the contributions of the value loss and entropy regularization, respectively. By maximizing the total loss L, PPO effectively updates both the policy and value functions to enhance performance while maintaining stability and encouraging sufficient exploration.

1.3 Autoencoder-Based Representation Learning

1.3.1 Autoencoders

Autoencoders are a type of unsupervised neural network designed to learn efficient data representations by compressing input data into a lower-dimensional latent space and then reconstructing it (Hinton & Salakhutdinov, 2006). The architecture consists of an encoder that maps the input data to a compressed encoding and a decoder that reconstructs the input from this encoding.

This process forces the network to capture robust and high quality features of the data, effectively performing dimensionality reduction and feature extraction. Autoencoders are beneficial because they can learn nonlinear relationships in the data, making them more powerful than linear methods like Principal Component Analysis (PCA). They are widely used for tasks such as data compression (Hinton & Salakhutdinov, 2006), noise reduction (Vincent et al., 2008), and anomaly detection (Sakurada & Yairi, 2014).

1.3.2 Autoencoder Pretraining

Leveraging pretrained autoencoders can enhance learning efficiency by providing rich, high-level feature representations that simplify subsequent tasks (Bengio et al., 2013). By encoding data into compressed forms, models can focus on the most relevant aspects of the input, potentially addressing challenges associated with high-dimensional data. This approach can improve performance and convergence speed in various machine learning tasks, as the compressed representations capture essential features while reducing noise and redundancy. Integrating representation learning techniques like autoencoders into machine learning workflows offers significant advantages in terms of efficiency and effectiveness.

Chapter 2

Design

2.1 Motivation

Compilers play a crucial role in translating source code into executable programs, but their effectiveness depends on the optimizations they support (Bacon et al., 1994). Optimizing code involves a vast and complex search space of parameters and configurations that are traditionally determined manually by experts, making the process lengthy and complex.

RL offers a promising approach to automate this process by enabling agents to autonomously explore large search spaces and identify optimal sequences of transformations. Previous attempts to apply RL within the Tiramisu compiler framework have shown potential in enhancing optimization processes (Lamouri & Merad, 2023). These efforts aim to achieve the following objectives:

- **Acceleration of Program Execution Times:** The primary goal is to optimize programs by reducing their execution times. By employing a reinforcement learning approach within Tiramisu, we aim to surpass existing methods and achieve better performance.
- **Automation of Optimization:** Developing learning agents capable of exploring the optimization search space in Tiramisu allows us to automate the compiler optimization process. This automation saves time and reduces reliance on human experts.
- **Generalization of Knowledge:** Creating optimization agents that can generalize knowledge acquired from a set of programs to new, unseen programs within Tiramisu results in more robust optimizations applicable to a wide range of use cases.

However, implementing RL in compiler optimization presents several challenges. Our work is motivated by the need to address these challenges and enhance the effectiveness of RL in this domain:

- **Improving Generalization:** RL agents may struggle to generalize effectively due to the complexity and variability of code representations in Tiramisu. By integrating pretrained autoencoders, we aim to capture the most relevant features of the compiler's state, helping the agent generalize better to new programs.
- **Optimizing Memory Requirements:** High-dimensional state spaces in Tiramisu can lead to increased memory consumption. Dimensionality reduction through autoencoder-based representation reduces the size of the state representation, optimizing memory usage.
- **Accelerating Convergence Speed:** Large state spaces can slow down the convergence rate during training. Providing the RL agent with a more informative and compact state representation can speed up the learning process and improve convergence rates.

2.2 Reinforcement Learning System

2.2.1 Environment

The environment in our reinforcement learning system is the Tiramisu compiler itself. This compiler processes C++ code and executes it, serving as the platform where the agent interacts and applies optimizations. A Python API facilitates this interaction by handling the application of code transformations and performing legality checks to ensure that any modifications maintain the correctness of the code. Additionally, the environment provides essential services such as compiling the code, running it, and collecting performance metrics. This setup enables the reinforcement learning agent to interact seamlessly with the compiler, apply optimization actions, and receive feedback based on the resulting performance improvements.

2.2.2 State Representation

Abstract Syntax Tree

In the RL system described by (Lamouri & Merad, 2023), the state representation is based on the Abstract Syntax Tree (AST) of the Tiramisu program. The AST provides a hierarchical, tree-like structure that represents the nested loops and computations within the program as illustrated in Figure 2.1. Each node in the AST corresponds to either an iteration (loop) or a computation, which facilitates the processing and analysis of the program's data.

Tiramisu (Baghdadi et al., 2019) allows for the extraction of numerical information from this AST, such as the upper and lower bounds of loops, memory access matrices, mathematical computations performed in each computation node, and the transformations that have already been applied. These data form a numerical representation of the program, transforming the textual code into a format that the reinforcement learning

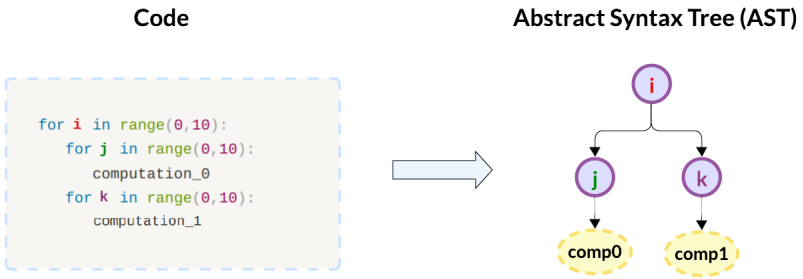

Figure 2.1: An Example of Transforming Code into an AST (Lamouri & Merad, 2023)

agent can effectively process. This structured and detailed state representation enables the agent to make informed decisions when selecting optimization actions.

Node Feature Representation

Each node in the AST is represented by a fixed-size feature vector to accommodate both iterator nodes and computation nodes Figure 2.2. The node type is specified by a boolean value in the first element of the vector (0 for iterator, 1 for computation). To differentiate between node types while maintaining a uniform vector size, different features are placed at opposite ends of the vector with appropriate padding.

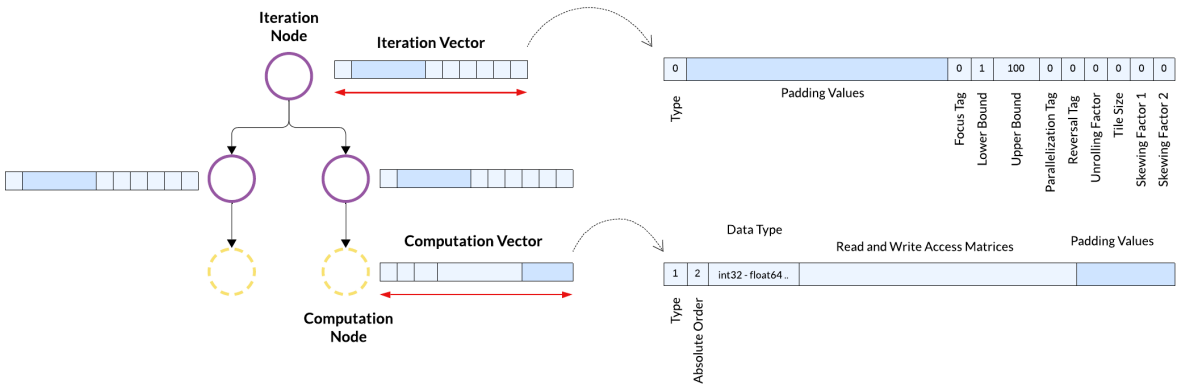

Figure 2.2: AST with Feature Vectors for Iterator and Computation Nodes

For **computation nodes**, features such as absolute order, data type, and read and write access matrices are positioned at the *start* of the vector, followed by padding. Conversely, for **iterator nodes**, padding is added at the start, and the features are placed at the *end* of the vector. These features include a *Focus Tag* indicating if the iterator is targeted for optimization, and transformation tags like *Parallelization* and *Reversal* that represent whether those actions have been applied.

By stacking these feature vectors, we form the input node representation matrix X, which serves as the structured input for the RL agent.

