Department of Electronic and Telecommunication Engineering
University of Moratuwa, Sri Lanka

Unified Framework for Performance Prediction of Tensor Programs in Domain Specific Languages

SUPERVISORS:

Dr. Subodha Charles

CO-SUPERVISORS:

Prof. Charith Mendis
Mr. Chamika Sudusinghe

MEMBERS:	GROUP NO 28
Kowrisaan S.	200312L
Jathurshan S.	200239T
Dharmasri N.T.S.	200126U
Methsarani H.E.N.	200395P

Final Year Project Mid Review Report
submitted in partial fulfillment of the requirements for the course module EN4203

January 28, 2025

Abstract

Domain Specific Language (DSL)s such as Halide and Tiramisu are used for optimizing tensor computations. They rely on cost models to select the best schedule for program execution. However, the diverse methodologies in their cost models make it difficult to predict performance accurately across different systems. Current solutions lack a unified approach, leading to inefficiencies and the constant need for new adaptations.

In response, we propose a unified framework for performance prediction of tensor programs in these DSLs by developing a common cost model. Our solution leverages machine learning to create a shared representation of program schedules enabling a robust and adaptable performance prediction mechanism. By analyzing and integrating the similarities in code transformations and scheduling techniques of these DSLs, our model will enhance the efficiency of code generation. This adaptive framework will significantly improve the optimization process, making it resilient to evolving challenges in DSL performance prediction..

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Domain Specific Languages (DSLs) are types of programming languages used in a particular application domain to solve a specific problem. These DSLs offer high-level abstraction compared to general programming languages so that programmers do not need to consider low-level hardware details. Image processing, machine learning, and scientific computing are a few examples of tensor programs that involve multi-dimensional computations implemented using DSLs. Achieving optimal performance in tensor computation heavily depends on developing cost models that predict the computational cost of a program and guide the compiler optimizations.

1.1 Motivation

In existing approaches, distinct cost models are specifically designed for each DSL, leading to inefficiencies and inconsistencies across different systems. These methods prevent generalizing and consistently applying optimizations across various DSLs, increasing the complexity of maintaining, adapting, and extending these cost models.

Optimizing tensor programs involves significant manual effort to account for language-specific constructs, hardware-specific parameters, and diverse scheduling strategies. This complexity increases development time and causes inefficiencies in performance tuning, making it difficult to achieve optimal results across various configurations.

To address these limitations, a unified framework for the DSLs is required. This framework will provide a generalized approach to predict the performance across different DSLs and hardware architectures by eliminating redundancy and integrating optimization processes. This framework will reduce the manual effort required for tuning, and enable automated, generalizable performance prediction. This approach would allow developers to focus on higher-level tasks rather than spending time adapting different DSLs or hardware. In existing approaches, distinct cost models are specifically designed for each DSL, leading to inefficiencies and inconsistencies across different systems. These methods prevent generalizing and consistently applying optimizations across various DSLs, increasing the complexity of maintaining, adapting, and extending these cost models.

Optimizing tensor programs involves significant manual effort to account for language-specific constructs, hardware-specific parameters, and diverse scheduling strategies. This complexity increases development time and causes inefficiencies in performance tuning, making it difficult to achieve optimal results across various configurations.

To address these limitations, a unified framework for the DSLs is required. This framework will provide a generalized approach to predict the performance across different DSLs and hardware architectures by eliminating redundancy and integrating optimization processes. This framework will reduce the manual effort required for tuning, and enable automated, generalizable performance prediction. This approach would allow developers to focus on higher-level tasks rather than spending time adapting different DSLs or hardware.

1.2 Problem Statement

The problem that needs to be addressed is the lack of a unified approach to performance prediction for tensor programs in DSLs. Because existing cost models are separate for respective DSLs which is not scalable and sustainable.

The need of distinct cost models for each DSL and hardware architecture, reduces the scalability of optimizing performance, making it difficult to extend these approaches to novel environments as each distinct cost model requires a significant amount of data for training, which is impractical and resource intensive when applying to novel DSLs or hardware architectures. Also, performance tuning for different hardware platforms is an inefficient process as it requires more manual effort which needs to be repeated for each DSL.

These limitations highlight a need for a unified framework that can predict performance and optimize tensor programs which would be efficient and automate optimization across various DSLs and hardware platforms, reducing redundancy, and increasing scalability and adaptability in performance tuning.

1.3 Primary Objectives

The primary objectives of the project are:

1. **Develop a Common Representation**

   Create a common representation for consistent analysis, optimization, and performance prediction across different DSLs.

2. **Integrate Deep Learning Techniques**

   Explore and implement state-of-the-art deep learning models to capture complex dependencies within program schedules and predict performance.

3. **Develop an Advanced Cost Model**

   Develop a robust and adaptable cost model that will predict the runtime and performance for various schedules, architectures, and DSLs accurately.

4. **Enhance Autotuning Algorithms**

Improve current autotuning methods by incorporating innovative techniques for search space pruning and feature extraction ensuring high accuracy and adaptability.

1.4 Project Scope

This project aims to deliver a Unified Framework for optimizing tensor programs that can be applied across various DSLs. The unified framework is designed to ensure efficiency and flexibility across various DSLs and hardware architectures by improving performance prediction and optimization. This unified framework includes:

1. **Common Representation**

   A common representation that facilitates consistent analysis, optimization, and performance evaluation across different DSLs.

2. **Cost Model**

   An advanced machine learning-based cost model which is capable of accurately predicting runtime and performance for different schedules, DSLs and hardware architectures.

3. **Autotuning Approach**

   Enhanced autotuning algorithms with improvements in performance through intelligent pruning strategies, fine-tuning, and dynamic adaptation to hardware.

By addressing fragmentation in cost models and hardware-specific constraints, this project will deliver a comprehensive solution for developers to design, optimize, and fine-tune tensor programs seamlessly across different DSLs and hardware platforms.

1.5 Novelty and the Uniqueness of the Project

This project introduces a novel and unified approach to address the critical challenge of performance prediction and optimization for tensor programs across various DSLs. Unlike existing fragmented solutions, which rely on separate cost models and hardware-specific tuning techniques, this project designs a universal framework that transcends the boundaries of individual DSLs and different hardware architectures. The integration of deep learning, innovative optimization techniques, and hardware adaptability distinguishes this framework as a state-of-the-art tool for advancing performance prediction and optimization in the domain of tensor programs.

1.6 Beneficiaries and Potential Applications

This unified framework which optimizes tensor programs will benefit researchers, developers, and industries that rely on high-performance tensor computations.

Researchers and developers who work in high-performance computing, scientific simulations, and machine learning will benefit from this unified framework, as this simplifies the optimization process, eliminates the need for manual tuning, and enhances the efficiency of tensor programs. This will accelerate research timelines and enable the development of cutting-edge applications.

Industries such as healthcare, finance, autonomous systems, and media will also benefit from this unified framework. In healthcare, optimized tensor computations can improve medical imaging processes like MRI and CT scans, bioinformatics pipelines, and real-time patient monitoring systems. In finance, the ability to optimize tensor programs will enhance algorithmic trading systems, risk analysis, and fraud detection, while reducing latency and improving scalability. This framework can improve autonomous systems, such as autonomous vehicles, drones, and robots that require real-time sensor data processing and decision-making. Additionally, the media and entertainment sector can use the framework to optimize tasks like video rendering, transcoding, and real-time image enhancement.

This framework is also useful for Android and iOS-based mobile applications and software like Adobe Photoshop that rely on image processing. Tasks such as real-time image filtering, photo enhancement, and video editing can be optimized using this framework, which will speed up performance and reduce energy consumption on mobile devices. Halide has already been adopted in the industry for high-performance image processing tasks, and this framework can further refine and generalize such optimizations across devices and platforms.

This project's integration of a unified approach for optimizing tensor computations ensures its applicability across a wide range of industries and use cases, from Android mobile apps and photo editing software to scientific research and real-time analytics, making it a transformative solution for performance-critical applications.

By offering a unified, scalable, and adaptable optimization framework, this project has the potential to change how tensor computations are performed across different industries, significantly improving efficiency, accuracy, and scalability.

1.7 Navigation to the Chapters

This report is organized into six chapters.

1.7.0.1 Chapter 1 - Introduction

Introduces the motivation, problem statement, primary objectives, project scope, novelty and uniqueness, and beneficiaries and potential applications of the project.

1.7.0.2 Chapter 2 - Literature Review

This chapter provides a comprehensive exploration of the theoretical and practical foundations necessary for the project. Alternative strategies and methodologies are also observed to identify innovative solutions and techniques that could address current limitations, providing a strong foundation for the proposed framework.

1.7.0.3 Chapter 3 - Methodology

This chapter outlines the methods and approaches used to achieve project objectives, providing a detailed explanation of the design and implementation processes. Resource requirement, budget consideration, project completion percentage, task delegation, timeline, and project deliverables are addressed in this chapter, ensuring a structured approach to the project.

1.7.0.4 Chapter 4 - Results

This chapter presents the results obtained from the major tasks of the project. Each task is analyzed in detail, focusing on the key outcomes, percentage of completion, and how they contribute to the overall framework.

1.7.0.5 Chapter 5 - Discussion

In this chapter, the results are interpreted and analyzed according to the project objectives. It examines the significance of the findings and their alignment with expectations. Challenges encountered during the project are discussed, along with their resolutions and potential impact on the outcomes.

1.7.0.6 Chapter 6 - Conclusion

The final chapter summarizes the progress of the project, reaffirming its objectives and milestones achieved so far, by emphasizing the importance of the project in advancing performance prediction for DSLs and expresses confidence in successfully achieving the remaining goals.

Chapter 2

Literature Review

This literature review provides a comprehensive analysis of the foundational concepts, challenges, and advancements in optimizing tensor programs using DSLs, focusing on Halide and Tiramisu. Both DSLs decouple algorithms from scheduling, enabling efficient performance tuning for diverse hardware architectures. The review explores their unique methodologies and evaluates their respective cost models, optimization strategies, and the role of auto-tuning in systematically exploring scheduling techniques using heuristic algorithms, performance metrics, and empirical testing. The review also examines alternative optimization strategies and methodologies.

2.1 Overview of Halide and Tiramisu

Halide [1] and Tiramisu [2] are DSLs used in optimizing tensor programs. The uniqueness of these two is their ability to decouple the algorithm and schedule to get optimal performance. Both these languages offer specific abstractions to enhance the performance of multi-dimensional data processing and have the ability to adapt to different hardware architectures.

The primary purpose of Halide is high-performance image processing. By keeping the algorithm and scheduling distinct, it enables developers to specify scheduling techniques independently and write computations in an understandable and straightforward way. This division makes it easier to test various scheduling techniques and determine which configuration works best for a certain target architecture. Halide uses a scheduling language that uses intervals to represent iteration space.

Tiramisu relies on flexible representation based on a polyhedral model and a scheduling language with novel commands to explicitly manage the complexity. It uses the expressive polyhedral model to represent iteration space. It uses four level intermediate representation that allows full seperation between the algorithms, loop transformations, data layout and communications. By offering dependency analysis, it expands the possible schedules than Halide could offer.

Both Halide and Tiramisu utilize a range of transformation and optimization strategies, including loop fusion, tiling, and vectorization, to produce high-performing code for various architectures such as GPUs and CPUs.

2.2 Existing Cost Models

Cost models are the important components in the optimization of Domain-Specific Languages (DSLs), as they provide means to estimate the performance of different implementation strategies. These models play a crucial role in automating the process of selecting the most efficient schedules and transformations by predicting execution times. With accurate cost models, compilers can make informed decisions to optimize the execution of programs across various hardware architectures. Below, we look into the cost models used in two prominent DSL frameworks, Halide and Tiramisu, each of which employs a distinct approach to performance optimization.

2.2.1 Halide's Cost Model

This cost model focuses on key metrics such as execution time, memory usage, and data locality, which are critical for efficient program execution. Key Features of Halide's Cost Model:

1. **Execution Time Estimation:** Halide's model estimates the time required to execute a given schedule by analyzing the operations involved, including the number of computations, the data movement, and the memory access patterns.

2. **Memory Usage and Data Transfer:** The model accounts for memory hierarchy considerations, including data transfer between different levels of memory.

3. **Architecture-Specific Optimization:** The cost model is highly tailored to the specific hardware architecture on which programs are trained and executed. It incorporates architecture-specific parameters, such as the number of cores, cache sizes, and memory bandwidth, to refine its predictions.

By incorporating these factors, Halide's cost model provides a detailed evaluation of various schedules, allowing developers to explore trade-offs and identify the most efficient implementation strategies for their programs.

2.2.2 Tiramisu's Cost Model

Unlike Halide, Tiramisu's cost model leverages the polyhedral representation of programs to perform detailed dependency analysis and assess the impact of loop transformations and scheduling decisions. Key Features of Tiramisu's Cost Model:

1. **Polyhedral Analysis and Dependency Information:** Tiramisu's cost model utilizes the polyhedral representation of loop nests to capture precise dependency information. This allows the model to analyze how loop transformations, such as tiling, fusion, or unrolling, affect dependencies and data access patterns. By leveraging this detailed information, the model can make accurate predictions about the performance impact of different transformations.

2. **Geometric Representation of Loop Nests:** The polyhedral model provides a geometric view of loop nests, which enables the analysis of the iteration space and the relationships between loops. This representation helps Tiramisu's cost model evaluate the effect of transformations on data locality, parallelism, and execution time.

3. **Data Locality Optimization:** Tiramisu's cost model places a strong emphasis on data locality. By analyzing the memory access patterns of loop nests, it predicts the cache behavior and identifies opportunities to improve cache utilization.

4. **Hardware-Aware Performance Assessment:** The model is designed to consider device-specific factors and limitations, making it suitable for optimizing programs across a variety of hardware architectures, including CPUs, GPUs, etc. By incorporating hardware-specific parameters, such as the number of processing units, memory bandwidth, and computation capabilities, Tiramisu's cost model ensures that the selected schedules are well-suited to the target platform.

This approach enables the model to generalize effectively across different programs and architectures. By integrating these features, Tiramisu's cost model provides a comprehensive framework for evaluating the performance of various schedules and transformations. Its reliance on polyhedral analysis and detailed dependency information makes it particularly well-suited for optimizing loop-intensive programs on diverse hardware platforms.

2.3 Existing auto-tuning approaches

After predicting the runtime using cost models, to improve the efficiency of the model, auto-tuning which is the process of automatically adjusting the parameters of the model to achieve optimal performance, is used. In the context of Halide and Tiramisu, auto-tuning involves systematically exploring different scheduling techniques and transformations to identify the best configuration for a given application and target hardware.

Auto-tuning frameworks explore the vast space of potential optimizations using a combination of performance models, empirical testing, and heuristic algorithms. The goal is to optimize computing performance while minimizing memory and power consumption. Key components of an auto-tuning system include:

- **Search Algorithms:** Optimization spaces are often explored using methods such as genetic algorithms, simulated annealing, and Bayesian optimization.
- **Performance Metrics:** Metrics such as power consumption, memory utilization, and execution time guide the optimization process.
- **Empirical Testing:** Potential configurations are tested on real hardware to assess performance and refine tuning.

2.4 Alternative Strategies and Methodologies

Chapter 3

Methodology

This chapter includes the details about the approaches and methods used to create the unified framework, why the methods are selected, how they are implemented and detailed explanation of the outcomes.

3.1 Project Approach

Describe the overall approach or framework adopted for the project (e.g., experimental, analytical, or design-based). Include a block diagram/flow charts etc. to show the architecture of the project. Mention any relevant theories, models, or principles guiding the methodology. There are three main stages in developing a unified framework. Following are the three important stages in the taks:

- **Creating a common representation:** In this task, we create a common program and schedule representation applicable for the Domain Specific languages.
- **Cost model:** We create a learned cost model to predict the speedup of the programs for several schedules.
- **Auto-tuning:** This stage we traverse through the schedule search space using an appropriate search algorithm and find the schedule with minimum execution time.

