A Deep Learning Based Cost Model for Automatic Code Optimization in Tiramisu

Thesis · October 2020

CITATIONS
0

READS
591

5 authors, including:

Massinissa Merouani
Ecole Nationale Supérieure d'Informatique
1 PUBLICATION **0** CITATIONS
SEE PROFILE

Mohamed-Hicham Leghettas
Ecole Nationale Supérieure d'Informatique
1 PUBLICATION **0** CITATIONS
SEE PROFILE

Riyadh Baghdadi
Massachusetts Institute of Technology
27 PUBLICATIONS **288** CITATIONS
SEE PROFILE

Taha Arbaoui
Université de Technologie de Troyes
23 PUBLICATIONS **38** CITATIONS
SEE PROFILE

People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research

École nationale Supérieure d'Informatique
ex. INI (Institut National de formation en Informatique)
National School of Computer Science
Algiers, Oued Smar

Graduation Thesis

A Deep Learning Based Cost Model for Automatic Code Optimization in Tiramisu

Authors:

Massinissa Merouani
fm_merouani@esi.dz

Mohamed-Hicham Leghettas
fm_leghettas@esi.dz

Supervisors:

Riyadh Baghdadi
CSAIL, MIT

Taha Arbaoui
LOSI, UTT

Karima Benatchba
LMCS, ESI

This thesis is submitted in partial fulfillment of the requirements of the
Engineering Degree in Computer Science, Computer Systems

October 2020

Acknowledgements

First of all, we would like to express our gratitude to our project supervisor Dr. Riyadh Baghdadi who offered us the opportunity to work on this project. His expertise, experience and patience were invaluable throughout the duration of this work. We thank him again for his efforts to allow us to work with the COMMIT team at the Massachusetts Institute of Technology.

Our special thanks go to our adviser Prof. Karima Benatchba for her precious time and guidance. Her advice and experience were valuable and of great importance.

We would like also to send special and sincere thanks to our project supervisor Dr. Taha Arbaoui for his expertise, involvement and patience. His continuous support and advice were indispensable throughout this work.

We thank the members of the jury for their interest and for taking the time to evaluate this work.

Many thanks go to the COMMIT team at the Massachusetts Institute of Technology. We would like to thank particularly Prof. Saman Amarasinghe for his time and advice, and for welcoming us to the team.

We would also like to acknowledge the teachers and staff of our university : l'École nationale Supérieure d'Informatique.

Finally, we must express our very profound gratitude to our parents for providing us with unfailing support and continuous encouragement throughout our years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them.

Thank you.

Abstract

Programmers spend a lot of time and effort optimizing their code to make it run faster, this has led compiler researchers to focus on developing automatic optimization techniques that allow to automatically improve program performance. Such techniques aim to transform programs to exploit more efficiently the underlying hardware. Efficient implementations of automatic optimization techniques require precise cost models, they evaluate whether applying a sequence of code transformations reduces the execution time of the program. Building an analytical cost model to do so is hard in modern x86 architectures due to the complexity of the microarchitecture. In this work, we present a novel deep learning-based cost model for automatic code optimization. This cost model is integrated into an auto-scheduler that enables Tiramisu compiler to select the best code transformations for a given program. The input of the proposed model is a tree-structured set of high-level features representing the unoptimized code along with a sequence of code transformations, the cost model predicts the speedup expected when the code transformations are applied. To train the model, we built a 1.8 million points dataset consisting of transformed synthetic Tiramisu programs. The proposed model achieves a low error rate of 16% $MAPE$ on predicting speedups. Furthermore, the model is proven effective at ranking code transformation candidates with an $nDCG$ score of 98%. Experiments on real-world image processing, deep learning, and scientific computing programs show that the auto-scheduler finds competitive solutions 95 times faster when it uses the proposed cost model.

Keywords : *Deep Learning, Automatic Code Optimization, High-Performance Computing, Compilers, Auto-Scheduling, Tiramisu Compiler.*

Table of Contents

Acknowledgements iii

Abstract iv

Table of Contents v

List of Figures ix

List of Tables xiii

List of Abbreviations and Notations xv

Introduction 1

1 Project Presentation 5

1.1 Hosting Research Institutions . . . 6

1.2 Background: Code Optimization and TIRAMISU Framework . . . 6

1.2.1 Background on Code Optimization . . . 6

1.2.2 TIRAMISU Framework . . . 10

1.3 Parent Project: TIRAMISU Auto-Scheduler . . . 15

1.4 This Project: TIRAMISU Deep Learning Based Cost Model . . . 16

1.4.1 Problem Statement . . . 16

1.4.2 Scope of This Project . . . 17

1.4.3 Organization of the Project . . . 18

1.5 Conclusion . . . 19

2 Machine Learning Cost Models for Auto-scheduling: A Literature Review 21

2.1 Background on Machine Learning . . . 21

2.1.1 Machine Learning . . . 22

2.1.2 Deep Learning . . . 24

2.2 Machine Learning Based Cost Models: State-of-the-Art . . . 29
2.2.1 Classifying by Objective Types . . . 30
2.2.2 Classifying by Data Acquisition Methods . . . 31
2.2.3 Classifying by Data Representation Approaches . . . 33
2.2.4 Classifying by Model Architectures . . . 35
2.2.5 Classifying by Training Policies . . . 38
2.2.6 Classifying by Evaluation Methodologies . . . 39
2.2.7 Synthesis of the Literature Review . . . 42
2.3 Conclusion . . . 43

3 Design and Implementation of the TIRAMISU Cost Model 45

3.1 Dataset Construction . . . 45
3.1.1 Design Discussion . . . 46
3.1.2 The Design of the Synthetic Programs' Generator . . . 47
3.1.3 Dataset Construction Workflow . . . 55
3.1.4 Dataset Construction Results . . . 59
3.1.5 Assessing the Execution Time's Imprecision . . . 61
3.2 Program Characterization . . . 63
3.2.1 Design Discussion . . . 63
3.2.2 The Proposed Program Characterization Approach . . . 64
3.2.3 A Detailed Characterization Example . . . 68
3.3 Model Architecture . . . 70
3.3.1 Design Discussion . . . 70
3.3.2 The Recursive LSTM Architecture . . . 71
3.3.3 Model Architecture Details and Training Methodology . . . 74
3.3.4 Other Explored Architectures . . . 75
3.4 Implementation Details . . . 77
3.5 Conclusion . . . 78

4 Tests and Evaluation 79

4.1 Model-Only Evaluation . . . 80

4.1.1 Used Metrics: Definition and Discussion . . . 80
4.1.2 Description of the Test Set . . . 81
4.1.3 Overall Scores of the Cost Model . . . 82
4.1.4 Comparing the Predicted and the Measured Speedups . . . 83
4.1.5 Visualizations of the Performances of the Cost Model . . . 85
4.2 Exploration+Model Evaluation . . . 86
4.2.1 Used Benchmark Programs . . . 87
4.2.2 Schedule Exploration Module . . . 89
4.2.3 Results of the Model-Guided Exploration . . . 90
4.3 Design Alternatives Comparison . . . 93
4.4 Conclusion . . . 94

Conclusion **95**

A Assessing the Execution Time's Imprecision: Experiment Details **99**

B Normalized Discounted Cumulative Gain: Further Explanation and Examples **101**

C Results Details of the Model-Guided Exploration **103**

Bibliography **105**

List of Figures

1.1 Tiling impact on the memory access of the array *a* . . . 7

1.2 Parallelization of a DOALL loop . . . 8

1.3 Loop interchange example . . . 8

1.4 Memory layout of a two dimensional array in C++ . . . 9

1.5 Loop unrolling with a factor of 8 . . . 9

1.6 Loop fusion of two loop nests . . . 10

1.7 TIRAMISU's four-layer design [Baghdadi et al., 2019] . . . 11

1.8 A simple TIRAMISU program . . . 12

1.9 A C/C++ loop nest equivalent to *computation A* from Figure 1.8 . . . 13

1.10 A scheduled generalized matrix multiplication in TIRAMISU and its equivalent pseudo code . . . 14

1.11 Overall design of the TIRAMISU Auto-scheduler . . . 16

1.12 Overall project organization and interaction between the different modules . . . 19

2.1 The interaction between the agent and the environment in reinforcement learning [Sutton and Barto, 2018] . . . 24

2.2 The relation between Deep Learning and different AI disciplines [Goodfellow et al., 2016] 25

2.3 A simplified modelisation of a feedforward neural networks showing the function computed during the forward pass . . . 26

2.4 Typical structure of a sequential convolutional neural network [Baldominos et al., 2018] 27

2.5 Basic implementation of the RNN cell . . . 27

2.6 The implementation of the LSTM cell . . . 28

2.7 The implementation of the GRU cell . . . 28

2.8 Halide's cost model architecture . . . 36

2.9 Ithemal system architecture . . . 36

2.10 Ithemal DAG-RNN architecture . . . 37

2.11 PolyScientist model . . . 38

2.12 Actual vs predicted runtimes for 100 random programs . . . 40

2.13 Heatmap for measured and predicted throughput values under Ithemal model for basic blocks with measured throughput values less than 1000 cycles for the Haswell micro-architecture . . . 40

2.14 Rank vs. Regression objective function for the GBT- and TreeGRU-based models . . . 41

2.15 fdtd-2d evaluation . . . 41

2.16 Performance of code variants for Resnet-50 layers . . . 42

3.1 Synthetic TIRAMISU program generator overview . . . 47

3.2 TIRAMISU code for zero initialization . . . 48

3.3 Example of simple assignment TIRAMISU computation . . . 48

3.4 Stencil examples . . . 49

3.5 Example of stencil (Horizontal Blur) . . . 49

3.6 Matrix multiplication in TIRAMISU . . . 50

3.7 Convolution implementation in TIRAMISU . . . 50

3.8 Example of TIRAMISU algorithm where all computation are fused in the innermost loop 51

3.9 TIRAMISU implementation of the "*convolution relu maxpooling*" algorithm . . . 51

3.10 The tree representation of the "*convolution relu maxpooling*" algorithm presented in Figure 3.9 . . . 52

3.11 A random synthetic program example . . . 54

3.12 Examples of the impact that code transformations make on the tree representation of the TIRAMISU program presented in Figure 3.11. parallelization and unrolling are intentionally not represented because they don't change the structure. . . . 54

3.13 Overall process of the dataset construction . . . 55

3.14 Overall structure of the dataset file. . . . 56

3.15 An example TIRAMISU program . . . 57

3.16 Program annotation for the example presented in Figure 3.15. . . . 57

3.17 Schedule annotation for the example presented in Figure 3.15. . . . 58

3.18 Execution time measurement's procedure. . . . 58

3.19 Our characterization of a typical program. . . . 64

3.20 Loop nest representation example . . . 65

3.21 Right-hand side assignment representation example . . . 66

3.22 Left-hand side assignment representation example 66

3.23 Loop transformation representation example 67

3.24 An example input TIRAMISU program. 69

3.25 The full representation of the program presented in Figure 3.24. 69

3.26 Examples of two trees representing TIRAMISU programs 71

3.27 Processing an example input program through the cost model's architecture. 72

3.28 Loop embedding unit 72

3.29 Processing a 3-computation TIRAMISU program with the feedforward architecture . . . 75

3.30 Processing the program presented in Figure 3.27a through the three layers of the RNN-based cost model. 76

3.31 Processing the program presented in Figure 3.27a through the three layers of the Tree-LSTM based model. 76

4.1 Predicted speedups compared to measured speedups over 100 × 32 synthetic programs. The speedups are ordered in ascending order. 83

4.2 Measured vs Predicted speedup on 16 random programs from the test set, each blue dot represents a schedule with respect to its measured speedup and its predicted speedup. 84

4.3 Cumulative distribution of the MAPE error over the test set. Reading example: 67% of the test set has an error <16.6% 85

4.4 Distribution of the error rates over measured speedups and execution times. 85

4.5 Visualizing the ranking performance of the cost model 86

4.6 Heat simulation stencils 88

4.7 Example of converting an RGB image to grayscale 89

4.8 Example of applying *blur* to an image 89

4.9 Comparison of the auto-scheduling results using Beam Search method with both the cost model and ground truth measurements. 91

4.10 Comparison of the auto-scheduling results using MCTS method with both the cost model and ground truth measurements. 91

A.1 Relative change of the measured speedups after each run. 99

A.2 Relative change of the measured speedups after each run in comparison to the reference speedups. 100

List of Tables

1.1 Table showing a sample of TIRAMISU scheduling commands (*C* and *P* are computations defined by *i* and *j* loop variables) . . . 13

2.1 Classification of the reviewed cost models . . . 43

3.1 Specifications of the machines on which the execution time of the dataset's programs was measured. . . . 59

3.2 Comparison between our dataset and the existing dataset . . . 60

3.3 A detailed listing of the features that composes the *Computation Vector*. . . . 68

3.4 Processing steps of the program presented in Figure 3.27a through the three layers of the cost model. . . . 73

3.5 Specifications of the NVIDIA DGX-1 server . . . 75

4.1 Description of the synthetic test set. . . . 82

4.2 Overall scores of the cost model on various metrics . . . 82

4.3 Search time improvement compared to performance degradation when using the cost model for auto-scheduling . . . 92

4.4 Comparison between the explored model architectures. . . . 93

B.1 Example of measured and predicted speedups by two cost models and their evaluation 101

B.2 Example of measured and predicted speedups . . . 101

C.1 Schedules found by the different search methods. . . . 104

C.2 The parameters for the search methods for each benchmark . . . 104

Abbreviations and Notations

ANN	–	Artificial Neural Network
API	–	Application Programming Interface
AST	–	Abstract Syntax Tree
BLAS	–	Basic Linear Algebra Sub-programs
BS	–	Beam Search
CPU	–	Central Processing Unit
DAG	–	Directed Acyclic Graph
DL	–	Deep Learning
DNN	–	Deep Neural Network
DSL	–	Domain-Specific Language
FPGA	–	Field-Programmable Gate Array
GBT	–	Gradient Boosted Trees
GCC	–	GNU Compiler Collection
GPU	–	Graphical Processing Unit
GRU	–	Gated Recurrent Unit
LSTM	–	Long Short-term Memory
MAPE	–	Mean Absolute Percentage Error
MCTS	–	Monte Carlo Tree Search
ML	–	Machine Learning
MLP	–	Multilayer Perceptron
nDCG	–	Normalized Discounted Cumulative Gain
NN	–	Neural Network
RNN	–	Recurrent Neural Network
SIMD	–	Single Instruction Multiple Data

Introduction

In modern programming languages, the same algorithm can be implemented in many different ways and yet still perform the same task. However different implementations of the same program may have different performance behaviors. A certain implementation may require more memory space to perform the task, or take a longer time to run, or consume more energy than another.

As a result of the growing complexity of modern computing workloads, such as in Deep Learning, Genomics, and Autonomous Driving, finding which implementation optimizes certain criteria has become a major concern for performance-critical applications. Therefore, research institutions and software companies invest strenuous efforts seeking the optimal implementation of their code. Indeed, a better performing code can be a game-changer in many industrial and scientific areas. For example, in Computational Chemistry, when designing new drugs, researchers need to simulate the protein dynamics of the viral agent. Such simulations are so computationally intensive that in 2020 the demand led to the creation of the world's first exascale computer[1] [Zimmerman et al., 2020]. Highly-optimized implementations of these simulations can be orders of magnitudes faster and can make an impactful difference in speeding up the productivity in drug design.

Complicating the task even more, the performance behavior of the same implementation may differ from a computer to another. In fact, when porting the top performant code version from a machine to another, we cannot always expect it to remain the optimal one. Indeed the performance measure of a program is highly dependent on the underlying hardware. This hardware dependence makes companies spend enormous resources rewriting a part of their codebase when upgrading their computing hardware or when trying to ship their software product to a wider range of consumer electronics.

Traditionally, the laborious task of deeply optimizing programs requires ninja programmers that have expert knowledge on hardware and software interactions. They spend countless hours and numerous attempts searching for the implementation that exploits better capabilities of the target hardware. Such manual optimization generally ends up with a very complicated and unportable code.

Different code implementations can often be expressed as applying a sequence of alterations to the base straightforward algorithm. These alterations are called, in the jargon, code transformations and mainly consist of loop optimizations and data layout transformations such as loop fission, fusion, tilling, unrolling, etc.

[1] Computing systems capable of performing 10^{18} floating-point operations per second (1 exaFLOPS)

Modern compilers solved a part of the problem by offering very convenient ways to apply transformations without having to rewrite the program [Ragan-Kelley et al., 2013, Baghdadi et al., 2019], just by expressing the desired transformations to apply alongside the base algorithm. However, the cumbersome of selecting the right set of transformations that optimizes the performance remains whole. In fact, applying a code transformation to a given program can improve its performance (e.g., reduce its execution time) as well as it can deteriorate it (e.g., make the program run slower). It is difficult to predict whether a code transformation is good or bad as it highly depends on many factors [Epshteyn et al., 2006] including the program's structure, the underlying hardware, and its interactions with the other applied transformations.

Techniques that automatically find the sequence of transformations that yield the best performance have always been a long-standing center of interest in the compiler community. Such techniques, referred to in the literature as auto-tuning or auto-scheduling [Mullapudi et al., 2016, Ashouri et al., 2018], captivate a growing portion of the efforts of compilers and high-performance computing researchers. Auto-scheduling provides greater productivity, smoother portability, and higher performance in computing-intensive areas. It also allows non-expert programmers and domain scientists to write and optimize their own code.

Selecting the best sequence of transformations can be modeled as a search problem that can be solved by leveraging classical search algorithms that explore the search space of applicable transformations [Memeti et al., 2018]. This exploration should be guided by an evaluation function that quantifies the execution time speedup produced by each candidate solution. A straightforward implementation of this evaluation function is to experimentally measure the execution time of the program after applying the candidate sequence of transformations. However, as compiling and running programs take a considerable amount of time, such evaluation renders the exploration of large search spaces infeasible in practice. In addition, the target hardware may not be available at compile time. Another way to evaluate the candidate solution is to use predictive cost models that estimate the speedup without running the program.

Manually designing precise cost models is a very hard task [Trifunovic et al., 2009, Bachir et al., 2013] due to the increasing diversity and complexity of modern hardware architectures (e.g., speculative execution, out-of-order execution, complex memory hierarchies, data prefetching, etc.). Moreover, interactions between transformations twist the problem much more.

In this work, instead of studying the transformations, their effects, and the complex interactions between them and with the hardware, we design a data-driven approach to automatically learn these complexities. We propose a deep learning based cost model that overcomes the previous obstacles, our cost model predicts the effect of code transformations on the execution time of a given program without running it. It takes as an input the untransformed program along with a sequence of code transformations and outputs an estimation of the speedup that these transformations would yield when applied. We design our model for

TIRAMISU [Baghdadi et al., 2019], a domain-specific language embedded in C++ for expressing fast and portable data-parallel algorithms. This cost model will be used in the TIRAMISU auto-scheduler to guide search algorithms through the space of code transformations.

Our contributions can be summarized as follows:

- An extensive literature review where we discuss state-of-the-art automatic optimization techniques that leverage data-driven cost models.
- A novel deep learning based cost model for automatic code optimization that supports full TIRAMISU programs. This cost model is a regression model that accurately predicts the effect of code transformations on a TIRAMISU program. We also provide an operational implementation of this cost model that can be used in conjunction with search techniques.
- A synthetic TIRAMISU program generator whose purpose is to efficiently generate training datasets that can be used when implementing data-driven approaches.
- A large dataset of code transformations applied on TIRAMISU programs that is used for training the proposed cost model.
- A rigorous evaluation of the proposed cost model on real-world use cases.

This manuscript is organized into 4 chapters. In the first chapter, we give a detailed presentation of the problematic addressed throughout this work, we will expose the context, scope, and objectives of the project along with the needed background knowledge. The second chapter is an extensive literature review where we explore and summarize state-of-the-art works that leverage machine learning cost models for automatic code optimization. The third chapter contains a meticulous presentation of our contributions going through the design choices and implementation process. The fourth chapter provides a rigorous evaluation of the proposed cost model through various experiments.

1

Project Presentation

The rising complexity and diversity of modern hardware architectures complicate more and more the task of optimizing programs. It makes the optimizations' quality, in addition to being program-dependent, dependent on the hardware environment. To manually optimize a program, the programmer has to take into account multiple interdependent factors including the program's structure and data-layouts, the interactions between optimizations, the memory hierarchy, the enabled hardware-level optimizations, etc. A growing amount of effort from the compilers community is focused toward building automatic optimization techniques, often referred to as auto-schedulers or auto-tuners [Mullapudi et al., 2016, Ashouri et al., 2018]. These techniques aim to offer an easy way to improve the computing performances across scientific and industrial areas.

The work presented in this manuscript inserts as a core piece into the TIRAMISU Auto-scheduler project. This parent project aims to build a framework that automatically optimizes programs written in TIRAMISU. In short terms, our contribution consists of the design and implementation of a deep learning based cost model that evaluates the effects of code transformations on the execution time of programs.

This chapter provides a detailed presentation of our project. After presenting the host institutions of this project, we will expose the essential background needed to understand the topic, we will go through introducing the TIRAMISU compiler and some basic notions about code optimization. Then, to better appreciate the importance of our work, we will present the parent project on which we are contributing, i.e., the TIRAMISU Auto-scheduler. We conclude this chapter with a detailed presentation of the subproject we are working on, i.e., the TIRAMISU cost model where we include a formal problem statement and a scope delimitation.

1.1 Hosting Research Institutions

The present project is a fruit of a collaboration between three research institutions:

Compilers at MIT (*COMMIT*), U.S.A [1]: A research group at *CSAIL* (Computer Science & Artificial Intelligence Laboratory) at *MIT* (Massachusetts Institute of Technology) that puts focus on finding novel approaches to improve the performance of modern computer systems without unduly increasing the complexity faced by application developers, compiler writers, or computer architects.

Laboratoire des Méthodes de Conception des Systèmes (*LMCS*), Algeria [2]: A research laboratory at École nationale Supérieure d'Informatique d'Alger (*ESI*) that focuses on building methods (Optimization, Knowledge Representation, Classification, Robustness, etc.) for system design (Information, Embedded, Hypermedia, Communication and Recognition systems)

Logistics and Optimization of industrial Systems (*LOSI*), France [3]: A research laboratory at Université de technologie de Troyes (*UTT*) that works on developing decision-making tools to optimise the performance of complex systems, from the design phase through to the operational phase, in terms of efficiency and competitiveness.

1.2 Background: Code Optimization and TIRAMISU Framework

In this section, we will give an overview on code optimizations with the aim of introducing concepts that are required to define the scope of this project. Additionally, we will provide quick insights on the TIRAMISU compiler, its design, and its syntax that is judged mandatory to understand the rest of this manuscript.

1.2.1 Background on Code Optimization

In the most general form, code optimization refers to the attempts that a compiler makes to produce a new code that is better than the original one without changing the meaning of the compiled program. Usually better means faster, but other objectives may be desired, such as shorter code or a code that consumes less power [Aho, 2006]. Nowadays, code optimizations are increasing in importance and complexity. They are more complex because processor architectures have become more complex, yielding more opportunities to improve the

[1] https://www.csail.mit.edu/research/commit-group
[2] http://lmcs.esi.dz/
[3] https://recherche.utt.fr/logistics-and-optimization-of-industrial-systems-losi

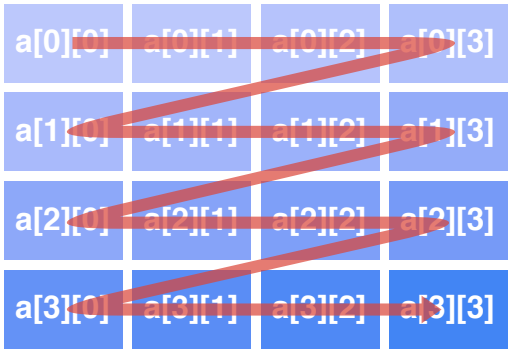
way code executes. They are more important because massively parallel computers require substantial optimization, or their performance suffers by orders of magnitude [Aho, 2006].

There may be an infinite number of possible code transformations to optimize a given program [Aho, 2006]. Some may achieve better performances than others, and some may even harm the performance, hence the need for carefully selecting code transformations to apply.

Since nested loops always take up the majority of computing time in most compute-intensive programs [Liu et al., 2019], it is more fruitful to explore code transformations targeting loop nests. In this section, we present the most common code transformations.

Tiling Tiling is a program transformation that is used to improve the spatial and/or temporal memory locality of a loop nest by changing its iteration order and by controlling the granularity of its parallel execution. Tiling reduces the synchronization overhead, the communication overhead, the cache coherency traffic, the number of external memory accesses, the amount of memory required to execute a loop nest, or the number of cache misses. Tiling adds some control overhead because the number of loops is doubled, and reduces the amount of parallelism available in the outermost loops. The n initial loops are replaced by n outer loops used to enumerate the tiles, and n inner loops used to execute all the iterations within a tile (Figure 1.1). This reordering should not modify the program semantics. Hence, several issues are linked to tiling as to any other program transformations. Tiling can be applied again, recursively or hierarchically, to increase locality at the different cache levels (L0, L1, L2, L3,...) and even at the register level by using very small tiles compatible with the number of registers [Irigoin, 2011].

```
1 for (i=1; i<n; i++)
2 for (j=1; j<m; j++)
3     a[i][j] = 0;
4
```



(a) Order of memory accesses before tiling

```
1 for (i0=1; i0<n; i+=t1)
2 for (j0=1; j0<m; j+=t2)
3 for (i1=i0; i1<min(i0+t1, n); i1++)
4 for (j1=j0; j1<min(j0+t2, m); j1++)
5     a[i1][j1] = 0;
6
```

(b) Order of memory accesses after tiling

Figure 1.1: Tiling impact on the memory access of the array a

Parallelization Parallelization refers to the problem of finding parallelism in a perfect nest of sequential loops. It typically deals with transformations that change the execution order of iterations of such a nest L by creating a different nest L', such that L' is equivalent to L and contains parallel loops [Banerjee, 2011]. A parallel loop is a loop whose iterations are executed concurrently by several threads or processes. The parallel processors execute the same code region, namely, the loop body, but with different data [Wismüller, 2011]. Figure 1.2a represents a doall loop (a parallel loop where the iterations are completely independent) executed in parallel by four processors. Figure 1.2 depicts one possible iteration domain distribution over the four processors.

```
1 for (i=1; i<101; i++)
2     a[i] = sin(PI*i/100);
3
```

(a) A DOALL loop

Processor 1:	1	2	···	25
Processor 2:	26	27	···	50
Processor 3:	51	52	···	75
Processor 4:	76	77	···	100

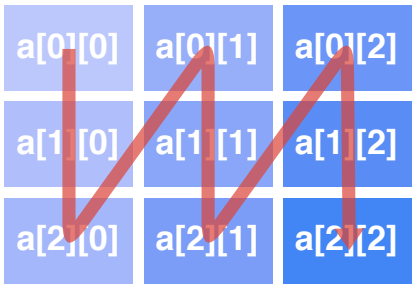
(b) One possible distribution of the iterations over 4 processors

Figure 1.2: Parallelization of a DOALL loop

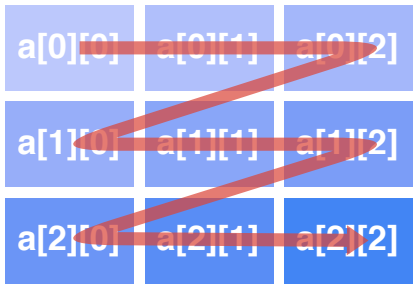
Interchange Loop interchange consists of reordering nested loops. This loop optimization is mainly used to maximize parallelism: relocates the loop that is the most profitable to parallelize so that it is the outermost. On the other hand, establishing an adequate loop order optimizes inner-most memory access. For instance, many programming languages like C and C++, access arrays in row-major order. Interchanging loop nest (Figure 1.3b) facilitates row-by-row access. The new access matches the memory layout (Figure 1.4), and therefore improves data locality.

```
1 for (j=0; j<m; j++)
2     for(i=0; i<n; i++)
3         a[i][j] = b[i][j] + c[i][j];
4
```

```
1 for(i=0; i<n; i++)
2     for (j=0; j<m; j++)
3         a[i][j] = b[i][j] + c[i][j];
4
```



(a) Order of memory accesses before interchange



(b) Order of memory accesses after interchange

Figure 1.3: Loop interchange example

Unrolling The loop unroll transformation (Figure 1.5) is primarily intended to increase register exploitation and decrease memory loads and stores per operation within an iteration of

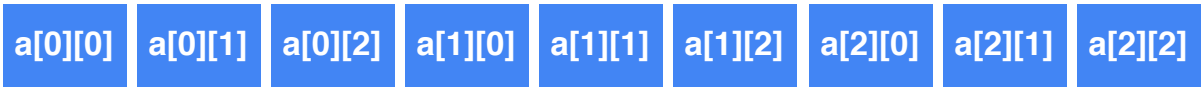


Figure 1.4: Memory layout of a two dimensional array in C++

a nested loop. Improved register usage decreases the need for main memory accesses and allows better exploitation of certain machine instructions. Unroll factor values are supposed to align data references in the innermost loop on cache boundaries. As a result, references to the consecutive memory regions in the innermost loop can have very high cache hit ratios. To illustrate this situation, let us assume that the code in Figure 1.5 is executed in a CPU with cache line size of 64 bytes : unroll factors of 9 or 10 may not be good choices because most array element sizes are either 4 bytes or 8 bytes. Therefore, an unroll factor of 8 or 16 is more likely to effectively exploit cache line reuse for the references that access consecutive memory regions.

```
1 for (i=0; i<n; i++)
2     a[i] = 2 * b[i] + 1;
3
```

(a) before unrolling

```
1  for (i=0; i<n; i+=8){
2      a[i] = 2 * b[i] + 1;
3      a[i+1] = 2 * b[i+1] + 1;
4      a[i+2] = 2 * b[i+2] + 1;
5      a[i+3] = 2 * b[i+3] + 1;
6      a[i+4] = 2 * b[i+4] + 1;
7      a[i+5] = 2 * b[i+5] + 1;
8      a[i+6] = 2 * b[i+6] + 1;
9      a[i+7] = 2 * b[i+7] + 1;
10     }
11
```

(b) after unrolling

Figure 1.5: Loop unrolling with a factor of 8

Fusion Loop fusion or loop jamming is an optimization that seeks reusing the data across two independent loops. Considering two loops that calculate the minimum and the maximum of a vector (Figure 1.6), the data locality can be improved by fusing them to a single loop. If the value of N is large enough that the array does not fit in the cache, the original loop suffers from conflict misses that are minimized by the fused loop [Srikant and Shankar, 2002].