2.2.3 Action Space

The agent's action space consists of 56 discrete actions, each representing a specific loop transformation with predefined parameters applied to **the targeted branch** in the AST to optimize code performance. These actions are:

- **Interchange** $I(i,\ i+1)$: Swapping loop levels i and $i+1$, where $i \in \{0,\ 1,\ 2,\ 3\}$.
- **Reversal** $R(i)$: Reversing the iteration order of loop level i, where $i \in \{0,\ 1,\ 2,\ 3,\ 4\}$.
- **Skewing** $S(i,\ i+1)$: Adjusting the iteration space of loops i and $i+1$, where $i \in \{0,\ 1,\ 2\}$.
- **Parallelization** $P(i)$: Parallelizing the loop level i, where $i \in \{0,\ 1\}$.
- **Tiling** $T(i,\ i+1,\ x,\ y)$: Breaking down loops i and $i+1$ into smaller blocks with tile sizes x and y, where $i \in \{0,\ 1,\ 2,\ 3\}$. The tile sizes include:
  - *Square Tiles*: $x = y \in \{32,\ 64,\ 128\}$
  - *Rectangular Tiles*: $(x,\ y) \in \{(32,\ 64),\ (32,\ 128),\ (64,\ 32),\ (64,\ 128),\ (128,\ 32),\ (128,\ 64)\}$
- **Unrolling** $U(f)$: Unrolling the innermost loop by factor f, where $f \in \{2,\ 4,\ 8,\ 16,\ 32\}$.
- **Next**: Progressing to the next branch in the AST. This action allows the agent to traverse the program's structure and apply optimizations across different code regions.

By combining these transformations with their specified parameters, the agent explores a comprehensive range of optimization strategies, totaling 56 actions. The diversity in loop levels, tile sizes, and unrolling factors allows the agent to test various configurations that may lead to significant performance gains. The inclusion of the **Next** action ensures that the agent can navigate through the AST, optimizing the entire program comprehensively.

2.2.4 Reward Function

To effectively guide the reinforcement learning agent in optimizing compiler performance, the reward function employs a logarithmic transformation of the speedup metric. Specifically, the reward is defined as $R = \log_4(S)$, where S represents the speedup achieved by the transformed code relative to the previous state, i.e., $S = \frac{\text{current\_speedup}}{\text{previous\_speedup}}$. This logarithmic scaling converts the multiplicative objective of speedup into an additive reward structure, which is more suitable for reinforcement learning algorithms that aim to maximize cumulative rewards. Additionally, using the logarithm reduces the variance of the reward signal, promoting more stable and efficient learning. The scaling is intuitive:

speedup values greater than one yield positive rewards, encouraging beneficial optimizations, while values less than one result in negative rewards, discouraging actions that degrade performance (Lamouri & Merad, 2023).

The implemented reward function incorporates several key components to ensure effective training of the agent. When a transformation action is deemed legal, the agent receives a reward based on the logarithm of the achieved speedup. If the action involves switching to a different branch in the AST, a predefined penalty is applied to discourage excessive branching. Similarly, illegal actions incur a fixed penalty to prevent the agent from selecting transformations that violate compiler constraints. Formally, the reward can be expressed as:

$$R = \begin{cases} \log_4(S) & \text{if the action is legal and not a branch switch,} \\ \log_4(0.9) & \text{if the action is a branch switch,} \\ \log_4(0.9) & \text{if the action is illegal.} \end{cases}$$

This structured approach guides the agent to perform actions that consistently enhance performance while penalizing actions that either lead to inefficiencies or violate compiler rules. By balancing positive rewards for beneficial transformations and negative rewards for undesirable actions, the reward function effectively directs the agent towards optimizing the compiler's performance in a stable and efficient manner.

2.2.5 Agent

The agent in the RL system adopts an actor-critic architecture (Sutton, 2018). The architecture leverages Graph Neural Networks (GNN) (Wu et al., 2020) as its backbone to effectively process the structured state representations derived from the AST as illustrated in Figure 2.3. Specifically, we utilize Graph Attention Networks (GAT) (Brody et al., 2021) to capture the intricate relationships and dependencies between different nodes in the AST. The architecture comprises two primary components: a policy network and a value network. The policy network, denoted as $\pi(s)$, is responsible for selecting actions based on the current state, while the value network, denoted as $V(s)$, estimates the expected return from a given state. Both networks share a common set of initial layers that process the graph-structured input, ensuring that the agent can efficiently extract and utilize relevant features for decision-making.

Our agent employs the PPO (Schulman et al., 2017) algorithm to update its policy and value networks, ensuring stable and efficient learning. The PPO loss function is designed to balance the trade-off between exploration and exploitation by incorporating a clipped surrogate objective, value loss, and entropy regularization. Mathematically, the loss is expressed as:

$$\mathcal{L}(\theta, \varphi) = \mathcal{L}_{\text{Clip}}(\theta) - c_1 \mathcal{L}_{\text{Value}}(\varphi) + c_2 \mathcal{S}(\pi_\theta)$$

where:

- $\mathcal{L}_{\text{Clip}}(\theta)$ is the clipped surrogate objective, which constrains the policy updates by clipping the probability ratio $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ to the interval $[1-\varepsilon, 1+\varepsilon]$. This

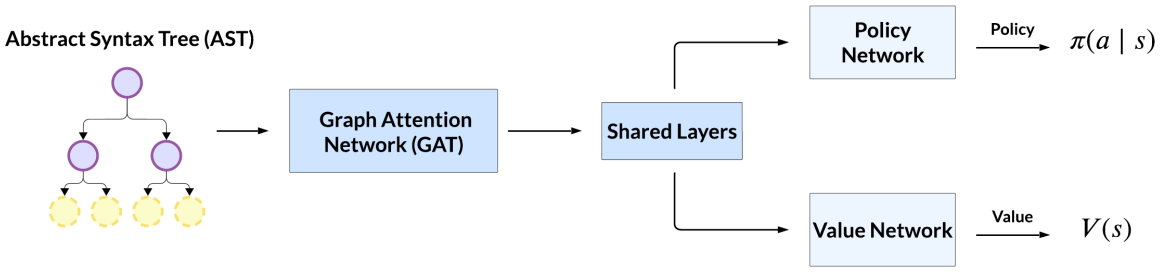

Figure 2.3: Actor-Critic Agent Architecture with AST Input and Graph Attention Network

prevents large, destabilizing policy updates by ensuring that the new policy does not deviate excessively from the old policy.

- c_1 and c_2 are hyperparameters that weight the importance of the value loss and entropy regularization, respectively.
- $\mathcal{L}_{\text{Value}}(\varphi)$ is the value loss, which measures the mean squared error between the predicted value $V_\varphi(s_t)$ and the actual return R_t, guiding the value network to produce accurate value estimates:

$$\mathcal{L}_{\text{Value}}(\varphi) = \frac{1}{N} \sum_{t=1}^{N} \left(V_\varphi(s_t) - R_t\right)^2$$

- $\mathcal{S}(\pi_\theta)$ is the entropy bonus, which encourages exploration by maximizing the entropy of the policy distribution:

$$\mathcal{S}(\pi_\theta) = -\sum_{a} \pi_\theta(a \mid s_t) \log \pi_\theta(a \mid s_t)$$

By optimizing this loss function using gradient ascent, the PPO algorithm ensures that the agent incrementally improves its policy while maintaining stability and promoting effective exploration. This structured approach enables the agent to efficiently learn optimal loop transformations, enhancing the overall performance and scalability of the compiler optimization process.

2.3 Autoencoder-Based Pretraining

2.3.1 Access Matrices

In the RL system[1] for the Tiramisu compiler, the state representation incorporates read and write access matrices derived from the AST. Each computational vector within the

[1]Lamouri Djamel Rassem, and Kourta Smail. 2024. **Reinforcement Learning Agent for Automatic Code Optimization in Tiramisu Compiler**. GitHub repository: https://github.com/Tiramisu-Compiler/gnn_rl

AST contains these matrices, which encode how data is accessed during program execution. As shown in Figure 2.4, the read access matrix captures the indices used to load data from input arrays, while the write access matrix represents the indices for storing results into output arrays. Each matrix is structured with dimensions corresponding to the number of loop iterators and array dimensions, effectively mapping the relationship between loop levels and data access patterns. These matrices play a crucial role in the state representation by providing detailed information about data dependencies and access patterns, enabling the RL agent to make informed decisions about loop transformations and optimizations.