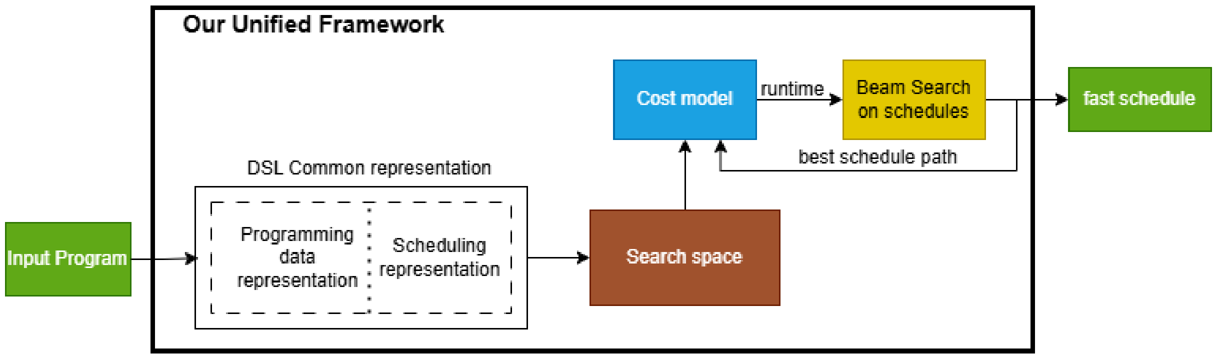


Figure 3.1: Project architecture

Figure 3.1 shows the project architecture diagram of our unified framework. It takes a program as input and delivers the best schedule as output. The input program is first converted into a common program data representation and schedule representation. Then auto-tuner searches the search space using a beam search algorithm and gives a schedule as input to the cost model. Cost model then predicts the speedup for that schedule and autotuner and cost model interact with each other till finding the best schedule with minimum execution time.

3.2 Techniques and Tools

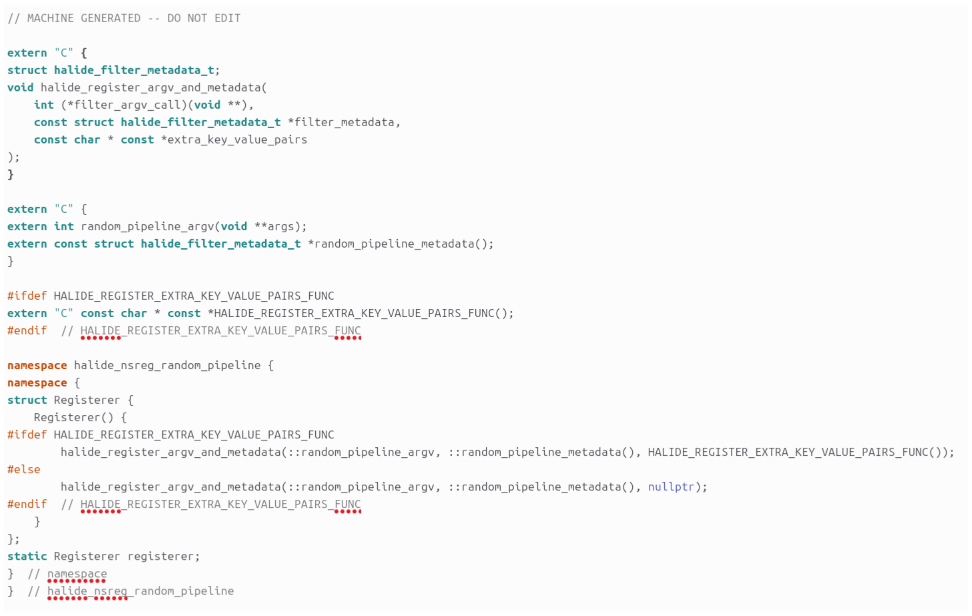
Outline the specific techniques, tools, or software used for research, design, development, or analysis. Explain why these were chosen and how they align with the project requirements.

3.3 Resource requirements and Budget

3.4 Stages of implementation

3.4.1 Generating Training Data

To create our dataset, we utilized a random pipeline generator to produce Halide code, which was executed across 465 batches. Each batch was reiterated 32 times, resulting in a total of 14,880 data samples. For every execution, both `stdout` (Standard Output) and `stderr` (Standard Error) files were generated for further analysis.



```
// MACHINE GENERATED -- DO NOT EDIT

extern "C" {
struct halide_filter_metadata_t;
void halide_register_argv_and_metadata(
    int (*filter_argv_call)(void **),
    const struct halide_filter_metadata_t *filter_metadata,
    const char * const *extra_key_value_pairs
);
}

extern "C" {
extern int random_pipeline_argv(void **args);
extern const struct halide_filter_metadata_t *random_pipeline_metadata();
}

#ifdef HALIDE_REGISTER_EXTRA_KEY_VALUE_PAIRS_FUNC
extern "C" const char * const *HALIDE_REGISTER_EXTRA_KEY_VALUE_PAIRS_FUNC();
#endif  // HALIDE_REGISTER_EXTRA_KEY_VALUE_PAIRS_FUNC

namespace halide_nsreg_random_pipeline {
namespace {
struct Registerer {
    Registerer() {
#ifdef HALIDE_REGISTER_EXTRA_KEY_VALUE_PAIRS_FUNC
        halide_register_argv_and_metadata(::random_pipeline_argv, ::random_pipeline_metadata(), HALIDE_REGISTER_EXTRA_KEY_VALUE_PAIRS_FUNC());
#else
        halide_register_argv_and_metadata(::random_pipeline_argv, ::random_pipeline_metadata(), nullptr);
#endif  // HALIDE_REGISTER_EXTRA_KEY_VALUE_PAIRS_FUNC
    }
};
static Registerer registerer;
}  // namespace
}  // halide_nsreg_random_pipeline
```

Figure 3.2: Random Pipeline Generating Halide Code

3.4.1.1 Why Additional Standard Error and Standard Output Files?

The inclusion of both `stdout` and `stderr` files is crucial for capturing comprehensive details about the execution of Halide pipelines. These files provide insights into different aspects of the process, which are vital for performance evaluation and debugging.

3.4.1.1.1 `stdout` (Standard Output)

The `stdout` file contains detailed information about the pipeline operations. It includes the various stages and transformations applied to the data, along with the overall flow of execution. This documentation helps:

- Understand the steps involved in processing the data.
- Verify that the pipeline is functioning as expected.
- Debug any issues related to pipeline execution.

3.4.1.1.2 `stderr` (Standard Error)

The `stderr` file captures computational and scheduling data related to the pipeline. It records:

- Execution times and memory usage.
- Any error messages encountered during execution.
- Scheduling details crucial for optimization and reliability analysis.

By logging these details, we can:

- Diagnose and resolve issues efficiently.
- Optimize pipeline performance.
- Ensure the reliability and accuracy of scheduling data.

3.4.1.1.3 Halide's Inbuilt `LoopNest` Function

Halide provides an inbuilt `LoopNest` function, which is instrumental in collecting detailed computational and scheduling data. This function:

- Enables developers to gain in-depth insights into loop structures, memory accesses, and other critical computational aspects.
- Facilitates performance tuning by providing clear visibility into the pipeline execution.
- Assists in debugging and optimizing Halide programs effectively.

3.4.2 Common Representation

3.4.2.1 DSL Input Program Structure

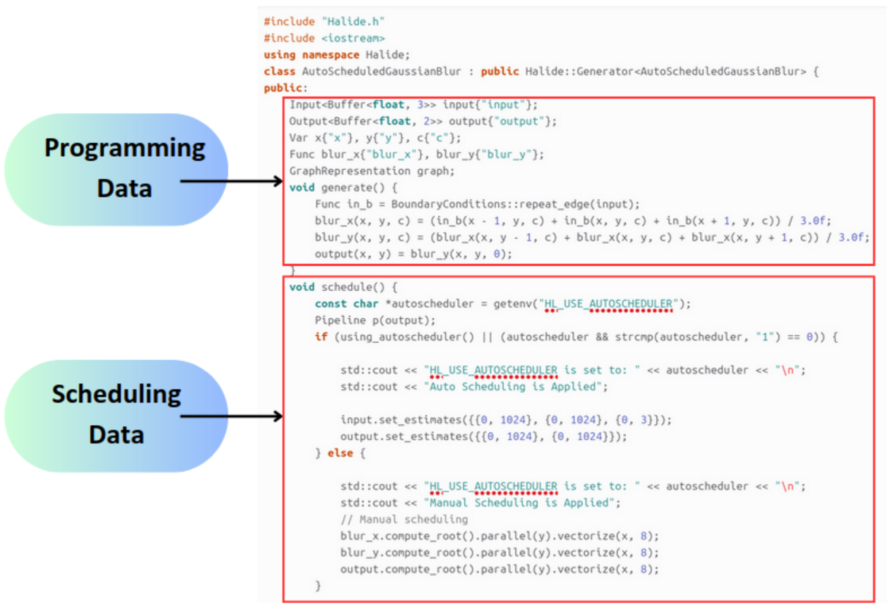
A DSL (Domain-Specific Language) program typically consists of two main sections:

a. Programming Data Section

This section defines the variables and computations that the program will perform.

Components

- **Variable Declarations**: Specifies the variables and data structures used in the program.
- **Computations**: Describes the operations and transformations applied to the data.



```cpp
#include "Halide.h"
#include <iostream>
using namespace Halide;
class AutoScheduledGaussianBlur : public Halide::Generator<AutoScheduledGaussianBlur> {
public:
    Input<Buffer<float, 3>> input{"input"};
    Output<Buffer<float, 2>> output{"output"};
    Var x{"x"}, y{"y"}, c{"c"};
    Func blur_x{"blur_x"}, blur_y{"blur_y"};
    GraphRepresentation graph;
    void generate() {
        Func in_b = BoundaryConditions::repeat_edge(input);
        blur_x(x, y, c) = (in_b(x - 1, y, c) + in_b(x, y, c) + in_b(x + 1, y, c)) / 3.0f;
        blur_y(x, y, c) = (blur_x(x, y - 1, c) + blur_x(x, y, c) + blur_x(x, y + 1, c)) / 3.0f;
        output(x, y) = blur_y(x, y, 0);
    }
    void schedule() {
        const char *autoscheduler = getenv("HL_USE_AUTOSCHEDULER");
        Pipeline p(output);
        if (using_autoscheduler() || (autoscheduler && strcmp(autoscheduler, "1") == 0)) {

            std::cout << "HL_USE_AUTOSCHEDULER is set to: " << autoscheduler << "\n";
            std::cout << "Auto Scheduling is Applied";

            input.set_estimates({{0, 1024}, {0, 1024}, {0, 3}});
            output.set_estimates({{0, 1024}, {0, 1024}});
        } else {

            std::cout << "HL_USE_AUTOSCHEDULER is set to: " << autoscheduler << "\n";
            std::cout << "Manual Scheduling is Applied";
            // Manual scheduling
            blur_x.compute_root().parallel(y).vectorize(x, 8);
            blur_y.compute_root().parallel(y).vectorize(x, 8);
            output.compute_root().parallel(y).vectorize(x, 8);
        }
```

Figure 3.3: Description of the image

b. Scheduling Data Section

This section outlines how the computations are executed, including optimization strategies.

Components

- **Manual Scheduling**: Defines specific instructions on how to execute the computations, such as parallelization and vectorization.
- **Auto Scheduling**: Automatically determines the optimal execution strategy based on certain criteria.

3.4.2.2 Representations in Different DSLs

Domain-Specific Languages (DSLs) adopt various representations to model input programs, which significantly impact their efficiency and expressiveness. The following are examples of such representations:

- **Halide**: Halide employs a Directed Acyclic Graph (DAG) representation. This structure allows Halide to break down complex image processing pipelines into individual stages, where each node represents an operation, and edges indicate dependencies [1].
- **Tiramisu**: Tiramisu uses a tree-like representation, where computations are organized hierarchically. Each node in the tree corresponds to a computation or a loop nest and child nodes are representing nested computations or loops. [2].

3.4.2.3 Reasons for Choosing Tree Representation

The choice of a tree representation for a common framework in optimizing tensor programs is motivated by several compelling advantages:

- **Hierarchical Representation of Programs**: Trees represent hierarchical data where each node depends only on its parent, reducing complexity compared to structures like DAGs where a node may have multiple parents.
- **Efficient Traversal**: Trees are inherently optimized for recursive operations, enabling efficient traversal methods such as Depth First Search (DFS) and Breadth First Search (BFS).
- **Flexibility and Modularity**: Trees provide flexibility in breaking down and recombining computations, essential for adapting to different hardware architectures and optimizing performance.
- **Parallelism**: Trees can represent dependencies explicitly, enabling parallelization by identifying independent subtrees.

3.4.2.4 Programming Pipeline Diagram

When considering about programming pipeline, it consists of several stages. Each stage we represented by a node, and the connections between each stage are represented as edges.

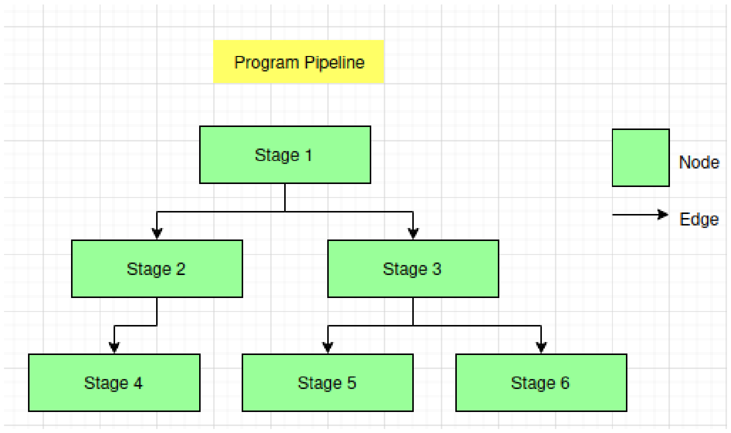


Figure 3.4: Programming Pipeline

3.4.2.5 Computational Data Representation

Our computational data representation involves parsing and extracting detailed information from input files and organizing it into a structured JSON format.

Node Details

- **Memory Access Patterns**: Refers to how data is accessed and manipulated during computations.

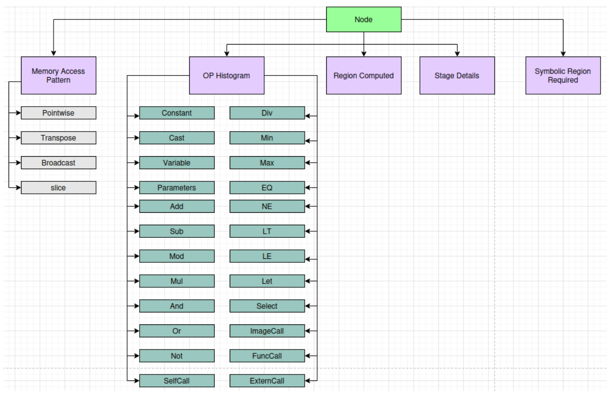


Figure 3.5: Programming Data Tree Structure for a Node

  - **Pointwise**: Operations where each output element is computed independently from corresponding input elements.
  - **Transpose**: Operations where the dimensions of the input are permuted, changing the layout of the data.
  - **Broadcast**: Operations where data from a smaller tensor is replicated to shape of a larger tensor.
  - **Slice**: Operations where specific sub-sections of a tensor are selected and operated upon.

- **Op Histogram**: Represents the frequency of different operations.
  - **Constant**: Number of constant operations.
  - **Cast**: Number of cast operations.
  - **Variable**: Number of variable operations.
  - **Param**: Number of parameter operations.
  - **Add**: Number of addition operations.
  - **Sub**: Number of subtraction operations.
  - **Mod**: Number of modulo operations.
  - **Mul**: Number of multiplication operations.
  - **Div**: Number of division operations.
  - **Min**: Number of minimum operations.
  - **Max**: Number of maximum operations.
  - **EQ**: Number of equality operations.
  - **NE**: Number of not equal operations.
  - **LT**: Number of less than operations.
  - **LE**: Number of less than or equal operations.
  - **And**: Number of logical AND operations.
  - **Or**: Number of logical OR operations.
  - **Not**: Number of logical NOT operations.

  - **Select**: Number of select operations.
  - **ImageCall**: Number of image call operations.
  - **FuncCall**: Number of function call operations.
  - **SelfCall**: Number of self-call operations.
  - **ExternCall**: Number of external call operations.
  - **Let**: Number of let operations.

- **Region Computed**:
  - **Details**: This includes the minimum and maximum values computed for different regions, aiding in understanding the range of data and optimizing processing strategies.