Vectorization Vectorization is a hardware optimization that converts a scalar implementation that processes one pair of operands at a time, to a vector implementation that processes in parallel many pairs. Modern conventional computers, including specialized supercomputers, often have vector operations (SIMD operations) that perform n assignment operations simultaneously.

```
min = a[0];
for (i=0; i<n; i++)
    if (a[i]<min)
        min = a[i];

max = a[0];
for (i=0; i<n; i++)
    if (a[i]>max)
        max = a[i];

```

(a) before fusion

```
min = a[0];
max = a[0];
for (i=0; i<n; i++){
    if (a[i]<min)
        min = a[i];
    if (a[i]>max)
        max = a[i];
    }

```

(b) after fusion

Figure 1.6: Loop fusion of two loop nests

1.2.2 TIRAMISU Framework

TIRAMISU[4] [Baghdadi et al., 2019] is a polyhedral compiler designed to express fast and portable data-parallel programs. TIRAMISU is used to write applications for image processing, deep learning, and scientific computing. It currently generates high-performance code for shared-memory CPUs, GPUs, FPGAs, and distributed systems. TIRAMISU offers an easy way to apply code transformations through high-level commands without changing the original core algorithm.

TIRAMISU the Polyhedral Compiler

The polyhedral model is a compilation model based on an algebraic representation of programs that involves loop nests, and array manipulations. Polyhedral compilers are capable of performing complex loop nest restructuring on static-control, regular loop nests and general data-dependent control-flow program parts [Benabderrahmane et al., 2010]. Polyhedral techniques can work at the granularity of their elements, i.e., at the granularity of a loop iteration and instance of a statement, and at the granularity of an array element. This allows the polyhedral model to construct complex sequences of loop transformations, therefore this model is considered a powerful framework for automatic optimization.

TIRAMISU is an open source DNN (Deep Neural Network) compiler that can optimize sparse DNNs. It can optimize cyclic data-flow graph computations which makes it suitable for sequential data processing in deep learning classes: RNN, LSTM, GRU, etc. On image processing benchmarks, TIRAMISU matches or outperforms state-of-the-art compilers [Baghdadi et al., 2019], thanks to its affine transformation capabilities and dependence analysis. It can even express non-rectangular loop nests in domains where an iterator condition depends on other iterator values.

[4] http://tiramisu-compiler.org

The four-level intermediate representation of the compiler separates architecture independent algorithms from loop transformations, data layout, and communication. As a result, the same high-level code can be ported through multiple architectures: end users can obtain a highly optimized code without having to bear the burden of the heterogeneity of the target environment or the correctness of the optimization.

TIRAMISU API

TIRAMISU has two main qualities: flexible algorithm representation and a rich set of code transformations commands, both yielded from the polyhedral background. TIRAMISU takes advantage of the conceptual separation between iteration domains and maps in the polyhedron model to introduce a unique four-layer design (Figure 1.7). In the absence of memory-based dependencies (Layer III commands), the compiler applies user-defined loop optimizations (layer II)to the abstract algorithm (layer I) without any data-layout limitation. The next two sections will further detail the API of the first three layers. The communication management layer is beyond the scope of this manuscript.

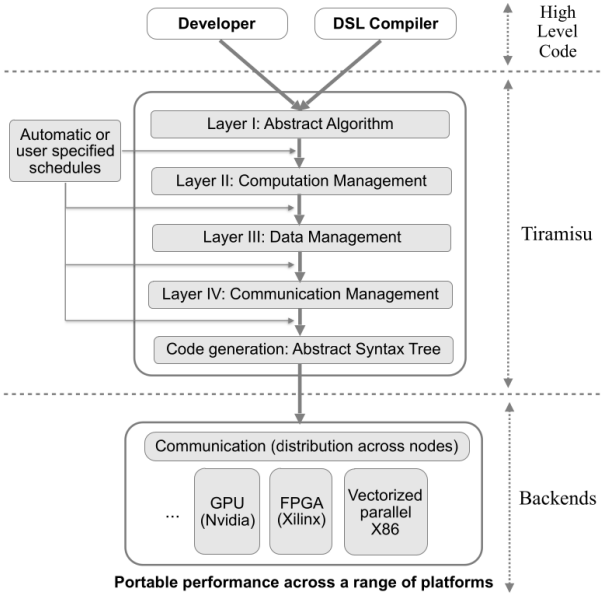


Figure 1.7: TIRAMISU's four-layer design [Baghdadi et al., 2019]

A simple use case. The common use case of TIRAMISU is that its API is utilized to produce an object file which is linked later to a C++ program. For illustration, the program presented in Figure 1.8 is an implementation in TIRAMISU of a function called *function_mult*. Running this program will generate an object file named *generated_code.o* (line 22) that will contain the compiled version of that function which can later be invoked from a C++ program that links the object file. Line 6 of the example program sets the name of the function to *function_mult* using the instruction *tiramisu::init*. Line 8 declares two constant scalars *N* and *M*. Those scalars are used to define loop iterator bounds and buffer sizes. Loop iterators are defined in line 10. Usually we declare a *var* (a loop iterator) and then use it for the declaration of

computations. The range of that variable defines the range of the loop around the computation (its iteration domain). When used to declare a buffer, it defines the buffer size, and when used with an input, it defines the input size. Line 12 declares an input object *B* to represent the input passed to the function *function_mult*. Computations are expressions associated with iteration domains. A computation indicates "what" needs to be computed without any specification of the execution order or the data-layout.The computation *A* is equivalent to the C/C++ statement in Figure 1.9. Buffers defined in lines 16 and 17 are equivalent to arrays in C/C++ language. Buffers are supposed to represent input arguments to functions like *buf_B* or to store computation results : *buf_A*. When the computation results are intended to be function outputs, the associated buffer is declared with the argument *a_output*. If the computation's results are temporary, the argument *a_temporary* should be used instead. The *store_in* function stipulates where computations and inputs should be stored in memory. It maps the computation to the data layout (buffers).

```
1  #include "tiramisu/tiramisu.h"
2  using namespace tiramisu;
3
4  int main(int argc, char **argv)
5  {
6      tiramisu::init("function_mult");
7
8      constant N("N", 128), M("M", 64);
9
10     var i("i", 0, N), j("j", 0, M);
11
12     input B("B", {i,j}, p_int32);
13
14     computation A("A", {i,j}, 2 * B(i, j));
15
16     buffer buf_B("buf_B", {N,M}, p_int32, a_input);
17     buffer buf_A("buf_A", {N,M}, p_int32, a_output);
18
19     B.store_in(&buf_B);
20     A.store_in(&buf_A);
21
22     tiramisu::codegen({&buf_B, &buf_A}, "generated_code.o");
23
24     return 0;
25 }
```

Figure 1.8: A simple TIRAMISU program

TIRAMISU algorithms A TIRAMISU algorithm is a pure function that has inputs, outputs, and is composed of a sequence of computations bound via producer-consumer relationships. A computation is defined by a statement and an iteration domain. The algorithm provides a compact functional description of what needs to be computed at each point in its iteration

```
1 for(i=0; i<N; i++)
2     for (j=0; j<M; j++)
3         A[i][j] = 2 * B[i][j];
4
```

Figure 1.9: A C/C++ loop nest equivalent to *computation A* from Figure 1.8

domain without specifying the order in which different points in the domain are computed, or the order of intermediate computations necessary to produce these points. As an example, lines 11 and 13 in Figure 1.10a define the formula of the Generalized Matrix Multiplication (*gemm*) which is $C \leftarrow \alpha AB + \beta C$.

TIRAMISU schedules A schedule is a high-level strategy for mapping an abstract algorithm to a parallel machine. More concretely, a schedule is an ordered sequence of code transformations describing how to globally optimize the algorithm's execution. TIRAMISU offers a scheduling language that enables programmers to easily apply code transformations without rewriting the core program. Based on the user-defined schedule, TIRAMISU generates an optimized code targeting the architecture in question. Table 1.1 shows some examples of scheduling commands that are judged related to this project. The result of applying a schedule to an algorithm is a transformed program (or a scheduled program). By way of illustration, the schedule presented in lines 15 to 20 of Figure 1.10a, produces the optimized code of Figure 1.10b.

Commands for loop nest transformations	
Command	**Description**
C.tile(i, j, t1, t2, i0, j0, i1, j1)	Tile the loop levels i and j by $t1 \times t2$
C.interchange(i, j)	Interchange levels i and j
C.unroll(i, v)	Unroll level i by v
C.then(P, i)	Fuse the computations C and P at level i
Commands for mapping loop levels to hardware	
C.parallelize(i)	Parallelize level i
C.vectorize(i, v)	Vectorize level i by v

Table 1.1: Table showing a sample of TIRAMISU scheduling commands (C and P are computations defined by i and j loop variables)

```
constant N("N", 2048), M("M", 1024), K("K", 256);
constant ALPHA("ALPHA", 5.0f), BETA("BETA", 6.0f);

var i("i", 0, N), j("j", 0, M), k("k", 0, K);
var i_0("i0"), i_1("i1"), j_0("j0"), j_1("j1");

input A("A", {i, k}, p_float32);
input B("B", {k, j}, p_float32);
input C("C", {i, j}, p_float32);

computation gemm_init("gemm_init", {i, j}, expr(o_mul, BETA, C(i, j)));
computation gemm("gemm", {i, j, k}, p_float32);
gemm.set_expression(gemm(i, j, k) + expr(o_mul, ALPHA,  A(i, k) * B(k, j)));

gemm_init.then(gemm, j);
gemm_init.parallelize(i);
gemm.parallelize(i);
gemm_init.tile(i, j, 64, 128, i_0, j_0, i_1, j_1);
gemm.tile(i, j, 64, 128, i_0, j_0, i_1, j_1);
gemm.unroll(k, 2);
```

(a) GEneralized Matrix Multiplication (*gemm*) in TIRAMISU with scheduling commands

```
parallel (c1, 0, 32) {
for (c3, 0, 8) {
  for (c5, 0, 64) {
    for (c7, 0, 128) {
     b_C[c3*128 + c7 + (c1*64 + c5)*1024] ← b_C[c3*128 + c7 + (c1*64 + c5)*1024]*6.0f
    }
  }
  for (c5, 0, 64) {
    for (c7, 0, 128) {
      for (c9, 0, 128) {
        b_C[c3*128 + c7 + (c1*64 + c5)*1024] ← b_C[c3*128 + c7 + (c1*64 + c5)*1024] +
↪  b_A[c9*2 + (c1*64 + c5)*256]*b_B[c3*128 + c7 + c9*2048]*5.0f
        b_C[c3*128 + c7 + (c1*64 + c5)*1024] ← b_C[c3*128 + c7 + (c1*64 + c5)*1024] +
↪  b_A[c9*2 + (c1*64 + c5)*256 + 1]*b_B[c3*128 + c7 + c9*2048 + 1024]*5.0f
      }
    }
  }
 }
}
```

(b) Equivalent pseudo code of the previous TIRAMISU transformed program. The buffers *b_A*, *b_B* and *b_C* correspond to the inputs *A*, *B*, and *C*, respectively. We omitted the buffer allocations and constant declarations for readability

Figure 1.10: A scheduled generalized matrix multiplication in TIRAMISU and its equivalent pseudo code

1.3 Parent Project: TIRAMISU Auto-Scheduler

The TIRAMISU Auto-Scheduler project aims to build a framework that automatically optimizes TIRAMISU programs in terms of execution speed. This framework takes an unoptimized TIRAMISU program as input and searches for the sequence of code transformations (i.e. schedules) that minimizes its execution time.

Finding the best set of code transformations that minimizes execution time is an NP-hard problem [Stephenson et al., 2003], and the space of candidate solutions is exponentially big [Manseri, 2018]. A certain schedule can yield a performance improvement when applied to a certain program, and yet the same schedule can deteriorate the performance of another [Memeti et al., 2018]. Furthermore, the effects of a schedule on a program may differ between different hardware architectures [Baghdadi et al., 2019]. In fact, the impact of code transformations on a program's performance depends on the program's structure, the interaction between applied transformations, the underlying hardware specifications (memory hierarchy, cache behavior, branch evaluation policy, datapaths, etc.), and other factors that amplify the unpredictability of schedules influence on performance [Xhafa and Dika, 2016, Goss, 2013].

Auto-scheduling can be modeled as a search problem [Memeti et al., 2018, Manseri, 2018] and use search algorithms (such as metaheuristics) to explore the exponential sized space of possible schedules. Such techniques generally require evaluating candidate solutions in order to guide the exploration and select the best schedules, but how to implement such evaluation raises an important challenge when designing auto-schedulers, we will discuss this challenge in the next section (Section 1.4).

The overall design of the Auto-scheduler can be split into two modules as shown in Figure 1.11:

1. **The exploration module** that browses the search space generating new candidate schedules for the input program, this module can be implemented using classical metaheuristics.

2. **The evaluation module** that estimates the effects of the candidate solutions on the execution time of the input program.

The two modules interact with each other through a feedback loop, the search module sends new generated candidates for evaluation, and the evaluation module sends back the respective estimations that guide the exploration. At the end of the process, when the search modules can no longer generate new candidates, the best schedule found so far is returned by the Auto-scheduler.

Our contribution consists in the design and implementation of a Deep Learning based cost model that serves as a fast and accurate evaluation module for the Auto-scheduler. Two

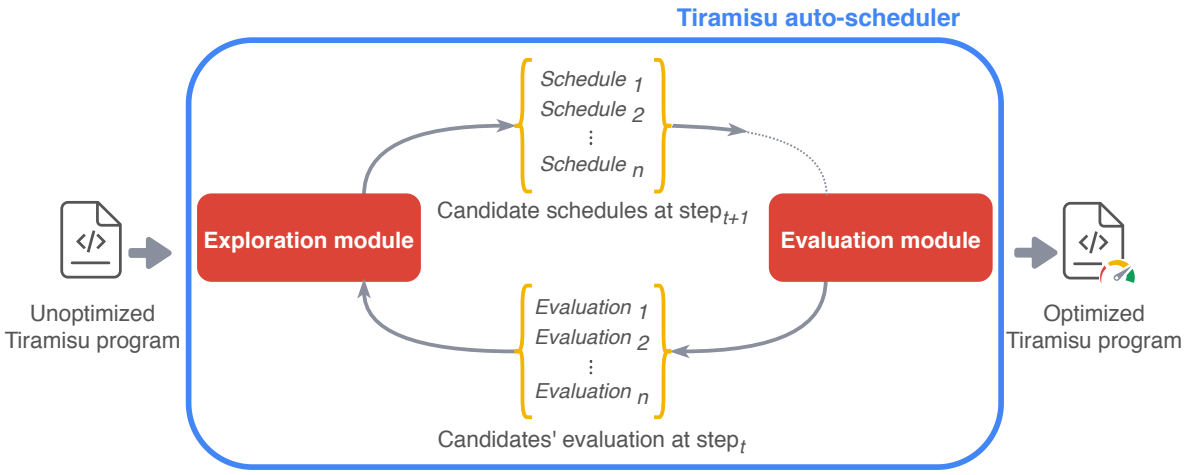


Figure 1.11: Overall design of the TIRAMISU Auto-scheduler

previous works done by ESI [5] students were engaged into this project. The first [Henni and Mekki, 2019] was an early attempt to build a cost model. Their work was restricted to predicting the execution time speedup on the specific class of TIRAMISU programs that contains a single computation. The second [Abdous, 2020] built an exploration module that offers the choice between two search methods (Beam Search and Monte Carlo Tree Search) and integrated the Auto-scheduler into the TIRAMISU Compiler. This latter work was progressing parallelly to ours and we collaborated in order to synchronize our contributions into a fully working Auto-scheduler.

1.4 This Project: TIRAMISU Deep Learning Based Cost Model

In this section, we will provide a detailed description of the issue we are addressing throughout this work, including a formal problem statement, a scope delimitation, and a breakdown of the project into submodules.

1.4.1 Problem Statement

As stated before in Section 1.3, in order to guide a search method through the space of possible code transformations, we need an objective function that is, **(1)** fast enough to allow the exploration of large search spaces and **(2)** accurate enough that it does not depreciate the quality of the search results. This objective function should evaluate the execution time speedup that would be yielded by applying a candidate schedule on a program.

The most straightforward implementation of such an objective function is the experimental evaluation. It consists in measuring and comparing the execution time of the program

[5] École nationale supérieure d'informatique, Algiers

before and after applying each schedule. Such measurements require to compile and run the program after applying each candidate schedule. However, as each compilation and execution consumes a considerable amount of time, such evaluation renders the search algorithm impractical when it comes to exploring thousands of candidate solutions [Ragan-Kelley et al., 2013, Manseri, 2018]. Therefore, we need to rely on predictive models that guarantee precise enough estimations without the computational overhead cost.

A predictive model can be implemented by simulating the interactions between the program and the hardware in order to estimate its execution time, such models are referred to as analytical models in literature [Chen et al., 2006, Renganarayana et al., 2006]. Building such models is known to be a hard and error-prone task that requires profound knowledge [Epshteyn et al., 2006, Trifunovic et al., 2009, Bachir et al., 2013]. Attempts to manually build such estimators often result on low precision, non-portable cost model with a narrow range of support [Di Biagio and Davis, 2018, Mendis et al., 2019, Laukemann et al., 2019].

Instead of studying the code transformations, their effects, and their complex interactions, we propose a data-driven approach to automatically learn those complexities via statistical analysis. This kind of approach currently motivates a growing interest from the compiler community [Adams et al., 2019, Zheng et al., 2020, Cummins et al., 2020]. Concretely, we propose a deep learning-based cost model as an implementation of the objective function. This model takes an unoptimized program and schedule as an input and predicts the execution time speedup that the schedule would yield when applied to the program.

In practice, deep learning (or machine learning in general) models are used to approximate complex functions [Goodfellow et al., 2016] (see Section 2.1 for background about machine learning). In our case, we try to approximate the function that maps the transformed programs to the resulting speedups. Formally, let P denote the set of non-scheduled TIRAMISU programs (TIRAMISU programs with no scheduling commands). Given p in P, we can refer to S_p as the space of schedules that can be applied to p, s_0 from S_p is the special schedule that contains zero transformations. Let's denote that applying a schedule s_i from S_p to the program p results in the transformed program p_{s_i}, hence $p_{s_0} = p$. Let t be the function that measures the execution time of a TIRAMISU program and f the function that returns the execution time speedup yield by applying a schedule s_i to a program p, we can define f as follows:

$$f(p, s_i) = \frac{t(p_{s_0})}{t(p_{s_i})} \tag{1.1}$$

In this work, we propose to use a deep learning model as an approximation of the function f.

1.4.2 Scope of This Project

We design the cost model to support programs that can be expressed by the TIRAMISU language. TIRAMISU is designed for expressing data parallel algorithms, especially those that operate over dense arrays using loop nests and sequences of statements, such programs are

often found in image processing, deep learning, dense linear algebra, tensor operations, and stencil computations.

The schedules supported by the cost model are any combination of loop fusion, loop interchange, loop tiling (2D and 3D), and loop unrolling in this particular order with a wide range of parameters. These transformations are considered to yield the most interesting performance improvements and are the most challenging to predict when combined [Wolf et al., 1996, Herruzo et al., 2004]. Parallelization is considered to be a simpler transformation and the cost model implicitly supports the application of parallelization (see Section 3.1.2).

Although our cost model supports any TIRAMISU program without restrictions on its size (in terms of the number of computations), we exclude the support of some code patterns that are very unlikely to appear in real-world programs (e.g. excessively deep loop nesting, an exorbitant number of memory accesses in a single instruction). We set these restrictions for the sake of better performance and efficiency of the system. While designing the cost model, we made sure that most of these restrictions can easily be extended in case the need arises in future works, details about each restriction are provided in the relevant sections.

1.4.3 Organization of the Project

Following the standard process of building deep learning models, and as illustrated in Figure 1.12, we split the project into the following modules:

Dataset Construction. Data-driven approaches rely on inferring statistical properties from an important collection of data, in our case the data that we need to train our model on is a set of full TIRAMISU programs from where the effects of schedules can be analyzed. As TIRAMISU is relatively new, such dataset does not exist, and there are not enough programs written in TIRAMISU to form a dataset. Therefore, we need to construct our own synthetic dataset. We expose our dataset construction process in Section 3.1.

Program Characterization. Deep learning models cannot operate efficiently over raw program source code [Choi et al., 2017]. We need to design an intermediate representation that will serve as a model-friendly input. Designing a characterization for programs is a significant challenge, due to the program's complex nature. It is not straightforward to map programs into compact representations without losing relevant information. We present our characterization design in Section 3.2.

Model Architecture. The model's architecture defines how the input data is processed and hence how the statistical analysis is performed. It is important to design an architecture that benefits from the input's nature in order to achieve high efficiency, as it is the case for Convolutional networks on images or Recurrent networks on language processing. We describe our model architecture in Section 3.3.

Evaluation. Since there are no general rules to guide the design of deep learning based systems, design choices are mainly guided by intuition and experiments. Therefore the evaluation process is a critical step as it provides feedback over the design choices. As shown in Chapter 4. We propose to evaluate the cost model on both synthetic programs and real-world programs (i.e., benchmarks), we will first evaluate it independently of the auto-scheduler in terms of various accuracy metrics, and then we will evaluate its performance when combined with search algorithms.

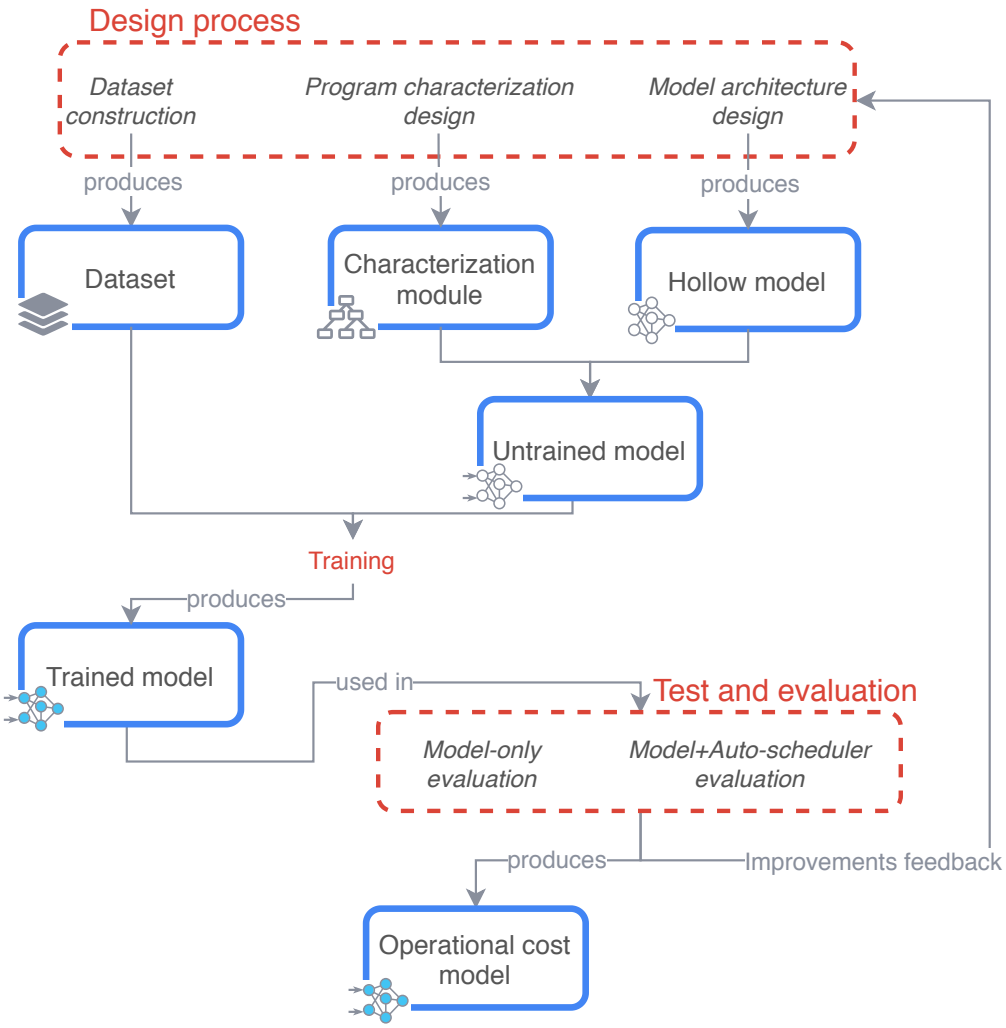


Figure 1.12: Overall project organization and interaction between the different modules

1.5 Conclusion

This chapter provided a presentation of our project detailing its context and objectives. After introducing the three research institutions that hosted our project, we gave the essential background information on code optimization techniques and the TIRAMISU framework.

Code optimizations are transformations that can be applied to a program's source code in order to improve its performance (reduce execution time in our case) without changing its algorithm. TIRAMISU is a Domain Specific Language and a compiler that enables easily applying a wide range of loop transformations to programs without having to rewrite it.

Then to better appreciate the importance of our work, we presented the parent project on which we are contributing: the TIRAMISU Auto-scheduler. This project aims to build a framework that automatically optimizes TIRAMISU programs by finding a sequence of code transformations (a schedule) that minimizes its execution time.

Finally, we concluded this chapter with a detailed presentation of our project including a problem statement, a scope delimitation, and a breakdown into submodules. In short terms, our project consists in building a deep learning based cost model that serves as a fast and accurate evaluation module for the auto-scheduler. Our cost model takes an unoptimized TIRAMISU program along with a schedule as an input and predicts the execution time speedup that the schedule should yield when applied to the program. This project is split into three main submodules: Dataset Construction, Program Characterization, and Model Architecture. Each submodule design and implementation will be detailed in Chapter 3, but before diving into it, we will first provide an extensive literature review in Chapter 2 that will help in understanding the design choices.

2

Machine Learning Cost Models for Auto-scheduling: A Literature Review

Deep learning-powered systems have been able to defeat the best human player in Go, one of the most abstract strategy board games where the number of possible board positions (approximately 2×10^{170}) is vastly greater than the number of atoms in the universe. Playing against a world class Go player requires an immense tactical knowledge and intuition. These deep learning models capture the knowledge of the target field through experience and data. Similarly, evaluating a program optimization among millions of possible variants requires knowledge about the machine architecture, the cache sizes and replacement algorithms, memory layout, etc. Can deep learning models capture these complex behaviors? The affirmative answer will be detailed in this chapter, but before that, we give an overview on Machine Learning. An important portion of this review is dedicated to Deep Learning, an ML subfield building large parametric approximators by composing simpler functions [Bengio et al., 2020].

2.1 Background on Machine Learning

In traditional programming, in order to solve a problem, we manually write a set of rules and instructions that a computer executes. Yet for many modern problems and tasks that we want computers to solve, we no longer know how to explicitly tell a computer how to do it, we may ourselves be good at those tasks, tasks like driving cars or identifying objects in a picture or translating a text from a language to another, but we do not know how directly make a computer perform those tasks. Instead, methods that take a collection of examples and then learn implicit rules from that data seem to be the best approaches to solving these problems. Furthermore, we need systems that can adapt to changing conditions, that can be user-friendly by adapting to the needs of their individual users, and that can improve performance over time.

In the following section, we will give an overview about machine learning starting by introducing the general notions and its taxonomy, then we will put the focus on deep learning by reviewing state-of-the-art models and practices.

2.1.1 Machine Learning

Machine learning is the scientific study of statistical approaches that enable computers to accomplish tasks without the need for explicit human-engineered instructions like in classical algorithms. Instead, it relies on implicit patterns inferred from previous experiences and observations. A Machine Learning algorithm is an algorithm that tunes the parameters θ (by training) of a function h (the model), by fitting them to a set of input examples (training set) in order to make the outputs of the function (the predictions) for a new set of inputs (the test set) optimize a certain criterion J (minimize the prediction error), in other words, a machine learning model can be seen as a function $h_{\hat{\theta}}$ where $\hat{\theta} = \arg\min_{\theta} J(h_{\theta}(D_{training}))$, $D_{training}$ is the training set.