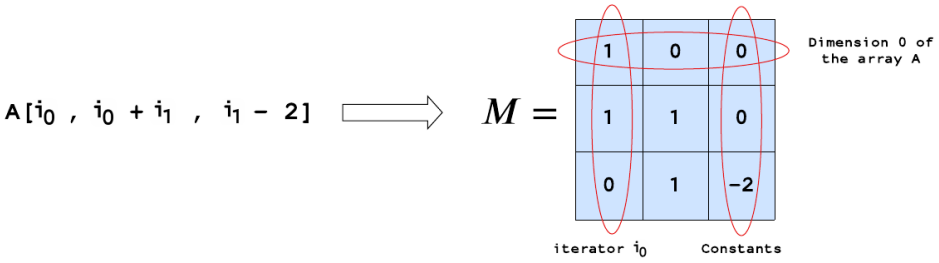

Figure 2.4: Example of an access matrix (Baghdadi et al., 2021)

However, the high dimensionality of the read and write access matrices presents significant challenges for the RL agent. The read and write access matrices occupy 704 out of the total 718 dimensions of the computational vector, making them a dominant component of the state space. This extensive dimensionality can lead to increased memory requirements and slower convergence rates during training, as the agent must process and learn from a vast amount of information.

2.3.2 Autoencoder Model

To address the high dimensionality of the read and write access matrices in the state representation, an autoencoder-based pretraining approach is employed[2]. The autoencoder model, as presented in Figure 2.5, consists of three primary components: an encoder, a regularization layer, and a decoder. The encoder is implemented using a bidirectional Long Short-Term Memory (LSTM) network (Hochreiter, 1997), denoted as Encoder, which processes the high-dimensional access matrices by capturing both forward and backward dependencies in the data. This bidirectional architecture ensures that the model comprehensively understands the intricate patterns within the access matrices.

Following the encoding phase, a dropout regularization layer with a dropout rate of $\gamma = 0.05$ is applied to prevent overfitting and promote generalization. The decoder, denoted as Decoder, is a linear layer that reconstructs the original access matrices from

[2]Chunting Liu. 2024. **Pretraining Read and Write Access Matrices Autoencoder in Tiramisu**. GitHub repository: https://github.com/Tiramisu-Compiler/cost_model_pretrain/tree/main/pretrain_access_matrices

the compressed latent representations generated by the encoder. Specifically, the decoder maps the concatenated hidden states from the bidirectional LSTM back to the original input size. Mathematically, the reconstruction can be expressed as:

$$\text{Reconstructed\_Output} = \text{Decoder}(\text{Dropout}(\text{Encoder}(X)))$$

where X represents the input access matrices. By learning to reconstruct the input data, the autoencoder effectively compresses the high-dimensional access matrices into a lower-dimensional latent space, preserving essential information while reducing computational complexity. This dimensionality reduction facilitates more efficient processing by the reinforcement learning agent, enhancing memory usage and accelerating the convergence of the learning algorithm.

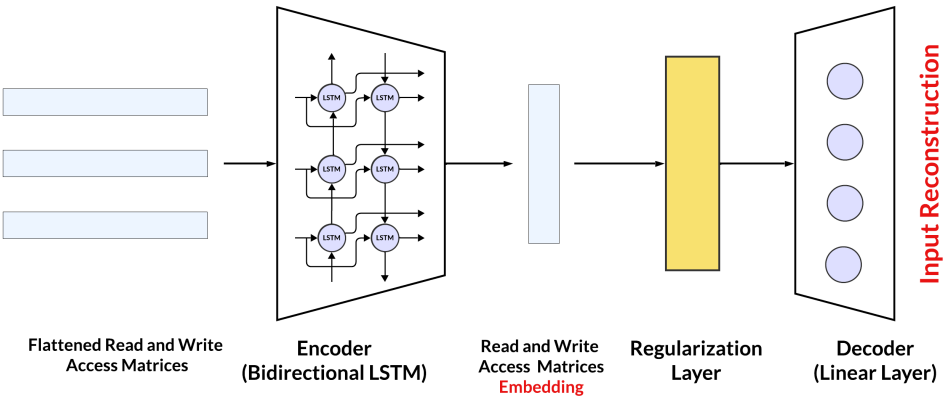

Figure 2.5: Pretrained Autoencoder Architecture

2.3.3 State Representation Update

To enhance the RL agent's performance, we modify the state representation by embedding the high-dimensional read and write access matrices using the encoder from the autoencoder model as shown in Figure 2.6. This embedding reduces the state dimensionality from 703 to a lower dimension, thereby alleviating memory constraints and improving training efficiency. By transforming the extensive access matrices into compact latent embeddings, we streamline the input space, enabling the agent to process and learn from the state more effectively.

We explore and evaluate various embedding types to identify the most effective configuration for capturing the essential features of the access matrices. The embedding types and their corresponding dimensions are as follows:

- **Final Hidden State** h_n (80 dimensions): Captures the last hidden state from the bidirectional LSTM, representing the sequential information processed by the encoder.
- **Final Cell State** c_n (80 dimensions): Represents the last cell state from the bidirectional LSTM, encapsulating the long-term dependencies learned during encoding.

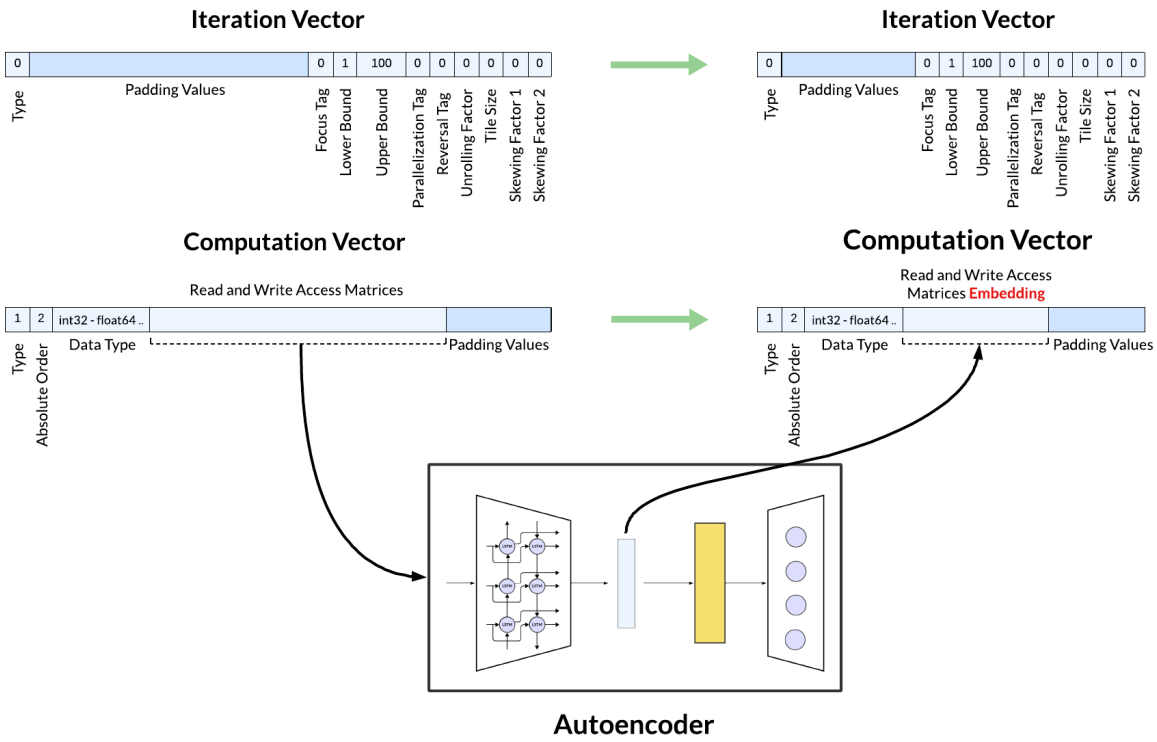

Figure 2.6: State Representation update using the Pretrained Autoencoder

- **Concatenated Final Hidden and Cell States** $h_n + c_n$ (160 dimensions): Combines both hidden and cell states to provide a comprehensive representation of the encoded information.
- **Flattened Output** (640 dimensions): Reshapes the entire encoder output into a single vector, retaining all encoded information.
- **Mean Pooling Output** (16 dimensions): Computes the mean across all time steps of the encoder's output, summarizing the overall information.
- **Max Pooling Output** (16 dimensions): Extracts the maximum value across all time steps, highlighting the most significant features.
- **Concatenated Max Pooling Output and Final Hidden State** (96 dimensions): Merges the max-pooled output with the final hidden state to balance feature highlighting and sequential information.