- **Stage Data**: Provides information about different stages of computations, including their associated minimum and maximum values.
  - **Stage 0**: Each stage's data includes detailed information about the computations performed at that stage and their corresponding minimum and maximum values. This helps in tracking the progress and results of computations at each stage.

- **Symbolic Region Required**:
  - **Details**: The symbolic region's required values indicate the necessary minimum and maximum limits for the region and ensure that computations adhere to these constraints and maintain accuracy.

Edge Details

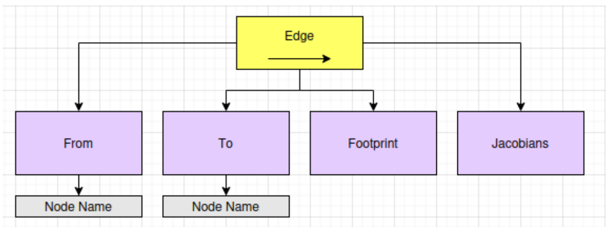
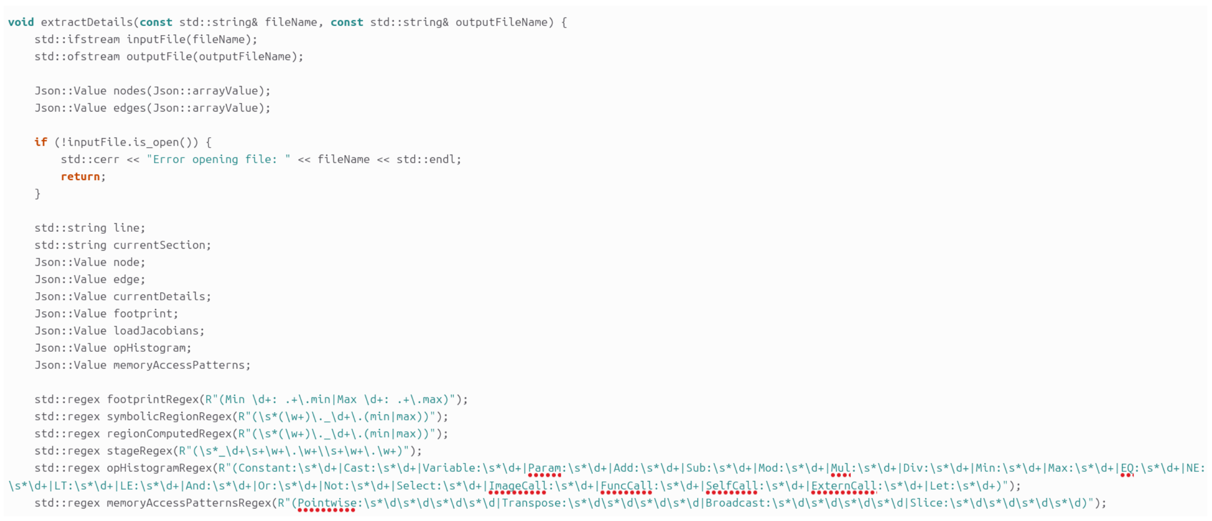


Figure 3.6: Programming Data Tree Structure for a Edge

- **Footprint**: Represents the memory footprint with min and max values for different dimensions.
  - **Min and Max values**: The memory footprint is computed by determining the minimum and maximum memory requirements for different dimensions during the execution of the pipeline stages. This helps in understanding the memory usage and optimizing resource allocation.

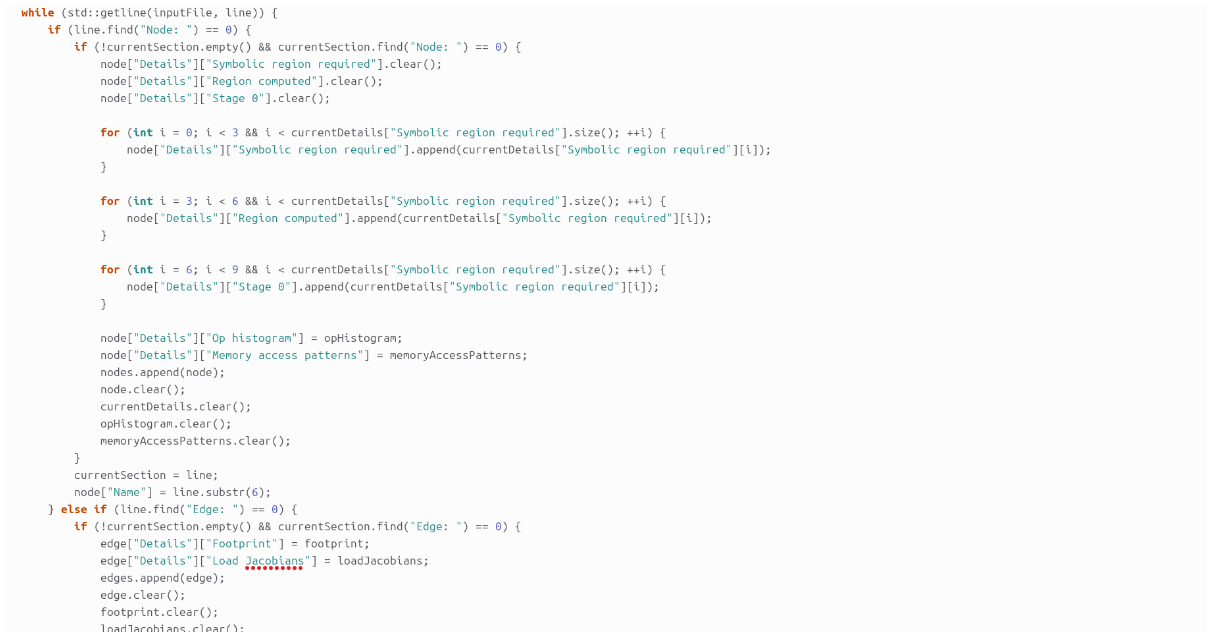
- **Load Jacobians**: The Jacobian matrix entries indicate how changes in the input variables affect the output variables. This is crucial for sensitivity analysis and understanding the impact of input variations on the overall computations.

3.4.2.6 Code Implementation For Programming Data Tree Structure



```cpp
void extractDetails(const std::string& fileName, const std::string& outputFileName) {
    std::ifstream inputFile(fileName);
    std::ofstream outputFile(outputFileName);

    Json::Value nodes(Json::arrayValue);
    Json::Value edges(Json::arrayValue);

    if (!inputFile.is_open()) {
        std::cerr << "Error opening file: " << fileName << std::endl;
        return;
    }

    std::string line;
    std::string currentSection;
    Json::Value node;
    Json::Value edge;
    Json::Value currentDetails;
    Json::Value footprint;
    Json::Value loadJacobians;
    Json::Value opHistogram;
    Json::Value memoryAccessPatterns;

    std::regex footprintRegex(R"(Min \d+: .+\.min|Max \d+: .+\.max)");
    std::regex symbolicRegionRegex(R"(\s*(\w+)\._\d+\.(min|max))");
    std::regex regionComputedRegex(R"(\s*(\w+)\._\d+\.(min|max))");
    std::regex stageRegex(R"(\s*_\d+\s+\w+\.\w+\\s+\w+\.\w+)");
    std::regex opHistogramRegex(R"(Constant:\s*\d+|Cast:\s*\d+|Variable:\s*\d+|Param:\s*\d+|Add:\s*\d+|Sub:\s*\d+|Mod:\s*\d+|Mul:\s*\d+|Div:\s*\d+|Min:\s*\d+|Max:\s*\d+|EQ:\s*\d+|NE:
\s*\d+|LT:\s*\d+|LE:\s*\d+|And:\s*\d+|Or:\s*\d+|Not:\s*\d+|Select:\s*\d+|ImageCall:\s*\d+|FuncCall:\s*\d+|SelfCall:\s*\d+|ExternCall:\s*\d+|Let:\s*\d+)");
    std::regex memoryAccessPatternsRegex(R"(Pointwise:\s*\d\s*\d\s*\d\s*\d|Transpose:\s*\d\s*\d\s*\d\s*\d|Broadcast:\s*\d\s*\d\s*\d\s*\d|Slice:\s*\d\s*\d\s*\d\s*\d)");
```

Figure 3.7: Computational Data Pre-Processing



```cpp
    while (std::getline(inputFile, line)) {
        if (line.find("Node: ") == 0) {
            if (!currentSection.empty() && currentSection.find("Node: ") == 0) {
                node["Details"]["Symbolic region required"].clear();
                node["Details"]["Region computed"].clear();
                node["Details"]["Stage 0"].clear();

                for (int i = 0; i < 3 && i < currentDetails["Symbolic region required"].size(); ++i) {
                    node["Details"]["Symbolic region required"].append(currentDetails["Symbolic region required"][i]);
                }

                for (int i = 3; i < 6 && i < currentDetails["Symbolic region required"].size(); ++i) {
                    node["Details"]["Region computed"].append(currentDetails["Symbolic region required"][i]);
                }

                for (int i = 6; i < 9 && i < currentDetails["Symbolic region required"].size(); ++i) {
                    node["Details"]["Stage 0"].append(currentDetails["Symbolic region required"][i]);
                }

                node["Details"]["Op histogram"] = opHistogram;
                node["Details"]["Memory access patterns"] = memoryAccessPatterns;
                nodes.append(node);
                node.clear();
                currentDetails.clear();
                opHistogram.clear();
                memoryAccessPatterns.clear();
            }
            currentSection = line;
            node["Name"] = line.substr(6);
        } else if (line.find("Edge: ") == 0) {
            if (!currentSection.empty() && currentSection.find("Edge: ") == 0) {
                edge["Details"]["Footprint"] = footprint;
                edge["Details"]["Load Jacobians"] = loadJacobians;
                edges.append(edge);
                edge.clear();
                footprint.clear();
                loadJacobians.clear();
```

Figure 3.8: Process of Creating the Tree Structure for Computational Data

3.4.2.7 Scheduling Data Tree Structure

We created our scheduling data as a tree.

Parent Nodes

- Represent specific scheduling entities (e.g., scheduling features for a particular task).
- Contain various scheduling features as key-value pairs.

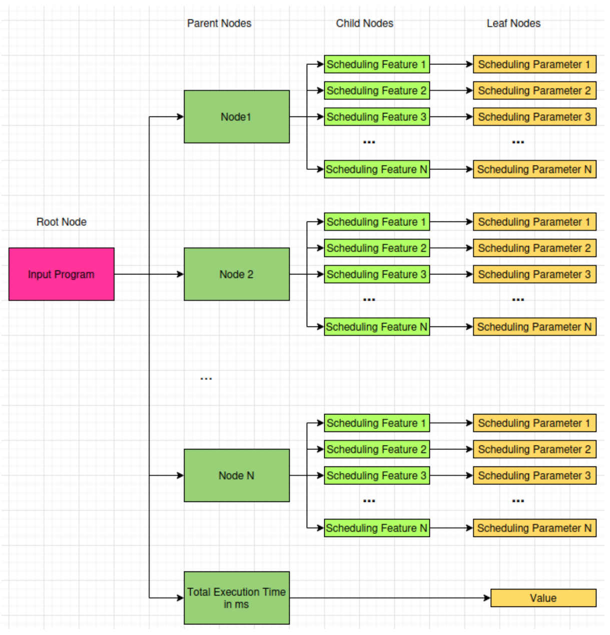


Figure 3.9: Scheduling Data Output Tree Structure

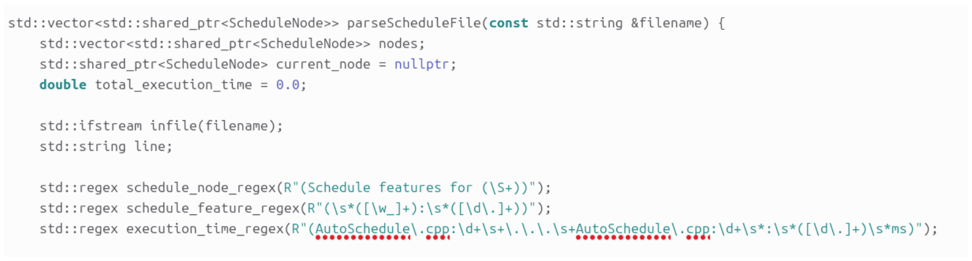
Child Nodes

- Represent individual scheduling parameters related to the parent node.

Separate Parent Node for Execution Time

- Represents the total execution time for all tasks in ms.

3.4.2.8 Code Implementation For Scheduling Data Tree Structure



```
std::vector<std::shared_ptr<ScheduleNode>> parseScheduleFile(const std::string &filename) {
    std::vector<std::shared_ptr<ScheduleNode>> nodes;
    std::shared_ptr<ScheduleNode> current_node = nullptr;
    double total_execution_time = 0.0;

    std::ifstream infile(filename);
    std::string line;

    std::regex schedule_node_regex(R"(Schedule features for (\S+))");
    std::regex schedule_feature_regex(R"(\s*([\w_]+):\s*([\d\.]+))");
    std::regex execution_time_regex(R"(AutoSchedule\.cpp:\d+\s+\.\.\.\s+AutoSchedule\.cpp:\d+\s*:\s*([\d\.]+)\s*ms)");
```

Figure 3.10: Scheduling Data Pre-Processing

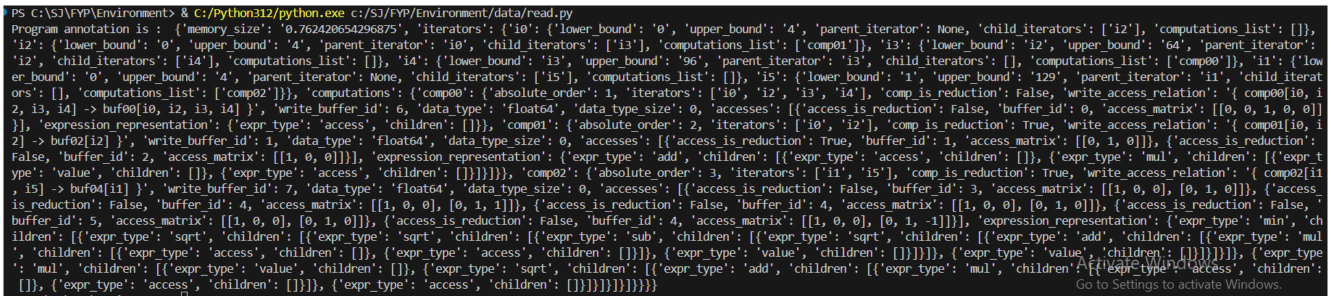
```
    while (std::getline(infile, line)) {
        std::smatch match;
        if (std::regex_search(line, match, schedule_node_regex)) {
            if (current_node) {
                nodes.push_back(current_node);
            }
            current_node = std::make_shared<ScheduleNode>();
            current_node->name = match[1].str();
        } else if (current_node && std::regex_search(line, match, schedule_feature_regex)) {
            current_node->scheduling_features[match[1].str()] = std::stod(match[2].str());
        } else if (std::regex_search(line, match, execution_time_regex)) {
            total_execution_time += std::stod(match[1].str());
        }
    }
    if (current_node) {
        nodes.push_back(current_node);
    }

    // Add execution time as a separate node
    auto execution_time_node = std::make_shared<ScheduleNode>();
    execution_time_node->name = "execution_time";
    execution_time_node->scheduling_features["total_execution_time_ms"] = total_execution_time;
    nodes.push_back(execution_time_node);

    return nodes;
}
```

Figure 3.11: Process of Creating the Tree Structure for Scheduling Data

3.4.3 Cost Model

Our cost model takes a dictionary contains several keys as the input for the training of the of the model. The keys are

1. **filename:** Name of the program
2. **initial_execution_time:** Time taken to execute the program
3. **program_annotation:** Program details including the memory usage of the programs, computations and loops in the program and data access matrices. Figure 3.12 shows the program_annotation of a sample program.