The most quoted formal definition is probably the one given by T. Mitchell in his work:

> "*A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.*" ([Mitchell et al., 1997])

For illustration, we can cite the example of a program that filters spam emails, the task T is to classify emails into two categories spams and not spams. The experience E is a set of labeled emails and the performance P is the ratio of correctly classified emails.

Machine learning algorithms can be classified based on multiple perspectives. In this section, we will describe the machine learning taxonomy that is the most frequent in literature. It consists in classifying machine learning algorithms into three main classes: supervised, unsupervised, and reinforcement learning. According to the type of experiences that the model learns from, each of these classes can be split further based on the type of task the model learns to perform.

a) Supervised learning. Is the learning algorithms that learn from experiences that are composed of both a set of features that characterize the experience and a label or a target. The objective of supervised learning algorithms is to predict, for a given experience, the most likely label that can be associated with that experience depending on its features. Supervised learning algorithms can be classified, according to the type of task they perform, into the following classes:

- **Classification tasks.** In this kind of task, the desired outcome of the algorithm is to predict to which of N categories a given input belongs to. The output can either be discrete representing the most likely category or continuous by giving the probability distribution of belonging to each of the N categories. A good example of a classification task is the handwritten digit recognition were the experiences consist of labeled images of handwritten digits. Then the model is asked to learn to classify an image into ten categories representing the ten decimal digits. Some of the most used classification algo-

rithms are: Support Vector Machines [Cortes and Vapnik, 1995], Decision tree [Belson, 1959], Random forests [Breiman, 2001], K-Nearest Neighbors [Altman, 1992].

- **Regression tasks.** In this case, the learning algorithm is asked to build a mathematical function that fits one of the continuous features of the experiences. In other words, regression tasks consist in predicting a continuous value that should be close to the experience's target. An example of a regression task is the prediction of house prices based on features like the house's location, area, and the number of rooms.

b) Unsupervised learning. Is where the learning algorithm learns from a dataset that consists of unlabeled data. Each experience is characterized only by features and does not have a particular target value. Unsupervised learning generally aims to extract interesting implicit properties of the underlying distribution of the dataset. The algorithm tries to make sense of the data without being guided to how the results should be. The tasks performed by unsupervised learning can be classified as follows:

- **Clustering tasks.** Clustering tasks consist in dividing a population of data samples into several groups so that data points belonging to the same group are similar to each other than to those of other groups. In other words, clustering aims to segregate groups that share the same traits from a population. An example of a clustering application is market segmentation where customers are grouped based on purchase history and interests. This classification would help the company target specific clusters of customers for specific campaigns. Some popular clustering algorithms are K-means [MacQueen et al., 1967] and DBSCAN [Ester et al., 1996].
- **Dimensionality reduction.** It aims to reduce the number of features needed to characterize a data point by replacing the feature set with a smaller set of principal features. In the kind of tasks, the model tries to find the best tradeoff between minimizing the number of features and maximizing the amount of information that can be preserved from the original features set. Dimensionality reduction can be used to improve the computational efficiency of other Machine Learning algorithms since it reduces the size of the original data, it is also used to avoid the curse of dimensionality phenomenon [Keogh and Mueen, 2017]. Some well-known machine learning algorithms for dimensionality reduction are Autoencoders [Rumelhart et al., 1985] and self-organizing maps [Kohonen, 1982].

c) Reinforcement learning. The third paradigm of machine learning is reinforcement learning. In reinforcement learning, the algorithms consist of an autonomous agent that learns to perform a task by trial and error without any guidance from the human operator. The agent interacts with a dynamic environment where it repeatedly has to decide what is the next action to perform [Sutton and Barto, 2018]. For each sequence of decisions, the agent receives either a reward or a penalty depending on how fruitful the results of its actions were. By trying to maximize the reward, the agent learns to make good decisions. A good example to illustrate

these notions is a robot that learns to move around on a terrain that has obstacles. The robot is the agent, its surroundings are the environment, the actions are the directions to take, the robot learns to move by getting a penalty each time it hits an obstacle. Figure 2.1 illustrates the reinforcement learning procedure.

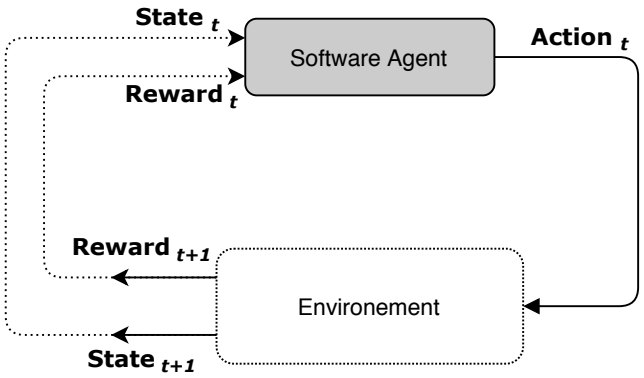


Figure 2.1: The interaction between the agent and the environment in reinforcement learning [Sutton and Barto, 2018]

Some widely used reinforcement learning algorithms are Q-learning [Watkins and Dayan, 1992], SARSA [Rummery and Niranjan, 1994] and Deep Q-learning [Mnih et al., 2013].

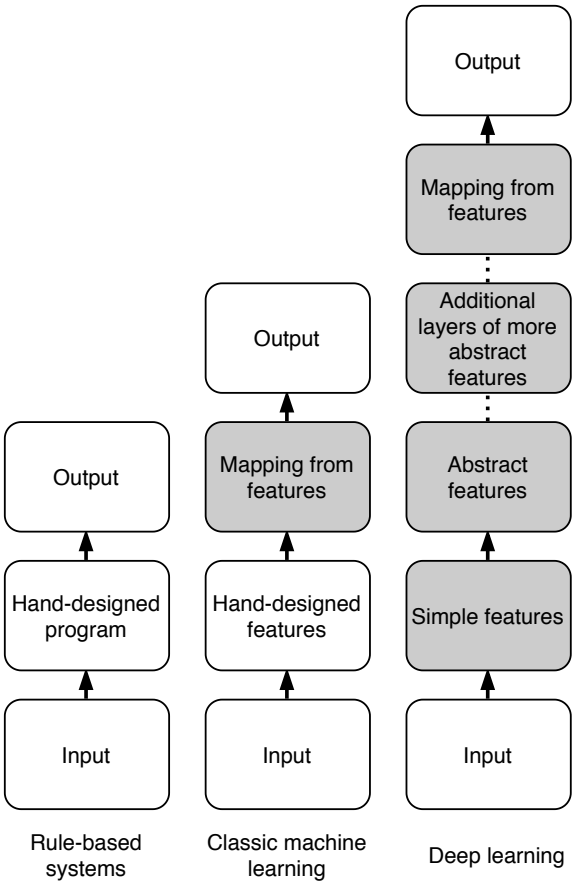
2.1.2 Deep Learning

Deep learning is a particular kind of machine learning that learns to represent the world as a nested hierarchy of concepts. Each concept is defined in relation to simpler concepts, and more abstract representations are computed in terms of less abstract ones [Goodfellow et al., 2016]. It transforms raw data, through multiple layers of representation, to progressively extract higher levels of features. For example in image recognition applications, the lower layer may identify the edges then the next layer may compose the arrangement of edges while the higher levels may identify concepts more relevant to humans such as faces or letters or digits.

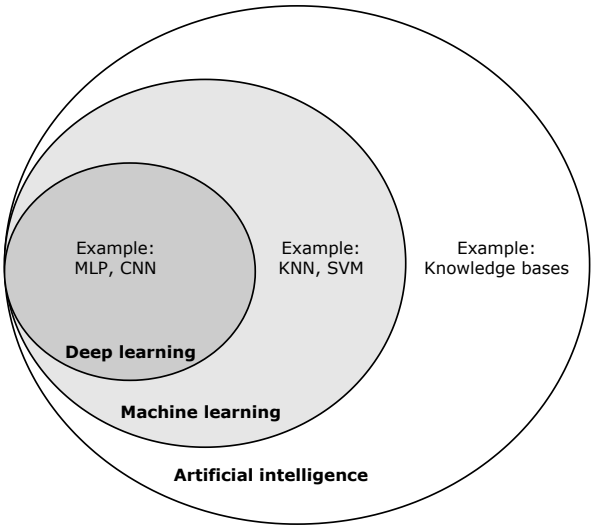
Figure 2.2a illustrates the relationship between Deep Learning and different AI (Artificial Intelligence) disciplines. Figure 2.2b gives a high-level schematic of how each works.

In the following sections, we will explore the main building blocks that are used to construct state-of-the-art Deep Learning models.

a) Feedforward neural networks. Feedforward neural networks, also called multilayer perceptrons, are the quintessential deep learning models. It consists of several neuron-like processing units organized in layers such that the neurons of a layer have direct weighted connections to the neurons of the following layer. These processing units, called artificial neurons or perceptrons, are a loosely imitation of biological neurons in the sense that each unit receives



(a) Flowcharts showing how the different parts of an AI system relate to each other within different AI disciplines. Shaded boxes indicate components that are able to learn from data.



(b) A Venn diagram of AI disciplines

Figure 2.2: The relation between Deep Learning and different AI disciplines [Goodfellow et al., 2016]

its inputs from many other units and computes its own activation value. The activation value of a neuron consists of a linear combination of the previous layer's neurons' activation values and a set of weights that are proper to the neuron. The weights of all the neurons together encode the knowledge of the network. As shown in Figure 2.3, in feedforward neural networks, the information flows through the network from the input to the output layer by layer. Each one can be seen as a vector function f_i that performs multiple linear combinations of its input and a set of weights W_i. The whole network is a composition of all those functions $F = f_{n-1} \circ \ldots \circ f_1 \circ f_0(x)$. Figure 2.3 illustrates a simplified example of how a forward pass is computed in a four-layer feedforward neural network.

Feedforward neural networks are universal function approximators as shown by Cybenko's universal approximation theorem [Cybenko, 1989]. The theorem states that simple neural networks can represent a wide variety of interesting functions when given appropriate parameters. This made them widely used in deep learning models for diverse fields of applications such as image recognition [Hijazi et al., 2015], speech recognition [Deng et al., 2013], and natural language processing [Goldberg, 2016].

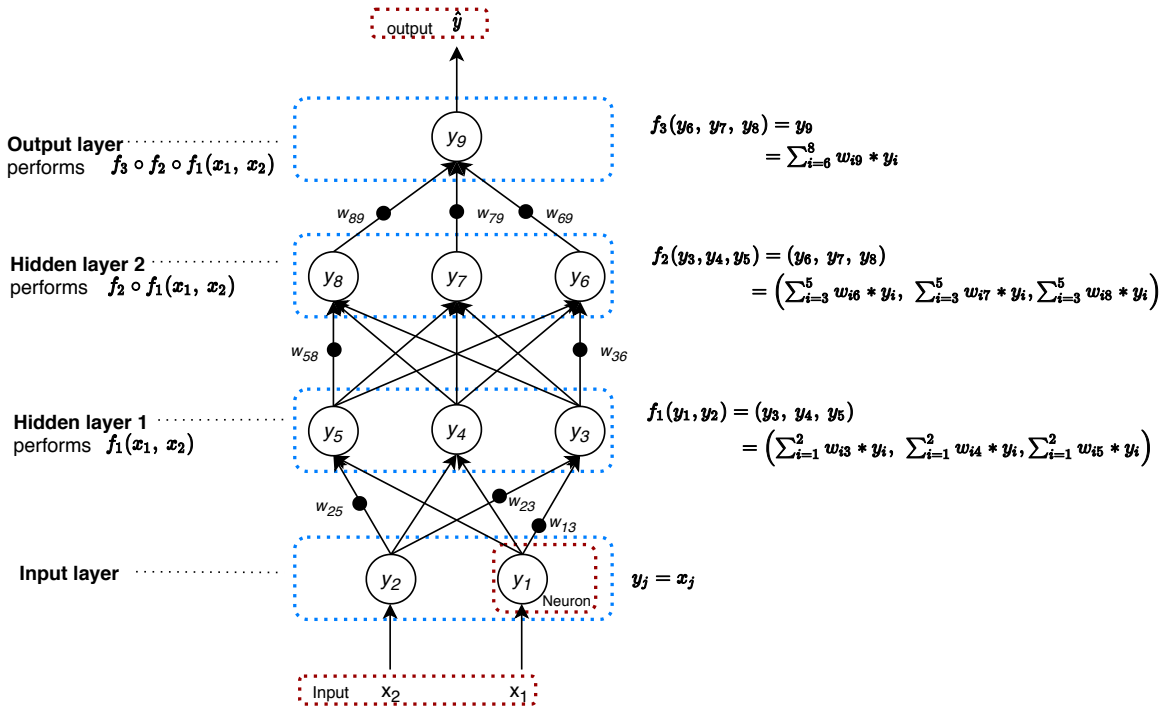


Figure 2.3: A simplified modelisation of a feedforward neural networks showing the function computed during the forward pass

b) Convolutional neural networks. Convolutional neural networks [LeCun et al., 1999] are an adapted version of feedforward neural networks that take advantage of the hierarchical pattern in data. They were inspired by a biological organization of neurons in the animal visual cortex where individual neurons respond to stimuli only in a restricted region of the visual field known as the receptive field. The receptive fields of different neurons partially overlap such that they cover the entire visual field. [Hubel and Wiesel, 1968].

As illustrated in Figure 2.4, in addition to having fully connected neuron layers like feedforward neural networks, Convolutional neural networks use convolution layers and pooling layers. Convolution is a mathematical operation that is a specialized kind of linear operation. It consists of a sliding dot product between the input and a set of weights called filters or kernels. Pooling layers are used to reduce the dimensions of the data by combining multiple adjacent neurons output from the previous layers into a single one by taking their max in case of Max pooling or averaging them in the case of Average pooling.

c) Recurrent neural networks. Recurrent neural networks [Rumelhart et al., 1985] are a type of neural network where previous outputs are fed along with new inputs to the network making information go through a feedback loop. Unlike in feedforward neural networks, where inputs of a layer are independent of its previous outputs, this feedback loop is proven useful for processing sequential data where the information held by each piece of data depend on the whole sequence. For example in natural language processing, the meaning on a word is highly dependant of the sentence where it occurred. This feedback loop adds a memory effect to the network since it allows previous information to persist.

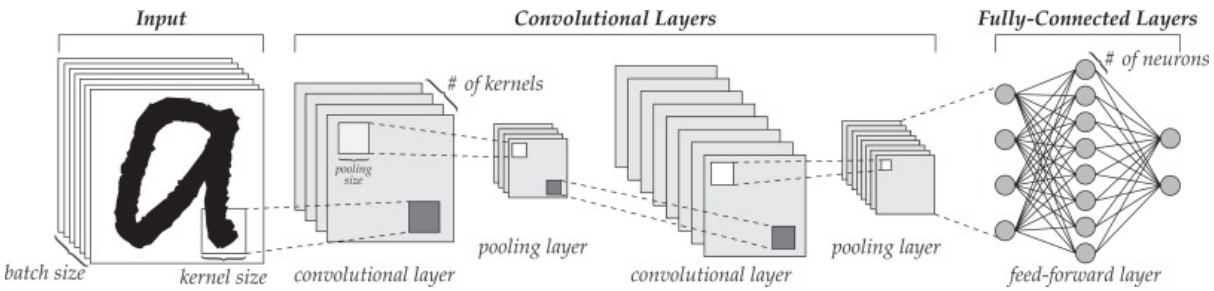


Figure 2.4: Typical structure of a sequential convolutional neural network [Baldominos et al., 2018]

Multiple variants of Recurrent Neural Networks exist, we introduce the most commonly used in real-world applications:

- **Basic Recurrent neural networks.** The basic RNN performs a weighted sum of a new input and its previous output (called hidden state) and bias. The set of weights and the bias are learnable parameters. This simple implementation has the particularity of being the most computationally efficient compared to the other implementations. However, it is also known that it is not very effective for processing long sequences due to the gradient vanishing and exploding problem [Hochreiter et al., 2001].

  Figure 2.5 shows the basic implementation of Recurrent neural networks.

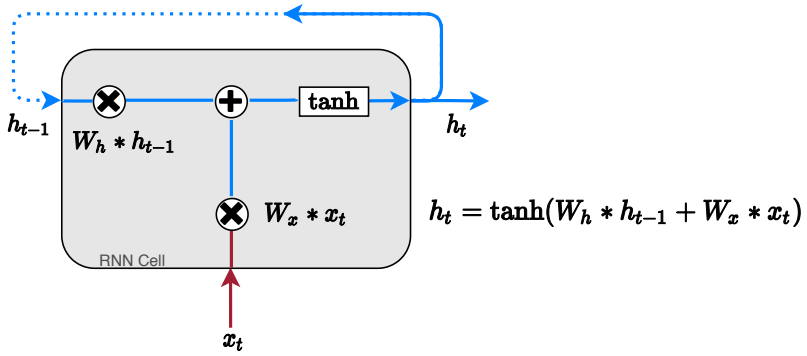


Figure 2.5: Basic implementation of the RNN cell

- **Long Short-term memory.** This implementation of RNN was introduced to deal with the gradient exploding and vanishing problem that limits the uses of basic RNN implementation [Hochreiter and Schmidhuber, 1997]. As shown in Figure 2.6 , a common LSTM unit is composed of a memory cell, an update gate, an output gate and a forget gate. The memory cell remembers values over an arbitrary sequence of steps and the three gates regulate the flow of information into and out of the cell. The LSTM implementation is less computationally efficient than the basic RNN, but it gives the best results when dealing with long sequences.
- **Gated Recurrent Unit.** Gated recurrent unit is an adaptation of LSTM with fewer parameters and thus more computationally efficient [Cho et al., 2014]. GRU's performance was found similar to that of LSTM on certain tasks [Chung et al., 2014]. However, [Weiss

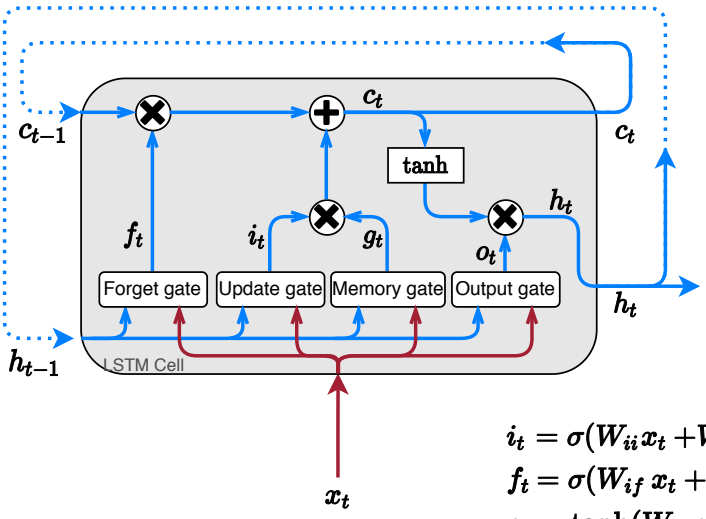


Figure 2.6: The implementation of the LSTM cell

et al., 2018] showed that the LSTM is "strictly stronger" than the GRU as it can easily perform unbounded counting while the GRU cannot.

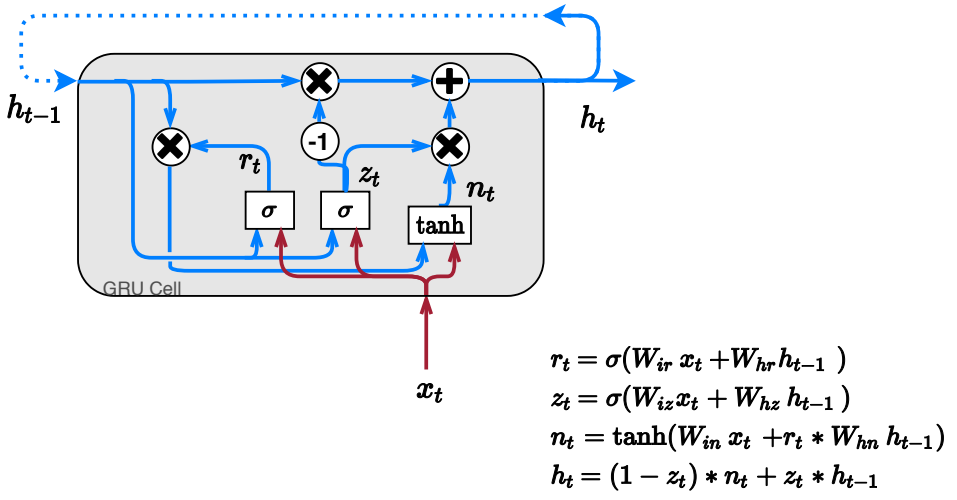


Figure 2.7: The implementation of the GRU cell

2.2 Machine Learning Based Cost Models: State-of-the-Art

With the increasing ability of compilers to implement more code optimizations, the need to establish automatic optimization systems becomes more persistent. These systems (referred to as auto-schedulers) explore many combinations of code optimizations with various parameters by following a search technique. It can be as simple as the Beam Search, or more extensive, such as Monte Carlo Tree Search. No matter how complex the search technique is, evaluating each optimization candidate is required. The evaluation can be done by compiling the program variant and measuring the running time on the target machine. The experimental evaluation is too expensive to compute, on the other hand, manually designing an evaluation cost function that abstracts all program and machine complexities is a tedious task.

Machine learning techniques have been widely used in Combinatorial Optimization problems to replace heavy computations with fast approximations [Bengio et al., 2020]. Similarly, compilation frameworks turn to deep learning to accomplish accurate and robust approximation. The main motivation to use this particular machine learning subfield is that the program characteristics and code optimization are often encoded in vectors or in sequences of vectors, and deep learning excels when dealing with such high-dimensional inputs. However, training deep learning based models requires large datasets.

When designing a new machine learning based cost model for auto-scheduling, there are various conceptual choices that need to be decided. The choices should be defined based on the cost model's goal, the intended task, and the desired performance. We propose to categorize these conceptual choices into six aspects:

- **Objective types:** Covers the different possible tasks of the cost model and its prediction targets.
- **Data acquisition methods:** Describes on what data the model was trained and how the dataset is collected.
- **Data representation approaches:** Covers the different approaches of data characterization.
- **Model's architecture:** Covers the different learning algorithms and neural network architecture that are used for implementing the cost model.
- **Training policies:** Covers the different methodologies that are used for training the cost models.
- **Evaluation methodologies:** Describes how the cost model's performances can be evaluated according to its goal.

We selected these particular aspects amongst others because they offer a good coverage of all the major design choices and they are the most helpful for designing new cost models.

In the following sections, we will review and classify state-of-the-art cost models used in auto-scheduling according to each of the six aspects discussed previously

2.2.1 Classifying by Objective Types

In automatic code optimization, when the need of estimating the effects of schedules on a given code arises, the most accurate way to do it is to take ground truth measurements, this involves compiling and running each candidate code variant on the target hardware as many times as necessary until a stable measure can be taken. And as it sounds. This way is often too expensive to be practical when the number of candidates is considerable, or when the time constraints are tight. The other way to get these estimations is to use approximation models that satisfy the accuracy requirements and are more affordable in terms of time cost. Given the nature of the characteristics that need to be estimated, building accurate analytical models is often very difficult due to the complexity of the function to approximate. This is where machine learning techniques are leveraged since it offers a good trade-off between the accuracy of the approximation, time efficiency, and ease of building.

Machine learning models are often used as cost functions that guide exploration algorithms through the search space of possible code transformations. The most common cost targeted by code optimization is the measured runtime since most works aim to minimize this value. Evaluating it is the most frequent purpose of deep learning models in auto-tuning techniques like in [Adams et al., 2019] and [Rahman et al., 2010].

In Halide's auto-scheduler [Adams et al., 2019], in order to guide the exploration algorithm in the optimization search space, they used an ANN model that predicts the relative throughput (the inverse of recorded runtime) of a given program. This value is computed relatively to the fastest schedule within each batch in order to bring algorithms with very different runtimes into the same dynamic range. This neural network is trained to minimize the L2 error (Mean Squared Error) on that target value. They found that this encourages the network to parse differences between schedules rather than predicting precisely how slow very bad schedules are.

[Rahman et al., 2010] addressed the problem of using a machine learning approach for predicting the distribution of performance improvement when applying the tiling optimization on a given program. They trained a neural network to minimize the mean squared error on predicting the execution time. A quite similar task is performed in [Mendis et al., 2019] where the objective is to predict the number of clock cycles a processor takes to execute a block of assembly instructions, minimizing the absolute percentage error.

Rather than the execution time itself, another frequent prediction target is the relative speedup between different implementations of the same program [Dubach et al., 2007, Ashouri et al., 2016]. The relative speedup of two programs is the ratio between their execution time. This value is used for comparing the effectiveness of different optimizations. Compared to predicting the execution time, this kind of target is useful for reducing the model's complexity since it no longer has to learn to predict the exact execution time of each optimized program but just the relative difference between them.

In [Dubach et al., 2007], a machine learning model is used to predict the speedup yielded when applying a sequence of transformations to a program. The speedup is computed relatively to the execution time of the base program where no optimization is applied. This model is used to predict the speedup distribution of the optimization space of a program after being trained on a random sample. [Ashouri et al., 2016] trained a machine learning model to predict the relative speedup between two optimized programs. This speedup is the expected result of adding one more compiler optimization flag to a sequence of flags. This model is used to attractively select the next best optimization flag to add to a previous sequence of optimization flags.

Another common task for deep learning models in code optimization is ranking a set of code variants. This enables to find the top-performing ones without needing to know a precise performance value like in [Chen et al., 2018] and [Tavarageri et al., 2020].

In [Chen et al., 2018], machine learning models are used to guide the search of the best tensor operator implementations. Since the search algorithm cared only about the relative order of program run times rather than their absolute value, they trained the model using a special rank loss [Burges et al., 2005]. The model outputs a value that can be used for ranking multiple code variants of the same algorithm and selects the top-performing implementations. [Tavarageri et al., 2020] developed a relative ranking algorithm that ranks multiple variants of the same code based on the potential performance of their cache behavior. This algorithm uses a DNN model that performs a relative ranking of two code variants at a time. This model performs a classification-like task by predicting which of the two input code variants is most likely better performing than the other. Using this model, the algorithm performs a tournament-based ranking system to assign a rank to each code variant by playing each code variant against every other code variant. The higher the number of wins of a code variant has, the higher its rank is.

2.2.2 Classifying by Data Acquisition Methods

Machine learning and data-driven approaches are based on inferring abstract relations from a set of data. Hence the dataset composition has a major influence on the model's performance and behavior. The same model may behave differently depending on the dataset on which it was trained on. Therefore choosing the dataset's content induces indirectly choos-

ing the model's behavior. The main choices that need to be done when building the dataset are about the diversity of training data in terms of code patterns and program application domains. Building a dataset with a specific set of programs with patterns drawn from precise real-world use cases will make the model very accurate for that specific use case, and will require a small amount of data for training the model. However, it will likely make the model fail when used for other patterns. On the other hand, opting for a very diverse dataset by including completely random programs may make the model very general but will require very important amounts of training data so that the model can learn from all possible patterns.

[Adams et al., 2019] in their work constructed their training dataset from a random algorithm generator that synthesizes random Halide[1] programs. These programs are composed of computation patterns that are drawn from real-world applications. Each program is a combination of computations randomly selected from a set of computation types commonly used in image processing and deep learning (e.g., convolution, pooling, relu, upsampling ...). These random algorithms did not need to be long or even meaningful since their cost model reasons locally about computations in their immediate context. Therefore it was sufficient to generate algorithms long enough to cover all sorts of situations in which a computation can find itself. Since the model needed to learn the influence of schedules in order to guide the search algorithm, they used this same search algorithm to synthesize a set of schedules for each program by generating candidate schedules by importance sampling paths down the search tree. This approach iteratively focuses the network on the kind of schedules that the exploration algorithm is most likely to visit. Each schedule is benchmarked on the target architecture in order to get the measured runtime needed for training the model. [Chen et al., 2018] used a quite similar approach for building the training dataset by gathering the schedules visited by the search algorithm. Since their model is trained for each new input program, they did not need an initial set of programs for training the model but instead, they used transfer learning to accelerate the optimization by using historical data collected during the optimization of previous input programs.

[Dubach et al., 2007], [Rahman et al., 2010], [Ashouri et al., 2016], and [Tavarageri et al., 2020] made their training set by collecting real-world programs of precise application domains, [Tavarageri et al., 2020] built the training set by gathering different convolution layers of a range of popular state-of-the-art image recognition neural models including Resnet-50 [He et al., 2016], Fast R-CNN [Girshick, 2015], Mask R-CNN [He et al., 2017], Xception [Chollet, 2017]. They used a code generator to create a number of code variants for each program. These variants were generated by applying loop interchange and tiling transformations with different parameters to the original convolutions. [Rahman et al., 2010] for their experiment selected 7 linear algebra benchmarks (doitgen, dgemm, dtrmm, dsyr2k, jacobi2d, fdtd2d, lu). For each benchmark, a sequence of affine loop transformations was applied to make the loop nests fully permutable and hence enable the tilling transformation. The search space of tile sizes is formed by evaluating along each dimension of every benchmark 22 possible tile sizes

[1] Halide is an open-source domain-specific language for the complex image processing pipelines. https://halide-lang.org/

from a defined set of sizes. [Dubach et al., 2007] trained their model on different programs from the UTDSP benchmark suite [Lee and Stoodley, 1998]. For each program, the training set consisted of a set of random transformations applied to the target program. For the training set, the transformations were generated by randomly combining 13 different code transformations into sequence up to a length of 5. Whereas in the evaluation set, the transformations are sequences of length 20 selected from 54 different transformations. [Ashouri et al., 2016] trained their model using a subset of *cBench benchmark suite* [Fursin, 2010] consisting of six different programs. Each program is compiled with different sequences of LLVM [Lattner and Adve, 2004] optimization flags combined in 4 different genes. These genes consist of 30 compiler optimizations in total and 13 unique optimizations. The full dataset is formed by generating different permutations with repetitions and dynamic sequence length leading to a total of 2046 datapoint. The leave-one-benchmark-out cross-validation procedure is used for evaluating the model.