Each embedding type offers a unique perspective on the encoded access matrices, balancing dimensionality reduction and information preservation. By systematically testing these embedding strategies, we identify the best configuration that maintains important data dependencies and access patterns while minimizing computational overhead. This approach ensures that the RL agent operates on a compact and informative state space, facilitating more efficient learning and better generalization in loop transformation and optimization tasks.

Chapter 3

Implementation

3.1 Tools

The implementation of the RL system and the pretrained autoencoder for compiler optimization leverages several key tools and libraries that facilitate the development, training, and deployment of the agent. These tools ensure efficiency, scalability, and reproducibility throughout the project. The primary tools utilized in this project include PyTorch, torch_geometric, Ray, and MLflow.

- **PyTorch** (Paszke et al., 2019) is an open-source deep learning framework widely used for developing and training neural networks. It provides dynamic computational graphs and automatic differentiation, which are essential for implementing complex models such as GAT and LSTM networks. In our project, PyTorch serves as the foundational library for building the agent's neural network architectures, enabling efficient model training and experimentation.
- **torch_geometric** (Fey & Lenssen, 2019) is an extension library for PyTorch designed to handle graph-structured data. It offers a wide range of utilities and pre-implemented layers for building Graph Neural Networks, including GAT. We utilize torch_geometric to implement the GATv2Conv layers in our agent, allowing us to effectively process the AST and capture the intricate relationships between its nodes.
- **Ray** (Moritz et al., 2018) is a distributed computing framework that simplifies the scaling of machine learning and reinforcement learning workloads. It provides tools for parallel and distributed execution, enabling efficient utilization of computational resources. In our system, Ray is employed to parallelize the training and evaluation processes of the reinforcement learning agent, significantly reducing training time and enhancing scalability.
- **MLflow** (Zaharia et al., 2018) is an open-source platform for managing the end-to-end machine learning lifecycle, including experimentation, reproducibility, and deployment. It offers capabilities for tracking experiments, packaging code into reproducible runs, and deploying machine learning models. We integrate MLflow

into our project to systematically track the performance of different embedding strategies, manage experiment metadata, and facilitate the reproducibility of our results.

3.2 Tiramisu Python API

The Tiramisu Python API (Lamouri & Merad, 2023) serves as the crucial interface between the RL environment and the Tiramisu compiler, enabling seamless communication and interaction. Its primary role is to translate the actions generated by the RL agent into concrete transformations applied to the Tiramisu functions, thereby facilitating automated loop optimizations. The API is architected into two main layers: **Core** and **Scheduler**, each comprising specialized services that handle distinct aspects of the optimization process.

3.2.1 Core Layer

The Core layer is responsible for managing data processing and direct interaction with the Tiramisu compiler. It encompasses two primary services:

- **Tiramisu Service**: This service handles the retrieval of program annotations, which include various features derived from the Tiramisu functions. These annotations provide essential insights into the program's structure and data dependencies, equipping the RL agent with the necessary information to make informed optimization decisions.
- **Compiling Service**: The Compiling service manages the compilation process, including the verification of action legality, the compilation of annotated functions, and the execution of transformed code. Utilizing Python's `subprocess` package, this service ensures that transformations are correctly applied and their legality is assessed, thereby maintaining the integrity and performance of the optimized code.

3.2.2 Schedule Layer

The Schedule layer orchestrates the application of transformations and oversees the progression of optimization tasks. It comprises three key services:

- **Legality Service**: This service is dedicated to verifying the legality of proposed actions by the RL agent. It employs both logical checks and compiler-based verification to determine whether a transformation can be safely and effectively applied, preventing invalid or inefficient optimizations.
- **Schedule Service**: The Schedule service coordinates the creation and traversal of AST branches, applying actions as directed by the RL agent. It maintains the state

of the program's optimization schedule, managing multiple branches and ensuring that transformations are systematically integrated into the AST.

- **Prediction Service**: Responsible for estimating the acceleration resulting from applied transformations, the Prediction service utilizes real execution time measurements to evaluate the effectiveness of optimizations. Unlike traditional cost models, this project relies on actual performance data to provide accurate feedback to the RL agent, enhancing the reliability and accuracy of the optimization process.

Through its well-defined structure and specialized services, the Tiramisu Python API effectively bridges the RL agent and the Tiramisu compiler. It facilitates the automated application of loop transformations, ensuring that optimizations are both effective and legally sound. This seamless integration is pivotal for achieving automated compiler optimizations, enabling the RL agent to iteratively improve the performance of Tiramisu-generated code.

3.3 Pretrained Autoencoder

As outlined in the **Design** chapter, we leverage a pretrained autoencoder to manage the high dimensionality of the read and write access matrices in the RL agent's state representation. The autoencoder model comprises three primary components: an encoder, a regularization layer, and a decoder. The architecture is designed to capture both forward and backward dependencies in the data using a bidirectional LSTM network. Table 3.1 below provides a detailed overview of each component:

Component	**Layer Type**	**Parameters**
Encoder	Bidirectional LSTM	Input Size: 44 Hidden Size: 20 Number of Layers: 2 Max Sequence Length: 16
Regularization	Dropout	Dropout Rate: 0.05
Decoder	Linear	Input Features: $20 \times 2 \times 16 = 640$ Output Features: $44 \times 16 = 704$

Table 3.1: Pretrained Autoencoder Architecture

3.4 Reinforcement Learning System

The RL system orchestrates the automated optimization of Tiramisu-generated code through a structured framework comprising three primary components: **Graph Utils**, **Rollout Worker**, and the **Policy-Value Neural Network**. Each component plays a vital role in ensuring efficient interaction between the RL agent and the compiler, facilitating effective learning and optimization.

3.4.1 Graph Utils

Graph Utils encompasses a suite of methods designed to create and manipulate the Tiramisu program's AST representation. This includes generating iterator and computation vectors, as well as applying various transformations to AST nodes. A key feature of Graph Utils is the integration of the highlighted **pretrained autoencoder**, which embeds the high-dimensional access matrices within the computational vectors. This embedding process reduces the complexity of the state representation, enabling the RL agent to process and learn from the AST more effectively.

3.4.2 Rollout Worker

The **Rollout Worker** is responsible for collecting trajectories that the RL agent uses for training. It operates by loading Tiramisu functions designated for optimization and sequentially applying exploratory actions proposed by the RL agent. Throughout each episode, the Rollout Worker records the sequence of states, actions, and rewards, thereby generating the data necessary for the agent's learning process. This component ensures that the agent receives diverse and comprehensive experiences, which are crucial for developing robust optimization strategies.

3.4.3 Policy-Value Neural Network

The **Policy-Value Neural Network** is a central component of the RL system, embodying an actor-critic architecture enhanced by a GAT backbone as clearly detailed in the design chapter. This network is responsible for processing the AST representations of Tiramisu programs and generating informed decisions for loop transformations. The architecture is meticulously designed to capture intricate relationships within the AST while providing robust policy and value estimations for the RL agent. The network is structured into three primary components:

- The **backbone** of the network employs two GATv2Conv (Brody et al., 2021) layers to process the AST's graph structure. These layers capture the intricate dependencies and interactions between nodes, enabling the network to understand the program's structural nuances. Each GATv2Conv layer is followed by a linear transformation and a SELU (Klambauer et al., 2017) activation function to introduce non-linearity and facilitate feature refinement. To capture global graph information, we perform global mean and max pooling after each linear transformation over each batch, concatenate the results into summary vectors, and then combine these from both layers to create a comprehensive feature vector for further processing.
- The **Value Network** ($V_\varphi(s)$) serves as the critic, estimating the expected return from the current state. Similar to the Policy Network, it comprises a series of linear layers with SELU activations. The final output layer produces a single scalar value representing the estimated return. This estimation provides a baseline that helps

in evaluating the effectiveness of the actions taken by the Policy Network, thereby stabilizing the training process.

- The **Policy Network** ($\pi_\theta(s)$) is an actor component that takes the refined features from the backbone and outputs a probability distribution over the possible actions. It consists of a sequence of linear layers interleaved with SELU activations, culminating in an output layer that corresponds to the number of actionable transformations (56 in this case). The network is designed to guide the RL agent in selecting the most promising loop transformations based on the current state.

The architecture of the model is detailed in Table 3.2. Each layer is designed to progressively extract and refine features from the AST, culminating in the generation of actionable policies and accurate value predictions. The 'input_size' is dynamically determined based on whether and how access matrices are embedded using the pretrained autoencoder.