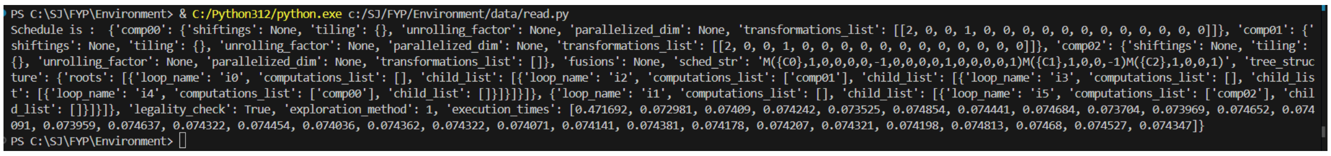
```
PS C:\SJ\FYP\Environment> & C:/Python312/python.exe c:/SJ/FYP/Environment/data/read.py
Program annotation is :  {'memory_size': '0.762420654296875', 'iterators': {'i0': {'lower_bound': '0', 'upper_bound': '4', 'parent_iterator': None, 'child_iterators': ['i2'], 'computations_list': []},
'i2': {'lower_bound': '0', 'upper_bound': '4', 'parent_iterator': 'i0', 'child_iterators': ['i3'], 'computations_list': ['comp01']}, 'i3': {'lower_bound': 'i2', 'upper_bound': '64', 'parent_iterator':
'i2', 'child_iterators': ['i4'], 'computations_list': []}, 'i4': {'lower_bound': 'i3', 'upper_bound': '96', 'parent_iterator': 'i3', 'child_iterators': [], 'computations_list': ['comp00']}, 'i1': {'low
er_bound': '0', 'upper_bound': '4', 'parent_iterator': None, 'child_iterators': ['i5'], 'computations_list': []}, 'i5': {'lower_bound': '1', 'upper_bound': '129', 'parent_iterator': 'i1', 'child_iterat
ors': [], 'computations_list': ['comp02']}}, 'computations': {'comp00': {'absolute_order': 1, 'iterators': ['i0', 'i2', 'i3', 'i4'], 'comp_is_reduction': False, 'write_access_relation': '{ comp00[i0, i
2, i3, i4] -> buf00[i0, i2, i3, i4] }', 'write_buffer_id': 6, 'data_type': 'float64', 'data_type_size': 0, 'accesses': [{'access_is_reduction': False, 'buffer_id': 0, 'access_matrix': [[0, 0, 1, 0, 0]]
}], 'expression_representation': {'expr_type': 'access', 'children': []}}, 'comp01': {'absolute_order': 2, 'iterators': ['i0', 'i2'], 'comp_is_reduction': True, 'write_access_relation': '{ comp01[i0, i
2] -> buf02[i2] }', 'write_buffer_id': 1, 'data_type': 'float64', 'data_type_size': 0, 'accesses': [{'access_is_reduction': True, 'buffer_id': 1, 'access_matrix': [[0, 1, 0]]}, {'access_is_reduction':
False, 'buffer_id': 2, 'access_matrix': [[1, 0, 0]]}], 'expression_representation': {'expr_type': 'add', 'children': [{'expr_type': 'access', 'children': []}, {'expr_type': 'mul', 'children': [{'expr_t
ype': 'value', 'children': []}, {'expr_type': 'access', 'children': []}]}]}}, 'comp02': {'absolute_order': 3, 'iterators': ['i1', 'i5'], 'comp_is_reduction': True, 'write_access_relation': '{ comp02[i1
, i5] -> buf04[i1] }', 'write_buffer_id': 7, 'data_type': 'float64', 'data_type_size': 0, 'accesses': [{'access_is_reduction': False, 'buffer_id': 3, 'access_matrix': [[1, 0, 0], [0, 1, 0]]}, {'access_
is_reduction': False, 'buffer_id': 4, 'access_matrix': [[1, 0, 0], [0, 1, 1]]}, {'access_is_reduction': False, 'buffer_id': 4, 'access_matrix': [[1, 0, 0], [0, 1, 0]]}, {'access_is_reduction': False, '
buffer_id': 5, 'access_matrix': [[1, 0, 0], [0, 1, 0]]}, {'access_is_reduction': False, 'buffer_id': 4, 'access_matrix': [[1, 0, 0], [0, 1, -1]]}], 'expression_representation': {'expr_type': 'min', 'ch
ildren': [{'expr_type': 'sqrt', 'children': [{'expr_type': 'sqrt', 'children': [{'expr_type': 'sub', 'children': [{'expr_type': 'sqrt', 'children': [{'expr_type': 'add', 'children': [{'expr_type': 'mul
', 'children': [{'expr_type': 'access', 'children': []}, {'expr_type': 'access', 'children': []}]}, {'expr_type': 'value', 'children': []}]}]}, {'expr_type': 'value', 'children': []}]}]}]}, {'expr_type
': 'mul', 'children': [{'expr_type': 'value', 'children': []}, {'expr_type': 'sqrt', 'children': [{'expr_type': 'add', 'children': [{'expr_type': 'mul', 'children': [{'expr_type': 'access', 'children':
 []}, {'expr_type': 'access', 'children': []}]}, {'expr_type': 'access', 'children': []}]}]}]}]}}}}
```

Figure 3.12: Program Annotation of a program

4. **schedules_list:** List of all the possible schedules. Figure 3.13 shows a possible schedule of the above program with the applied transformations and parameters.

3.4.3.1 How the model works?

To train the model for the speedup prediction, we first need to convert the input data into a suitable representation to represent the program for every schedule. Data conversion

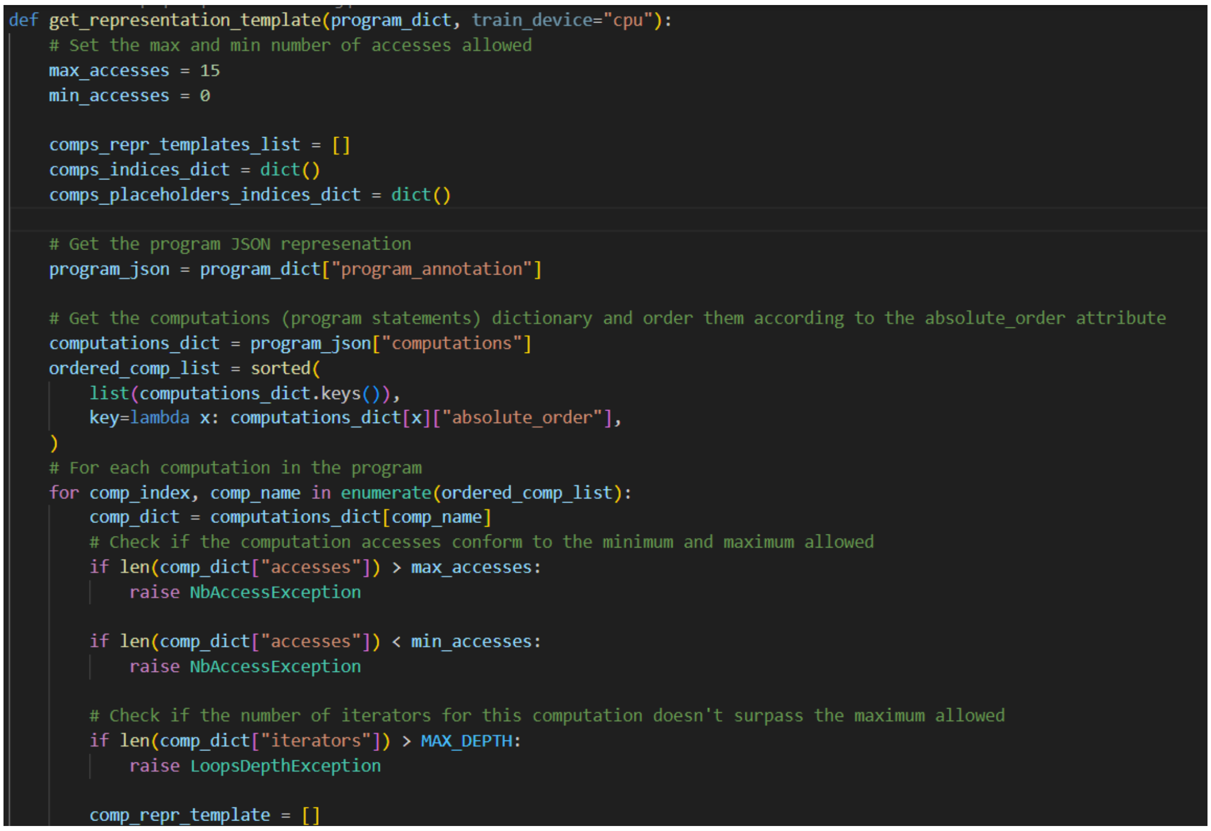


```
PS C:\SJ\FYP\Environment> & C:/Python312/python.exe c:/SJ/FYP/Environment/data/read.py
Schedule is :  {'comp00': {'shiftings': None, 'tiling': {}, 'unrolling_factor': None, 'parallelized_dim': None, 'transformations_list': [[2, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]}, 'comp01': {'
shiftings': None, 'tiling': {}, 'unrolling_factor': None, 'parallelized_dim': None, 'transformations_list': [[2, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]}, 'comp02': {'shiftings': None, 'tiling':
{}, 'unrolling_factor': None, 'parallelized_dim': None, 'transformations_list': []}, 'fusions': None, 'sched_str': 'M({C0},1,0,0,0,0,-1,0,0,0,0,1,0,0,0,0,1)M({C1},1,0,0,-1)M({C2},1,0,0,1)', 'tree_struc
ture': {'roots': [{'loop_name': 'i0', 'computations_list': [], 'child_list': [{'loop_name': 'i2', 'computations_list': ['comp01'], 'child_list': [{'loop_name': 'i3', 'computations_list': [], 'child_lis
t': [{'loop_name': 'i4', 'computations_list': ['comp00'], 'child_list': []}]}]}]}, {'loop_name': 'i1', 'computations_list': [], 'child_list': [{'loop_name': 'i5', 'computations_list': ['comp02'], 'chil
d_list': []}]}]}, 'legality_check': True, 'exploration_method': 1, 'execution_times': [0.471692, 0.072981, 0.07409, 0.074242, 0.073525, 0.074854, 0.074441, 0.074684, 0.073704, 0.073969, 0.074652, 0.074
091, 0.073959, 0.074637, 0.074322, 0.074454, 0.074036, 0.074362, 0.074322, 0.074071, 0.074141, 0.074381, 0.074178, 0.074207, 0.074321, 0.074198, 0.074813, 0.07468, 0.074527, 0.074347]}
PS C:\SJ\FYP\Environment> 
```

Figure 3.13: A schedule of this program

contains the following steps

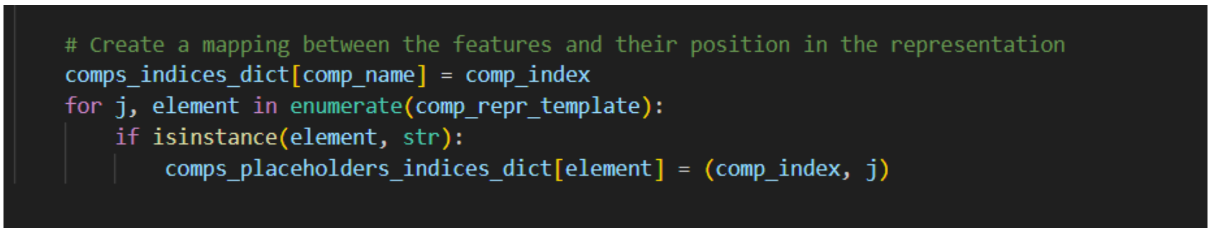
1. **Create a representation template:** We define the get_representation_template function to create the program representation template. First we load the program details with no schedules applied from the zeroth index of the scheduleslist and create representation template lists for the computations and loops. This is shown in the figure 3.14.
   It also creates the indices dictionaries for the transformation placeholders and then maps the placeholders with their respective indices which is shown in the figure 3.15 .
   This function gives the following five templates called the prog_tree, comps_repr_list, loops_repr_list, comps_placeholders_indices_dict and loop_placeholders_indices_dict as the outptut as shown in the figure 3.16.



```python
def get_representation_template(program_dict, train_device="cpu"):
    # Set the max and min number of accesses allowed
    max_accesses = 15
    min_accesses = 0

    comps_repr_templates_list = []
    comps_indices_dict = dict()
    comps_placeholders_indices_dict = dict()

    # Get the program JSON represenation
    program_json = program_dict["program_annotation"]

    # Get the computations (program statements) dictionary and order them according to the absolute_order attribute
    computations_dict = program_json["computations"]
    ordered_comp_list = sorted(
        list(computations_dict.keys()),
        key=lambda x: computations_dict[x]["absolute_order"],
    )
    # For each computation in the program
    for comp_index, comp_name in enumerate(ordered_comp_list):
        comp_dict = computations_dict[comp_name]
        # Check if the computation accesses conform to the minimum and maximum allowed
        if len(comp_dict["accesses"]) > max_accesses:
            raise NbAccessException

        if len(comp_dict["accesses"]) < min_accesses:
            raise NbAccessException

        # Check if the number of iterators for this computation doesn't surpass the maximum allowed
        if len(comp_dict["iterators"]) > MAX_DEPTH:
            raise LoopsDepthException

        comp_repr_template = []
```

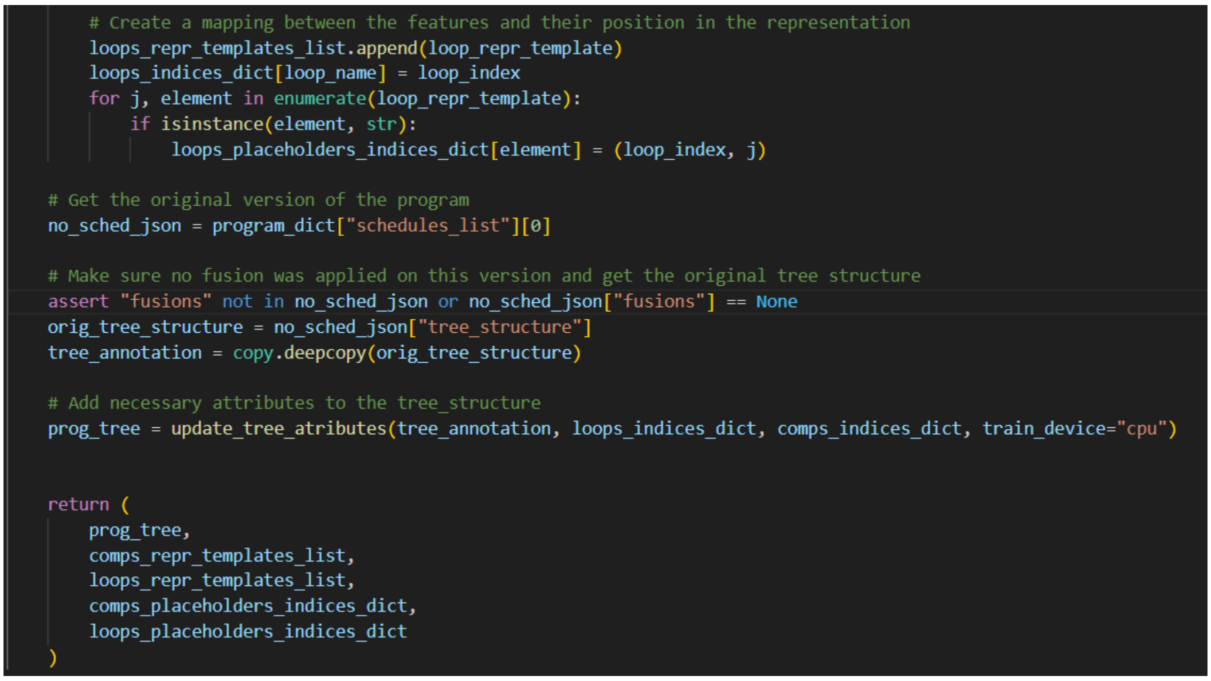
Figure 3.14: get_representation_template function

2. **Update the template:** This is done using a function called get_schedule_representation. First this function gets the output of the above template creation function and updates the templates for each schedule with the associated parameters of the schedules. Figure 3.17 shows some lines of the code of this function.



```
# Create a mapping between the features and their position in the representation
comps_indices_dict[comp_name] = comp_index
for j, element in enumerate(comp_repr_template):
    if isinstance(element, str):
        comps_placeholders_indices_dict[element] = (comp_index, j)
```