[Mendis et al., 2019] designed the training dataset to include a diverse set of real-world applications from diverse application domains while covering a wide range of x86-64 instructions. It consists of benchmark suites including SPEC2017 [SPEC, 2017] and Polybench-3.1 [Pouchet et al., 2012], as well as end-user applications and libraries used in day to day computing including Linux shared libraries, Firefox, and Open-Office. Each program in the dataset is compiled targeting Intel Haswell architecture, then the basic blocks (sequence of instruction with no branches or jumps) are extracted and executed 100 times in order to reach the steady-state behavior and get a stable measurement. After removing the repeated occurrences of basic blocks the dataset contains about one and a half million unique datapoints.

2.2.3 Classifying by Data Representation Approaches

Another critical choice when building machine learning systems is how the data should be represented to the model, and what are the features of an input program that are the most relevant and sufficient to make accurate predictions about the target characteristics. Opting for representing an input program with a set of low-level featurization, by extracting hand-engineered features, will enable the use of simple models such as classical machine learning models or shallow neural networks that are cost-efficient and can make fast predictions. However, this approach will require a considerable effort in feature engineering in order to build the right set of features. Moreover, it can also make the model more prone to overfitting the dataset if these features are not diverse enough and informative to represent the input space. On the other hand, relying on high-level features can bypass the feature engineering burden since it will be left to the model to build itself the needed abstract features that are the most representative of the input space. But, it will require building deep learning models complex enough to do the feature abstraction. This approach generally requires more important amounts of training data computing resources.

[Adams et al., 2019] opted to use hand-engineered features. Each computation in an algorithm is represented by a set of features that describes the computation itself and the relationship between that computation and the other computations. This set of features can be sorted into two types, algorithm-specific features and schedule-dependent features. Algorithm-specific features, which are invariant to the schedule, consist of histogram operations which counts how many times each basic operation (e.g., add, multiply, subtract, ...) is performed and Jacobian of accesses which describes, for each access made to another computation or input buffer, how the coordinates accessed vary in respect to the loop dimensions of the consumer. These jacobians are used to classify memory accesses into several subcategories (e.g., pointwise operation vs broadcasting). Schedule-dependant features are either event counts (e.g., the number of SIMD vectors computed or the number of parallel tasks launched) or memory footprint characteristics (e.g., the number of bytes touched and the number of contiguous segments of memory touched) making a total of thirty-nine schedule dependent features per computation.

Similarly, [Tavarageri et al., 2020] also opted for hand-engineered features extracted from input programs. Their work focused on ranking different code variants based on the cache behavior of the program. They represented a code variant by its working set sizes at different levels of the memory hierarchy. The used DNN model compares between two code variants at a time. It takes as input the working set size at three cache levels and at the memory level of the two code variants making a total of eight input features.

On the far opposite side, [Mendis et al., 2019] uses very high-level featurization for feeding the input data to the model by just performing some basic restructurations to the raw input code. Ithemal takes a compiled assembly block, disassembles it, and maps it to a list of instructions where each instruction consists of a list of tokens representing its operation code (opcode, like add, mov ..), source operand, and destination operands, separated by delimiter tokens. All constant operands (like integer constants and memory addresses) are mapped to the same token representing constants, then each token is replaced by its embedding which is a real-valued n-dimensional vector, and then fed to the model.

[Chen et al., 2018] tested in their work two distinct statistical models. The first one is a machine learning model based on gradient-boosted trees [Friedman, 2001] that relies on low-level domain-specific features extracted from a given low-level abstract syntax tree. These features include loop structure information (e.g., memory access count and data reuse ratio) and schedule information (e.g., vectorization unrolling thread binding), and context relation features that summarizes useful relations between feature axes (such as loop count vs touched memory size). The second model is a recursive neural network that learns from the abstract syntax tree of the input program hence leveraging the hierarchical loop structure of the input program. Unlike in [Rahman et al., 2010] where they did not need to characterize the input program at all, characterizing only schedules was enough since they used a different model for each input program. The only transformation type that was considered is tiling. As a result, they just needed the tiling parameters as a representation and the model was able to

build abstract features based on the reaction obtained by applying tiling with those particular parameters.

[Dubach et al., 2007] constructed their feature vector from the intermediate representation of the program, meaning that the features are extracted after a compilation pass. The selected feature represents information about the code size, the executed instructions, the availability of parallelism, and memory accesses. In addition to these static features, dynamic features were also collected during the first execution of the base program such as the number of execution of each control-flow structure. The total number of selected features using this method is reduced using principal components analysis from 118 to 10.

[Ashouri et al., 2016] used a dynamic profiling framework for constructing the feature vector of the optimized programs. The selected profiling framework provides a high-level Microarchitectural Independent Characterization of Applications (MICA) [Hoste and Eeckhout, 2007]. This framework reports data on instruction type, memory and register access pattern, potential instruction-level parallelism, and a dynamic control flow analysis in terms of branch predictability. A total of 99 different features are extracted and then reduced to 10 using principal components analysis.

2.2.4 Classifying by Model Architectures

After describing the characterization of the programs fed to the cost models, we detail in the following section each cost model's architecture. The architectures range from a shallow feedforward neural network handling 3-dimensional vectors to a Directed Acyclic Graph Recurrent Neural Networks (DAG-RNNs) [Shuai et al., 2015] operating over hierarchically structured representations.

[Adams et al., 2019] feed to their cost model two sets of hand-designed features extracted from Halide programs: Algorithm-specific features and schedule-specific features as described in Section 2.2.3. The schedule features have a large dynamic range and they are most naturally combined multiplicatively. Therefore, they are first processed through a *log* layer which is proved to be superior to the standard MLP for unbounded non-linear function approximation [Hines, 1996]. The output of the log layer and the algorithm features are both fed to two independent fully connected layers then concatenated to form a larger vector of size 32. This hidden vector is refined through the output layer and a *relu* activation function (Figure 2.8). Instead of predicting the run time directly, the model predicts a vector of positive coefficients. The run time is obtained by a dot product between the output vector and a vector composed of hand-designed features. Therefore the network assigns more or less importance to a hand-designed feature by predicting its coefficient. The neural network is free to give a zero coefficient to any feature judged irrelevant.

Ithemal model [Mendis et al., 2019] has a hierarchical LSTM architecture. Before feeding the block of assembly instructions to the model, the instructions are canonized first. The

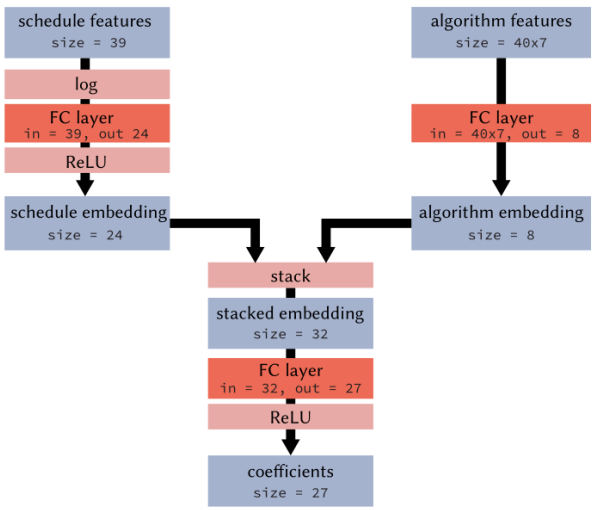


Figure 2.8: Halide's cost model architecture

source and destination operands are tagged differently and the end of each instruction is marked. The model breaks down into three layers: the Token layer, the Instruction layer, and the Prediction layer (Figure 2.9. The token layer can be considered as a token embedding lookup table that maps a code operation (mov, add, ...) or a register (eax, edx, ...) to a representation vector of size 256. The role of the instruction layer is to resume a set of representation vectors corresponding to the same instruction, to a hidden vector of fixed size. This is achieved by an LSTM unit.

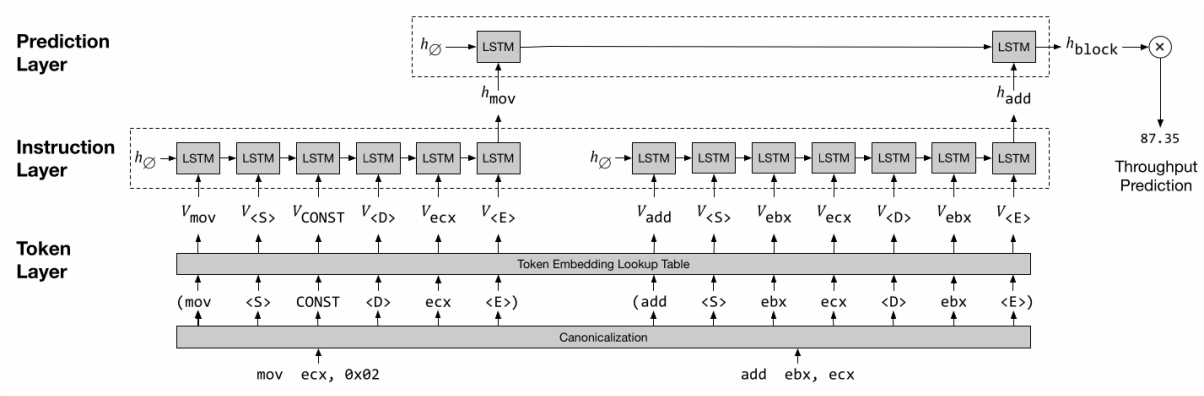


Figure 2.9: Ithemal system architecture

Similarly, The prediction layer is another LSTM unit that proceeds the hidden vectors of instructions and generates a single hidden vector. This hidden vector is supposed to hold all the necessary information about the block to predict the global throughput (the number of CPU cycles). Finally, the hidden vector of the block is transformed into throughput by a linear layer. [Mendis et al., 2019] present two other deep learning architectures: Token LSTM and DAG-RNN. The Token LSTM sequentially precedes the assembly block tokens (the output of the Token layer in Figure 2.9), regardless of the instructions hierarchy. The DAG-RNN model takes an opposite approach by adding producer-consumer dependency between the instruc-

tions. The Token and the Instruction layers are kept the same. However, rather than running an LSTM sequentially over the instructions in the prediction layer, the DAG-RNN constructs a producer-consumer dependence graph of the instructions in the basic block as shown in Figure 2.10. The constructed graph is fed to a DAG-Recurrent Neural Network [Shuai et al., 2015].

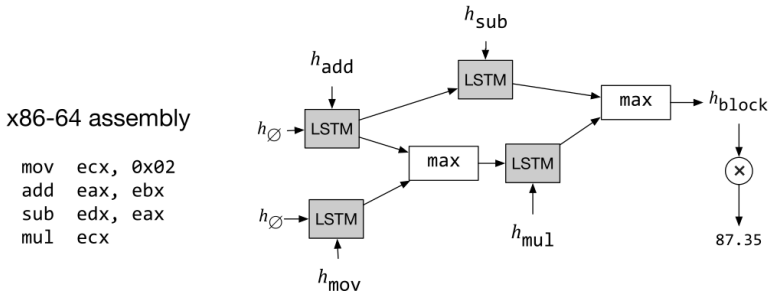


Figure 2.10: Ithemal DAG-RNN architecture

The AutoTVM framework [Chen et al., 2018] supports two statistical cost models. The first is based on Gradient Boosted Trees [Friedman, 2001]. This machine learning algorithm is feature-based. That is why the programs are characterized by a set of hand-designed features extracted from the low-level abstract syntax trees. Static feature extraction makes the inference time very low even when running exclusively on CPUs. Unlike the first model, the TreeGRU neural network [Tai et al., 2015] requires no feature engineering. As detailed in Section 2.2.3, the program is considered as a tree. Recursively, the neural network encodes the abstract syntax tree of a program into an embedding vector. The embedding vector is fed to a linear layer to predict the execution time. The computational cost of this approach is relatively high, but the quality of the learned features is proven superior to those designed by hand in most cases.

In another example, [Rahman et al., 2010] uses a fully connected feedforward neural network with three layers to predict tile size impact. Each tile size configuration is presented to the neural network as a three-dimensional input vector. Accordingly, The input layer has 3 units: one unit for each loop level tile size. Since the target is the execution time, the output layer is a single perceptron. The hidden layer consists of 30 hidden neurons. Two activation functions are applied in this ANN: a logistic activation function for the hidden layer and a linear activation function for the output layer.

Based on the cache reuse algorithm outputs, PolyScientist system [Tavarageri et al., 2020] defines a cost function that considers the working set sizes, the latency, and the bandwidth of the levels of cache (typically L1, L2, and L3). Instead of predicting an absolute value (execution time, speedup, etc) and using it later to rank programs, the model is trained to perform relative ordering of two code variants. Given a set of generated program optimizations, the model is run to order each combination of two variants. Then the variants are globally ranked based on the accumulated number of wins. The advantage of this technique is that the model does not need to predict the scores of the candidates (which is more challenging). In return,

the evaluation of n candidates requires the execution of the model $\frac{n(n-1)}{2}$ times. The model architecture should be shallow enough to keep the inference time below the acceptable threshold. The model takes an 8-dimensional vector as input (4 dimensions for each variant) and outputs 2 binary neuron values as shown in Figure 2.11. If the first neuron fires a 1, then the first variant is considered the winner. If the second neuron fires a 1, then the second variant is considered the winner. If both of them are zero, then it is a draw between the two variants. The model has four hidden layers of decreasing sizes. Two types of activation functions are used for the intermediate layers : *relu* and *softsign*. For the output layer, the *softmax* activation function is applied so that the elements of the 2-dimensional output lie in the range [0,1] and sum to 1. If the output value is above a threshold θ, it is considered a 1, otherwise a 0.

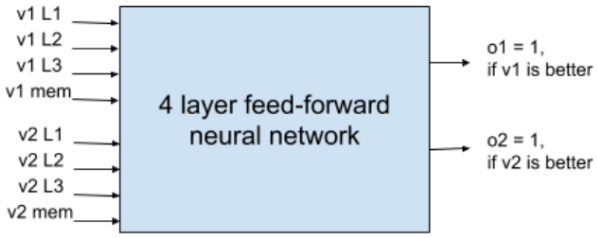


Figure 2.11: PolyScientist model

2.2.5 Classifying by Training Policies

All the models presented above are trained using supervised learning techniques although some special-purpose metrics are invoked, the rank loss function for example. This section presents the optimization algorithms and their parameters as well as other details that relate to each case in particular.

Halide cost model [Adams et al., 2019] is implemented and trained using the Halide framework itself. The criterion used to train the network is *Mean Squared Error* which is sensitive to outliers and leverages larger errors. Targeting run time makes the model predict large values more accurately (bad schedules) and it is less precise for programs with small run time (good schedules). The problem is solved by targeting the throughput (the inverse of the run time). The training batch is composed of 32 schedules for a single random program. The neural network is trained using Adam optimizer, with standard hyper-parameters.

Ithemal [Mendis et al., 2019] is trained on 80% of the dataset, the other 20% is assigned to the test set. All model parameters are learnable from token embedding to the affine linear layer weights. The used loss function is *MAPE*, a normalized error metric based on the *L1* norm. For the optimizer, Ithemal's best performing model is achieved by the asynchronous *SGD* (Stochastic Gradient Descent) algorithm with momentum and an initial learning rate of 0.1. The batches are formed from 4 blocks of assembly code. The 6 parallel trainers converge to the optimal result after 5 epochs.

Mean squared error, the common regression loss function, is used to train the models to be more accurate in predicting the run time. The run time of a given schedule is compared to other schedules during space exploration to select the best candidates. For this reason, the relative order of the schedules is more important than the absolute value of the target. [Chen et al., 2018] use a rank loss function designed particularly to serve this purpose (Equation 2.1).

$$\sum_{i,j} \log(1 + e^{-sign(y_i - y_j)(f(x_i) - f(x_j))}) \tag{2.1}$$

[Chen et al., 2018] exploit other techniques like batching and GPU acceleration to improve the inferring time of the TreeGRU model. Nevertheless, this is not enough because the training must be done for several tensor operators with different input shapes and data types. [Chen et al., 2018] solves this via transfer learning. The system stores knowledge collected from the previously seen data and uses it to speed up the next training.

[Rahman et al., 2010] present a particularity: The 3-dimensional vector fed to the model does not contain any information about the loop nest bounds or the statements of the program. So the model must be trained on samples of the same referred program. In this case, the program features (other than tiles sizes) are invariant for the cost model and they could be omitted from the input. In the experiments presented in the paper, the search space is composed of 223 tile size configurations (3 loop levels with 22 possible tile sizes for each loop level). A random sample of 5% of the search space is used for training. Out of these 530 data points, approximately 10% are separated for validation while the remaining 90% are used for training. The neural network is developed and trained using the Stuttgart Neural Network Simulator. The model is trained to minimize the sum squared error between the predicted and ground truth execution time. The used training algorithm is resilient back-propagation with a learning rate of 0.001. To prevent overfitting, the Early Stopping test is invoked at the end of each epoch.

The PolyScientist system [Tavarageri et al., 2020] collects and normalizes the compiler-generated statistics before processing with the DNN model. Since the goal is to learn the relative performance of two program variants, the difference between the working set sizes must be emphasized. Normalizing each variant entry independently of one another can break the relative difference. The adopted normalization policy is to gather the working set sizes of the two variants together and divide the individual statistic by the result.

2.2.6 Classifying by Evaluation Methodologies

Most models are used to approximate an objective function. Once integrated into a search algorithm, the models do not need to predict the target exactly (run time, speedup, ...). What makes the real difference in their ability to rank the candidate optimizations or variants in the right order. In addition to the usual regression error metrics like MSE (Mean Square Error) and

MAE (Mean Absolute Error), other metrics could be used to evaluate models in ranking tasks like Spearman rank correlation coefficient [Zar, 1972]. The Spearman coefficient is defined as the correlation between two series of rank variables.

Halide model [Adams et al., 2019] is evaluated on one hundred unseen random programs. The programs are formed from pipelines of length up to 10 stages. The programs carry out image processing primitives like stencils and deep learning operations like convulsions. The model has achieved an R^2 correlation of 96%.

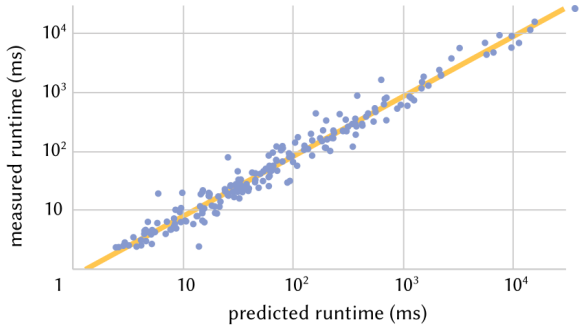


Figure 2.12: Actual vs predicted runtimes for 100 random programs

Ithemal [Mendis et al., 2019] accuracy is evaluated against two state-of-the-art, hand-written analytical models. The test dataset is formed from assembly blocks extracted from Linux Standard Libraries, polyhedral compilation test suite, and various other simulation and end-user applications. The ground truth throughput is measured by running the test set blocks on three different Intel microarchitectures. Ithemal has shown the best precision for all considered micro-architectures. Figure 2.13 Presents the heatmap relating actual and predicted values for the Haswell architecture, the corresponding Mean Absolute Percentage error is 8.9% while the Spearman and Pearson correlations are 96% and 91.8% respectively.

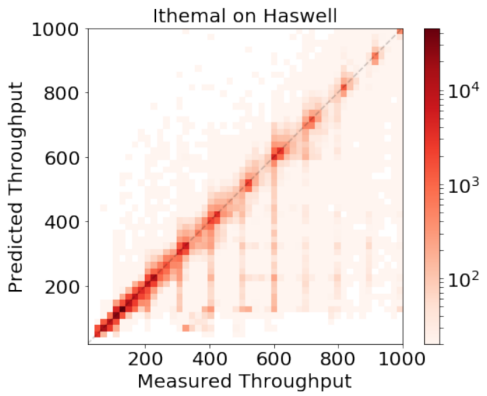


Figure 2.13: Heatmap for measured and predicted throughput values under Ithemal model for basic blocks with measured throughput values less than 1000 cycles for the Haswell micro-architecture

Both AutoTVM [Chen et al., 2018] cost models (the GBT- and TreeGRU-based) are evaluated with the MSE (Mean Squared Error) and the rank objective functions. The rank-based

objective either outperformed or performed the same as the regression-based objective in all presented experiments. The test set consists of conv2d operators extracted from an Image Recognition neural network inference (ResNet-18). In addition to the height and width of the layer inputs, other convolutional parameters have also been varied to cover a larger test space. Figure 2.14 shows one of these configurations (height and width are 56, Input and output channels are 64, kernel and stride sizes are 1). The Number of trials corresponds to the number of evaluations on the real hardware.

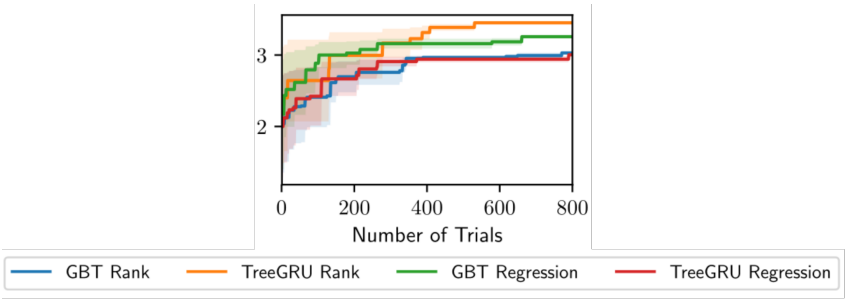


Figure 2.14: Rank vs. Regression objective function for the GBT- and TreeGRU-based models

In [Rahman et al., 2010], The cost model is evaluated on linear algebra kernels and decompositions (dgemm, doitgen, lu,...) and image processing stencils (fdtd-2d and jacobi-2d). The neural network is able to accurately model the execution time in particular regarding the best performing tile size values. Figure 2.15 left shows a plot of the predicted vs the actual execution time. The tile sizes (x-axis index) are sorted in ascending order according to their performance. For most benchmarks, it predicts more than 90% of the search space with less than 10% deviation compared to the actual execution time, even with chaotic distributions such as Figure 2.15 right.

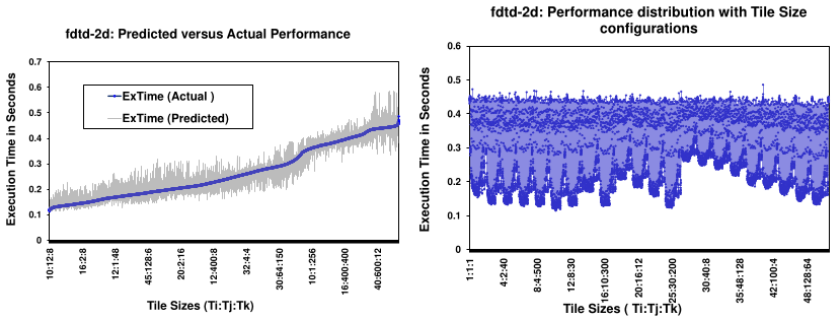


Figure 2.15: fdtd-2d evaluation

The Polyscentist DNN [Tavarageri et al., 2020] cost model is compared to a hand-written analytical model by using both during a search algorithm and reporting the best found optimization. The optimized programs are layers of 10 state-of-the-art image recognition neural networks. Figure 2.16 corresponds to 20 layers of the ResNet-50 architecture. For most test set layers, the deep learning based cost model achieved better performance.

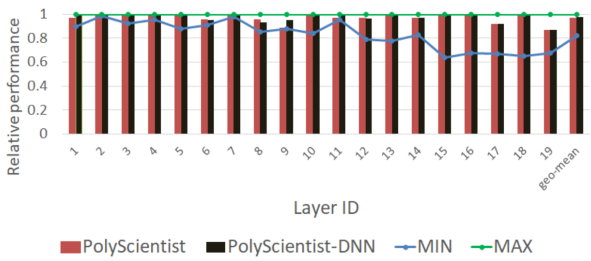


Figure 2.16: Performance of code variants for Resnet-50 layers

2.2.7 Synthesis of the Literature Review

In the previous sections, we have established six classifications from six different perspectives. The perspectives cover a variety of design choices, training strategies, and evaluation methods. The most prominent criteria are: the type of approximate function, the type of features fed into the model, and the inclusiveness of the input. In this section, we focus on those criteria to give a comprehensive wrap-up.

We can distinguish two types of approximated functions : regression and rank functions. A regression function returns the overall score of a candidate which is execution time or speedup. Put another way, the output is a continuous positive value that reflects the quality of the input. In contrast, a rank function takes as an input two candidates and outputs a binary variable that indicates the best between the two. Regression functions report more information than rank functions, and therefore, it is more difficult to design and train models that approximate them.

The features mirroring the programs are either hand-designed or automatically extracted. Automatically extracting features from high-level program characteristics (source code or other compilation phases) is more challenging than operating on hand-engineered features. In such cases, cost model architectures are designed to handle the variable length of the “raw” inputs and they incorporate deeper neural networks. The models targeting hand-designed features are relatively shallow and have fewer parameters since the burden of selecting relevant and pertinent combinations already falls on the designers of the system.

The inputs fed to a cost model can include both program and optimization features or they can be limited to optimization parameters. In the first case, once the model is trained on hundreds of thousands of program-optimization combinations, there is no need to retrain it when facing a new program because the model has learned to take into consideration any program characteristics. This is not true for the second case, where the models are trained using new samples from the optimization space belonging to the new program. In other words, for a new program, the system generates several possible optimizations, executes each optimization, builds a data set specific to that program, and performs training via transfer learning. This could be a major limitation if the system is required to give quick responses.

As detailed above, we classified the reviewed cost models according to three criteria, even more, each criterion gives rise to two classes. Classes yielded from different criteria are not mutually excluded. The global classification is summarised in Table 2.1. For instance, Halide auto-scheduler's model [Adams et al., 2019] approximates a regression function, uses hand-engineered features, and has inputs that include both program features and optimization parameters (full programs).

<table>
<tr><td></td><td></td><td colspan="2">Features</td><td></td><td></td></tr>
<tr><td></td><td></td><td>Hand-engineered</td><td>Automatically extracted</td><td></td><td></td></tr>
<tr><td rowspan="3">Type of the approximated function</td><td rowspan="2">Regression</td><td>Halide auto-scheduler [Adams et al., 2019] ,
Predictive phase-ordering [Ashouri et al., 2016] ,
Compiler Optimisation Evaluation [Dubach et al., 2007]</td><td>Ithemal [Mendis et al., 2019] ,
AutoTVM (TreeGRU) [Chen et al., 2018]</td><td>Full programs</td><td rowspan="3">Inclusiveness of the input</td></tr>
<tr><td>Tile Selection [Rahman et al., 2010],
AutoTVM (GBT) [Chen et al., 2018]</td><td></td><td rowspan="2">Optimization parameters</td></tr>
<tr><td>Rank</td><td>PolyScientist [Tavarageri et al., 2020]</td><td></td></tr>
</table>

Table 2.1: Classification of the reviewed cost models

2.3 Conclusion

In this chapter, after giving a brief background on machine learning and deep learning, we covered the use of machine learning to build cost models for automatic code optimization.

The first design choice made by auto-scheduler developers is whether the approximated function is a regression or a rank function. One of the main challenges is data acquisition. In order to build high quality and sufficiently large datasets, many approaches have been adopted. The collection of code from publicly available repositories is exploited where possible. For newer frameworks that do not have this advantage, the generation of random algorithms is considered. Cost model architectures are designed to take advantage of data rep-

resentation, sequence modeling techniques, and long-range semantic dependencies among input features. Training policies are strongly coupled to the architecture of the model and its use case once integrated into search algorithms. Some models are trained offline where the model parameters are frozen throughout the production phase, unlike online learning where the model parameters are updated each time new programs are confronted.

3

Design and Implementation of the TIRAMISU Cost Model

In this chapter, we present our proposed implementation of the TIRAMISU cost model, we will provide a detailed description of our approach for building the different stages of our contribution, and will discuss the different design choices that we made at each stage.

This chapter is divided into three sections:

- **Dataset Construction.** We will explain the process of constructing our 1.8 million points dataset, from the design of a novel synthetic TIRAMISU program generator to our highly optimized dataset construction pipeline.
- **Program Characterization**. We discuss the challenges of designing a characterization that can mirror complete TIRAMISU programs. Then, we introduce our proposed AST-based characterization that combines program features and schedule parameters.
- **Model Architecture.** We will discuss the challenges of designing an efficient architecture and we will detail the architecture that we ended up with for the cost model, the Recursive LSTM. Finally, we give a quick overview of other intermediate architectures that we explored.