Component	**Layer**	**Parameters**
Backbone	GATv2Conv 1	Heads: 4 `input_size` $\rightarrow 64 \times 4$
	Linear 1	$256 \rightarrow 64$
	Pooling & Concatenation 1	Mean + Max Pooling (batch_size, 64) $\rightarrow 128$
	GATv2Conv 2	Heads: 4 $64 \rightarrow 64 \times 4$
	Linear 2	$256 \rightarrow 64$
	Pooling & Concatenation 2	Mean + Max Pooling (batch_size, 64) $\rightarrow 128$
	Concatenation (x1 + x2)	$128 + 128 \rightarrow 256$
	Convs Summarizer	$256 \rightarrow 128$
	Shared Linear 1	$128 \rightarrow 64$
Policy Network	3 Linear Layers	$64 \rightarrow 64 \rightarrow 64 \rightarrow 56$
Value Network	3 Linear Layers	$64 \rightarrow 64 \rightarrow 64 \rightarrow 1$

Table 3.2: Policy-Value Neural Network Architecture

3.5 Training Workflow

Our training workflow leverages the PPO algorithm to efficiently train the Policy-Value Neural Network. By employing parallel trajectory collection with multiple workers, we

enhance data efficiency and exploration. We utilize GAE to compute returns and advantages, improving learning stability. The training process emphasizes a balance between policy improvement and exploration by using a clipped surrogate objective for the policy loss and incorporating entropy regularization. Dynamic adjustment of hyperparameters and strategic updates of model parameters via gradient ascent contribute to effective convergence. This approach ensures robust policy learning and improved performance over time.

Algorithm 1: Training Workflow of the Policy-Value Neural Network using PPO

Input: Configuration parameters (e.g., hyperparameters, embedding settings, training device, dataset path)
Output: Trained Policy-Value Neural Network π_θ and V_θ

Initialize **model** parameters θ
Initialize **Adam** optimizer with learning rate α
Log training **configuration**
for *each update step* $u = 1$ **to** $N_{updates}$ **do**
  Adjust entropy coefficient c_2 based on **schedule**
  for *each worker* $w = 1$ **to** $N_{workers}$ ***in parallel*** **do**
    Collect **trajectory** τ_w using current policy π_θ
    Store **states** s_t, **actions** a_t, **rewards** r_t, **log probabilities** $\log \pi_\theta(a_t|s_t)$, **values** $V_\theta(s_t)$, **entropies** $H(\pi_\theta(s_t))$
  end
  Combine trajectories $\{\tau_w\}$ into **batch** $\mathcal{B}$
  Compute **returns** R_t and **advantages** A_t using GAE
  for *each epoch* $e = 1$ **to** N_{epochs} **do**
    Shuffle batch $\mathcal{B}$ and divide into **minibatches** $\{\mathcal{B}_i\}$
    for *each minibatch* $\mathcal{B}_i$ **do**
      Compute **policy** loss L_{policy} using clipped surrogate objective
      Compute **value function** loss L_{value}
      Compute **entropy** loss L_{entropy} using entropy bonus
      Compute **total** loss $L = L_{\text{policy}} - c_1 L_{\text{value}} + c_2 L_{\text{entropy}}$
      Update model parameters θ using **gradient ascent** to **maximize** L
    end
  end
  Log training **metrics** (e.g., loss values, rewards, entropies)
  if *performance improved* **then**
    Save current model parameters θ
  end
end

Chapter 4

Evaluation

4.1 Experimental Setup

4.1.1 Experimental Environment

Component	**Specification**
Operating System	Ubuntu 24.04.1 LTS
Kernel Version	6.8.0-48-generic
CPU	Intel(R) Core(TM) i5-10500H CPU @ 2.50GHz
CPU Cores	6 physical cores, 12 threads
CPU Cache	L1d: 192 KiB x6, L1i: 192 KiB x6, L2: 1.5 MiB x6, L3: 12 MiB x1
GPU	NVIDIA GeForce GTX 1650 Mobile / Max-Q
GPU VRAM	4 GB
GPU Driver	560.35.03
CUDA Version	11.6
PyTorch Version	1.13.0
Memory (RAM)	24 GB
Disk	2 x NVMe SSDs, Total Size: 256 GB each

Table 4.1: Environment Specifications for Training and Evaluation of the RL Agent

4.1.2 Dataset

The dataset employed for training and evaluating the RL agent comprises a small set of C++ functions optimized using the Tiramisu compiler. These functions are crafted

to represent a balanced range of computational patterns and optimization challenges prevalent in scientific computing and image processing applications. By encompassing various optimization scenarios such as loop transformations, parallelism enhancements, and memory access optimizations, the dataset provides a comprehensive benchmark to assess the RL agent's effectiveness in improving computational performance and efficiency.

- **function_mvt_MEDIUM**: Performs matrix-vector multiplication.
- **function_doitgen_MEDIUM**: Generates synthetic data through matrix operations.
- **function_blur_MEDIUM**: Applies a blurring filter to images.
- **function_seidel2d_MEDIUM**: Implements a 2D Seidel stencil computation.
- **function_jacobi2d_MEDIUM**: Executes a 2D Jacobi stencil computation.
- **function_heat2d_MEDIUM**: Performs 2D heat diffusion computations.
- **function_heat3d_MEDIUM**: Simulates 3D heat diffusion using finite difference methods.
- **function_cvtcolor_MEDIUM**: Converts image color spaces.

4.1.3 Training Hyperparameters

Hyperparameter	**Value**
Number of Updates (N_{updates})	500
Number of Epochs (N_{epochs})	4
Clipping Epsilon (ε)	0.3
Batch Size	512
Mini-Batch Size	64
Learning Rate (α)	0.0001
(Adam) Weight Decay	0.0001
Discount Factor (γ)	0.99
(GAE) Lambda (λ)	0.95
Value Coefficient (c_1)	2
(Decaying) Entropy Coefficient (c_2)	$0.5 \rightarrow 0$

Table 4.2: Training Hyperparameters of RL Agent

4.2 Results

4.2.1 Agent Training

The RL agent was trained using a pretrained autoencoder under various configurations to evaluate the impact of different embedding types on optimization performance. Specifically, the training configurations included the **default computational vector setting** without any embedding and several embedding strategies: Final Hidden State **(HS)**, Final Cell State **(CS)**, Concatenated Final Hidden and Cell States **(C(HS, CS))**, Flattened Output **(FO)**, Mean Pooling Output **(MnPO)**, Max Pooling Output **(MPO)**, and Concatenated Max Pooling Output with Final Hidden State **(C(MPO, HS))**. It is noteworthy that the Final Cell State **(CS)** and Mean Pooling Output **(MnPO)** embeddings were excluded from the final analysis due to their suboptimal performance in preliminary experiments.

Figure 4.1 illustrates the training progression of the RL agent across the selected embedding configurations over 500 update steps. All methods began with initial mean rewards below -0.4, indicating a period of deceleration in terms of code optimization. As training advanced, the mean rewards increased, reflecting an overall acceleration in performance. Most embedding configurations, alongside the default method, exhibited comparable convergence speeds, with mean rewards stabilizing above 0.5 after approximately 300 update steps except for the Flattened Output **(FO)** embedding, which continued to improve and plateaued around 400 update steps.

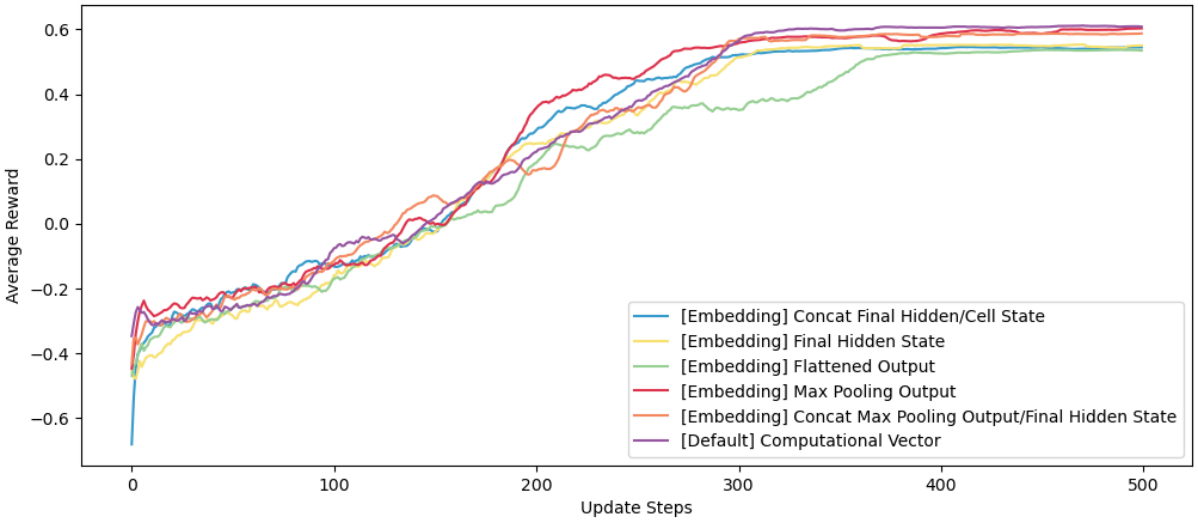

Figure 4.1: RL Agent's Training Progress: Default vs. Different Embedding Configurations

During the intermediate training phase, within [200, 300] update steps, certain embedding types, such as Max Pooling Output **(MPO)** and Concatenated Max Pooling Output with Final Hidden State **(C(MPO, HS))**, achieved higher mean rewards compared to others. This suggests that these embeddings may facilitate more effective learning dynamics in the early stages of training.