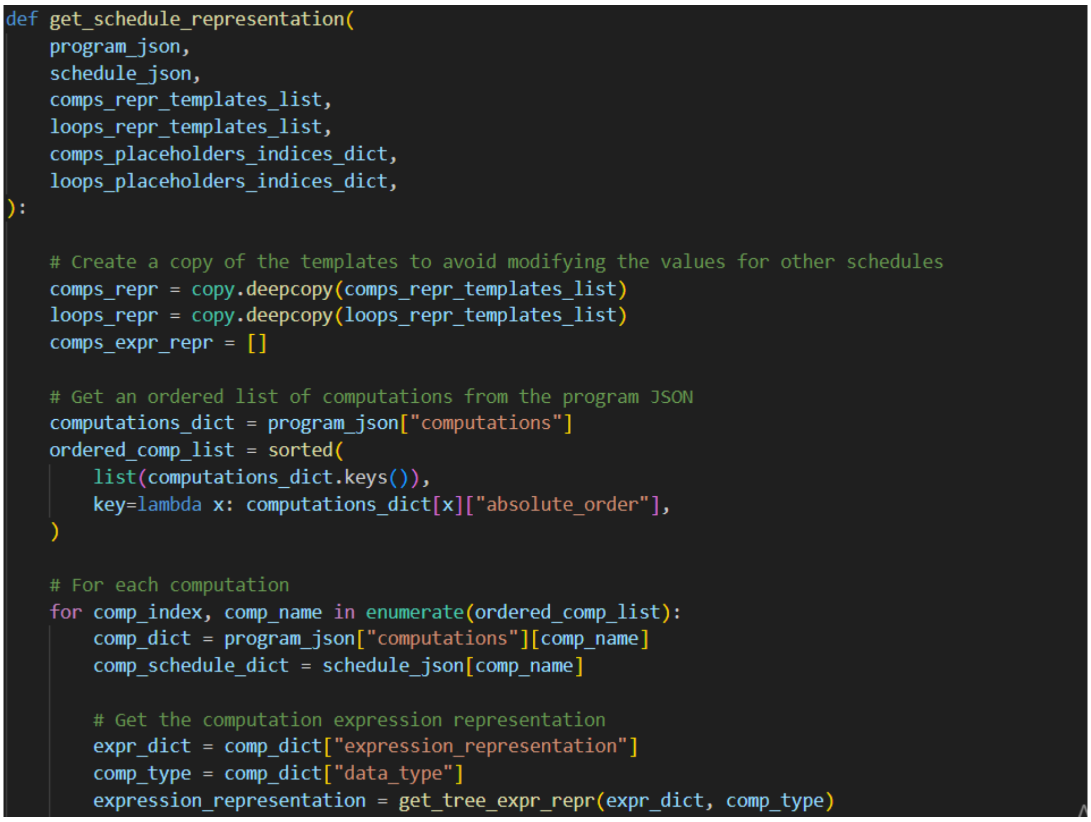
Figure 3.15: Mapping of the placeholder indices with nodes



```
        # Create a mapping between the features and their position in the representation
        loops_repr_templates_list.append(loop_repr_template)
        loops_indices_dict[loop_name] = loop_index
        for j, element in enumerate(loop_repr_template):
            if isinstance(element, str):
                loops_placeholders_indices_dict[element] = (loop_index, j)

    # Get the original version of the program
    no_sched_json = program_dict["schedules_list"][0]

    # Make sure no fusion was applied on this version and get the original tree structure
    assert "fusions" not in no_sched_json or no_sched_json["fusions"] == None
    orig_tree_structure = no_sched_json["tree_structure"]
    tree_annotation = copy.deepcopy(orig_tree_structure)

    # Add necessary attributes to the tree_structure
    prog_tree = update_tree_atributes(tree_annotation, loops_indices_dict, comps_indices_dict, train_device="cpu")


    return (
        prog_tree,
        comps_repr_templates_list,
        loops_repr_templates_list,
        comps_placeholders_indices_dict,
        loops_placeholders_indices_dict
    )
```

Figure 3.16: Output templates of the function

After converting the data into a suitable format that could represent the program semantically as well as suitable to give as input to the cost model, we designed the model architecture to predict the model. To process the structural data representation we created, we considered two possible model architectures.
After converting the data into a suitable format that represent the program structure semantically correct as well as suitable to feed as input to the learned cost model, we created the model architecture to predict the speedup for various schedules. To process the structural representation of the programs, we selected the Long Short Term Memory architecture and Graph Neural Network architecture as our options.
LSTMs are good at processing the sequential data patterns. We selected the LSTMs based on the following reasons. Programs are generally represented as a tree representation. LSTMs are a good choice for processing the hierarchical relationships of loops and computations in the program tree. Program trees may have variable tree width and tree depth. We do not need to have fixed length of input size for the LSTM model architecture. Other reason is the temporal nature of transformations. Transformations depend on their order also. LSTMs are ideal for capturing the sequential dependencies as they maintain a memory of prior transformations while processing the subsequent ones.
GNNs are good at processing graph based data representations. GNNs ca be chosen because of graphs' ability to represent the programs in more detailed manner than of the



```python
def get_schedule_representation(
    program_json,
    schedule_json,
    comps_repr_templates_list,
    loops_repr_templates_list,
    comps_placeholders_indices_dict,
    loops_placeholders_indices_dict,
):

    # Create a copy of the templates to avoid modifying the values for other schedules
    comps_repr = copy.deepcopy(comps_repr_templates_list)
    loops_repr = copy.deepcopy(loops_repr_templates_list)
    comps_expr_repr = []

    # Get an ordered list of computations from the program JSON
    computations_dict = program_json["computations"]
    ordered_comp_list = sorted(
        list(computations_dict.keys()),
        key=lambda x: computations_dict[x]["absolute_order"],
    )

    # For each computation
    for comp_index, comp_name in enumerate(ordered_comp_list):
        comp_dict = program_json["computations"][comp_name]
        comp_schedule_dict = schedule_json[comp_name]

        # Get the computation expression representation
        expr_dict = comp_dict["expression_representation"]
        comp_type = comp_dict["data_type"]
        expression_representation = get_tree_expr_repr(expr_dict, comp_type)
```

Figure 3.17: get_schedule_representation function

tree representation. In tree representation, edges represent the parent-child relationships between the nodes while in graph representation, edges are used to represent the parent-child relationship as well as the read and write access relationships between the nodes. Similar to the LSTMs, GNNs also can take input of variable length. We do not need to limit the input into a fixed length. Unlike LSTMs, by using the GNNs we can capture the global dependencies between the nodes by aggregating and message passing between the nodes. But the problem in using the GNNs is the proper data representation for the model. Generally, programs are in a tree format, to use them in GNN we have to convert the tree representation into a directed acyclic graph representation. That's why we initially used the recursive LSTM architecture to train the model for the speedup prediction.

3.4.4 Auto-tuning

3.4.4.1 Search Space & Exploration

Our auto-scheduler generates different combinations of code transformations, including fusion, unrolling, tiling, and interchange. Each unique code transformation is called a schedule, and the set of all such schedules forms the search space. We represent our search space as a tree structure, although there are many ways to structure it.

For example, let N be the number of schedules. Figure 2 illustrates how we represent our search space. The original program (i.e., without any schedule) serves as the parent node. At depth 1, we apply fusion to the original program. Then, at depth 2, we apply unrolling to the schedule. Subsequently, we apply tiling and interchange at depths 3 and

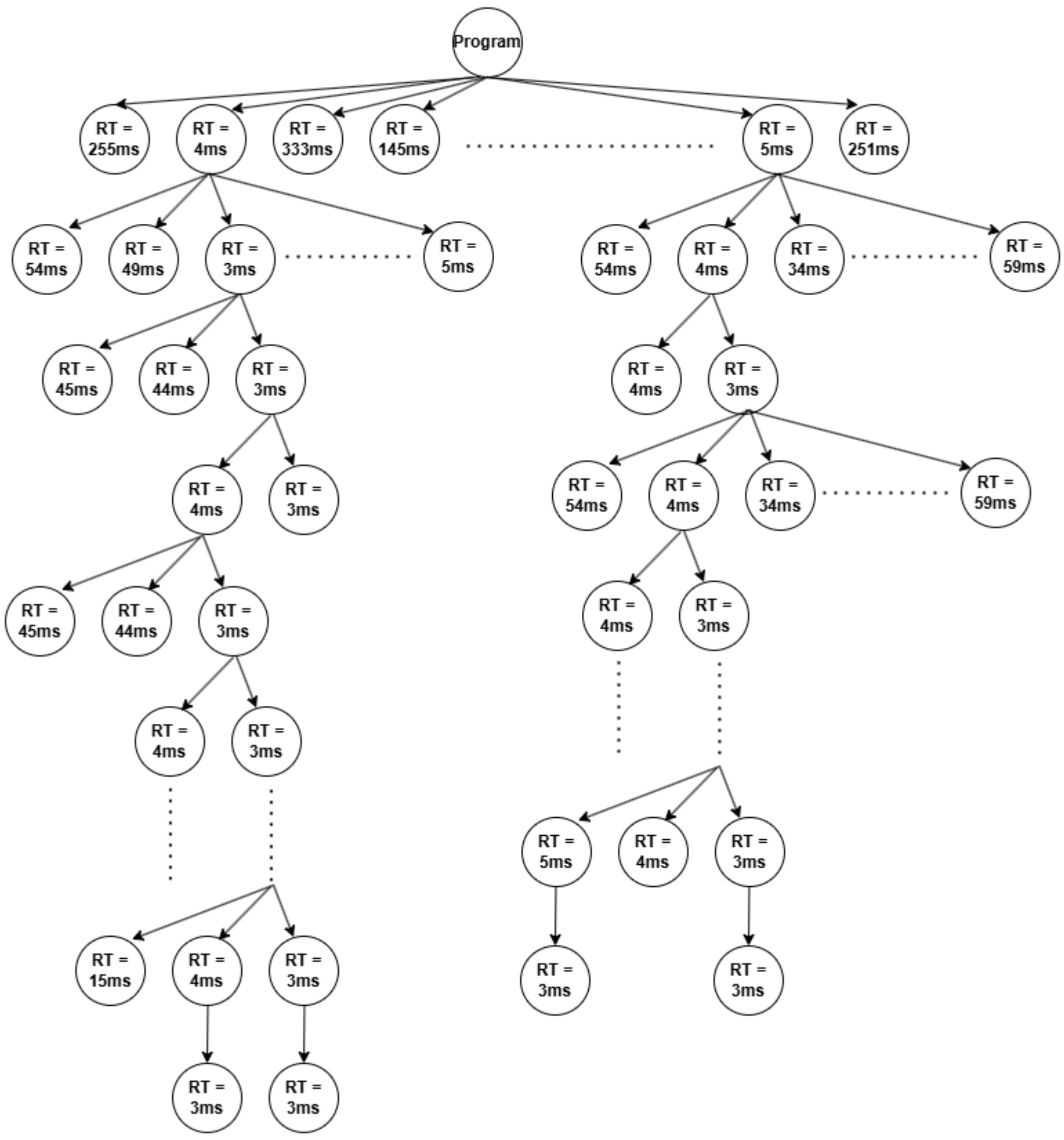


Figure 3.18: Search space as a tree structure

4, respectively. At depth 5, we apply fusion again to the output of depth 4. This process continues until we reach the desired depth. By changing the parameter values of the transformations at each depth, we generate different schedules for a single program.

Furthermore, the root node represents the original program, and each child node represents a schedule with any combination of the considered transformations. Regardless of the order in which we apply transformations, the size of the search space depends solely on the number of transformations we apply. This size grows exponentially with the number of code transformations.

To optimize the program, we select the best configuration of code transformation in terms of runtime. Since the search space contains various schedule patterns, we explore it using a search algorithm that identifies the best schedule within the search space. For this purpose, we utilize Beam Search (Russell and Norvig, 2009 [3]). This method is simple and efficient for optimizing programs. Moreover, Adams et al., 2019 [4], employ this algorithm as a search technique in their framework.

We represent our search space as a tree structure. Beam search starts with the root node (the original program). It then moves to the first depth of the tree and evaluates

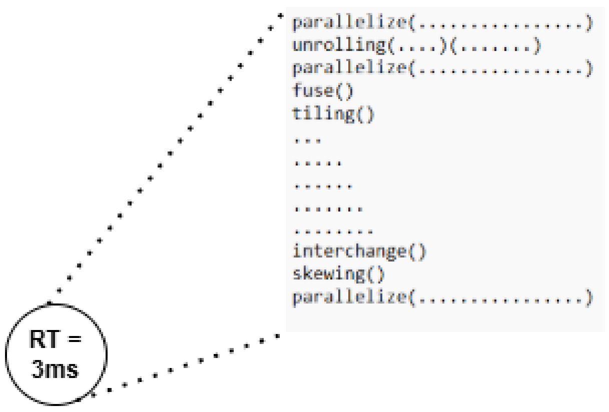

RT = 3ms
parallelize(................)
unrolling(....)(.......)
parallelize(................)
fuse()
tiling()
interchange()
skewing()
parallelize(................)

Figure 3.19: A node in Search space

runtimes for each node at that depth. After this evaluation, we retain only the top k schedules (determined by runtime performance), while we discard the remaining nodes. The algorithm then evaluates runtimes for the child nodes of the selected k schedules. We define the parameter k as the beam size, and it is user-defined. This process continues until we reach the deepest level of the tree structure, ultimately resulting in the best schedule for the program. Note: We perform runtime evaluation using the proposed cost model or by directly running the program on hardware. Figure 3 illustrates the search exploration with k = 2.

Algorithm 1 BeamSearch(ast, k)

```
Require: $ast$: an AST, $k$: the beam size
 1: if StopCriterion($ast$) then
 2:     Stop searching
 3: end if

 4: $children \leftarrow$ GenerateNextTreeLevel($ast$)
 5: for all $c \in children$ do
 6:     $c.evaluation \leftarrow$ EvaluationFunction($c$)
 7:     if $c.evaluation < best\_evaluation$ then
 8:         $best\_evaluation \leftarrow c.evaluation$
 9:         $best\_schedule \leftarrow c$
10:     end if
11: end for
12: Sort $children$ from best evaluation to worst

13: for $i = 0$ to $k$ do
14:     BeamSearch($children[i]$, $k$)
15: end for
```

Algorithm 1 demonstrates the step-by-step process of how beam search explores the search space. It requires the search space to be represented as an Abstract Syntax Tree (AST) and the beam size k. Using this technique, we can identify the best schedule of a given program within the search space.