3.1 Dataset Construction

In general, supervised learning techniques consist in approximating an unknown function by deducing it from a collection of examples called a training dataset [Goodfellow et al., 2016]. In our case, we need our cost model to predict the performance for any valid algorithm-schedule pair after having seen hundreds of thousands of examples.

Compared to the conventional machine learning approaches, training deep learning models is much more data-demanding and requires larger training datasets in order to be efficient. Furthermore, the training dataset has a direct and decisive influence on the generalization ability of DNNs, the learning time, and the prediction's accuracy. Therefore, in order

to efficiently train our cost model, we need to construct a large-enough high-quality dataset, this task is one of the least tractable challenges we encountered during this project.

This section will be about our dataset construction process, we will go through the challenges that we met and how we overcame them, then we will expose our design choices and the implementation methodology, and finally, we will give insights about the content and quality of our dataset.

3.1.1 Design Discussion

Ideally, we should construct the dataset by collecting algorithms from the publicly available user repositories and libraries. Unfortunately, only a small number of programs have ever been written in this novel framework, TIRAMISU. Therefore we decided to populate our dataset with synthetic TIRAMISU programs that will be created through an automatic generation process. This methodology is shown to be more efficient and generalizes more robustly than fitting the dataset to specific benchmarks or to a set of real-world programs [Adams et al., 2019].

As our dataset will be made of automatically generated synthetic programs, we need to ensure that the generated algorithms and schedules are drawn from a distribution that is representative of key patterns found in real programs. In other words, the generated algorithms should be diverse and general enough to cover most of the patterns present in real-world TIRAMISU applications, just the same that schedules should cover the most fruitful code optimization possibilities. The randomness of the automatic generation must be controlled to prevent producing unrealistic algorithms that are unlikely to exist in real applications, and which are not useful for our model to learn to predict their performances, on the contrary, such improbable algorithms can be considered as noisy data points. As the generated programs need to compiled and run in order to label the dataset, the algorithms must be correct to achieve successful compilation and execution and the schedules must be valid to get consistent results.

Each data point in the dataset is a tuple of the form (scheduled-program, speedup) where speedup is the target value. As detailed in Section 1.4.1, a program is the result of applying a schedule to an algorithm. The speedup is the ratio between the execution time of the original unoptimized program and the optimized one.

While defining our target value, we had the choice between the execution time and the speedup, we favored the speedup for the three following motivations.

- First, the execution time has a wide range of values. The execution times can range from about a few milliseconds for simple programs to hundreds of seconds for more complicated ones. We can view speedup as a normalization that brings all the execution

times to the same dynamic range centered on 1 for all programs (see Table 3.2). As a result, the cost model training will be easier and faster.

- Second, if the model is trained to predict execution times, it will learn to rely more on the algorithm characteristics like loop extents rather than schedule features. For short running algorithms, most correspondent schedules have an execution time around the same small value. Therefore the model can just predict the same center value for all schedules and still gets an acceptable accuracy. This behavior will not take place if the speedup is used instead of execution time.

- Third, we expect our cost model to make a difference between a good and bad schedule for a given algorithm. The most natural way to emphasize this difference is through speedup. For example, an optimized program with an execution time of 100 ms can be considered a good schedule if the original unoptimized program has an execution time of 200 ms and can be considered bad if the unoptimized program has an execution time of 50 ms. Therefore the execution time alone is not an indicator of the schedule's quality, but the speedup clearly highlights the quality by yielding 2 for the first case and 0.5 for the second.

3.1.2 The Design of the Synthetic Programs' Generator

The synthetic program generator is composed of two submodules: the algorithm generator and the schedule generator (Figure 3.1). The algorithm generator defines the logic of the produced programs by delineating three elements in this particular order: sub-tree structure (loop variables without computations), computations, and loop extents. The schedule generator applies valid combinations of code transformations to a previously generated algorithm to form schedules. In the following, we detail the policy of each submodule.

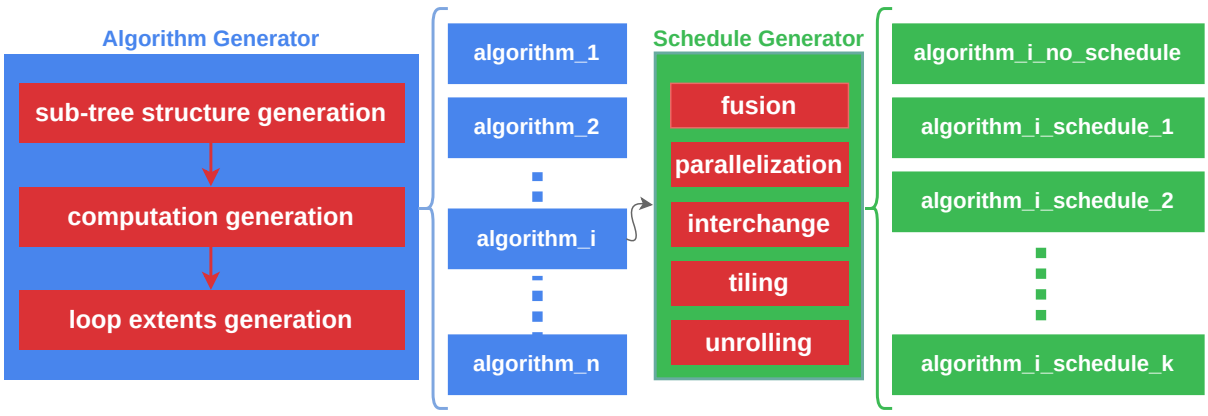


Figure 3.1: Synthetic TIRAMISU program generator overview

a) Algorithm generation policy. A TIRAMISU program is a sequence of computations where each computation is an assignment. Analyzing Deep Learning, Image processing, and scientific computing applications, we can identify five key patterns:

- **Arithmetic expressions.** These computations are used to initialize arrays with expressions without invoking memory accesses. The most common example is the zero initialization. Figure 3.2 shows an example of 3-dimensional zero initialization computation.

```
1 var i("i", 0, 64), j("j", 0, 128), k("k", 0, 128);
2
3 computation zero_init("zero_init", {i,j,k}, 0);
4
```

Figure 3.2: TIRAMISU code for zero initialization

- **Simple assignments** The right-hand side of the simple assignment computations, is a function of input arrays or array values computed previously. Matrix addition, matrix subtraction, and matrix-scalar multiplication are all considered simple assignments. Figure 3.3 illustrates a computation used to initialize a convolution tensor (4-dimensional array) with a bias vector.

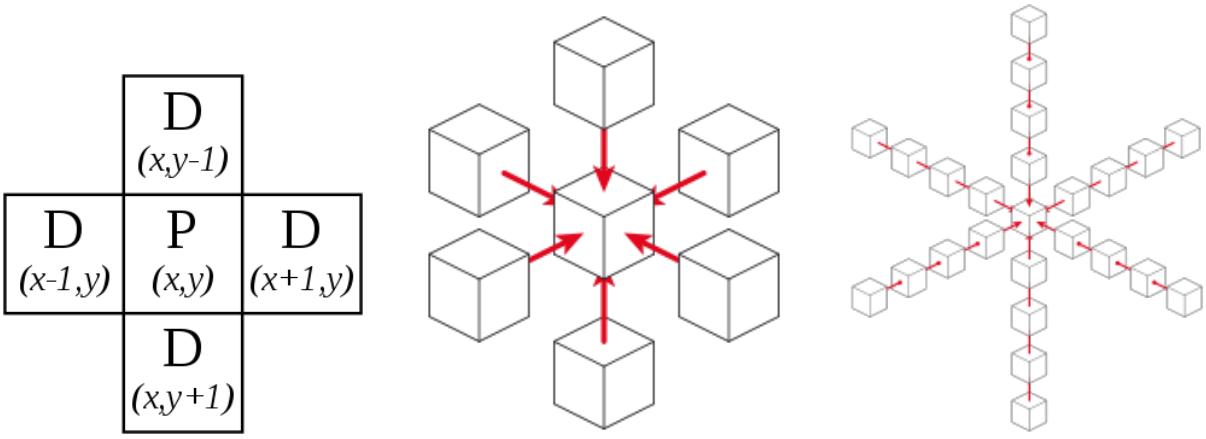
```
1 var b("b", 0, batch_size), o_y("o_y", 0, output_height), o_x("o_x", 0, output_width),
    ↪ nb_k("nb_k", 0, nb_kernels);
2
3 input bias("bias", {nb_k}, p_float64);
4
5 computation convolution_init("convolution_init", {b, nb_k, o_y, o_x}, bias(nb_k));
6
```

Figure 3.3: Example of simple assignment TIRAMISU computation

- **Stencils** Stencils are computations that iteratively update a grid of data using the same pattern. The output grid is updated by the same position in the input grid and its neighbors. The grid point and its neighbors can be scaled before they are added to generate the output. The shape of the neighborhood depends on the application itself. The stencil computations considered in this work use the Von Neumann neighborhood. Equation 3.1 defines the 2-D Von Neumann neighborhood of range r while Figure 3.4 illustrates some examples.

$$N^{v}_{(x_0,y_0)} = \{(x,y) : |x - x_0| + |y - y_0| \leqslant r\} \tag{3.1}$$

Stencils are widely applied in the field of image processing, scientific computing, and engineering applications. Stencil codes are used to solve partial differential equations



(a) Von Neumann stencil of range $r = 1$

(b) 7 points 3-D stencil with offset equals to 1

(c) 25 points 3-D stencil with offsets = 1..4

Figure 3.4: Stencil examples

and to simulate fluid dynamics. Figure 3.5 shows a TIRAMISU code that performs horizontal blur. Horizontal blur is a 2-D image processing stencil. The output cell *b(i,j)* is updated with the mean of the input cell *a(i, j)* and its neighbors *a(i, j - 1)* and *a(i, j +1)*.

```
1 var i("i", 1, N - 1), j("j", 1, M - 1);
2
3 input a("a", {N, M}, p_float64);
4
5 computation b("b", {i, j}, (a(i, j) + a(i, j - 1) + a(i, j + 1))/3);
6
```

Figure 3.5: Example of stencil (Horizontal Blur)

- **Reductions** Given n inputs $x_1, x_2, \ldots, x_n$, and an associative operation $\otimes$, a reduction algorithm computes the output $x_1 \otimes x_2 \otimes \ldots \otimes x_n$. The reduction operator can be any associative and often commutative function such as sum, max, min, or user defined. For example, matrix multiplication ($C = A.B$) can be formulated as a sum reduction computation (Figure 3.6). Each output element *C(i, j)* is the result of reducing the inputs : $A(i,0) * B(0,j), A(i,1) * B(1,j), \ldots, A(i,K-1) * B(K-1,j)$ according to Equation 3.2.

$$C(i,j) = \sum_k A(i,k) * B(k,j) \tag{3.2}$$

  The reduction is a frequent operation in parallel linear algebra (parallel dot product, parallel matrix multiplication, etc), sorting (quicksort and radix sort), graph algorithms (minimum spanning trees, connected components, maximum flow, etc), and computational geometry (convex hull, K-D tree building, closest pair in a plan, etc) [von Praun et al., 2011].

- **Convolutions** In the context of Machine Learning, a convolution is a computation that operates on two arguments : an *input* and a *kernel*. The *input* is a multidimensional

```
var i("i", 0, N), j("j", 0, M), k("k", 0, K);

input A("A", {i, k}, p_float64);
input B("B", {k, j}, p_float64);

computation C("C", {i, j, k}, p_float64);
C.set_expression(C(i, j, k) + A(i, k) * B(k, j));

```

Figure 3.6: Matrix multiplication in TIRAMISU

data array and the *kernel* is a multidimensional array of learnable parameters. The output is usually referred to as the *feature map*. For example, if the *input* is a two-dimensional image I and the *kernel* is a matrix K, the output matrix S is given by the Equation 3.3 [Goodfellow et al., 2016] . Figure 3.7 is a TIRAMISU implementation of a convolution that takes as an input a batch of 3-dimensional images (the first two dimensions are for image resolutions and the third dimension is the number of color channels). The code applies *nb_k* kernels to the input. Convolutions are commonly applied to analyze visual imagery and are widely used in other fields such as time series forecasting, recommender systems, natural language processing, etc.

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n) K(m,n) \tag{3.3}$$

```
var b("b", 0, batch_size), c("c", 0, nb_channels), i_y("i_y", 0, input_height), i_x("
    ↪ i_x", 0, input_width), o_y("o_y", 0, output_height), o_x("o_x", 0, output_width),
    ↪  k_y("k_y", 0, kernel_height), k_x("k_x", 0, kernel_width), nb_k("nb_k", 0,
    ↪ nb_kernels);

input input("input", {b, c, i_y, i_x}, p_float64);
input kernel("kernel", {nb_k, c, k_y, k_x}, p_float64);

computation convolution("convolution", {b, nb_k, o_y, o_x, c, k_y, k_x}, p_float64);
convolution.set_expression(convolution(b, nb_k, o_y, o_x, c, k_y, k_x) + input(b, c,
    ↪ o_y + k_y, o_x + k_x) * kernel(nb_k, c, k_y, k_x));

```

Figure 3.7: Convolution implementation in TIRAMISU

[Adams et al., 2019] has shown in their work that synthetic programs do not need to be long or meaningful to successfully train a predictive cost model. With a dataset covering the most common situations, the model learns to generalize to long real programs. Our program generator combines the five patterns detailed above to randomly build synthetic algorithms. Moreover, it is designed to be easily extensible to new patterns and new algorithm structures.

The generated algorithms can be classified into three classes, corresponding to their structure. To begin with, one computation algorithms are the most simple, all examples above

belong to this class. Furthermore, the more general class encloses algorithms with computations fused in the innermost loop. For these algorithms, all computations have the same iteration domain. Many image processing operations are included in this class. For example, Figure 3.8 shows a TIRAMISU code with two computations $blur\_x$ and $blur\_y$, that apply horizontal and vertical blur respectively. The two computations are defined by the same loop variables and thus have the same iteration domain over the input image. On top of that, the last class includes algorithms having the most general form. The structure of these algorithms can be modeled by a tree, such as the internal nodes represent loop variables and the leaves represent computations. A case in point is the *convolution relu maxpooling* algorithm illustrated in Figure 3.9, where the computations : *convolution_init*, *relu*, and *maxpool* iterate using the first-four loop variables *(b, nb_k, o_y, o_k)* while the *convolution* computation includes all seven loop variables to form its iteration domain. The tree representation (Figure 3.10) is able to encode the iteration domains of each computation and conserves the order of the computations.

```
var i("i", 1, N - 1), j("j", 1, M - 1), c("c", 0, 3);

input input_image("input_image", {N, M, c}, p_float64);

computation blur_x("blur_x", {i, j, c}, (input_image(i, j, c) + input_image(i, j - 1,
  ↪ c) + input_image(i, j + 1, c))/3);
computation blur_y("blur_x", {i, j, c}, (blur_x(i, j, c) + blur_x(i - 1, j, c) +
  ↪ blur_x(i + 1, j, c))/3);
```

Figure 3.8: Example of TIRAMISU algorithm where all computation are fused in the innermost loop

```
var b("b", 0, batch_size), c("c", 0, nb_channels), i_y("i_y", 0, input_height),
  ↪ i_x("i_x", 0, input_width), o_y("o_y", 0, output_height), o_x("o_x", 0,
  ↪ output_width), k_y("k_y", 0, kernel_height), k_x("k_x", 0, kernel_width),
  ↪ nb_k("nb_k", 0, nb_kernels);

input input_t("input", {b, c, i_y, i_x}, p_float64);
input kernel("kernel", {nb_k, c, k_y, k_x}, p_float64);
input bias("bias", {nb_k}, p_float64);

computation convolution_init("convolution_init", {b, nb_k, o_y, o_x}, bias(nb_k));

computation convolution("convolution", {b, nb_k, o_y, o_x, c, k_y, k_x}, p_float64);
convolution.set_expression(convolution(b, nb_k, o_y, o_x, c, k_y, k_x) + input_t(b, c,
  ↪  o_y + k_y, o_x + k_x) * kernel(nb_k, c, k_y, k_x));

computation relu("relu", {b, nb_k, o_y, o_x}, expr(o_max, convolution(b, nb_k, o_y,
  ↪ o_x, 0, 0, 0), cast(p_float64, 0)));

computation maxpool("maxpool", {b, nb_k, o_y, o_x}, p_float64);
maxpool.set_expression(expr(o_max, maxpool(b, nb_k, o_y, o_x),  relu(b, nb_k, o_y, o_x
  ↪ )));
```

Figure 3.9: TIRAMISU implementation of the "*convolution relu maxpooling*" algorithm

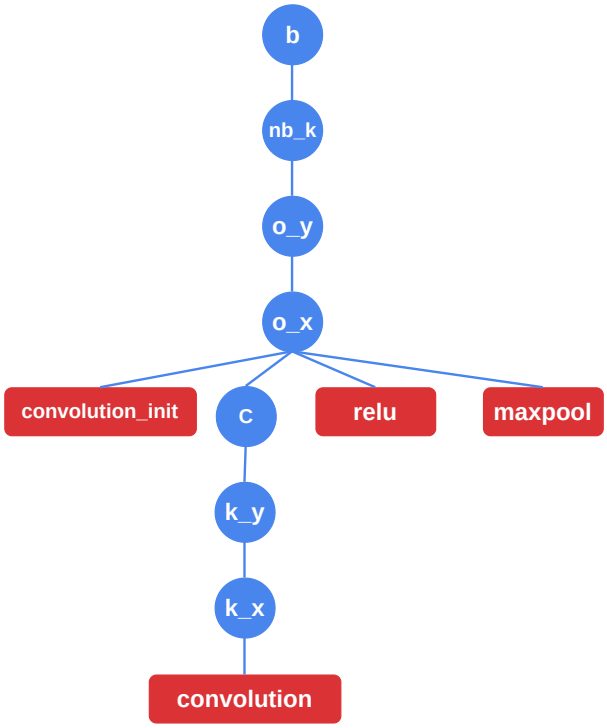


Figure 3.10: The tree representation of the "*convolution relu maxpooling*" algorithm presented in Figure 3.9

Randomly generated algorithms are correct by construction. A computation consumes either constants, input arrays, or values computed by previous computations. The synthetic program generator is designed to produce realistic algorithms. The sub-tree structure containing loop variables (without computations), is randomly generated according to a predefined set of probabilities. Thereafter, the number of computations and the type of each computation are always stochastic but conditioned to the position of the computation in the tree. For example, let's assume that the generated tree has seven levels (i.e seven loop variables). It is more likely that the type of the leaf computation will be a convolution rather than a stencil since stencils operate on images and they are limited to four loop variables maximum (two variables for the image dimensions, one dimension for colors and the extra fourth dimension for time step). This rule among others is inferred from observing TIRAMISU algorithms specialized in deep learning operations, image processing pipelines, and scientific computing applications. Finally, the loop extents are drawn from a distribution that covers all the space of real algorithms. The loop extents can be as low as 3 (for the number of color channels in an image, the size of a convolution kernel, etc.) and up to thousands (for the image resolution, the number of neurons in DNN architecture, etc.).

b) Schedule generation policy. The synthetic program generator is able to combine five different types of code optimization: parallelization, fusion, interchange, tiling, and unrolling. Two depths of tiling are supported, 2D and 3D. Tile sizes and unrolling factors are configurable. For all the datasets of this project, the tile sizes are set to $\{128, 64, 32\}$ and unrolling factors are set to $\{16, 8, 4\}$. Parallelization is considered a simpler optimization as it is most

likely beneficial when applied to the outermost loops [Kennedy and McKinley, 1992], therefore, we choose to apply parallelization to outermost loops as a default transformation for all programs. A schedule is a sequence of these code transformations.

Since the scheduling space is exponential depending on the number of loop variables and computations [Manseri, 2018], enumerating all schedules exhaustively becomes infeasible when dealing with large algorithms (which will usually be the case). As an illustration, for an algorithm with two computations and four loop variables (Figure 3.11), our program generator can produce 9096 different schedules. To redress the problem, the program generator offers the possibility of randomly selecting n schedules. Limiting the number of schedules per algorithm has another favorable impact on the balance between algorithms and schedules in datasets. When using a nearly exhaustive schedule list, the datasets have low algorithm diversity, despite having a high total number of schedules, due to the high number of schedules per algorithm. In a dataset with 1 million schedules constructed using an average of 2000 schedules/algorithm, the dataset will only contain about 500 algorithms. This is of course an example of lacking algorithm diversity. We have found that limiting the number of schedules for each algorithm to 32 produces the best results.

Specific rules are used to guarantee that the loop optimizations are valid. In particular, the parameters of tiling and unrolling should respect the loop extent, meaning we must check that the loop extent is smaller than either the tile size or the unrolling factor. On the other hand, applying some optimizations can make others invalid in upcoming steps. If we interchange the loop variables l and k of the algorithm in Figure 3.11, then the command *tile(k, l, 64, 128, k0, l0, k1, l1)* won't be allowed because the tiling should respect the order of variables. Then again, we can no longer unroll the loop k, since the variables l and k are replaced by *l0, k0, l1, k1* throughout tiling (by the correct tilling command : *tile(l, k, 64, 128, l0, k0, l1, k1)*). Accordingly, if we want to apply unrolling, we should target the new innermost loop *k1*. For these reasons, tracking the transformations that the AST (Abstract Syntax Tree) undergoes after each code optimization, is mandatory (see Figure 3.12 for illustration).

```
// original variables
var i("i", 0, 1024), j("j", 0, 2048), k("k", 0, 512), l("l", 0, 1280);
// new variables yield from tiling
var k_0("k0"), k_1("k1"), l_0("l0"), l_1("l1");

input input_0("input_0", {i, j, k}, p_int32);
input input_1("input_1", {i, j, k, l}, p_int32);

computation comp_0("comp_0", {i, j, k, l}, 2 * input_0(i, j, k));
computation comp_1("comp_1", {i, j, k, l},  input_0(i, j, k) + input_1(i, j, k, l));

comp_0.then(comp_1,j);
comp_0.parallelize(i);
comp_1.parallelize(i);
comp_0.interchange(k, l);
comp_1.interchange(k, l);
comp_0.tile(l, k, 64, 128, l0, k_0, l_1, k_1);
comp_1.tile(l, k, 64, 128, l0, k_0, l_1, k_1);
comp_0.unroll(k_1, 4);
comp_1.unroll(k_1, 4);
```

Figure 3.11: A random synthetic program example

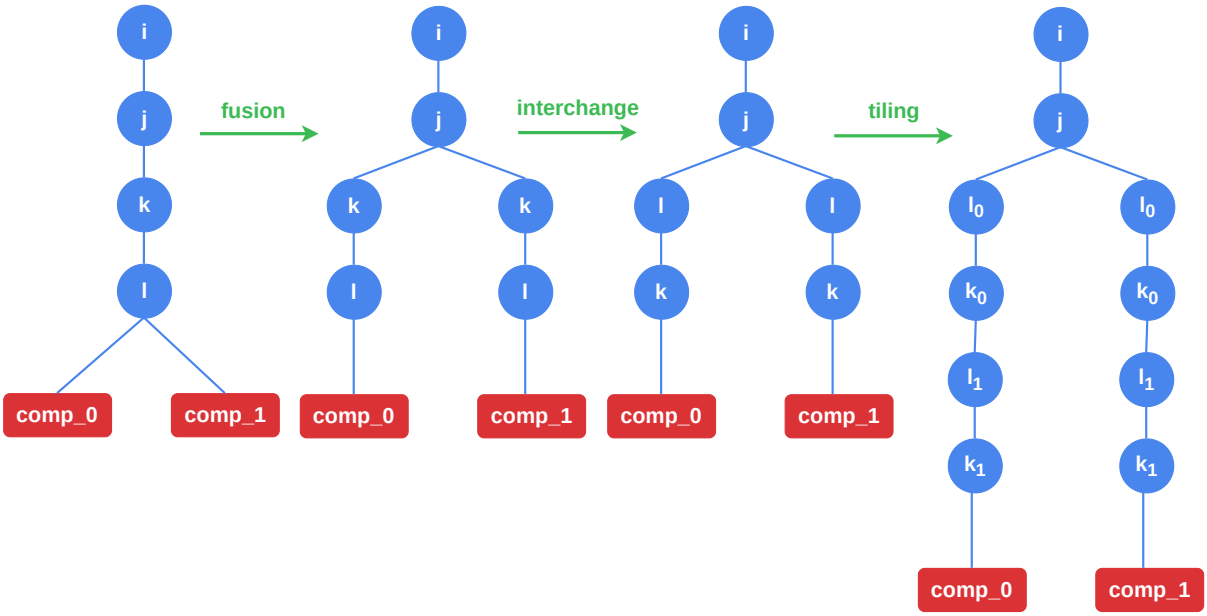


Figure 3.12: Examples of the impact that code transformations make on the tree representation of the TIRAMISU program presented in Figure 3.11. parallelization and unrolling are intentionally not represented because they don't change the structure.

3.1.3 Dataset Construction Workflow

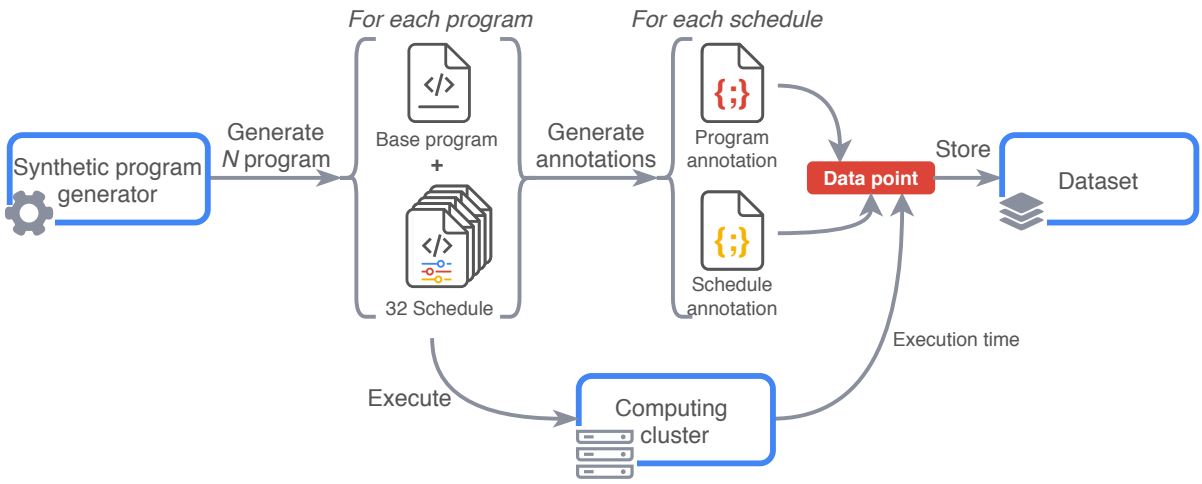


Figure 3.13: Overall process of the dataset construction

As summarized in Figure 3.13, our typical workflow for constructing a dataset consists on the following steps: generate a set of synthetic programs using our program generator, generate annotation files for each generated program, measure the programs' execution time by running them, and finally store the results into the dataset file. These actions were performed repeatedly until we reached a satisfying dataset size.

In the following paragraphs we will present the structure of our dataset file, we will detail the utility and the content of the annotation files, and we will explain how the execution time measurements are taken.

Dataset file structure. As the purpose of the intended dataset is to serve as a statistical base for building a model that is expected to predict the effects of schedules on the execution time, we design our dataset structure as a nested associative array (or a nested dictionary) that maps between programs, schedules applied on each program, and the resulting speedup. Programs and schedules are expressed by an annotation-like intermediate representation and execution times are evaluated by taking ground truth measurements. We designed this structure to offer a general framework that enables an easy reusability for testing different models and data-driven approaches. Figure 3.14 describes the general structure of the dataset file.

Program and schedules annotation. The annotations are used as an intermediate representation that structures information extracted from the source codes into a JSON[1] format [Crockford, 2006], these annotations contain all the information that characterizes a given scheduled program, and are structured in a way that enables easily building different representations that may be required by different models, hence it enables to bypass the need of parsing source codes for building new representations for each new model.

[1] JavaScript Object Notation

```
{"Program_1" : { "Program_annotation" : {...},
                 "Schedules" : { "Schedule_1" : { "Schedule_annotation" : {...},
                                                  "Speedup" : <value>
                                                },
                                 "Schedule_2" : { "Schedule_annotation" : {...},
                                                  "Speedup" : <value>
                                                },
                                   . . .              . . .

                                 "Schedule_n" : { "Schedule_annotation" : {...},
                                                  "Speedup" : <value>
                                                }
                                 }
                },
 "Program_2" : { "Program_annotation" : {...},
                 "Schedules" : {...}
               }
   . . .          . . .

 "Program_n" : { "Program_annotation" : {...},
                 "Schedules" : {...}
               }
}
```

Figure 3.14: Overall structure of the dataset file.

The program annotation contains information about the base program (the program with no transformations applied) such as the used buffers, loop extent, performed assignments, positional information about the computations, etc. The schedules annotations contain information about the transformations that are applied to the program, the application point of each transformation, and its parameters. An example of a program and schedule annotation are given in Figures 3.16 and 3.17 respectively, these annotations are representing the example program presented in Figure 3.15 . Some elements of the annotations that need further explanation (such as the coefficient matrix) will be described in Section 3.2.