To provide a more detailed comparison of the convergence behavior, Figure 4.2 presents a zoomed-in view of the last 100 training steps. In this phase, the **default configuration** (no embedding) attained mean rewards comparable to the top-performing embeddings, reaching approximately 0.6. Other embedding methods converged to mean rewards in the range of 0.53 to 0.55, demonstrating that while the default method slightly outperformed the embeddings overall, the embedding strategies achieved competitive performance.

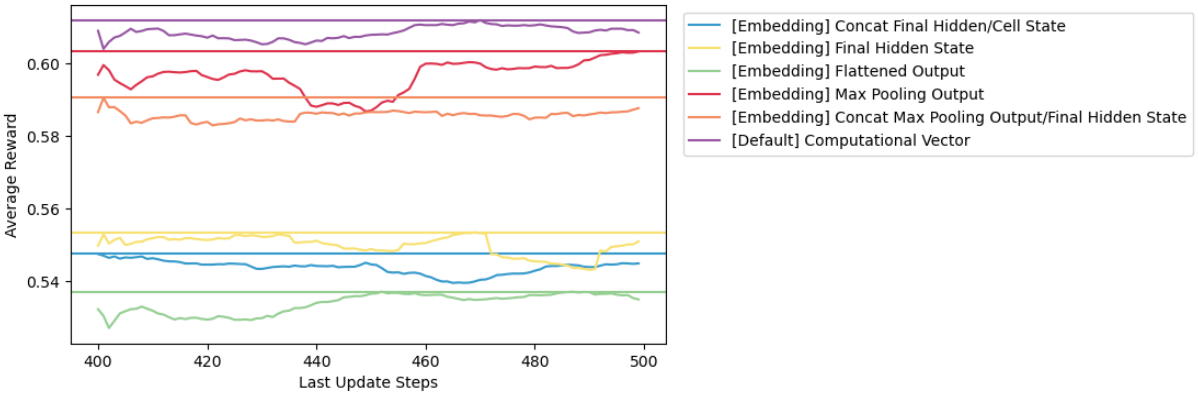

Figure 4.2: RL Agent's Training Progress on the last 100 Update Steps

Furthermore, Figure 4.3 depicts the maximum average reward achieved by each method during the training process. The bar plot clearly indicates that the default method achieved the highest mean reward, closely followed by Max Pooling Output **(MPO)** and Concatenated Max Pooling Output with Final Hidden State **(C(MPO, HS))**. The remaining embedding configurations attained slightly lower but still comparable mean rewards, underscoring their effectiveness in the optimization task.

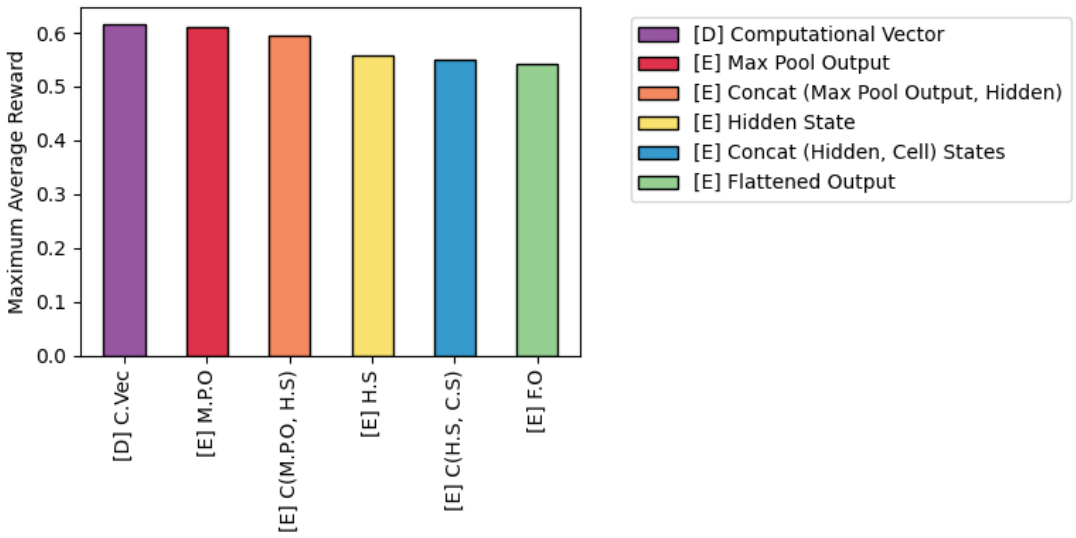

Figure 4.3: Maximum Training Average Reward achieved by RL Agent

In summary, the embedding methods successfully facilitated the RL agent in handling loop transformation optimizations, achieving acceleration in performance as evidenced by the increase in mean rewards. Although the **default method** without embed-

dings demonstrated superior overall performance, the embedding configurations exhibited promising potential by reaching comparable performance levels with slight marginal differences. Additionally, certain embeddings showed faster initial learning rates, suggesting that embedding strategies can enhance training efficiency. These findings highlight the viability of incorporating embedding techniques to improve the RL agent's optimization capabilities within the Tiramisu compiler framework.

4.2.2 Benchmark Evaluation

To evaluate the effectiveness of different embedding configurations, we employed the same set of functions used during the training phase as benchmarks. This decision was driven by hardware constraints, allowing us to conduct experiments efficiently on a smaller scale while focusing on the comparative effectiveness of pretrained embeddings.

The evaluation focused on comparing the performance of the **default representation** (without embedding) against various embedding configurations. Specifically, we examined whether embedding methods could surpass the default approach in terms of speedup across different benchmarks. The key observations from the benchmark evaluations, as illustrated in Figure 4.4, are as follows:

- **Superior Performance with Embeddings:** Several embedding configurations outperformed the default method in specific benchmarks:
  - Concatenated Final Hidden and Cell States **(C(HS, CS))** achieved higher speedup in **Seidel2D**.
  - Final Hidden State **(HS)** and Concatenated Max Pooling Output and Final Hidden State **(C(MPO, HS))** demonstrated better performance in **Blur**.
  - Concatenated Final Hidden and Cell States **(C(HS, CS))**, Flattened Output **(FO)**, and Concatenated Max Pooling Output and Final Hidden State **(C(MPO, HS))** showed enhanced speedup in **Heat2D**.
  - Final Hidden State **(HS)** surpassed the default in **Heat3D**.
  - Concatenated Final Hidden and Cell States **(C(HS, CS))** outperformed the default in **MVT**.
- **Comparable Performance:** In benchmarks where the **default method** remained dominant, certain embedding configurations still achieved comparable speedups in **Jacobi2D** and **DOITGEN**
- **Underperformance**: All configurations failed to accelerate **CvtColor**, consistently achieving a speedup of 1, indicating no performance improvement over the original code.

To quantify the overall performance, we computed the **geometric mean** of the speedups across all benchmarks, as depicted in Figure 4.5. These results are also summarized numerically in Table 4.3, providing both visual and quantitative comparisons.