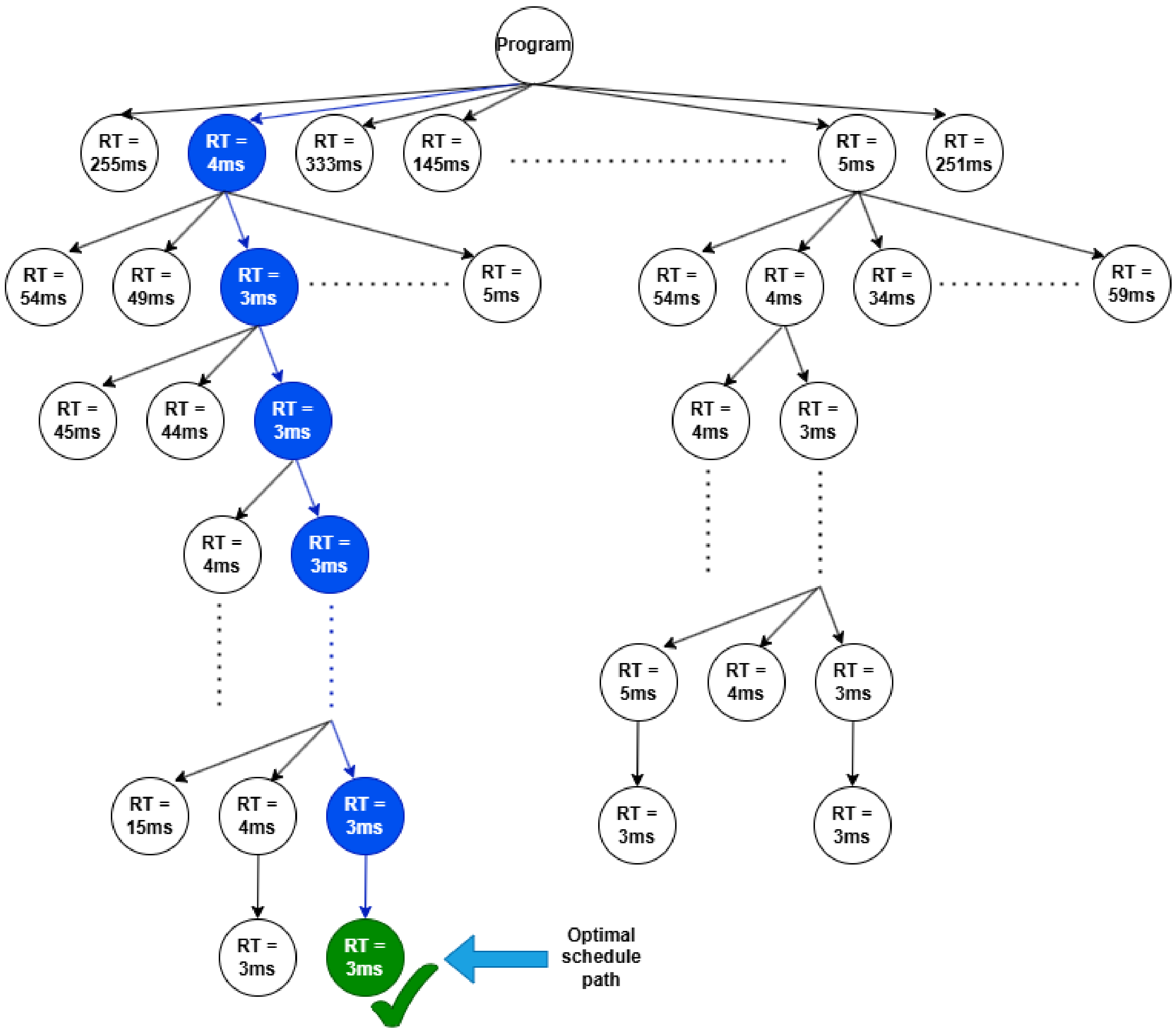


Figure 3.20: How Beam search explores the space

3.5 Task Delegation and Timeline

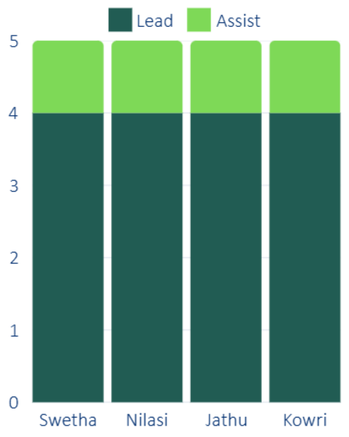
3.6 Task Delegation & Timeline

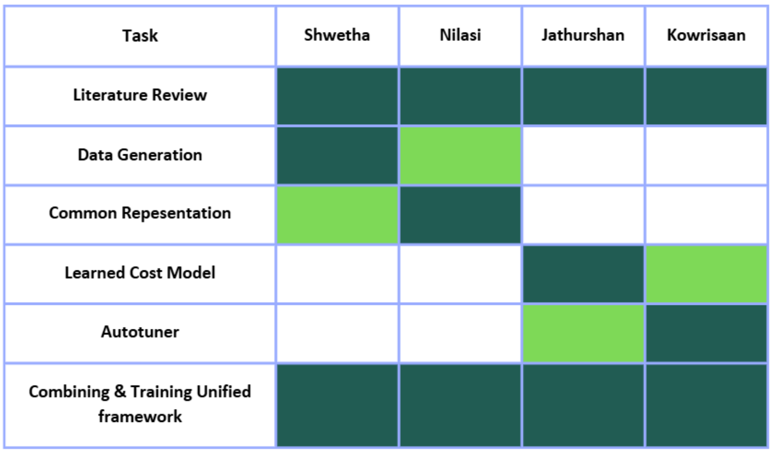
Figure 3.21 illustrates the main tasks, divided equally, as shown in the chart. Each task is assigned to at least two members to ensure efficiency and effectiveness. Figure 3.22 presents the planned timeline for completing the tasks. As scheduled, we proposed our representation for any DSLs by the end of December 2024. Currently, we have started model training and fine-tuning, while simultaneously working on the publication.

3.7 Project Deliverables

We will deliver a Unified Framework for optimizing tensor programs that can be used for any DSLs. The unified framework contains,

- **Common Representation**





Task	Shwetha	Nilasi	Jathurshan	Kowrisaan
Literature Review	⬛	⬛	⬛	⬛
Data Generation	⬛	🟩		
Common Repesentation	🟩	⬛		
Learned Cost Model			⬛	🟩
Autotuner			🟩	⬛
Combining & Training Unified framework	⬛	⬛	⬛	⬛

Figure 3.21: Task Delegation

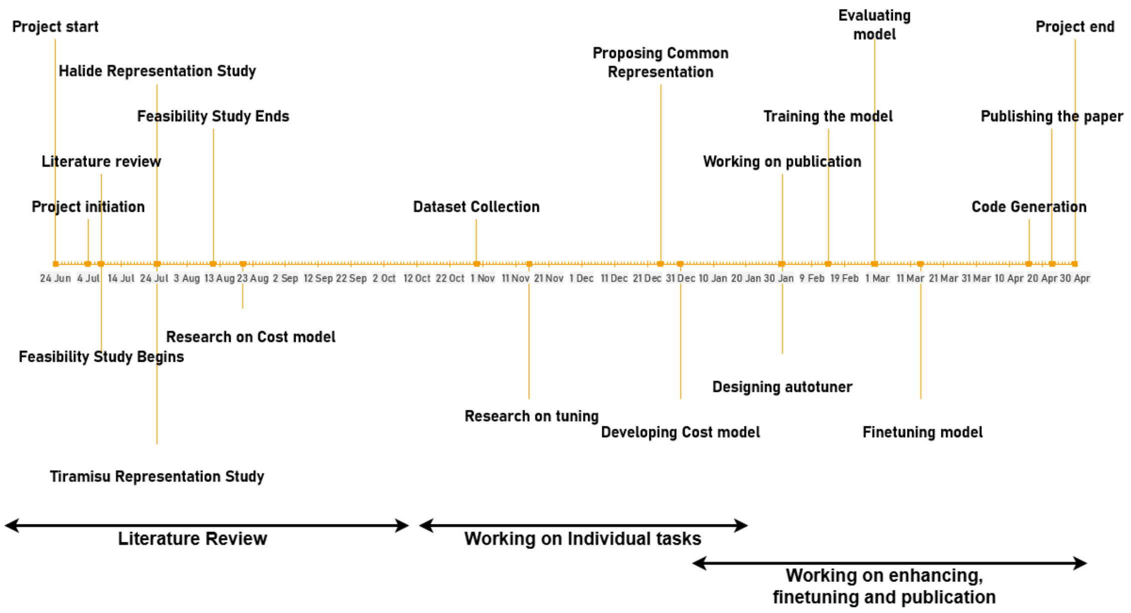


Figure 3.22: Timeline

- **Cost Model** - An LSTM architecture that predicts the speedup of several schedules applied to a program.
- **Autotuning approach** - A beam search algorithm that searches the global optimal schedule of a random program.

Chapter 4

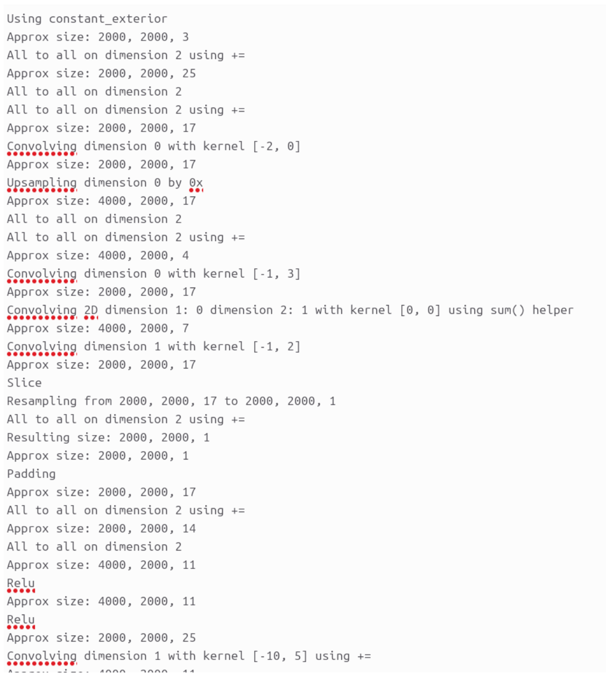
Results

This section contains the divided tasks and the results obtained in each task, key outcome from the obtained results and what percentage of each task has been completed. Training data generation, common representation creation, cost model development, and autotuning are the major tasks in developing the unified framework. A detailed explanation of the results obtained from the above tasks, key outcomes and the completed percentage are provided below.

4.1 Obtained results

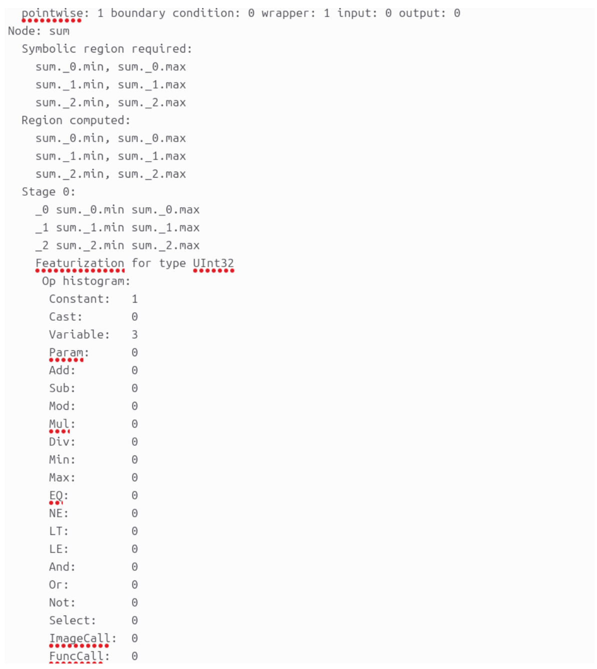
4.1.1 Generating Training Data

`stdout` and `stderr` are related to the random pipeline generation halide code. Figure 4.1 shows the **stdout** format and the figure 4.2 shows the **stderr** format.



```
Using constant_exterior
Approx size: 2000, 2000, 3
All to all on dimension 2 using +=
Approx size: 2000, 2000, 25
All to all on dimension 2
All to all on dimension 2 using +=
Approx size: 2000, 2000, 17
Convolving dimension 0 with kernel [-2, 0]
Approx size: 2000, 2000, 17
Upsampling dimension 0 by 0x
Approx size: 4000, 2000, 17
All to all on dimension 2
All to all on dimension 2 using +=
Approx size: 4000, 2000, 4
Convolving dimension 0 with kernel [-1, 3]
Approx size: 2000, 2000, 17
Convolving 2D dimension 1: 0 dimension 2: 1 with kernel [0, 0] using sum() helper
Approx size: 4000, 2000, 7
Convolving dimension 1 with kernel [-1, 2]
Approx size: 2000, 2000, 17
Slice
Resampling from 2000, 2000, 17 to 2000, 2000, 1
All to all on dimension 2 using +=
Resulting size: 2000, 2000, 1
Approx size: 2000, 2000, 1
Padding
Approx size: 2000, 2000, 17
All to all on dimension 2 using +=
Approx size: 2000, 2000, 14
All to all on dimension 2
Approx size: 4000, 2000, 11
Relu
Approx size: 4000, 2000, 11
Relu
Approx size: 2000, 2000, 25
Convolving dimension 1 with kernel [-10, 5] using +=
```

Figure 4.1: Format of stdout

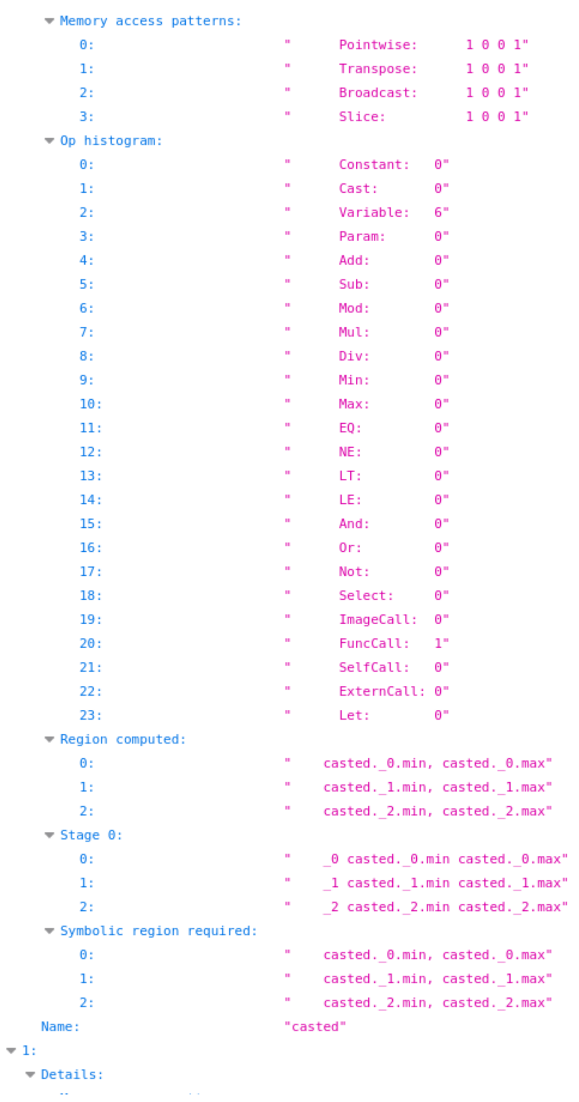


```
  pointwise: 1 boundary condition: 0 wrapper: 1 input: 0 output: 0
Node: sum
  Symbolic region required:
    sum._0.min, sum._0.max
    sum._1.min, sum._1.max
    sum._2.min, sum._2.max
  Region computed:
    sum._0.min, sum._0.max
    sum._1.min, sum._1.max
    sum._2.min, sum._2.max
  Stage 0:
    _0 sum._0.min sum._0.max
    _1 sum._1.min sum._1.max
    _2 sum._2.min sum._2.max
    Featurization for type UInt32
     Op histogram:
      Constant:   1
      Cast:       0
      Variable:   3
      Param:      0
      Add:        0
      Sub:        0
      Mod:        0
      Mul:        0
      Div:        0
      Min:        0
      Max:        0
      EQ:         0
      NE:         0
      LT:         0
      LE:         0
      And:        0
      Or:         0
      Not:        0
      Select:     0
      ImageCall:  0
      FuncCall:   0
```