Execution time measurements. In order to evaluate the speedup obtained by applying schedules, we need to get precise measurements of the execution time of each transformed version of the program. The execution times of programs are subject to an inevitable variability that can make the evaluated speedup unreliable [Zaparanuks et al., 2009, Mytkowicz et al., 2009, Chen and Revels, 2016], this variability is due to the fluctuations of the execution environment that can be caused by multiple factors such as OS (operating system) jitter, System interruptions, cluster workload, environment temperature, etc.

To minimize the impact of this variability on the quality of the dataset and reduce the overall noise in the evaluated speedups, we use the algorithm presented in Figure 3.18 to measure the execution time of each transformed program.

The program is first run one time without measuring the execution time, i.e., a dry run, this warm-up execution happens to be important to reduce the transient background noise of the environment as the first execution is often the most prone to high noise influence

```
var i_0("i_0", 0, 1024) , i_1("i_1", 0, 128) , i_2("i_2", 0, 128) ;
var i_3("i_3") , i_4("i_4") , i_5("i_5") , i_6("i_6") ;

input input_1("input_1", {i_0, i_1, i_2}, p_int32);
input input_0("input_0", {i_0, i_1}, p_int32);

computation comp_0("comp_0",{i_0, i_1}, 2*input_0(i_0, i_1+1) + 3*input_0(i_0, i_1-1)
  ↪ - 1);
computation comp_1("comp_1",{i_0, i_1, i_2}, input_1(i_0, i_2, i_1)*comp_0(i_0, i_1));

comp_0.then(comp_1, i_1);
comp_0.tile(i_0, i_1, 128, 64, i_3, i_4, i_5, i_6);
comp_1.tile(i_0, i_1, 128, 64, i_3, i_4, i_5, i_6);
comp_1.unroll(i_2, 8);

```

Figure 3.15: An example TIRAMISU program

```
{"function_name": "function1234",
"iterators": {
    "i_0": {"lower_bound": 0, "upper_bound": 1024},
    "i_1": {"lower_bound": 0, "upper_bound": 128 },
    "i_2": {"lower_bound": 0, "upper_bound": 128 }},
"inputs": {
    "input_1": {"buffer_id": 1, "data_type": "p_int32",
                "iterators_list": ["i_0", "i_1", "i_2"] },
    "input_0": {"buffer_id": 2, "data_type": "p_int32",
                "iterators_list": ["i_0", "i_1"] } },
"computations": {
    "comp_0": {
        "buffer_id": 3,
        "iterators": ["i_0", "i_1"], "dims_after_reduction": ["i_0", "i_1"],
        "additions": 1, "subtractions": 1, "multiplications": 2, "divisions": 0,
        "accesses": [{"buffer_name": "input_0", "coefficient␣_matrix": [[1, 0, 0 ],
                                                                        [0, 1, 1 ]]},
                     {"buffer_name": "input_0", "coefficient␣_matrix": [[1, 0,  0],
                                                                        [0, 1,-1]]}]},
    "comp_1": {
        "buffer_id": 4,
        "iterators": ["i_0","i_1","i_2"], "dims_after_reduction": ["i_0","i_1","i_2"],
        "additions": 0, "subtractions": 0, "multiplications": 1, "divisions": 0,
        "accesses": [{"buffer_name": "input_1", "coefficient␣_matrix": [[1, 0, 0, 0],
                                                                        [0, 1, 0, 0],
                                                                        [0, 0, 1, 0]]}
                     {"buffer_name": "comp_0", "coefficient␣_matrix" : [[0, 1, 0, 0],
                                                                        [1, 0, 0, 0]]}
    ↪ ]}
}
}
```

Figure 3.16: Program annotation for the example presented in Figure 3.15.

```
{"schedule_name": "function1234_schedule_12",

"comp_0": {"interchange_dims": null,
           "tiling": {"tiling_depth": 2,
                      "tiling_dims": ["i_0", "i_1"],
                      "tiling_factors": ["128", "64"] },
           "unrolling_factor": null },

"comp_1": {"interchange_dims": null,
           "tiling": {"tiling_depth": 2,
                      "tiling_dims": ["i_0", "i_1"],
                      "tiling_factors": ["128", "64"]},
           "unrolling_factor": "8" },

"tree_structure": {"loop_name": "i_0",
                   "computations_list": [],
                   "child_list": [{"loop_name": "i_1",
                                   "computations_list": ["comp_0"],
                                   "child_list": [{"loop_name": "i_2",
                                                   "computations_list":["comp_1"],
                                                   "child_list":[]}]}]}
}
```

Figure 3.17: Schedule annotation for the example presented in Figure 3.15.

Input: A compiled TIRAMISU program P.

Execute P once `// warm-up execution`
Decide a value for N_{runs}
Initialize L_{time} as an empty list
for $i = 0$ **to** N_{runs} **do**
    Execute P and measure its execution time T_p
    Append T_p to L_{time}
end
return the median value of L_{time}

Figure 3.18: Execution time measurement's procedure.

[Kalibera et al., 2005], then the program is run N_{runs} times saving the execution time of each run, running the program multiple time allows reaching a steady state where execution time becomes more stable. Multiple policies can be used to decide N_{runs} [Kalibera and Jones, 2013, Kalibera et al., 2005] and it depends on the importance we give to the ratio between the data quality of the data generation cost (in terms of time). The policy we used consists on running each transformed program 30 times and each base program (the one without any transformation applied) 45 times as the stability of the later is more important because it is used when calculating the speedup of all the transformed ones. We chose this policy as it provides an acceptable trade-off between generation cost and the measurements' quality. The median of this series of measurements is taken as the final execution time, we opt for the median as it is more robust to outliers compared to other central tendency measures such as the mean that can be significantly altered by the outliers.

The compilation and execution jobs of all the programs of the dataset are distributed on one of MIT's clusters composed of 16 identical nodes [2], the characteristics of each node are described in Table 3.1.

CPU model name	Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz
Architecture	x86_64
Number of sockets	2
Number of cores per socket	12
Number of threads per core	2
Total number of threads	48
L1 data cache size	32 KB
L1 instruction cache size	32 KB
L2 cache size	256 KB
L3 cache size	30720 KB
Memory size	128 GB
Operating system	Ubuntu 18.04.3 LTS

Table 3.1: Specifications of the machines on which the execution time of the dataset's programs was measured.

As all the measurements are taken on the same CPU architecture, the dataset becomes specialized for that specific architecture and hence the models that are built using this dataset can be used to predict speedups of this same architecture since a transformation can have different influences on a program depending on the architectures. But since the execution times of programs are the only architecture-dependent part of our whole approach, our work can easily be extended to support newer CPU architectures by just retaking measurements on the new target hardware.

Note that some ideas for this workflow are inspired from the work of [Henni and Mekki, 2019] and [Abdous, 2020], and our adaptation led to noticeable improvements in terms of both the dataset quality and the generation cost, a detailed comparison is given in the next section.

3.1.4 Dataset Construction Results

The dataset generation process took approximately 4 weeks to run on the cluster in order to generate a dataset that is composed of about 55 thousands programs with 32 transformations each, this makes a total of 1.8 million datapoints in the dataset. Table 3.2 presents the main

[2] http://groups.csail.mit.edu/commit/lanka/

characteristics of the generated dataset and the generation workflow and compares them to an existing dataset previously made by [Henni and Mekki, 2019].

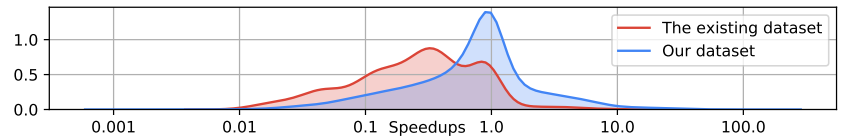
		Our dataset	The existing dataset
Dataset dimensions	Number of datapoints	1,789,833 Scheduled program	527,866 Scheduled program
	Number of programs	55,142 Program	1,000 Program
	Number of schedules per program	32 (fixed)	527.87 (on average)
Diversity	Speedups distribution		
	Speedups range	0.0012x → 143.15x	0.0073x → 20.28x
	Ratio of beneficial transformations (speedup greater than 1x)	31.98%	7.41%
	Ratio of very good transformations (speedup greater than 2x)	12.83%	1.88%
	Speedup variance	8.32	0.4402
	Code patterns	Contains programs with multiple loop nests and multiple computations, supported computation patterns are - Direct assignments - Input assignments - Stencil computations - Reduction - Convolution	Contains programs with a single loop nest and a single computation, supported computation patterns are - Direct assignments - Input assignments - Stencil computations
	Transformations	- Loop interchange - Loop tiling - Loop unrolling - Loop fusion	- Loop interchange - Loop tiling - Loop unrolling
Generation efficiency[3]	Generation overhead ratio	0.35%	93.62%
	Time needed to generate 1M datapoint	12.96 days	36.84 days
Measurements quality	Measurments stability (Coefficient of variation)	0.1768	0.2897
	Number of runs per schedule	30.5 (on average)	11 (fixed)

Table 3.2: Comparison between our dataset and the existing dataset

[3] Under the same conditions i.e. same number of runs per program (30 runs) on the same cluster.

The Generation overhead ratio is the percentage of time spent in running auxiliary operations over the total generation time, the lower is better.

$$\textit{Generation Overhead Ratio} = \frac{T_{aux}}{T_{total}} \tag{3.4}$$

Where T_{aux} is the amount of time spent on auxiliary operations such as preparing input buffers and saving intermediate results, and T_{total} is the total generation time that includes T_{aux} plus the time spent effectively running TIRAMISU programs.

The Coefficient of Variation (CV) is a standardized measure of dispersion that is commonly used to evaluate the stability of measurements from repeated experiments [Reed et al., 2002, Brown, 1998], the lower is better.

$$C_v = \frac{\sigma}{\mu} \tag{3.5}$$

where σ is the standard deviation of the measurement sequence and μ is the mean.

The table shows notable improvements in the dataset quality compared to the older dataset, the covered schedules space is bigger and the distribution of the generated speedups is more interesting thanks to the new code and schedule generation policies. Although our dataset is composed of more complex programs (multiple computations program), the data generation is 2.85 times faster (under the same conditions) as a result of numerous optimizations performed on the workflow including: minimizing the unnecessary I/O, parallelizing expensive operations such as buffer initialization. Our policy for choosing the number of runs per program also brought considerable improvements to the stability of the measures, this stability improvement happens to have a very important impact on the learning capabilities of the prediction models as discussed in the next Section 3.1.5.

3.1.5 Assessing the Execution Time's Imprecision

Programs' execution time is known to be an unstable and noisy measure [Zaparanuks et al., 2009, Chen and Revels, 2016, Mytkowicz et al., 2009], especially on modern CPU architectures, the execution time of the same program may considerably vary between different executions, and in practice, this variability is unpreventable [Mytkowicz et al., 2009, Kalibera et al., 2005] due to the lack of control on some preconditions that influences the execution time of the program. Factors that contribute to these fluctuations include: physical factors (such as CPU temperature and power availability), OS (Operating system) behavior (address space layout randomization, unpredictable activity from system daemons, and cluster managers), non-deterministic CPU behavior (speculative prefetching of cache lines, branch prediction, out-of-order execution), etc [Chen and Revels, 2016, Mytkowicz et al., 2009].

A common solution to reduce the impact of this issue is to measure the execution time of each program multiple times and retain a central tendency value [Kalibera et al., 2005] of this sequence of measurements (commonly the median), but as measuring the execution time of each program multiple times can be expensive, the question resides on what is the

optimal number of measurements that yields the best tradeoff between the time cost and measurements' stability, some works proposes complex heuristics [Chen and Revels, 2016, Kalibera et al., 2005, Kalibera and Jones, 2013] that choose dynamically the number of runs for each program, but as stated previously, we chose to use a simpler policy that offers a satisfying tradeoff.

As this dataset will be used to train a cost model on predicting the execution time speedup, this inexactness may have an impact on the model's prediction accuracy. Indeed, since theoretically a machine learning model cannot be more accurate than the data on which it was trained on, this imprecision can be considered as a lower bound when minimizing the model's prediction error. Therefore we need to evaluate this variability in order to make sure it is within an acceptable range, and in order to estimate what is the minimum prediction error a machine learning model can achieve when trained on this dataset.

To estimate this variability, we performed a rigorous experimental evaluation on the collected speedups, and we found that the speedups acquired by our running policy (30 runs per program plus an additional 15 runs for the base programs) are subject to a measurements noise of about 10.17%, the same experiment shows that this error rate cannot be reduced further without an important computational cost increase. Details about this experiment can be found in Appendix A.

3.2 Program Characterization

When using data-driven approaches on real-world data, we usually need to transform that data into an intermediate representation that serves as a model-friendly input, this process is also known as data characterization or featurization. The aim behind using an intermediate representation instead of the source data is to exhibit only the informative part from the raw data, this makes the model focus on features of interest in order to achieve better learning abilities, it also improves the computational efficiency of the training process.

In general, when designing an input characterization, we need to ensure a good representativity of the input space, the representation of each element should encode all the necessary features and the relationship between them and must be unique among all the other elements of the space, a good representation is the one from which the integrity of the original input can be restored without losing significant information.

3.2.1 Design Discussion

In our case, building the input characterization is an important challenge since the input data consists in programs, having to deal with programs means dealing with an unbounded input space of variable-sized elements, and therefore we cannot rely on the typical representation approach that consists of mapping the input space into fixed-size vectors, furthermore, the structure of a program naturally encodes complex features that cannot be efficiently represented by numerical values, hence our representation must include a non-numerical manner to encode these structural features.

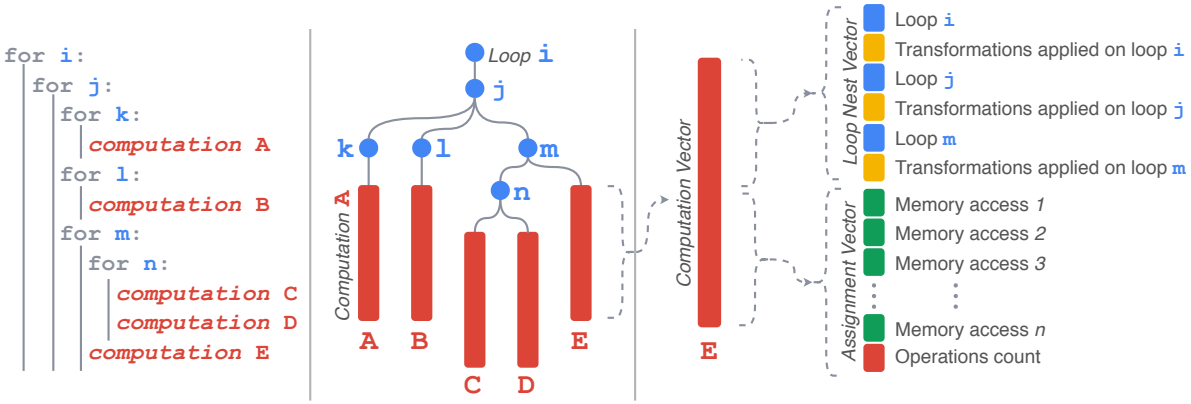
When designing a representation for programs, an intuitive approach to consider is to build the characterization upon features extracted after compiling and the program in question, such as features extracted from the assembly code or from the runtime behavior of the program in question (e.g. instruction count, performance counters, cache behavior ...), this approach may offer the opportunity to use information-rich features but comes with a considerable drawback, the overhead time required to compile or execute each candidate program during the exploration of the exponential search space of possible transformations renders this approach impractical. Another classical characterization approach that might seem interesting to consider consists on crafting a set of hand-engineered features that might simplify the learning process, but in addition to being tedious and time-consuming, this method introduces a portability flaw that makes the characterization generalize poorly when used on other hardware environments since these features are often dependent on the underlying architecture as proven in [Adams et al., 2019].

From this discussion we derive the criteria that must be fulfilled by our design choices:

- Our program characterization should model the program structure and encode the complex relationship between features.
- The feature extraction must be computationally efficient and fast enough to be usable during the exploration of the transformation search space, therefore should not involve the compilation and execution of input programs.
- Our program characterization should only depend on the programs themselves in order to be portable between different hardware architectures.

3.2.2 The Proposed Program Characterization Approach

Our representation approach consists in extracting simple high-level features directly from the program's source code, these features are then packed into a variable size representation that mirrors the program's structure. All the used features are built from the source code (i.e., we don't resort to compile or run the program) in order to guarantee the efficiency of the feature extraction process and the portability of the resulting representation. Our characterization does not employ any kind of complex feature engineering since all features are simple and high-level and hence relies entirely on the ability of the model to build the needed abstract features.



(a) Program pseudocode. (b) Program tree representation. (c) Computation vector.

Figure 3.19: Our characterization of a typical program.

We structure our representation as an ordered tree of computation vectors, the tree structure encodes the program's structure that is derived from AST (Abstract Syntax Tree) representation of the program as shown in Figure 3.19b, the computation vector depicted in Figure 3.19c wraps three representation components: the loop nest representation, the assignment representation, and the transformations' representation. In the following paragraphs, we describe each of these components, the key features encoded by each, and how these components are combined in a compact way.

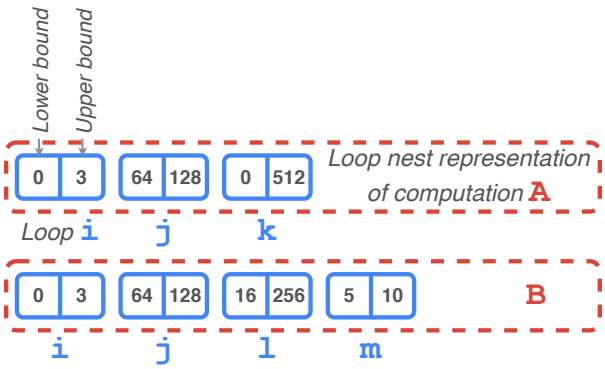
Loop nest representation. The loop nest representation encodes the sequence of loop imbrication where the computation occurs, the nesting relation is represented by ordering the loops from the outermost to the innermost as depicted in Figure 3.20, and for each loop we store its extent i.e. upper and lower bound.

```
for i in {0...3}
    for j in {64...128}
        for k in {0...512}
            A[i,j,k]←<...>
        for l in {16...256}
            for m in {5...10}
                B[i,j,l,m]←<...>
```

(a) Example pseudocode

```
var i("i", 0, 3), j("j", 64, 128), k("k", 0,
  ↪   512), l("l", 16, 256), m("m", 5, 10);
computation A("A", {i, j, k}, <...>);
computation B("B", {i, j, l, m},  <...>);
```

(b) TIRAMISU version



(c) Loop nest representation

Figure 3.20: Loop nest representation example

Assignments representation. The assignment representation encodes both the right and left-hand sides of the assignment.

The right-hand side of the assignment (the assignment expression) is represented by information about **(A)** the computed arithmetic operations and **(B)** the performed memory accesses. About **(A)** we just collect the count of how many times each arithmetic operation occurs in the expression, and to represent **(B)**, for each memory access we store a numerical ID that serves as an identifier of the accessed buffer (we assign a unique ID for each buffer declared in the program) followed by a coefficient matrix.

The coefficient matrix captures the memory access pattern into a $(k,\ n+1)$ matrix where k is the dimension of the access buffer and n is the depth of the loop nest where the assignment occurs, this matrix describes how the memory is accessed in respect to loop iterators, for example, if the following memory access $A[i_0,\ i_0+i_1,\ i_1-2]$ occurs inside a two dimensional loop nest the associated coefficient matrix M will be the following:

$$M = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & -2 \end{bmatrix} \tag{3.6}$$

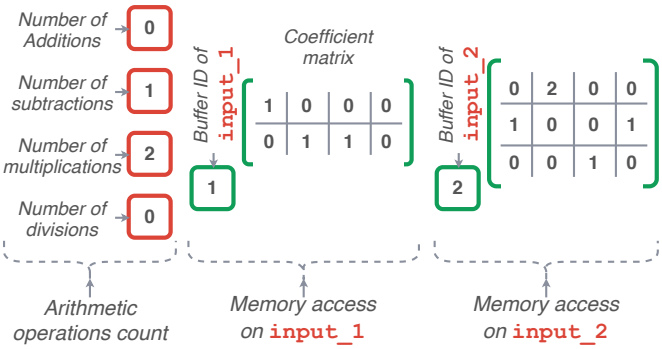
Figure 3.21 illustrates an example of the right-hand side assignment representation.

```
for i in {0...3}
    for j in {64...128}
        for k in {0...512}
            A[i,j,k]← 3 * input_1[i, j+k]
  ↪ -  2 * input_2[2*j, i+1, k]
```

(a) Example pseudocode

```
var i("i", 0, 3), j("j", 64, 128),
  ↪ k("k", 0, 512);
computation A("A",{i,j,k}, 3*input_1(i,j+k)
  ↪  - 2*input_2(2*j, i+1, k));
```

(b) TIRAMISU version



(c) Right-hand side assignment representation

Figure 3.21: Right-hand side assignment representation example

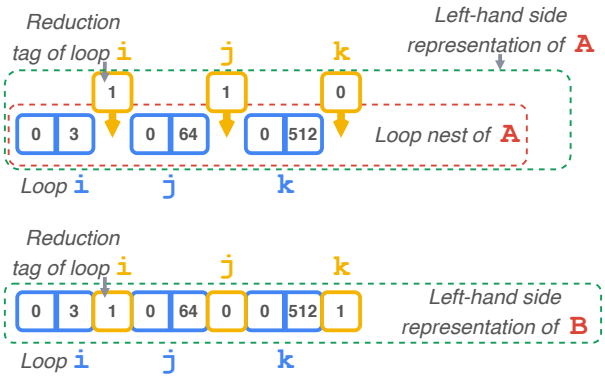
The left-hand side of the assignment is represented by storing the dimensions and the extent of each dimension of the buffer that receives the assignment, but as in TIRAMISU these dimensions always match some loop extents of the nest where the assignment occurs, we merge the left-hand assignment representation with the loop nest representation in order to avoid redundancy by adding reduction tags to each loop level. Reduction tags are boolean values that we use to differentiate between the loops that are contributing in defining the buffer's dimension from the others as depicted in Figure 3.22.

```
A ← int[3][64]
B ← int[3][512]
for i in {0...3}
    for j in {0...64}
        for k in {0...512}
            A[i,j] ← A[i,j] + C[j,k]
            B[i,k] ← B[i,k] + C[j,k]
```

(a) Example pseudocode

```
var i("i", 0, 3), j("j", 0, 64), k("k", 0,
  ↪  512);
computation A("A", {i, j, k});
A.set_expression(A(i,j,k) + C(j,k));
computation B("B", {i, j, k});
B.set_expression(B(i,j,k) + C(j,k));
buffer A_b("A_b", {i,j}, p_int32, a_output);
buffer B_b("B_b", {i,k}, p_int32, a_output);
A.store_in(&A_b);
B.store_in(&B_b);
```

(b) TIRAMISU version



(c) Representing the left-hand side assignment by inserting reduction tags to the *Loop nest vector*

Figure 3.22: Left-hand side assignment representation example

Loop transformation representation. Each transformation can be characterized by its type, its parameters, and its application point. Since transformations are applied to computations with respect to loop levels, we create a transformation vector for each loop level of the nest, each transformation vector contains a sequence of boolean tags that describes which transformations are applied along with the transformation parameters (when it applies),

these vectors are respectively inserted into the loop nest vector according to the affected loops as shown in Figure 3.23. The transformations that involve changing the structure of the program (e.g., loop fusion) are also applied to the program structure representation that we will describe next.

```
A ← int[128][64]
for i in {0...128}
    for j in {0...64}
        for k in {0...512}
            A[i,j] ← A[i,j] + C[j,k]

interchange(i, j) // interchange i with j
tile(j, i, 16, 32)// tile j and i with factors
  ↪ of 16 and 32
unroll(k, 8) // unroll k with a factor of 8
```

(a) Example pseudocode

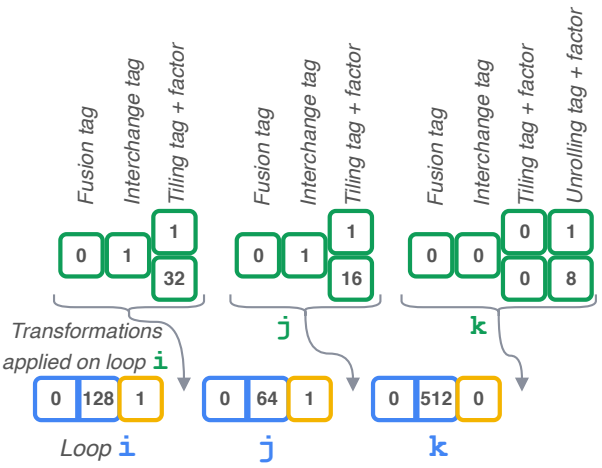
```
var i("i",0, 128), j("j",0, 64), k("k",0, 512)

computation A("A", {i, j, k});
A.set_expression(A(i,j,k) + C(j,k));

A.interchange(i, j);
A.tile(j, i, 16, 32)
A.unroll(k, 8)

buffer A_b("A_b", {i,j}, p_int32, a_output);
A.store_in(&A_b);
```

(b) TIRAMISU version



(c) Representing the loop transformations by inserting schedule features to the *Loop nest vector*

Figure 3.23: Loop transformation representation example

Program structure representation. The program is represented as a tree structure where leaves are the computation vectors and internal nodes are the loop levels modeling the AST (Abstract Syntax Tree) structure of the program as depicted in Figure 3.19b, we use this tree representation in order to capture positional features that can hardly be encoded with numerical values, such as the relative positions of assignments in respect to each other, the loop arborescence, and the relative distance between memory accesses.

Table 3.3 summarizes the full list of features used in our representation. All features are integer values except Tag features which are boolean values. In practice, we set $n = 7$ and $m = 21$ as the deepest loop nest in our dataset has a length of 7 nested loops and the maximum number of terms used in an assignment is 21. When needed, a zero-padding is added to the Loop Nest Vector and Assignment Vector.

Loop Nest Vector	Loop_1	Upper bound, Lower bound, Reduction tag.
		Fusion tag, Interchange tag, Tilling tag, Tilling factor.
	Loop_2	Upper bound, Lower bound, Reduction tag.
		Fusion tag, Interchange tag, Tilling tag, Tilling factor.
	⋮	⋮
	Loop_n	Upper bound, Lower bound, Reduction tag.
		Fusion tag, Interchange tag, Tilling tag, Tilling factor, Unroll tag, Unrolling factor.
Assignment Vector	Memory access $_1$	Coefficient matrix, Buffer ID.
	Memory access $_2$	Coefficient matrix, Buffer ID.
	⋮	⋮
	Memory access $_m$	Coefficient matrix, Buffer ID.
	Operations count	Number of Additions, Number of Multiplications, Number of Subtractions, Number of Divisions.

Table 3.3: A detailed listing of the features that composes the *Computation Vector*.

3.2.3 A Detailed Characterization Example

In this section, we provide a detailed example of how we characterize a full input program.

We take an example of a synthetic linear algebra algorithm that performs a weighted sum of two matrix multiplications, the performed operation is denoted as $R = \alpha \times A.B + \beta \times C.D$ where A, B, C, D are input matrices with sizes (64, 256), (256, 128), (64, 512), (512, 128) respectively, α and β are two scalar values 0.7 and 0.3 respectively, R is the (64, 128) matrix that receives the final result.

As shown in Figure 3.24, this algorithm can be implemented in TIRAMISU using three computations representing the assignments of the original program, the two first computations, $T1$ and $T2$, are reductions and the third, R, consumes their results, we assign to the buffers A, B, C, D, $T1$, $T2$, R the numerical IDs 1, 2, 3, 4, 5, 6, 7 respectively that we will later need in the representation. We apply a schedule to this program that consists in, for the sake of example, a 2D loop tiling for the two outermost loops (i, j) with parameters (16, 32) and loop unrolling for the two loops (k, l) with the same unroll factor of 8.

Figure 3.25 shows the full representation of the previous example program (Figure 3.24), the upper left corner illustrates the tree representation of the program, then the content of each of the three computation vectors is separately detailed in the rest of of the figure. All the features of each computation are flattened into a one-dimensional vector, zeros padding is added when needed in order to even out the computation vector sizes.

```
1  var i("i", 0, 64) , j("j", 0, 128) , k("k", 0, 256),  l("l", 0, 512);
2
3  input A("A", {i, k}, p_float32), B("B", {k, j}, p_float32), C("C", {i, l}, p_float32),
     ↪ D("D", {l, j}, p_float32);
4
5  computation T1("T1", {i, j, k});
6  T1.set_expression(T1(i,j,k) + A(i,k)*B(k,j);
7  computation T2("T2", {i, j, l});
8  T2.set_expression(T2(i,j,l) + C(i,l)*D(l,j);
9  computation R("R", {i, j}, 0.7*T1(i,j,0) + 0.3*T2(i,j,0));
10
11 T1.then(T2, j).then(R, j);
12 T1.tile(i, j, 16, 32); T2.tile(i, j, 16, 32); R.tile(i, j, 16, 32):
13 T1.unroll(k, 8); T2.unroll(l, 8);
14
15 buffer b_T1("b_T1", {i,j}, p_float32, a_temporary);
16 T1.store_in(&b_T1);
17 buffer b_T1("b_T2", {i,j}, p_float32, a_temporary);
18 T2.store_in(&b_T2);
19 buffer b_R("b_R", {i,j}, p_float32, a_output);
20 R.store_in(&b_R);
```

Figure 3.24: An example input TIRAMISU program.