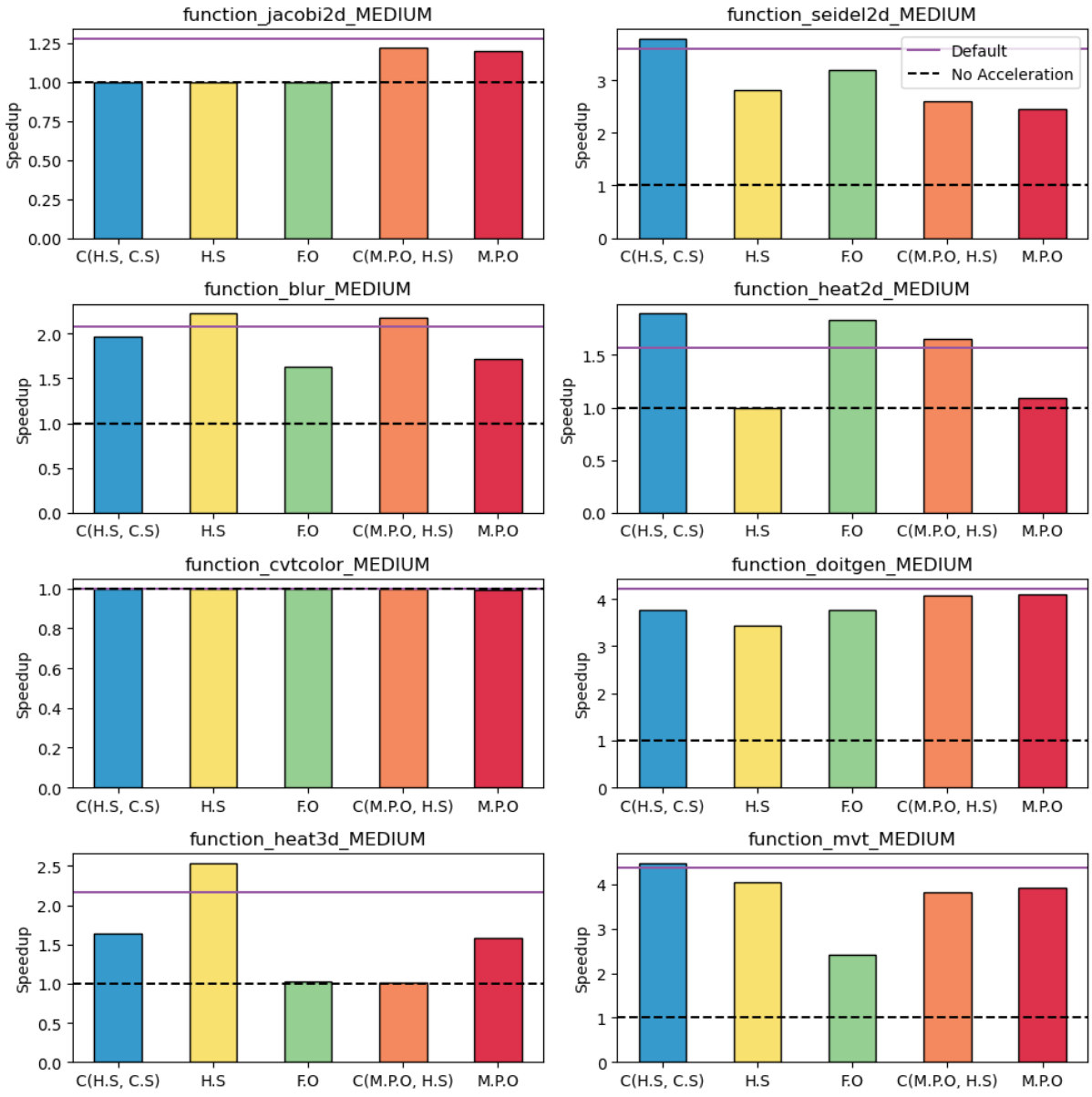

Figure 4.4: Speedup Evaluation of RL Agent on Benchmark Functions

Despite the default method achieving the highest overall speedup, embedding configurations demonstrated their potential by surpassing the default in specific benchmarks and maintaining competitive performance in others. The close geometric mean values suggest that embedding methods offer a balanced trade-off between performance and input representation efficiency.

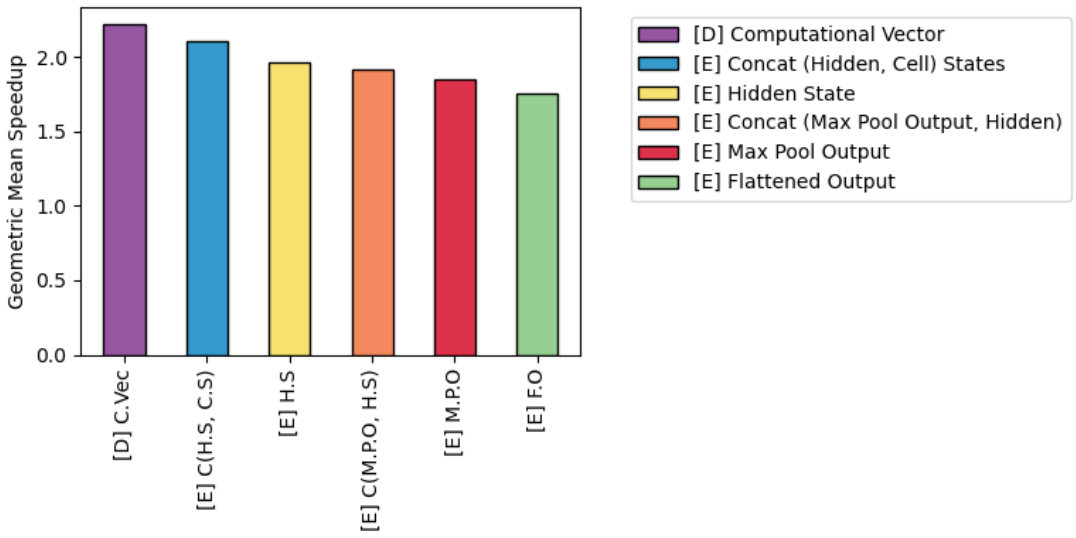

Figure 4.5: Geometric Mean Speedup of RL Agent Across Benchmark Functions

Embedding Configuration	**Geometric Mean Speedup**
Default (No Embedding)	**2.22**
C(HS, CS)	2.11
HS	1.96
C(MPO, HS)	1.92
MPO	1.85
FO	1.75

Table 4.3: Geometric Mean Speedup of RL Agent Across Benchmark Functions

This is further supported by Figure 4.6, which compares the dimensionality of the AST vectors for the default and various embedding configurations. The embeddings significantly reduce the input vector size, enhancing computational efficiency without substantially compromising speedup performance.

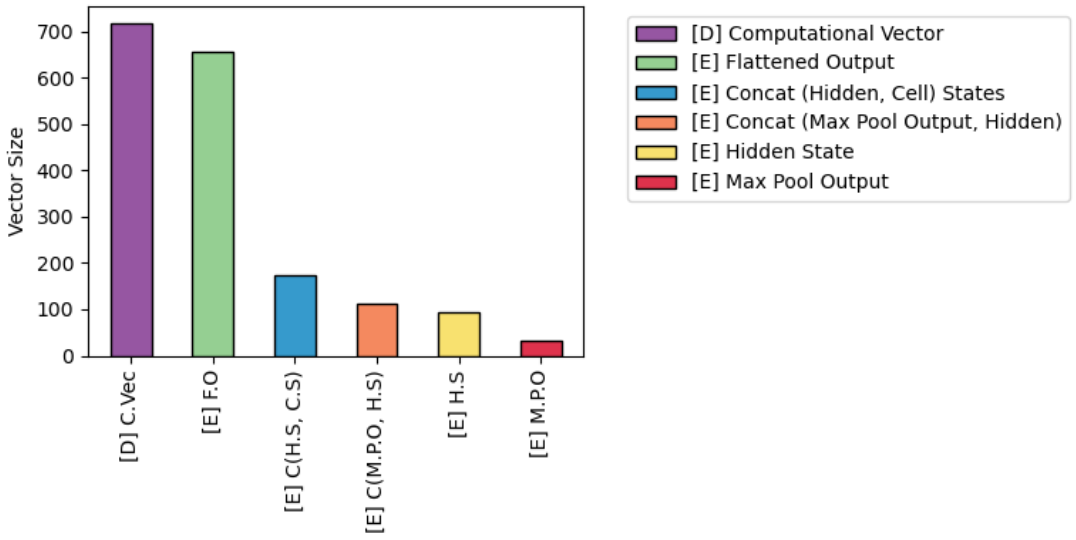

Figure 4.6: Comparison of AST Vector Dimensionality: Default vs. Different Embedding Configurations

In conclusion, while the default method remains the top performer overall, embedding configurations exhibit considerable promise by excelling in specific benchmarks and offering efficiency gains through reduced input dimensionality. These findings highlight the potential of embedding strategies to augment the RL agent's optimization capabilities within the Tiramisu compiler framework by providing more informative and compact input representations. These findings encourage further exploration and optimization using larger datasets and more advanced computational resources.