Figure 4.2: Format of stderr

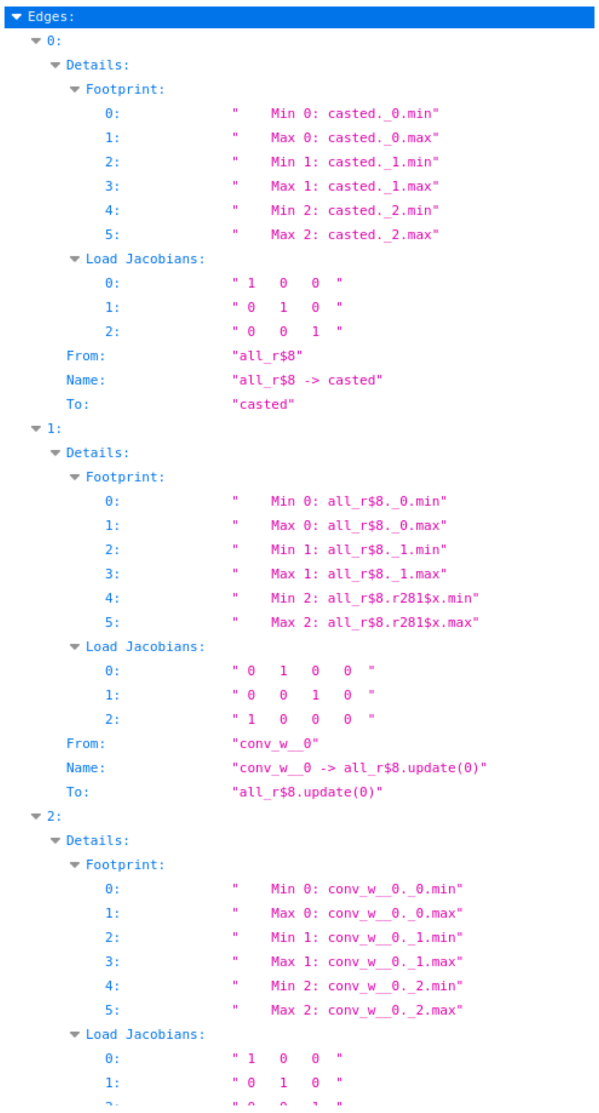
4.1.2 Common Representation

Figure 4.3 shows the programming tree structure of nodes which includes the details such as the memory access patterns and features of each node. Figure 4.4 shows the programming tree structure of edges and the associated parameters of the edges.



```
  Memory access patterns:
     0:                        "      Pointwise:      1 0 0 1"
     1:                        "      Transpose:      1 0 0 1"
     2:                        "      Broadcast:      1 0 0 1"
     3:                        "      Slice:          1 0 0 1"
  Op histogram:
     0:                        "      Constant:   0"
     1:                        "      Cast:       0"
     2:                        "      Variable:   6"
     3:                        "      Param:      0"
     4:                        "      Add:        0"
     5:                        "      Sub:        0"
     6:                        "      Mod:        0"
     7:                        "      Mul:        0"
     8:                        "      Div:        0"
     9:                        "      Min:        0"
     10:                       "      Max:        0"
     11:                       "      EQ:         0"
     12:                       "      NE:         0"
     13:                       "      LT:         0"
     14:                       "      LE:         0"
     15:                       "      And:        0"
     16:                       "      Or:         0"
     17:                       "      Not:        0"
     18:                       "      Select:     0"
     19:                       "      ImageCall:  0"
     20:                       "      FuncCall:   1"
     21:                       "      SelfCall:   0"
     22:                       "      ExternCall: 0"
     23:                       "      Let:        0"
  Region computed:
     0:                        "    casted._0.min, casted._0.max"
     1:                        "    casted._1.min, casted._1.max"
     2:                        "    casted._2.min, casted._2.max"
  Stage 0:
     0:                        "    _0 casted._0.min casted._0.max"
     1:                        "    _1 casted._1.min casted._1.max"
     2:                        "    _2 casted._2.min casted._2.max"
  Symbolic region required:
     0:                        "    casted._0.min, casted._0.max"
     1:                        "    casted._1.min, casted._1.max"
     2:                        "    casted._2.min, casted._2.max"
  Name:                        "casted"
1:
  Details:
```

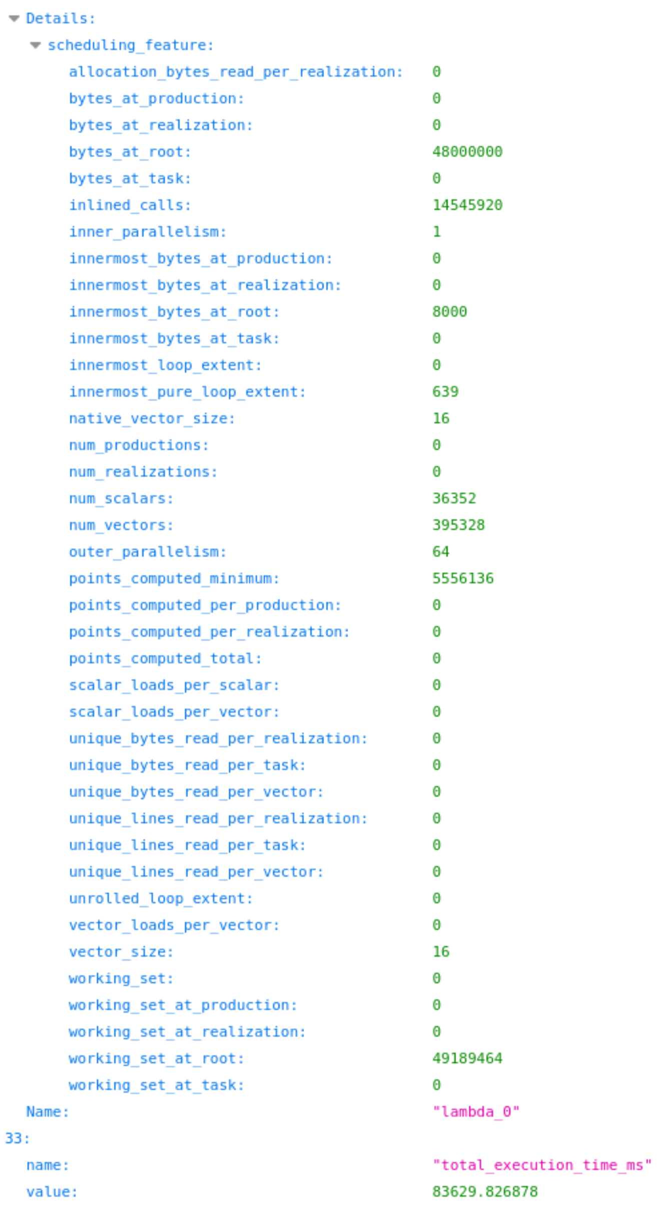
Figure 4.3: Tree Structure for Nodes



```
Edges:
  0:
    Details:
      Footprint:
         0:                "    Min 0: casted._0.min"
         1:                "    Max 0: casted._0.max"
         2:                "    Min 1: casted._1.min"
         3:                "    Max 1: casted._1.max"
         4:                "    Min 2: casted._2.min"
         5:                "    Max 2: casted._2.max"
      Load Jacobians:
         0:                " 1   0   0  "
         1:                " 0   1   0  "
         2:                " 0   0   1  "
    From:                  "all_r$8"
    Name:                  "all_r$8 -> casted"
    To:                    "casted"
  1:
    Details:
      Footprint:
         0:                "    Min 0: all_r$8._0.min"
         1:                "    Max 0: all_r$8._0.max"
         2:                "    Min 1: all_r$8._1.min"
         3:                "    Max 1: all_r$8._1.max"
         4:                "    Min 2: all_r$8.r281$x.min"
         5:                "    Max 2: all_r$8.r281$x.max"
      Load Jacobians:
         0:                " 0   1   0   0  "
         1:                " 0   0   1   0  "
         2:                " 1   0   0   0  "
    From:                  "conv_w__0"
    Name:                  "conv_w__0 -> all_r$8.update(0)"
    To:                    "all_r$8.update(0)"
  2:
    Details:
      Footprint:
         0:                "    Min 0: conv_w__0._0.min"
         1:                "    Max 0: conv_w__0._0.max"
         2:                "    Min 1: conv_w__0._1.min"
         3:                "    Max 1: conv_w__0._1.max"
         4:                "    Min 2: conv_w__0._2.min"
         5:                "    Max 2: conv_w__0._2.max"
      Load Jacobians:
         0:                " 1   0   0  "
         1:                " 0   1   0  "
```

Figure 4.4: Tree Structure for Edges

Figure 4.5 shows the scheduling data output tree structure of the program which contains the details of the scheduling parameters and the execution time value of the particular schedule.



```
▾ Details:
  ▾ scheduling_feature:
      allocation_bytes_read_per_realization:   0
      bytes_at_production:                     0
      bytes_at_realization:                    0
      bytes_at_root:                           48000000
      bytes_at_task:                           0
      inlined_calls:                           14545920
      inner_parallelism:                       1
      innermost_bytes_at_production:           0
      innermost_bytes_at_realization:          0
      innermost_bytes_at_root:                 8000
      innermost_bytes_at_task:                 0
      innermost_loop_extent:                   0
      innermost_pure_loop_extent:              639
      native_vector_size:                      16
      num_productions:                         0
      num_realizations:                        0
      num_scalars:                             36352
      num_vectors:                             395328
      outer_parallelism:                       64
      points_computed_minimum:                 5556136
      points_computed_per_production:          0
      points_computed_per_realization:         0
      points_computed_total:                   0
      scalar_loads_per_scalar:                 0
      scalar_loads_per_vector:                 0
      unique_bytes_read_per_realization:       0
      unique_bytes_read_per_task:              0
      unique_bytes_read_per_vector:            0
      unique_lines_read_per_realization:       0
      unique_lines_read_per_task:              0
      unique_lines_read_per_vector:            0
      unrolled_loop_extent:                    0
      vector_loads_per_vector:                 0
      vector_size:                             16
      working_set:                             0
      working_set_at_production:               0
      working_set_at_realization:              0
      working_set_at_root:                     49189464
      working_set_at_task:                     0
  Name:                                        "lambda_0"
▾ 33:
  name:                                        "total_execution_time_ms"
  value:                                       83629.826878
```

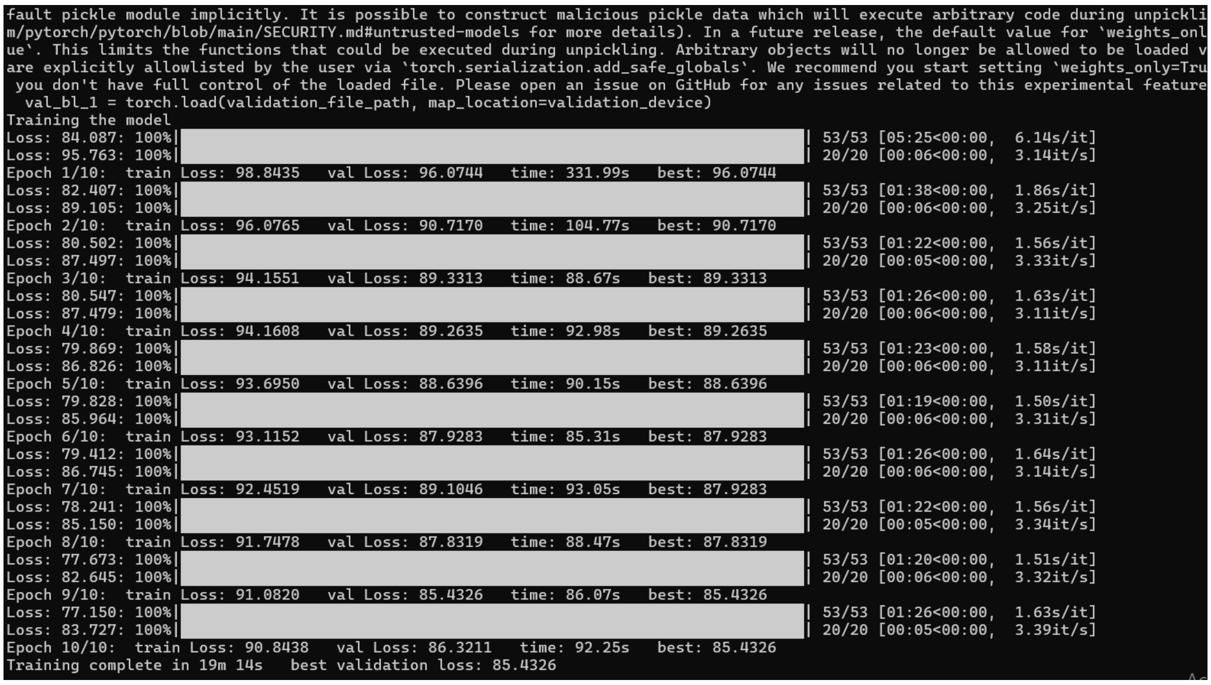
Figure 4.5: Scheduling Data Output Tree Structure

4.1.3 Cost Model

The model is trained on training data which consists of 500 programs with around 60 thousands of possible schedules. The model is validated using the validation data which consists of 125 programs with 20 thousands of possible schedules which are in pickle format. Then the losses for the training and validation data are calculated. The figure 4.6 shows a sample training procedure of the model architecture done for 10 epochs.After training the model, speedups of the programs are calculated for several schedules. Figure 4.7 shows the predicted speedups obtained of a sample program with 98 schedules.

4.1.4 Auto-tuning

As mentioned, we use the Beam search algorithm(1) to efficiently prune the search space. Figure 4.8 shows a sample search space for a program as a list. In this instance, the beam



```
fault pickle module implicitly. It is possible to construct malicious pickle data which will execute arbitrary code during unpickli
m/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for more details). In a future release, the default value for `weights_onl
ue`. This limits the functions that could be executed during unpickling. Arbitrary objects will no longer be allowed to be loaded v
are explicitly allowlisted by the user via `torch.serialization.add_safe_globals`. We recommend you start setting `weights_only=Tru
 you don't have full control of the loaded file. Please open an issue on GitHub for any issues related to this experimental feature
  val_bl_1 = torch.load(validation_file_path, map_location=validation_device)
Training the model
Loss: 84.087: 100%|                                                                    | 53/53 [05:25<00:00,  6.14s/it]
Loss: 95.763: 100%|                                                                    | 20/20 [00:06<00:00,  3.14it/s]
Epoch 1/10:  train Loss: 98.8435   val Loss: 96.0744   time: 331.99s   best: 96.0744
Loss: 82.407: 100%|                                                                    | 53/53 [01:38<00:00,  1.86s/it]
Loss: 89.105: 100%|                                                                    | 20/20 [00:06<00:00,  3.25it/s]
Epoch 2/10:  train Loss: 96.0765   val Loss: 90.7170   time: 104.77s   best: 90.7170
Loss: 80.502: 100%|                                                                    | 53/53 [01:22<00:00,  1.56s/it]
Loss: 87.497: 100%|                                                                    | 20/20 [00:05<00:00,  3.33it/s]
Epoch 3/10:  train Loss: 94.1551   val Loss: 89.3313   time: 88.67s   best: 89.3313
Loss: 80.547: 100%|                                                                    | 53/53 [01:26<00:00,  1.63s/it]
Loss: 87.479: 100%|                                                                    | 20/20 [00:06<00:00,  3.11it/s]
Epoch 4/10:  train Loss: 94.1608   val Loss: 89.2635   time: 92.98s   best: 89.2635
Loss: 79.869: 100%|                                                                    | 53/53 [01:23<00:00,  1.58s/it]
Loss: 86.826: 100%|                                                                    | 20/20 [00:06<00:00,  3.11it/s]
Epoch 5/10:  train Loss: 93.6950   val Loss: 88.6396   time: 90.15s   best: 88.6396
Loss: 79.828: 100%|                                                                    | 53/53 [01:19<00:00,  1.50s/it]
Loss: 85.964: 100%|                                                                    | 20/20 [00:06<00:00,  3.31it/s]
Epoch 6/10:  train Loss: 93.1152   val Loss: 87.9283   time: 85.31s   best: 87.9283
Loss: 79.412: 100%|                                                                    | 53/53 [01:26<00:00,  1.64s/it]
Loss: 86.745: 100%|                                                                    | 20/20 [00:06<00:00,  3.14it/s]
Epoch 7/10:  train Loss: 92.4519   val Loss: 89.1046   time: 93.05s   best: 87.9283
Loss: 78.241: 100%|                                                                    | 53/53 [01:22<00:00,  1.56s/it]
Loss: 85.150: 100%|                                                                    | 20/20 [00:05<00:00,  3.34it/s]
Epoch 8/10:  train Loss: 91.7478   val Loss: 87.8319   time: 88.47s   best: 87.8319
Loss: 77.673: 100%|                                                                    | 53/53 [01:20<00:00,  1.51s/it]
Loss: 82.645: 100%|                                                                    | 20/20 [00:06<00:00,  3.32it/s]
Epoch 9/10:  train Loss: 91.0820   val Loss: 85.4326   time: 86.07s   best: 85.4326
Loss: 77.150: 100%|                                                                    | 53/53 [01:26<00:00,  1.63s/it]
Loss: 83.727: 100%|                                                                    | 20/20 [00:05<00:00,  3.39it/s]
Epoch 10/10:  train Loss: 90.8438   val Loss: 86.3211   time: 92.25s   best: 85.4326
Training complete in 19m 14s   best validation loss: 85.4326
```

Figure 4.6: Training of the LSTM cost model

size is set to 32. After we pruned the search space, we had 32 schedules with less predicted runtime at the final transformation applied. Figure 4.9 shows the best 32 schedules with predicted runtime(in each row, the last element is runtime and the rest of the elements give schedule path).