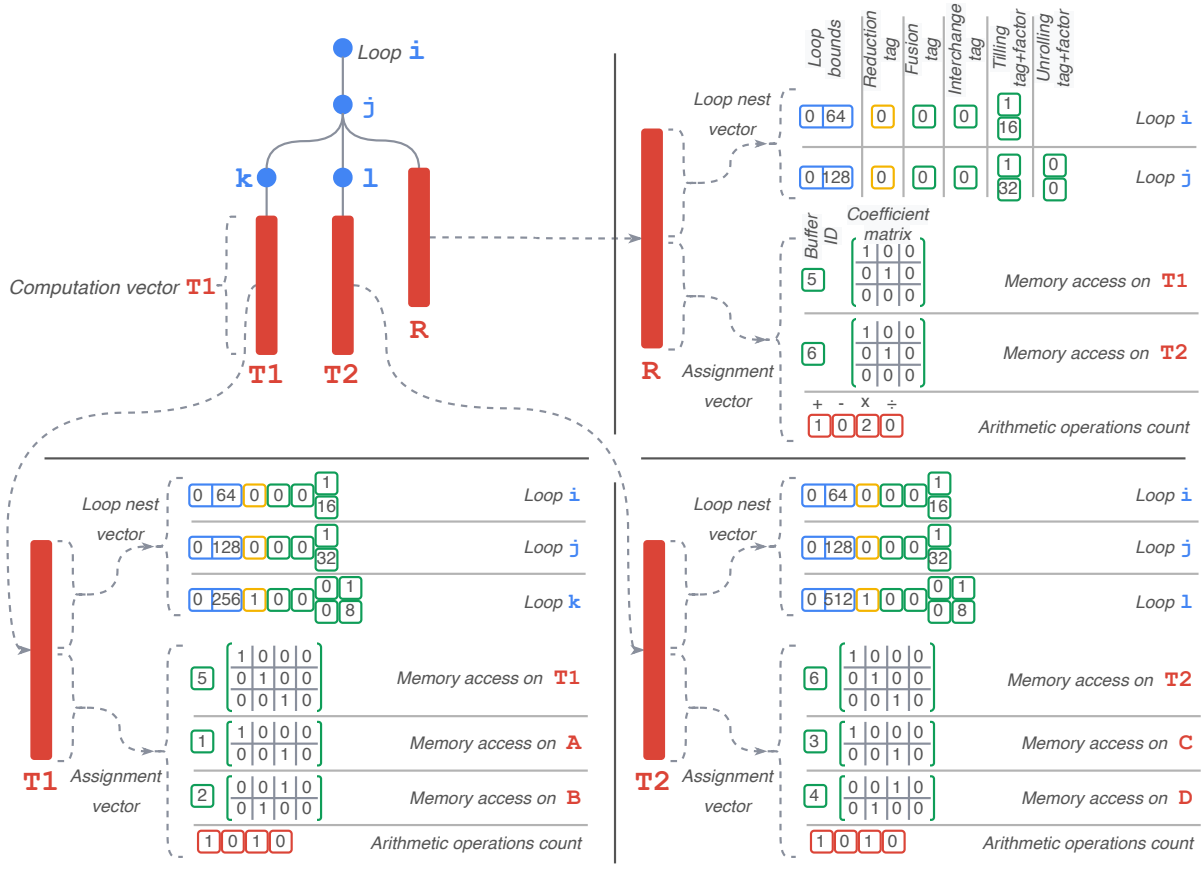


Figure 3.25: The full representation of the program presented in Figure 3.24.

3.3 Model Architecture

Building a cost model based on deep learning rather than on machine learning is motivated by the results that show the universality of neural networks and their exponential advantages over hand-crafted features [Kawaguchi et al., 2017]. In this cost model, the main architectural considerations are to choose the type of units (RNN, LSTM, gated NN, feedforward NN, etc.), the depth of each component, the size of each layer, and the activation functions linking different layers. Since our model is parametric (have a fixed number of parameters), making certain architectural decisions is equivalent to imposing assumptions about the form of the approximated function. This is known as inductive bias and it has a major influence on the accuracy of the model. The universal approximation theorem [Hornik et al., 1989] guarantees the existence of a large-enough neural network to achieve any degree of accuracy we desire. Unfortunately, there is no general method to find the architecture of this neural network. Besides, the “no free lunch" theorem states that there is no superior algorithm to train this desired architecture if we assume that it is found. Architectures with a high number of parameters generally have better performances, however they could lead to data overfitting and they are often harder to train. We must adopt an experimental approach to select the best design choices. The experimentation is guided by the generalization error, which is the evaluated error on new data that was not seen during the training (called the test set).

In this section, we will detail the final architecture that we propose for the cost model, a recursive LSTM based architecture, but as throughout this project we explored and developed multiple architectures before achieving satisfying results, we will also briefly expose the other architectures that were explored.

3.3.1 Design Discussion

We model the problem of speedup estimation as a regression problem: given an algorithm and a set of code transformations (schedules), our model predicts the speedup expected when applying the suggested code transformations compared to the base program (i.e. without applying any code transformations). The representations fed into the model do not rely on heavily engineered features, nor do they include any compilation information or performance counters. If that was the case, a simple neural network would be enough to combine those features and conclude the speedup. The inputs are only based on simple high-level source code features and embed the program structure. The simplicity of the input features must be overcome by a sophisticated model architecture able to extract the abstract program properties to predict the overall speedup.

Sequence modeling techniques such as RNNs, LSTMs, or Gated RNNs are the natural solutions to address the variable length of the adopted representation. In this case, the model maps an arbitrary length sequence of computation representations to a fixed-length vector that encodes the program properties. Unquestionably, the hierarchical structure of the AST

and the relative position of the computations are lost for most programs, yet some code transformations like fusion will be poorly represented. Figure 3.26 reflects two TIRAMISU programs having computations with the same expressions with the same loop extents, but two different structures. In regards to a simple sequence model, the two program representations are confused as a sequence of the same three computations. The non-fused loop *k* (Figure 3.26 left) is completely out of sight for such models. This major limitation pushed us to consider an architecture that, as sequence models, can support variable-length representation, but also takes into account the recursive and hierarchical nature of the program's AST. We propose a Recursive-LSTM-based architecture that is built by stacking multiple task-specific layers.

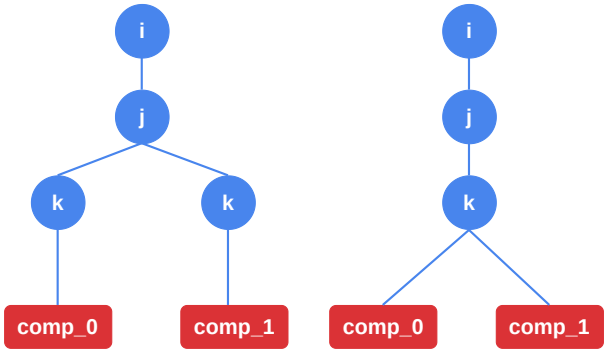


Figure 3.26: Examples of two trees representing TIRAMISU programs

3.3.2 The Recursive LSTM Architecture

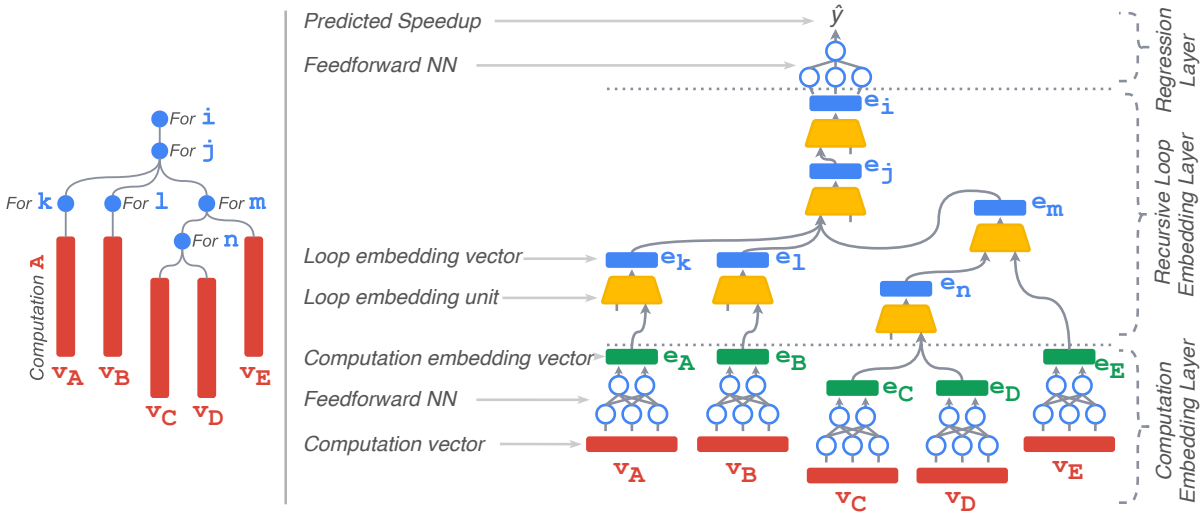
We design our cost model's architecture to support the variable size and recursive nature of our program characterization by combining Recurrent and Recursive Neural Networks. We dissect our model's architecture into three layers:

Computation embedding layer. This first layer preprocesses each computation independently by mapping the high-dimensional *computation vectors* into a compact embedding of abstract features. After *log*-transforming the non-boolean features, all *computation vectors* of the program are processed through a feedforward network. This *log*-transformation is necessary since these features have a large dynamic range and most of them are expected to be multiplied with other features to compute the final speedup.

Recursive loop embedding layer. The *computation embeddings* resulting from the previous layer are then processed recursively following the tree structure of the program using the *loop embedding unit* as shown in Figure 3.27b. At a given loop level, the *loop embedding unit* summarizes the program up to that loop into a *loop embedding vector*. The *loop embedding unit*, as depicted in Figure 3.28, is composed of two separate LSTM cells [Hochreiter and Schmidhuber, 1997] and a feedforward layer. Recalling that the internal nodes of a program's tree representation have two types of children: loop variables (other internal nodes) and computations (leaves). One LSTM cell is dedicated to each type. At each loop level, the first LSTM is fed with the embedding vectors of computations that are nested directly in that loop level

while the second LSTM is fed with the embedding vectors of the previous loop levels that resulted from the previous *loop embedding units*. At this point, two hidden states are produced from mapping those two sequences. The two hidden states of the LSTMs are merged using a feedforward layer into a *loop embedding vector*. The purpose of this recursive embedding is to selectively incorporate information from each computation respecting its positional relations with the other computations. At each loop level, the loop embedding vector is assumed to contain the needed set of automatically extracted features covering the corresponding subtree. Thereupon, the *program embedding vector* (which is the *loop embedding vector* of the root loop) covers the complete program.

Regression layer. The *program embedding vector* outputted by the previous layer is finally fed to a shallow feedforward neural network with one neuron output layer that performs a regression in order to predict the speedup.



(a) AST of an example input program.

(b) Processing the example program through the three layers of the cost model.

Figure 3.27: Processing an example input program through the cost model's architecture.

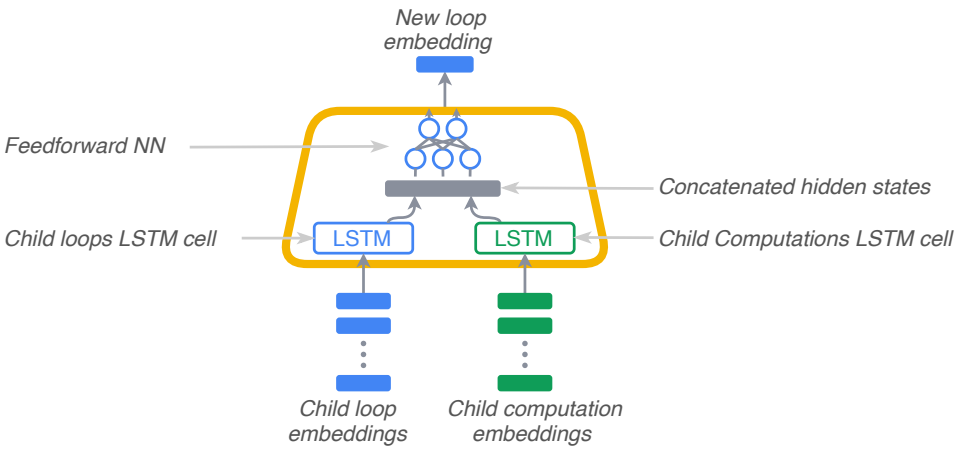


Figure 3.28: Loop embedding unit

To illustrate the workings of this architecture, we consider processing the program presented by its AST representation in Figure 3.27a. Figure 3.27b provides an overview of the process while Table 3.4 describes it in detail. The adopted notation is as follows:

- $\mathbf{v}_{comp}$: the representation vector of the computation *comp*.
- $\mathbf{e}_{comp}$: the embedding of the computation *comp*.
- $\mathbf{e}_x$: the embedding of the loop x.
- $\mathbf{h\_loop}_x$: the hidden state that summarizes the loop children of the loop x.
- $\mathbf{h\_comp}_x$: the hidden state that summarizes the computation children of the loop x.

Computation embedding layer	In parallel, the computation vectors v_A, v_B, v_C, v_D, v_E are fed into the feedforward NN of the *computation embedding layer* and transformed to the *computation embeddings* : e_A, e_B, e_C, e_D, e_E.	
Recursive loop embedding layer	Step 1	e_A the embedding of the single computation child of the loop k, is mapped by the child computation LSTM of the loop embedding unit to the computation hidden state h_comp_k. Since the loop has no loop children, the hidden state corresponding to the loop children of k (h_loop_k) is set to zeros. Finally, h_comp_k and h_loop_k are concatenated and processed by the feedforward NN of the loop embedding unit to produced the embedding of the loop k : e_k.
	Step 2	Same as Step 1, e_l the embedding of the loop l, is the result of handling its unique child computation embedding vector e_B.
	Step 3	The loop n has two children : computation C and D. Thus, the child computation LSTM of the loop embedding unit maps the sequence of computation embeddings (e_C, e_D) to the computation hidden state h_comp_n. The rest of the process until obtaining the embedding of the loop n (e_n) is similar to Step 1.
	Step 4	The loop m has two children with different types : the computation E and loop n. The computation hidden state h_comp_m is the result of mapping the computation embedding e_E by the child loop LSTM of the loop embedding unit. Rather than the previous steps where the loop children hidden states were zero initialized, h_loop_m is attained by applying the child loop LSTM cell over the loop embedding e_n. Finally, h_comp_m and h_loop_m are concatenated and processed by the feedforward NN of the loop embedding unit to produced the embedding of the loop m : e_m.
	Step 5	The loop j has three loop children k, l and m. Their embeddings e_k, e_l, and e_m are mapped to h_loop_j by the child loop LSTM cell. On the other hand, h_comp_j is zero initialized because the loop j has no computation children. The two hidden states h_loop_j and h_comp_j are transformed to e_j in the same fashion as the previous steps.
	Step 6	The loop i has one loop child j. The loop embedding e_j is diverted to e_i, much like Step 5.
Regression layer	At long last, The loop embedding of the root loop e_i is drawn up to the speedup scalar $\hat{y}$ by the feedforward NN of the regression layer.	

Table 3.4: Processing steps of the program presented in Figure 3.27a through the three layers of the cost model.

3.3.3 Model Architecture Details and Training Methodology

The computation embedding layer is a fully connected multilayer perceptron (i.e., a feedforward neural network) that takes 1235-dimensional computation vectors and generates 180-dimensional embeddings. We use 3 intermediate layers of 600, 350, 200 neurons respectively. The output of each layer is transformed by the *ELU* (Exponential Linear Unit) activation function and fed to a dropout layer with a dropout probability of 0.225, and then passed to the next layer. This succession of the activation function and the dropout layer is applied to all the neural networks of this model.

The two LSTM cells in the loop embedding unit have identical input and hidden vector sizes that correspond to the output of the computation embedding layer (180). The feedforward neural network that maps the concatenated hidden vectors to 180-dimensional loop embedding has one intermediate layer of size 200.

The regression layer that maps the *program embedding* vector to a speedup value, has two intermediate layers with 200 and 180 neurons. All model parameters are learnable from the computation embedding layer to the regression layer by way of the recursive loop embedding layer.

For the loss function, we used Mean Absolute Percentage Error (MAPE), a normalized metric based on L_1. This loss function is suitable for speedup prediction because the target value is positive by design. In addition, the function motivates the model to be equitably accurate in the wide range of speedups we have. The Glorot initialization [Glorot and Bengio, 2010] is adopted for all weights of the model.

We train our model using AdamW [Loshchilov and Hutter, 2017] with a weight decay coefficient of 0.0075. The learning rate is scheduled by the One Cycle Policy [Smith and Topin, 2017] with a maximum learning rate of 0.001. The other optimizer parameters are left on their default values.

The training set is processed in batches of 32 data points. Each batch is formed by code transformations belonging to the same algorithm. The rationale for this grouping is that it is faster to operate on data points having the same tree structure. The best accuracy is achieved after about 700 epochs of training.

The training is performed on a MIT's Nvidia DGX-1[4]. The machine is equipped with 8 NVIDIA Tesla V100-SXM2 [5] Graphics Processors Units. More details can be found in Table 3.5.

[4] https://www.nvidia.com/en-us/data-center/dgx-1/
[5] https://www.nvidia.com/en-us/data-center/v100/

GPUs	8X Tesla V100
Performance (GPU FP16)	1 petaFLOPS
NVIDIA CUDA Cores	40,960
GPU Memory	8X 32GB (256GB)
CPU	Intel Xeon E5-2698 v4 2.2 GHz
System Memory	512GB
Operating system	Ubuntu 16.04.6 LTS

Table 3.5: Specifications of the NVIDIA DGX-1 server

3.3.4 Other Explored Architectures

We explored several neural network architecture alternatives with varying levels of complexity before arriving at the best performing recursive LSTM described in Section 3.3. In the following paragraphs, we provide a brief description of the most significant architectures we tested throughout this project.

Feedforward Neural Networks based architecture. [Henni and Mekki, 2019] used a five-layer feedforward NN to handle single computation programs. We adopted the same architecture but with an expanded input layer to add the support for multiple computation programs. For this architecture, input programs are represented by concatenating the computation vectors into a fixed size input vector (Figure 3.29). This first straightforward choice has the considerable limitation of not supporting variable input sizes and hence supports only programs that contain up to a certain number of computations. For experimenting this architecture, we set the maximum number of computations to 4, if a program has less than 4 computations the remaining space is padded with zeros.

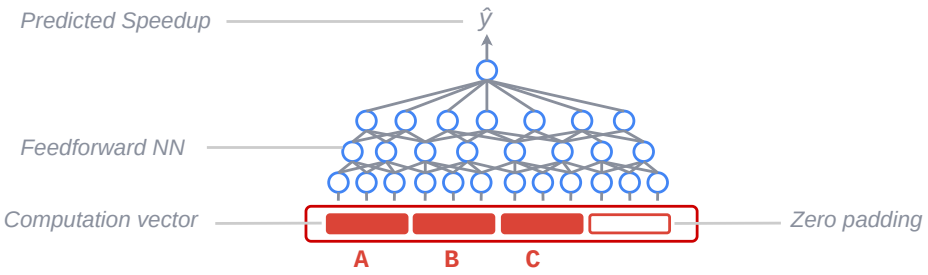


Figure 3.29: Processing a 3-computation TIRAMISU program with the feedforward architecture

Simple Recurrent Neural Networks based architecture. The second architecture that we explored is a three-layers RNN-based model. Maintaining the same vocabulary established in Section 3.3, the *computation embedding layer* and the *regression layer* are identical to the ones in the recursive LSTM architecture. In contrast, the *loop embedding unit* is replaced

with an RNN cell (Figure 3.30). The sequence of *computation embedding* vectors is processed sequentially without taking into consideration the loops hierarchy.

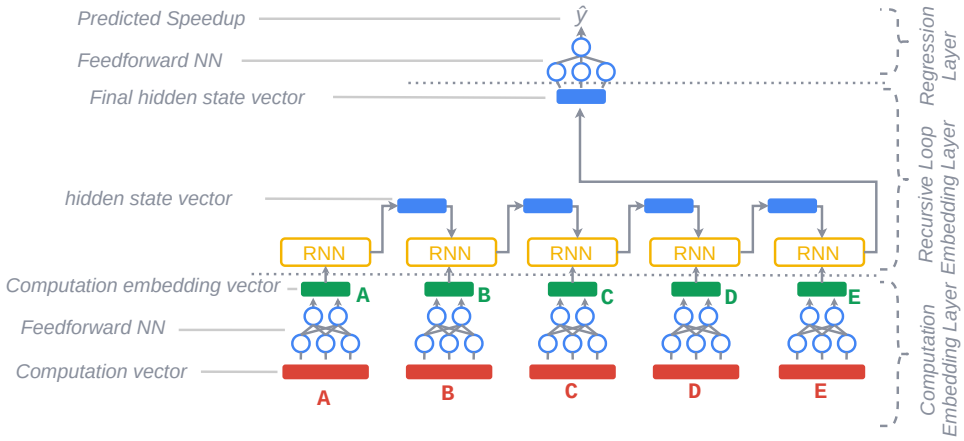


Figure 3.30: Processing the program presented in Figure 3.27a through the three layers of the RNN-based cost model.

Tree-LSTM based architecture. Like the previous architecture, the *computation embedding layer* and the *regression layer* are identical to the ones from the recursive LSTM architecture. A child-sum Tree-LSTM cell [Tai et al., 2015] is incorporated into the *Recursive loop embedding layer* replacing the *Loop embedding unit* (Figure 3.31). This generalized LSTM is able to incorporate information from multiple child units but because loop nodes can have two different types of children (loops and computations), the Tree-LSTM cell suffers from this heterogeneity.

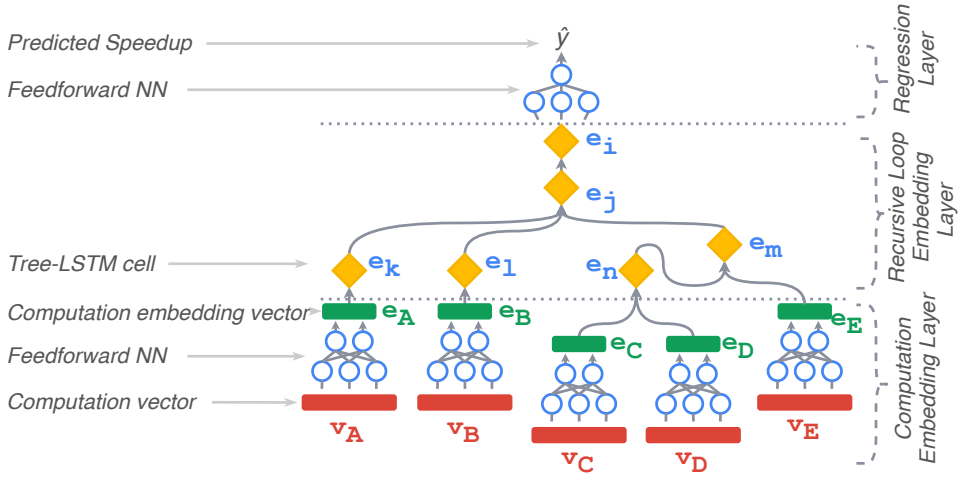


Figure 3.31: Processing the program presented in Figure 3.27a through the three layers of the Tree-LSTM based model.

3.4 Implementation Details

From data generation to evaluation experiments, we used many programming languages, libraries, and tools. In this section, we give details on the use of each of them.

Utilization of C++ Since TIRAMISU is embedded in C++, this programming language is extensively used in this project. Wrapper programs are a perfect example, those programs are intended to perform two roles. First, they declare and initialize multidimensional arrays via multi-threading. Those arrays will be passed as input buffers of a TIRAMISU program. Second, the wrappers implement Algorithm 3.18. Precise execution time measurements are taken using the *Time library*.

Utilization of Python Our synthetic program generator is entirely developed with Python 3 (version 3.8.1). We used Anaconda [6], a free and open-source distribution of Python that comes with over 250 packages automatically installed for machine learning applications, data science. Among other recurrent tasks, the compilation and the execution of TIRAMISU programs are also automated with Python. Alongside the build-in language's features, various Python packages are invoked to simplify some tasks, like manipulating code with regular expressions (*re* package), generating JSON annotation files (*json* package), using operating system interfaces (*os* package), generating probability distributions (*random* package), efficiently handling millions of matrices (*numpy* package), analyzing dataset's statistics and prediction results (*pandas* package), and creating visualizations (*matplotlib* package).

Utilization of PyTorch PyTorch [7] [Paszke et al., 2019] is an imperative style, high-performance deep learning library. PyTorch has a Python interface and relies on an efficient C++ core to implement the tensor data structure (multi-dimensional arrays), operators, and parallel primitives. As a result, PyTorch provides an array-based programming model accelerated by GPUs and differentiable via automatic differentiation.

All cost model architectures introduced in this project are implemented using version 0.4.1.post2 of PyTorch. PyTorch tensors can be created from Numpy arrays sharing the same memory, this feature allows us to optimize memory when dealing with millions of matrices.

Utilization of Slurm Slurm Workload Manager [8] is an open-source, fault-tolerant, and highly scalable cluster management and job scheduling system for Linux clusters. We used Slurm to distribute compilation and execution jobs of TIRAMISU programs over MIT's cluster. Slurm functions were extremely useful. Namely, the exclusive allocation of resources, which guarantees that the measured execution time of a TIRAMISU program is not disturbed by other programs running on the same node. Bearing in mind that some dataset construction jobs took up to a month, it would be very difficult to accomplish large datasets without Slum's monitoring framework.

[6] https://www.anaconda.com/products/individual
[7] https://pytorch.org/
[8] https://slurm.schedmd.com/

Utilization of JupyterLab JupyterLab [9] is a web-based interactive development environment for creating and sharing documents that contain live code, equations, visualizations, and narrative text. JupyterLab supports many programming languages including Python. It offers a powerful user interface to browse files and to visualize rich outputs. We used JupyterLab to remotely edit and execute code in MIT's DGX-1 machine.

3.5 Conclusion

In this chapter, we presented our proposed design and implementation of the TIRAMISU cost model. We provided a detailed description of our approach for building the three submodules of our contribution and we discussed the different design choices that we made for each submodule.

In the first section of this chapter, we explained the process of constructing our 1.8 million points dataset that will be used to train our cost model. Our dataset was built using a novel synthetic TIRAMISU program generator that we developed and a highly optimized dataset construction pipeline. In the second section, we discussed the challenges of designing a characterization that can mirror complete TIRAMISU programs. Then, we introduced our proposed AST-based characterization that combines program features and schedule parameters. In the third section of this chapter, after discussing the challenges of designing an efficient architecture, we detailed the architecture that we ended up with for the cost model, the Recursive LSTM. Then we gave a quick overview of other intermediate architectures that we explored throughout this project.

In the next chapter, we provide a rigorous evaluation of the proposed cost model on real-world use cases.

[9] https://jupyter.org/

4

Tests and Evaluation

In this chapter, we propose to evaluate our cost model through various experiments against various settings. This evaluation will demonstrate the quality of our model's predictions and its influence on the performance of the overall auto-scheduler. This evaluation will also serve as a justification for our design choices concerning the model's architecture by including a comparison of the impacts of the major design alternatives on the overall performance of the model.

We structure this evaluation section as follows:

Model-only evaluation: in this first part, we will evaluate the model independently from the auto-scheduler, this evaluation will put the focus on the quality of the model itself against diverse criteria including prediction accuracy and inference speed, this first evaluation will be done on synthetic TIRAMISU programs.

Exploration+Model evaluation: This second part consists in evaluating the performance of the auto-scheduler when it uses the model as a cost function, through this evaluation we will compare the performances of the exploration algorithm combined with our cost model in contrast to its performances when it relies on ground truth measurements, this second evaluation will be performed on real-world TIRAMISU programs (benchmark programs).

Architecture design alternatives comparison: In this last part, we will compare, through experiments, some major conceptual alternatives that we encountered when designing the cost model's architecture.

4.1 Model-Only Evaluation

The purpose of this first evaluation is to uncover the strengths and weaknesses of the cost model independently from the auto-scheduler, we will use a new set of programs (that did not figure in the training set) and compare the speedups predicted by the cost model to ground truth ones, the quality of these predictions will be quantified using various relevant metrics.

Before heading to the actual evaluation, we will first start by defining the metrics that will be used throughout this section, and we will describe the test set on which the experiments will be performed.

4.1.1 Used Metrics: Definition and Discussion

We used various metrics to evaluate the proposed cost model. Mean Absolute Percentage Error (MAPE) is one of the most popular metrics used in the literature for regression problems [Botchkarev, 2019]. As shown in Equation 4.1, the metric operates over two series of values, measured values (Y) and predicted values ($\hat{Y}$). First, the absolute point distance is calculated between the two series: $|y_i - \hat{y}_i|$. This point distance has non-negative values and avoids mutual cancellation of the positive and negative errors. Then, a division by the measured speedup y_i is invoked to normalize the point distance. Measured speedups are strictly positive by definition, so we don't need to worry about zero division. The normalization ensures that the metric penalizes small errors made in small values of speedups with the same order of magnitude as large errors in speedups of large values. The final MAPE scalar value is obtained by calculating the arithmetic mean of the normalized point distances over the data set.

$$MAPE(Y, \hat{Y}) = \frac{1}{n} \sum_{i=1}^{n} \frac{|y_i - \hat{y}_i|}{y_i} \tag{4.1}$$

As the cost model is intended to be integrated into a search space exploration, the common use case is as follows: the search algorithm feeds the model a set of scheduled programs; the model returns the predicted speedups; the search algorithm selects a subset based on the predicted values. The Greedy algorithms maintain the best K schedules. In this case, the ranking of the schedules is more important than the absolute precision of the prediction. We measure the ranking quality of the models by two specialized metrics: Spearman's correlation and *nDCG*.