General Conclusion

This manuscript introduced a novel system that integrates an automatic code optimization RL agent with a pretrained autoencoder to embed access matrices within the AST representations of Tiramisu programs. The primary objective was to investigate the effectiveness of pretraining in the context of ACO using RL within the Tiramisu compiler.

Chapter 1 provided a comprehensive background on compilers, particularly the Tiramisu compiler. It explored ACO techniques and detailed the state-of-the-art PPO algorithm used in RL. Additionally, the chapter highlighted the benefits of utilizing autoencoder-based representation learning, demonstrating how such approaches can enhance the learning process by reducing input dimensionality and capturing high-quality features.

In Chapter 2, the design of the proposed system was elaborated, encompassing all critical components. This included the RL agent, which interacts with the Tiramisu environment by utilizing AST-based state representations and receiving rewards based on execution speedup. The chapter also detailed the architecture of the policy-value neural network leveraging the PPO algorithm and the pretrained autoencoder that facilitates effective embedding of access matrices.

Chapter 3 focused on the implementation specifics, outlining the technical tools and frameworks employed for training and monitoring the system. It described the Tiramisu Python API that serves as the interface for the RL agent, as well as the intricate details of both the RL agent and autoencoder architectures. The training workflow, which employs the PPO algorithm, was clearly elaborated by the RL agent's training algorithm.

Finally, Chapter 4 presented a comprehensive evaluation of the system under different embedding configurations compared to the default representation. The results demonstrated that certain embedding methods facilitated faster convergence during training and outperformed the default approach in specific benchmarks. Although the default method achieved the highest training average reward and overall geometric mean speedup on the benchmark, the embedding configurations exhibited competitive performance with significantly reduced input dimensionality, underscoring their potential effectiveness.

Despite the promising results, the study was constrained by hardware limitations, which restricted the size of the training and evaluation dataset and the scalability of experiments. Future work aims to address that by employing larger and more diverse datasets along with more powerful computational resources. Such advancements are expected to further validate the viability and effectiveness of the pretrained autoencoder approach and enhance the RL agent's capability to optimize code at a larger scale.

Bibliography

Adams, A., Ma, K., Anderson, L., Baghdadi, R., Li, T.-M., Gharbi, M., Steiner, B., Johnson, S., Fatahalian, K., Durand, F., & Ragan-Kelley, J. (2019). Learning to optimize halide with tree search and random programs. *ACM Trans. Graph.*, *38*(4). https://doi.org/10.1145/3306346.3322967

Aho, A., & Lam, M. S. (2022). *Compilers principles techniques & tools*. Pearson.

Bacon, D. F., Graham, S. L., & Sharp, O. J. (1994). Compiler transformations for high-performance computing. *ACM Computing Surveys (CSUR)*, *26*(4), 345–420.

Baghdadi, R., Merouani, M., Leghettas, M.-H., Abdous, K., Arbaoui, T., Benatchba, K., et al. (2021). A deep learning based cost model for automatic code optimization. *Proceedings of Machine Learning and Systems*, *3*, 181–193.

Baghdadi, R., Ray, J., Romdhane, M. B., Del Sozzo, E., Akkas, A., Zhang, Y., Suriana, P., Kamil, S., & Amarasinghe, S. (2019). Tiramisu: a polyhedral compiler for expressing fast and portable code. *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, 193–205.

Bengio, Y., Courville, A., & Vincent, P. (2013). Representation learning: a review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, *35*(8), 1798–1828.

Brody, S., Alon, U., & Yahav, E. (2021). How attentive are graph attention networks? *arXiv preprint arXiv:2105.14491*.

Chen, T., Zheng, L., Yan, E., Jiang, Z., Moreau, T., Ceze, L., Guestrin, C., & Krishnamurthy, A. (2018). Learning to optimize tensor programs. *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, 3393–3404.

Cummins, C., Petoumenos, P., Wang, Z., & Leather, H. (2017). End-to-end deep learning of optimization heuristics. *PACT*.

Deng, Y., Bao, F., Kong, Y., Ren, Z., & Dai, Q. (2016). Deep direct reinforcement learning for financial signal representation and trading. *IEEE transactions on neural networks and learning systems*, *28*(3), 653–664.

Fey, M., & Lenssen, J. E. (2019). Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*.

Fursin, G., Kashnikov, Y., Memon, A. W., Chamski, Z., Temam, O., Namolaru, M., Yom-Tov, E., Mendelson, B., Zaks, A., Courtois, E., Bodin, F., Barnard, P., Ashton, E., Bonilla, E., Thomson, J., Williams, C. K. I., & O'Boyle, M. (2011). Milepost gcc: machine learning enabled self-tuning compiler. *International Journal of Parallel Programming*, *39*(3), 296–327. https://doi.org/10.1007/s10766-010-0161-2

Hinton, G. E., & Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. *science*, *313*(5786), 504–507.

Hochreiter, S. (1997). Long short-term memory. *Neural Computation MIT-Press*.

Klambauer, G., Unterthiner, T., Mayr, A., & Hochreiter, S. (2017). Self-normalizing neural networks. *Advances in neural information processing systems*, *30*.

Kober, J., Bagnell, J. A., & Peters, J. (2013). Reinforcement learning in robotics: a survey. *The International Journal of Robotics Research*, *32*(11), 1238–1274.

Lamouri, D., & Merad, D. (2023). Utilisation d'apprentissage par renforcement pour l'optimisation automatique de code dans tiramisu. *École nationale Supérieure d'Informatique (ESI ex-INI)*.

Liu, S., Cui, Y.-Z., Zou, N.-J., Zhu, W.-H., Zhang, D., & Wu, W.-G. (2019). Revisiting the parallel strategy for doacross loops. *Journal of Computer Science and Technology*, *34*, 456–475.

Mendis, C., Renda, A., Amarasinghe, D., & Carbin, M. (2019, June). Ithemal: accurate, portable and fast basic block throughput estimation using deep neural networks. In K. Chaudhuri & R. Salakhutdinov (Eds.), *Proceedings of the 36th international conference on machine learning* (pp. 4505–4515, Vol. 97). PMLR. https://proceedings.mlr.press/v97/mendis19a.html

Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M. I., et al. (2018). Ray: a distributed framework for emerging {ai} applications. *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, 561–577.

Nystrom, R. (2021). *Crafting interpreters*. Genever Benning.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. (2019). Pytorch: an imperative style, high-performance deep learning library. *Advances in neural information processing systems*, *32*.

Sakurada, M., & Yairi, T. (2014). Anomaly detection using autoencoders with nonlinear dimensionality reduction. *Proceedings of the MLSDA 2014 2nd workshop on machine learning for sensory data analysis*, 4–11.

Schulman, J. (2015). Trust region policy optimization. *arXiv preprint arXiv:1502.05477*.

Schulman, J., Moritz, P., Levine, S., Jordan, M., & Abbeel, P. (2015). High-dimensional continuous control using generalized advantage estimation. *arXiv:1506.02438*.

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

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *nature*, *529*(7587), 484–489.

Sutton, R. S. (2018). Reinforcement learning: an introduction. *A Bradford Book*.

Sutton, R. S., McAllester, D., Singh, S., & Mansour, Y. (1999). Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, *12*.

Vincent, P., Larochelle, H., Bengio, Y., & Manzagol, P.-A. (2008). Extracting and composing robust features with denoising autoencoders. *Proceedings of the 25th international conference on Machine learning*, 1096–1103.

Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., & Philip, S. Y. (2020). A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, *32*(1), 4–24.

Zaharia, M., Chen, A., Davidson, A., Ghodsi, A., Hong, S. A., Konwinski, A., Murching, S., Nykodym, T., Ogilvie, P., Parkhe, M., et al. (2018). Accelerating the machine learning lifecycle with mlflow. *IEEE Data Eng. Bull.*, *41*(4), 39–45.

Zhang, J., Kim, J., O'Donoghue, B., & Boyd, S. (2021). Sample efficient reinforcement learning with reinforce. *Proceedings of the AAAI conference on artificial intelligence*, *35*(12), 10887–10895.