4.2 Key Outcomes

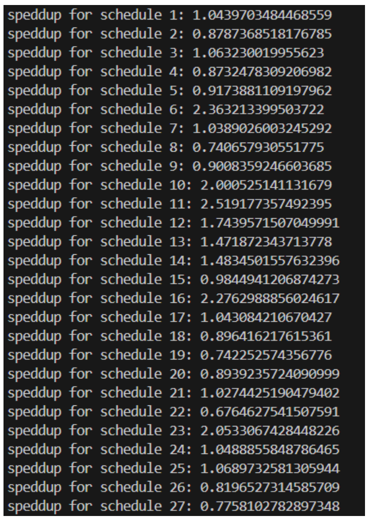
The following are the key outcomes in each of the tasks.

- Common Representation: A tree-based structure for both nodes and edges in programming pipelines enables efficient representation and scheduling of programming and scheduling data.
- Cost Model: An LSTM architecture that predicts the speedup of several schedules applied to a program.
- Auto-tuning approach: A beam search algorithm that searches the global optimal schedule of a random program.
- Common Representation:

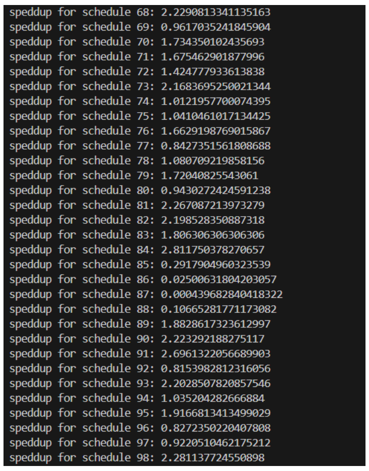
4.3 Percentage completed

We have three main individual tasks, which have been completed as follows:

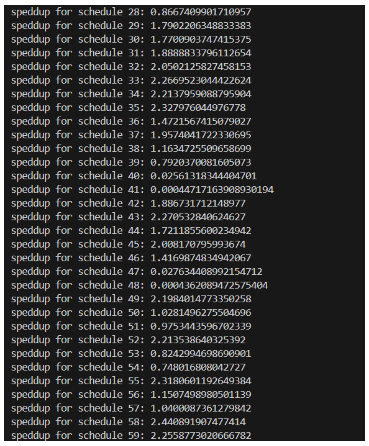
- Common Representation: 80%
- Cost Model Development: 40%



```
speddup for schedule 1: 1.0439703484468559
speddup for schedule 2: 0.8787368518176785
speddup for schedule 3: 1.06323001995562з
speddup for schedule 4: 0.8732478309206982
speddup for schedule 5: 0.9173881109197962
speddup for schedule 6: 2.363213399503722
speddup for schedule 7: 1.0389026003245292
speddup for schedule 8: 0.740657930551775
speddup for schedule 9: 0.9008359246603685
speddup for schedule 10: 2.000525141131679
speddup for schedule 11: 2.519177357492395
speddup for schedule 12: 1.7439571507049991
speddup for schedule 13: 1.471872343713778
speddup for schedule 14: 1.4834501557632396
speddup for schedule 15: 0.9844941206874273
speddup for schedule 16: 2.2762988856024617
speddup for schedule 17: 1.043084210670427
speddup for schedule 18: 0.896416217615361
speddup for schedule 19: 0.742252574356776
speddup for schedule 20: 0.8939235724090999
speddup for schedule 21: 1.0274425190479402
speddup for schedule 22: 0.6764627541507591
speddup for schedule 23: 2.0533067428448226
speddup for schedule 24: 1.0488855848786465
speddup for schedule 25: 1.0689732581305944
speddup for schedule 26: 0.8196527314585709
speddup for schedule 27: 0.7758102782897348
```

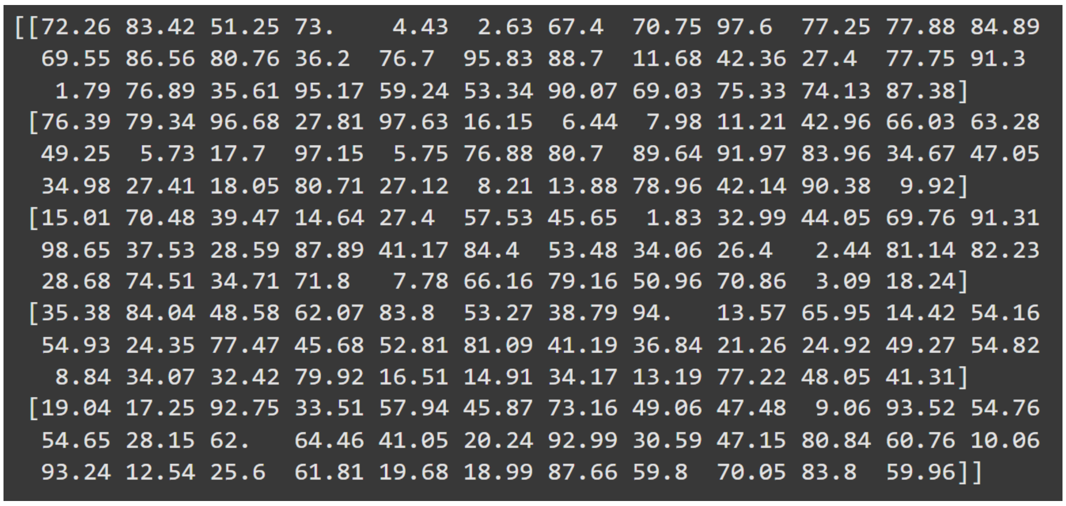


```
speddup for schedule 68: 2.2290813341135163
speddup for schedule 69: 0.9617035241845904
speddup for schedule 70: 1.734350102435693
speddup for schedule 71: 1.675462901877996
speddup for schedule 72: 1.424777933613838
speddup for schedule 73: 2.1683695250021344
speddup for schedule 74: 1.0121957700074395
speddup for schedule 75: 1.0410461017134425
speddup for schedule 76: 1.6629198769015867
speddup for schedule 77: 0.8427351561808688
speddup for schedule 78: 1.080709219858156
speddup for schedule 79: 1.72040825543061
speddup for schedule 80: 0.9430272424591238
speddup for schedule 81: 2.267087213973279
speddup for schedule 82: 2.198528350887318
speddup for schedule 83: 1.806306306306306
speddup for schedule 84: 2.811750378270657
speddup for schedule 85: 0.2917904960323539
speddup for schedule 86: 0.025006318042030 57
speddup for schedule 87: 0.00043968284041832 2
speddup for schedule 88: 0.10665281771173082
speddup for schedule 89: 1.8828617323612997
speddup for schedule 90: 2.223292188275117
speddup for schedule 91: 2.6961322056689903
speddup for schedule 92: 0.8153982812316056
speddup for schedule 93: 2.2028507820857546
speddup for schedule 94: 1.035204282666884
speddup for schedule 95: 1.9166813413499029
speddup for schedule 96: 0.8272350220407808
speddup for schedule 97: 0.9220510462175212
speddup for schedule 98: 2.281137724550898
```



```
speddup for schedule 28: 0.8667409901710957
speddup for schedule 29: 1.7902206348833383
speddup for schedule 30: 1.7700903747415375
speddup for schedule 31: 1.8888833796112654
speddup for schedule 32: 2.0502125827458153
speddup for schedule 33: 2.2669523044422624
speddup for schedule 34: 2.2137959088795904
speddup for schedule 35: 2.327976044976778
speddup for schedule 36: 1.4721567415079027
speddup for schedule 37: 1.9574041722330695
speddup for schedule 38: 1.1634725509658699
speddup for schedule 39: 0.7920370081605073
speddup for schedule 40: 0.02561318344404701
speddup for schedule 41: 0.00044717163908930194
speddup for schedule 42: 1.886731712148977
speddup for schedule 43: 2.270532840624627
speddup for schedule 44: 1.7211855600234942
speddup for schedule 45: 2.008170795993674
speddup for schedule 46: 1.4169874834942067
speddup for schedule 47: 0.027634408992154712
speddup for schedule 48: 0.00043620894725754 04
speddup for schedule 49: 2.1984014773350258
speddup for schedule 50: 1.0281496275504696
speddup for schedule 51: 0.9753443596702339
speddup for schedule 52: 2.213538640325392
speddup for schedule 53: 0.8242994698690901
speddup for schedule 54: 0.748016808042727
speddup for schedule 55: 2.3180601192649384
speddup for schedule 56: 1.1507498980501139
speddup for schedule 57: 1.0400087361279842
speddup for schedule 58: 2.440891907477414
speddup for schedule 59: 2.2558773020666782
```

Figure 4.7: Obtained speedups for various schedules of a program

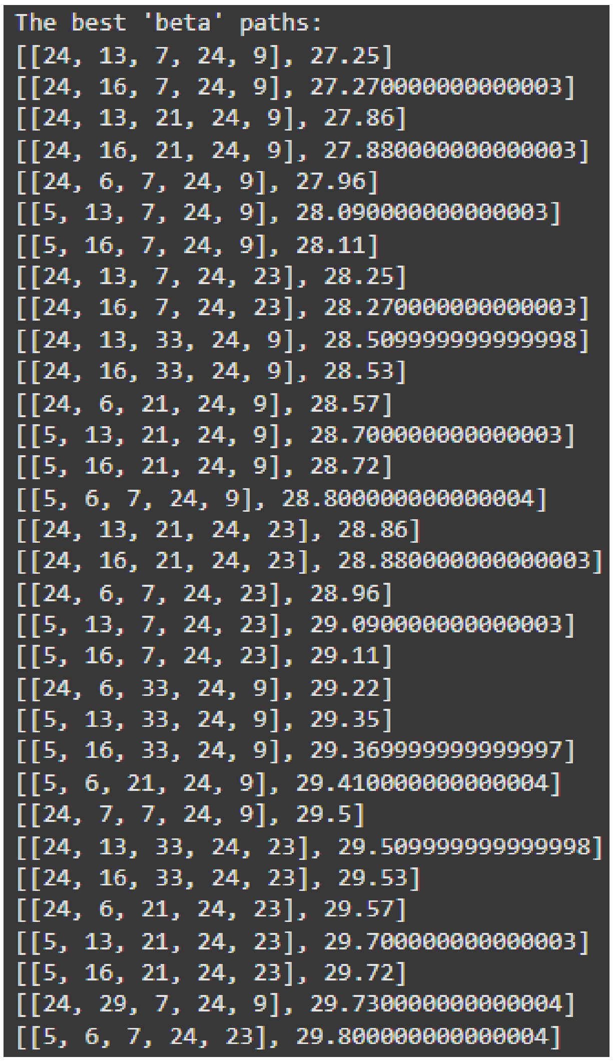


```
[[72.26 83.42 51.25 73.    4.43  2.63 67.4  70.75 97.6  77.25 77.88 84.89
  69.55 86.56 80.76 36.2  76.7  95.83 88.7  11.68 42.36 27.4  77.75 91.3
   1.79 76.89 35.61 95.17 59.24 53.34 90.07 69.03 75.33 74.13 87.38]
 [76.39 79.34 96.68 27.81 97.63 16.15  6.44  7.98 11.21 42.96 66.03 63.28
  49.25  5.73 17.7  97.15  5.75 76.88 80.7  89.64 91.97 83.96 34.67 47.05
  34.98 27.41 18.05 80.71 27.12  8.21 13.88 78.96 42.14 90.38  9.92]
 [15.01 70.48 39.47 14.64 27.4  57.53 45.65  1.83 32.99 44.05 69.76 91.31
  98.65 37.53 28.59 87.89 41.17 84.4  53.48 34.06 26.4   2.44 81.14 82.23
  28.68 74.51 34.71 71.8   7.78 66.16 79.16 50.96 70.86  3.09 18.24]
 [35.38 84.04 48.58 62.07 83.8  53.27 38.79 94.   13.57 65.95 14.42 54.16
  54.93 24.35 77.47 45.68 52.81 81.09 41.19 36.84 21.26 24.92 49.27 54.82
   8.84 34.07 32.42 79.92 16.51 14.91 34.17 13.19 77.22 48.05 41.31]
 [19.04 17.25 92.75 33.51 57.94 45.87 73.16 49.06 47.48  9.06 93.52 54.76
  54.65 28.15 62.   64.46 41.05 20.24 92.99 30.59 47.15 80.84 60.76 10.06
  93.24 12.54 25.6  61.81 19.68 18.99 87.66 59.8  70.05 83.8  59.96]]
```

Figure 4.8: A sample search space as list

- Auto Tuning: 40%

Overall, we have completed approximately 55% of the entire project.



```
The best 'beta' paths:
[[24, 13, 7, 24, 9], 27.25]
[[24, 16, 7, 24, 9], 27.270000000000003]
[[24, 13, 21, 24, 9], 27.86]
[[24, 16, 21, 24, 9], 27.880000000000003]
[[24, 6, 7, 24, 9], 27.96]
[[5, 13, 7, 24, 9], 28.090000000000003]
[[5, 16, 7, 24, 9], 28.11]
[[24, 13, 7, 24, 23], 28.25]
[[24, 16, 7, 24, 23], 28.270000000000003]
[[24, 13, 33, 24, 9], 28.509999999999998]
[[24, 16, 33, 24, 9], 28.53]
[[24, 6, 21, 24, 9], 28.57]
[[5, 13, 21, 24, 9], 28.700000000000003]
[[5, 16, 21, 24, 9], 28.72]
[[5, 6, 7, 24, 9], 28.800000000000004]
[[24, 13, 21, 24, 23], 28.86]
[[24, 16, 21, 24, 23], 28.880000000000003]
[[24, 6, 7, 24, 23], 28.96]
[[5, 13, 7, 24, 23], 29.090000000000003]
[[5, 16, 7, 24, 23], 29.11]
[[24, 6, 33, 24, 9], 29.22]
[[5, 13, 33, 24, 9], 29.35]
[[5, 16, 33, 24, 9], 29.369999999999997]
[[5, 6, 21, 24, 9], 29.410000000000004]
[[24, 7, 7, 24, 9], 29.5]
[[24, 13, 33, 24, 23], 29.509999999999998]
[[24, 16, 33, 24, 23], 29.53]
[[24, 6, 21, 24, 23], 29.57]
[[5, 13, 21, 24, 23], 29.700000000000003]
[[5, 16, 21, 24, 23], 29.72]
[[24, 29, 7, 24, 9], 29.730000000000004]
[[5, 6, 7, 24, 23], 29.800000000000004]
```

Figure 4.9: Best schedules

Chapter 5

Conclusion

We have aimed to develop a unified framework for optimizing tensor programs in Domain-Specific Languages. This framework focuses on addressing the limitations of existing cost models by introducing a common representation, a machine learning-based performance-predicting cost model, and enhanced auto-tuning strategies. Significant progress has been made toward achieving these objectives, including presenting a common representation, developing the cost model, and enhancing the autotuning algorithms of the unified framework.

Throughout the project, we encountered challenges such as the complexity of scaling cost models across diverse hardware architectures and integrating machine learning techniques into existing DSL workflows. These challenges were addressed by creating robust training datasets, implementing efficient feature extraction methods, and iteratively refining the framework based on empirical evaluations.

In the next phase, we will focus on completing the implementation of the unified framework, conducting extensive testing across diverse scenarios, refining its adaptability to various hardware architectures, integrating advanced search algorithms for auto-tuning, and evaluating the framework's performance in real-world applications. Finally we will combine individual tasks to build the unified framework.

This project reaffirms its significance by addressing a critical gap in optimizing tensor programs and enabling cross-DSL compatibility. The progress achieved so far aligns well with the initial objectives and goals. We remain confident in our ability to complete the remaining tasks, delivering a transformative solution for performance prediction of tensor programs in DSLs.

Bibliography

[1] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, “Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines,” *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 365–377, 2013.

[2] R. Baghdadi, J. Ray, E. Del Sozzo, M. Ben Romdhane, P. Suriana, S. Kamil, and S. Amarasinghe, “Tiramisu: A code optimization framework for high performance systems,” *arXiv preprint arXiv:1804.10694*, 2018, arXiv:1804.10694.

[3] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. USA: Prentice Hall Press, 2009.

[4] A. Adams, K. Ma, L. Anderson, R. Baghdadi, T.-M. Li, M. Gharbi, B. Steiner, S. Johnson, K. Fatahalian, F. Durand, and J. Ragan-Kelley, “Learning to optimize halide with tree search and random programs,” *ACM Transactions on Graphics*, vol. 38, no. 4, 2019.