Spearman 's correlation of two variables Y and $\hat{Y}$ is defined as the correlation coefficient between their rank values Rg and $\hat{Rg}$ (Equation 4.2). In in our case, to calculate the spearman's correlation between the measured speedups $(y_1, ..., y_n)$ and the predicted speedups $(\hat{y}_1, ..., \hat{y}_n)$, we form new two series $(rg_1, ..., rg_n)$ and $(\hat{rg}_1, ..., \hat{rg}_n)$, where rg_i is the rank of y_i

among the actual speedups, and similarly, $\hat{rg}_i$ is the rank of $\hat{y}_i$ among the predicted speedups. Then, we report the correlation between these two rank series.

$$Spearman(Y, \hat{Y}) = Corr(Rg, \hat{Rg}) = \frac{cov(Rg, \hat{Rg})}{\sigma_{Rg}\sigma_{\hat{Rg}}} \tag{4.2}$$

The Normalized Discounted Cumulative Gain ($nDCG$) is a family of ranking measures widely used in applications and recommender systems [Wang et al., 2013]. The measures have values between 0 and 1. The higher $nDCG_K$ values mean that the first K elements (according to the model prediction) are relevant. As formulated in Equation 4.3, We obtain the $nDCG_K$ score by summing the measured speedups (y_i) ranked in the order induced by the predicted speedups, after applying a logarithmic discount. Then, we divide by the best possible score: $iDCG_K$. $iDCG_K$ is the Ideal Discounted Cumulative Gain obtained for a perfect ranking (Equation 4.4), where REL_K is the list of schedules ordered by their measured speedups.

$$nDCG_K = \frac{1}{iDCG_K} \sum_{i=1}^{K} \frac{y_i}{log_2(1+i)} \tag{4.3}$$

$$iDCG_K = \sum_{i=1}^{|REL_K|} \frac{y_i}{log_2(1+i)} \tag{4.4}$$

$nDCG$ is a strong indicator of the performance of the cost model, especially when integrated with greedy algorithms (see Section 4.2.2). When cost models are plugged in a greedy search exploration algorithm, all that matters is their ability to correctly rank the candidate schedules. $nDCG_K$ is designed to target this quality. Unfortunately, this metric cannot be used to train models because it is not differentiable. A detailed justification of the relevance of the $nDCG$ metric over the other regular regression metrics (for our case) can be found in Appendix B along with examples and calculation details.

4.1.2 Description of the Test Set

This first evaluation will be done on a synthetic test set, this test set consists of synthetic programs generated by the code generator described in Section 3.1, the programs contained in this set are unseen during the training process meaning that the model hasn't been trained on those programs. This test set will show whether the model can generalize what it has learned to new inputs. Details of this test set are provided in table 4.1

Synthetic test set	
Number of data points	276017 scheduled program
Number of programs	8502
Number of schedules per program	32 (fixed)
Speedups range	0.0018x → 99.26x
Speedups variance	8.12

Table 4.1: Description of the synthetic test set.

4.1.3 Overall Scores of the Cost Model

In order to give quick insights about the model's overall performance, we summarized its scores on the most relevant metrics in the following table (Table 4.2)

Metric	Score
Mean absolute percentage error (*MAPE*)	16.32%
$MAPE$ above measurements-induced error[a]	06.15%
Spearman's rank correlation	0.9558
Average program-wise $nDCG$	0.9833
Average program-wise $nDCG_1$	0.9400
Average program-wise $nDCG_5$	0.9542
Average program-wise $nDCG_{10}$	0.9645

Table 4.2: Overall scores of the cost model on various metrics

[a] Estimated $MAPE$ after omitting the measurements-induced error described in Section 3.1.5

The Mean Absolute Percentage Error (*MAPE*) of the model's predictions is 16.32%, this means that the predicted speedups are different from the recorded ones by 16.32% on average. This error rate is considered low enough for the cost model, especially since the recorded speedups themselves are imprecise by about 10.17% as explained in Section 3.1.5.

Spearman's rank correlation of 0.9558 shows that the predicted ranking of speedups is very close to the original one. Furthermore, the Average program-wise $nDCG$ (average of the $nDCG$ calculated over the 32 schedules of each program) shows that the relevance of the model's ranking of schedules is 98.33% of the ideal ranking.

The $nDCG_1$ score shows that, for any given program, the best schedule selected by the model (the schedule that is predicted to yield the highest speedup) has on average a speedup of 94% of the real best schedule ($\frac{\textit{Speedup of the best schedule according to the model}}{\textit{Speedup of the real best schedule}} = 94\%$)

$nDCG_5$ and $nDCG_{10}$ are relevant for search techniques that use the model to select the top 5 or top 10 schedules in order to evaluate them experimentally.

4.1.4 Comparing the Predicted and the Measured Speedups

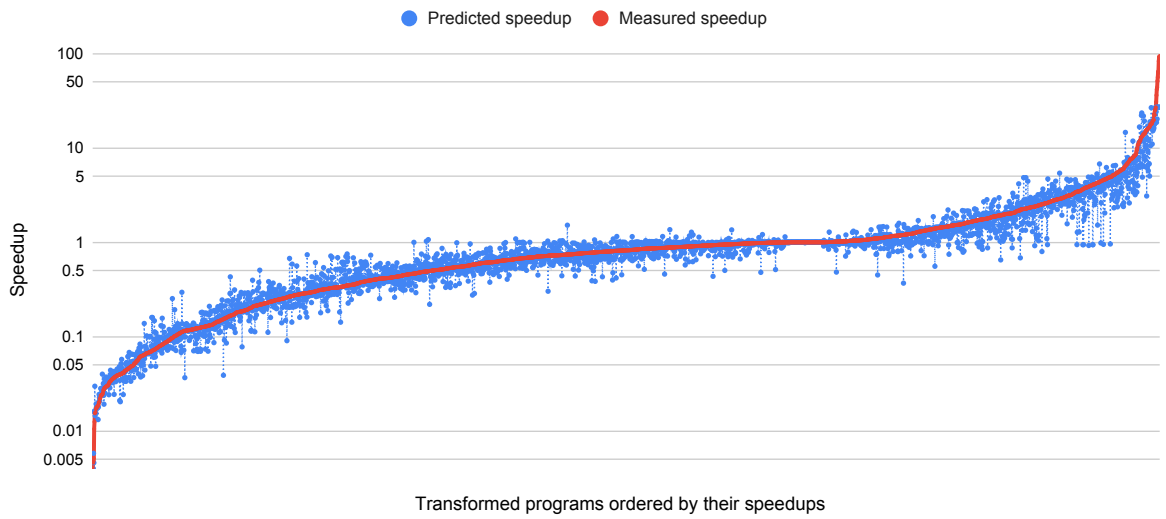


Figure 4.1: Predicted speedups compared to measured speedups over 100 × 32 synthetic programs. The speedups are ordered in ascending order.

We plot in Figure 4.1 the measured speedups alongside with the corresponding predicted speedups. The chart is a scatter of 3200 data points, which is 100 different programs randomly selected from the test set with 32 schedules for each. The speedups are arranged in ascending order to visualize the performance of the cost model under different speedup ranges. The figure shows that the model is equitably accurate on low ($\ll 1$) and high ($\gg 1$) speedups. We notice that the bias is much lower around 1. This is hustled by the fact that the ground truth speedup distribution is bell-shaped around 1. To put it differently, the measured speedup values between 0.75 and 1.25 represent 33.86% of the entire dataset. As a result, the model learns to be more precise in those frequent cases.

For some programs, most of the corresponding schedules have close speedup values. So an undesirable phenomenon that may occur that the model may ignore the schedules and just rely only on the program's characteristics to predict an average speedup value for each program. In order to prove that this phenomenon does not occur for our cost model, we separately analyze the predicted speedups of schedules belonging to the same program. Figure 4.2 gives an overview of the correlation between the predicted and measured speedups over 16 programs randomly selected from the test set. Each chart represents the 32 schedules applied on each program with blue dots, the closer a blue dot is to the red line the lower the prediction error is. This figure shows that the cost model's predictions fit well the distribution and the range of the speedups and does not just predict an average value for each program.

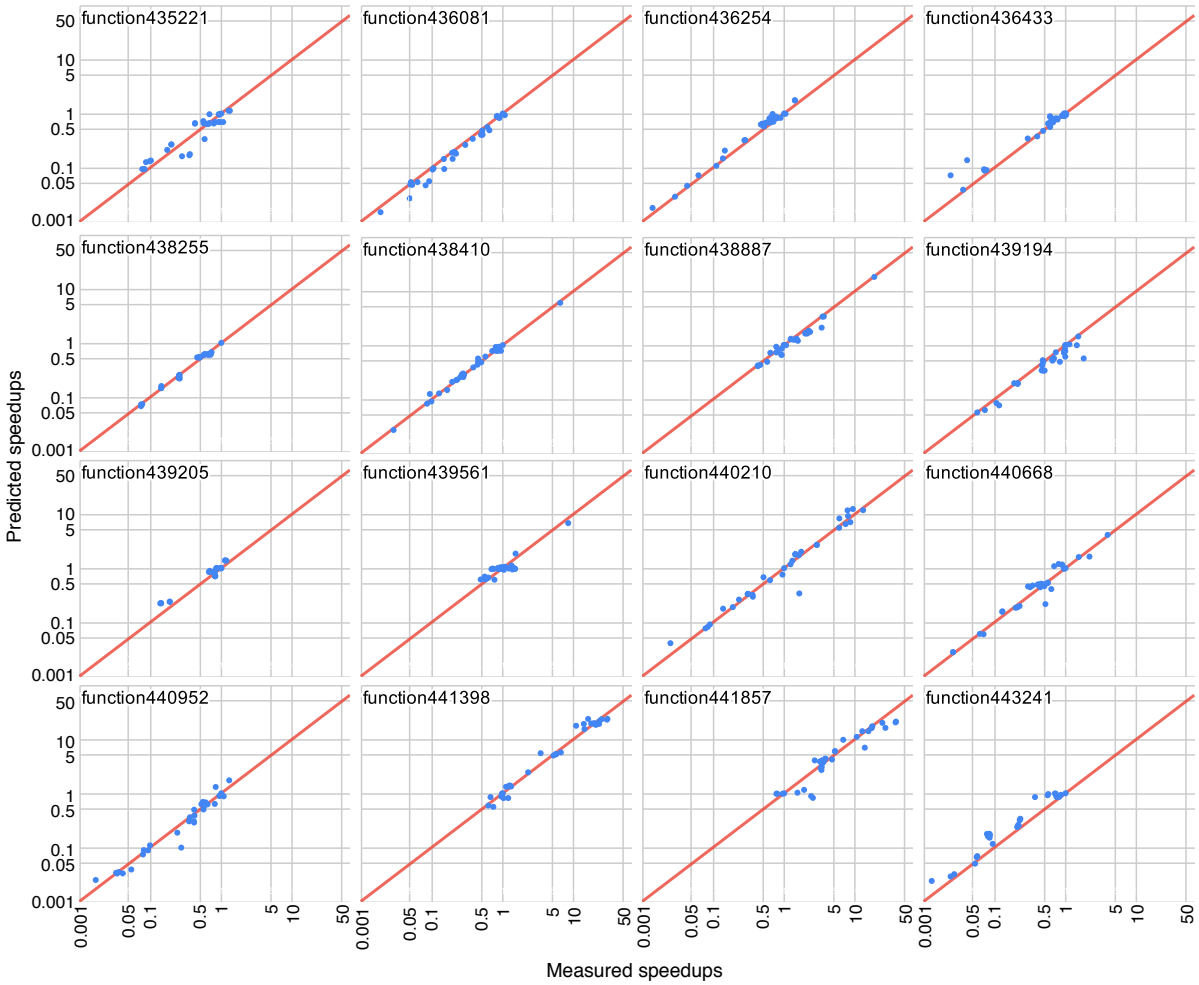


Figure 4.2: Measured vs Predicted speedup on 16 random programs from the test set, each blue dot represents a schedule with respect to its measured speedup and its predicted speedup.

4.1.5 Visualizations of the Performances of the Cost Model

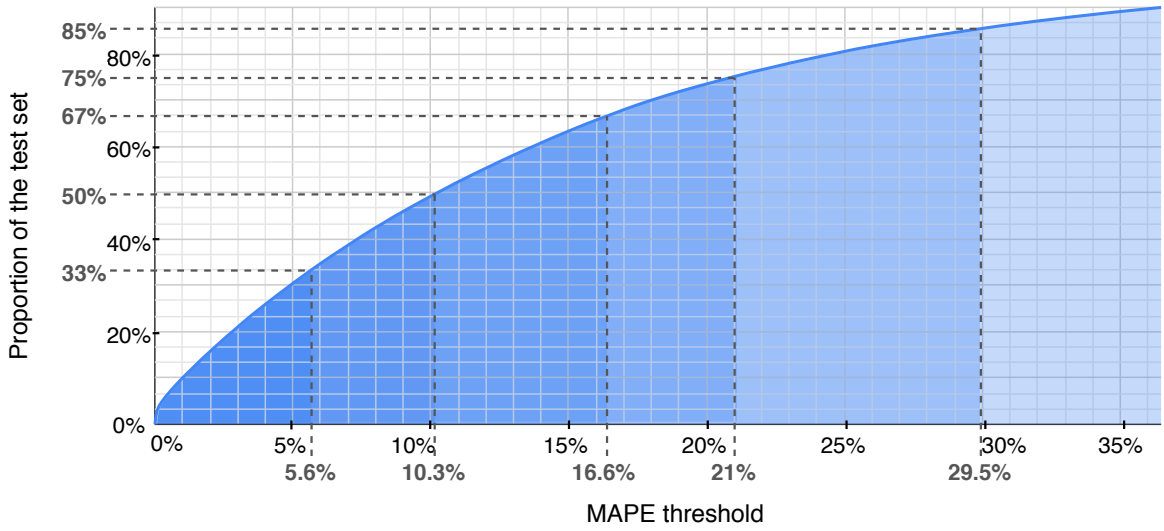
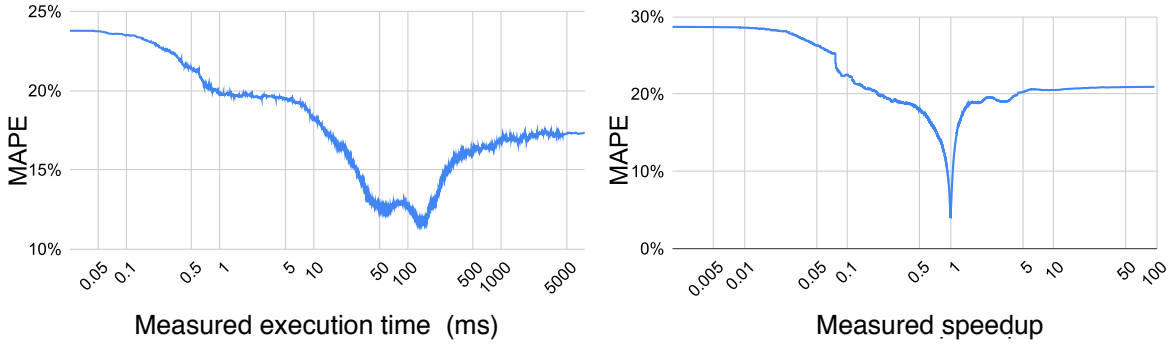


Figure 4.3: Cumulative distribution of the MAPE error over the test set. Reading example: 67% of the test set has an error <16.6%

After giving the average errors and scores over the test set in the previous section, we further analyze the distribution of these errors. Figure 4.3 shows the cumulative MAPE distribution. The x axis gives MAPE threshold values, and the y axis presents the percentage of the programs of which the error is below that threshold. The figure shows that 33% of the programs have MAPE lower than 5.6%. In addition, for half test programs, the speedup prediction presents an error lower than 10.3%. The same figure shows only 15% of the programs have MAPE higher than 30%.



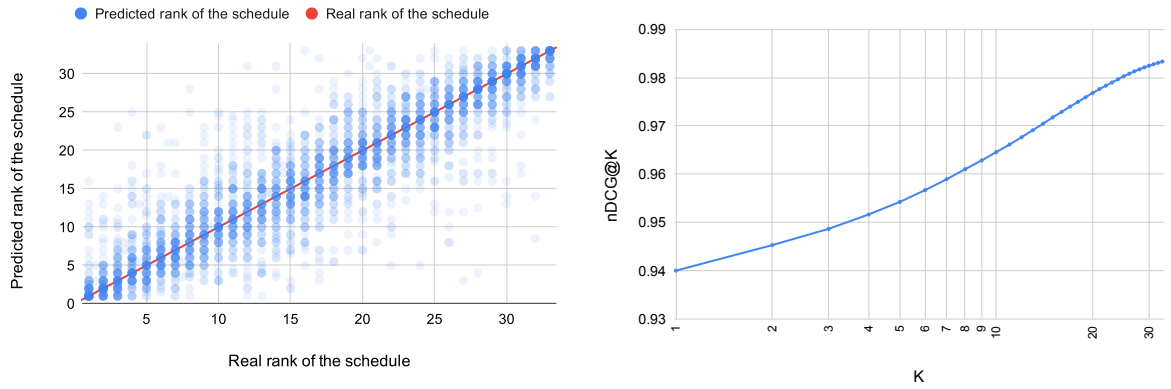
(a) MAPE over execution times.

(b) MAPE over measured speedups.

Figure 4.4: Distribution of the error rates over measured speedups and execution times.

We further study the distribution of MAPE made by the cost model in speedup predictions. The distribution is plotted over execution times (Figure 4.4a) and speedups (Figure

4.4b). Figure 4.4a shows that the model is more accurate for programs having mid-range execution time because they represent a large fraction of the training set. Since the execution time measurements are less stable for small execution time programs, the error among those programs is relatively higher. Figure 4.4b reveals the same cost model's behavior as figure 4.1. This behavior is already discussed in the previous section.



(a) Predicted versus real schedule rank of 100*32 scheduled program from the test set

(b) Average nDCG$_K$ over different values of K

Figure 4.5: Visualizing the ranking performance of the cost model

In order to prove the effectiveness of our cost model to rank schedules, we establish a ranking of schedules belonging to the same program based on their predicted speedups. Then, we plot them against the real ranks. The real ranks are obtained from the ground truth speedups. The results are shown in Figure 4.5a. The high density of dots around the identity line (the red line) depicts that the ranks yield from the predicted speedups are accurate.

Figure 4.5b provides a more detailed assessment of model capabilities for ordering schedules. For different K values, we calculate the $nDCG_K$ score among schedules that belong to the same program, and the average is reported. It is more challenging to have high $nDCG_K$ scores for small K values because the ordered list of schedules (ordered based on model speedup predictions) must contain high quality schedules in the first K elements. The model reaches a high score even for $nDCG_1$ (0.94), when the K value is greater than 22, the model's $nDCG_K$ score exceeds 0.98.

4.2 Exploration+Model Evaluation

The purpose of this second evaluation is to assess the behavior of the auto-scheduler when it uses our cost model as an objective function, we will use our cost model combined with search algorithms in order to find top performing schedules for a set of programs. We will use the two schedule search algorithms implemented by [Abdous, 2020], one based on Monte Carlos Tree Search (MCTS) and another based on Beam Search (BS), we will compare the results

of the exploration when the search algorithm relies on our cost model to evaluate candidate schedule versus when it takes ground truth measurements. This evaluation will be done on a set of real-world TIRAMISU programs (benchmark programs) and we will put our interest in the quality of the schedules found and the exploration time.

Before presenting the experiments' results, we will first describe the benchmark program on which this evaluation will be done, then we will give an overview of the exploration module as well as a description of the search algorithm that will be used.

4.2.1 Used Benchmark Programs

To guarantee that the cost model doesn't overfit to synthetic programs, we also evaluate it on a benchmark of real-world TIRAMISU programs. The benchmarks are an implementation of algorithms spanning various areas, including image processing, physics simulation, linear algebra, deep learning operations, etc. These programs are mainly implemented based on their C implementation offered by Polybench [1] [Pouchet et al., 2012] or adapted from the TIRAMISU repository [2]. The following paragraphs provide a description of each benchmark as well as its domain and utility.

- **convolution.** An implementation of the widely used neural network layer. The benchmark operates over a 4-D input tensor and a variable number of 2-D kernels. The input tensor shapes correspond to: number of images in the batch, number of color channels in each image, image height, and image width. This operation abstracts the images to a feature map with shape (number of images) × (feature map height) × (feature map width) × (feature map channels).

- **conv_relu.** In deep learning networks, the convolution layers are commonly followed by a rectified linear unit, commonly referred to as *relu*. The function is defined by $relu(x) = max(0, x)$. This benchmark computes a feature map from input images and kernels the same as above, and forwards the result to the *relu*activation function.

- **gemm.** Generalized Matrix Multiply is a Level 3 BLAS routine (Basic Linear Algebra Subprograms) [Dongarra et al., 1990]. Level 3 BLAS routines can be used to implement algorithms of numerical linear algebra, for instance, Cholesky factorization. Such implementations are portable and efficient, once the efficiency of BLAS routines is guaranteed [Dongarra et al., 1990]. *gemm* takes as inputs two scalars and three matrices. The output array C is updated following the formula : $C \leftarrow \alpha AB + \beta C$.

- **doitgen.** The *doitgen* kernel is part of a high-level software environment : Multiresolution ADaptive NumErical Scientific Simulation (MADNESS) [Harrison et al., 2016]. The software is used to solve integral and differential equations in chemistry and in several

[1] http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/
[2] https://github.com/Tiramisu-Compiler/tiramisu/

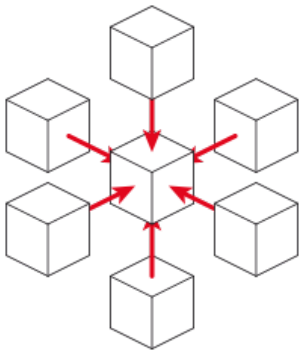
areas of physics. The kernel is taken from [Pouchet et al., 2012]. It updates each element of A (which is an array of dimension $R \times Q \times S$) according to Equation 4.5.

$$A(r,q,p) \leftarrow \sum_{s=0}^{S-1} A(r,q,s)x(p,s) \tag{4.5}$$

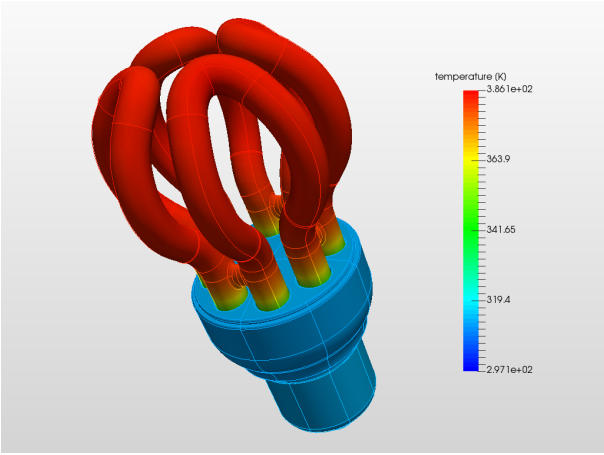
- **mvt.** A linear algebra kernel that represents a matrix vector multiplication composed with another matrix vector multiplication but with transposed matrix [Pouchet et al., 2012]. The two output vectors x_1 and x_2 are updated according to Equation 4.6

$$\begin{cases} x_1 \leftarrow x_1 + Ay_1 \\ x_2 \leftarrow x_2 + A^T y_2 \end{cases} \tag{4.6}$$

- **heat2d** and **heat3d**. Two stencils for heat propagation simulations in 2D and 3D spaces, respectively (see definition in Section 3.1.2). Figure 4.6a illustrates the neighborhood pattern of the *heat3d* stencil. The impact of such stencils is shown in Figure 4.6b.



(a) Neighborhood pattern of *heat3d* stencil



(b) image generated by a heat simulator

Figure 4.6: Heat simulation stencils

- **jacobi2d.** Jacobi-style stencil is part of Gauss-Seidel method for solving systems of linear equations [Pouchet et al., 2012]. The benchmark is composed of one computation taking the average of five points neighborhood pattern (see Figure 3.4a).

- **seidel2d.** Just like *jacobi2d*, this stencil comes from Gauss-Seidel method [Pouchet et al., 2012] but it takes the average of 9-points neighborhood pattern instead.

- **cvtcolor.** An image processing filter for converting the colors from one space to another. For instance, convert an input image from RGB (colored image) to grayscale (Figure 4.7).

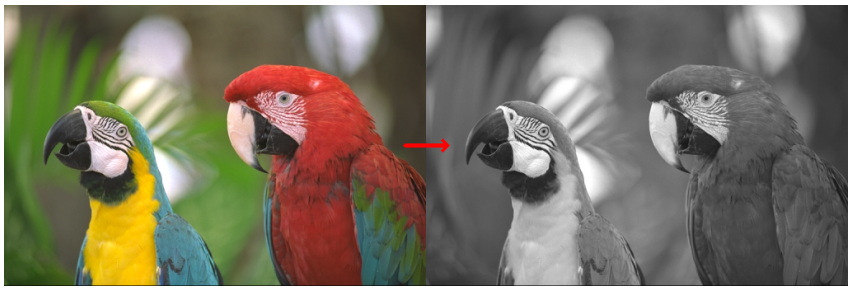


Figure 4.7: Example of converting an RGB image to grayscale

- **blur.** *box blur* or just *blur* for short, is a succession of two stencil computations that calculate horizontal and vertical blur by taking the average of 3-points neighborhood pattern (see Figure 4.8).

Figure 4.8: Example of applying *blur* to an image

4.2.2 Schedule Exploration Module

The space exploration module, developed by [Abdous, 2020], takes as an input an unscheduled TIRAMISU program and outputs an optimized scheduled version. The procedure to find the best schedule can be viewed as an iteration of three steps :

1. Generate candidate schedules following a policy.
2. Evaluate each generated schedule.
3. Keep a subset of the generated schedules to start within the next iteration.

Steps 1 and 2 are defined by the search algorithm, namely, Beam Search and Monte Carlo Tree Search. A brief description of each algorithm will be provided later in this section. The evaluation (step 2) can be done by compiling the base program with each schedule and measuring the speedups after execution on the target machine. This evaluation method is not applicable when the size of the search space increases considerably or when the target machine cannot be used during the search time. Our cost model offers a fast and accurate alternative for the evaluation method.

Beam Search. The search space is modeled as a tree. Beginning from the root, the node that contains no code transformation, the algorithm generates multiple child schedules based on one code transformation. For example, considering tiling, the algorithm can form schedules by applying 2D and 3D tiling on different loop iterators with different parameters (tiles). After evaluating each schedule, the algorithm selects the k best performing candidates and recursively proceeds them just like the root. Once the stopping criteria are reached, for instance, maximum depth, the algorithm returns the best-found schedule.

Monte Carlo Tree Search. Like Beam Search, the search space is modeled by a tree and even the generation policy of child schedules remains the same. Although, randomization is introduced to select the child schedules. Given a set of schedules $\{s_1, s_1, ..., s_n\}$, the probability of choosing a schedule s_i is q_i given by the *softmax* Equation 4.7, where $f(p, s_i)$ is the speedup of the schedule s_i applied to a program p (f is defined by Equation 1.1).

$$q_i = \frac{e^{f(p,s_i)}}{\sum_{j=1}^{n} e^{f(p,s_j)}} \tag{4.7}$$

Best performing schedules have a high probability to be selected, but in contrast to Beam search, even low speedup schedules can also be chosen to start with in the next iteration.

4.2.3 Results of the Model-Guided Exploration

Since both Beam Search and Monte Carlo Tree Search cares only about the relative order of different candidate schedules, the ranking abilities of the cost model happens to be more important than the predictions' precision. Therefore, in order to achieve better auto-scheduling results, we fine-tuned the proposed cost model to perform more accurate ranking by training it with the *NDCG-Loss2++*, a rank-based loss function from the *LambdaLoss* framework [Wang et al., 2018] that encourages the cost model to focus more on minimizing the ranking errors than on maximizing the precision of the predicted speedup.

In this section, we will compare the results of the auto-scheduler when the exploration module relies on the cost model for evaluating candidate schedules and when it relies on ground truth measurements (by executing each candidate schedule), for this purpose we perform an experiment where we gather the effects of each evaluation scheme on both the search performance (the speedup of the schedule found by the auto-scheduler) and efficiency (the time consumed by the auto-scheduling process).

Figures 4.9 and 4.10 represent the speedup of the best schedule found by the search techniques for each benchmark, this experiment highlights the influence of the cost model on both Beam Search (Figure 4.9) and Monte Carlo Tree Search (Figure 4.10). The baselines (the blue columns of each figure) are obtained by performing the auto-scheduling procedure using ground truth evaluation (i.e, each candidate schedule is evaluated by running it and measuring it's execution time), the other columns are the speedups of the schedules found by

the same search technique but with relying on the predictions of the cost model instead of ground truth measurements.

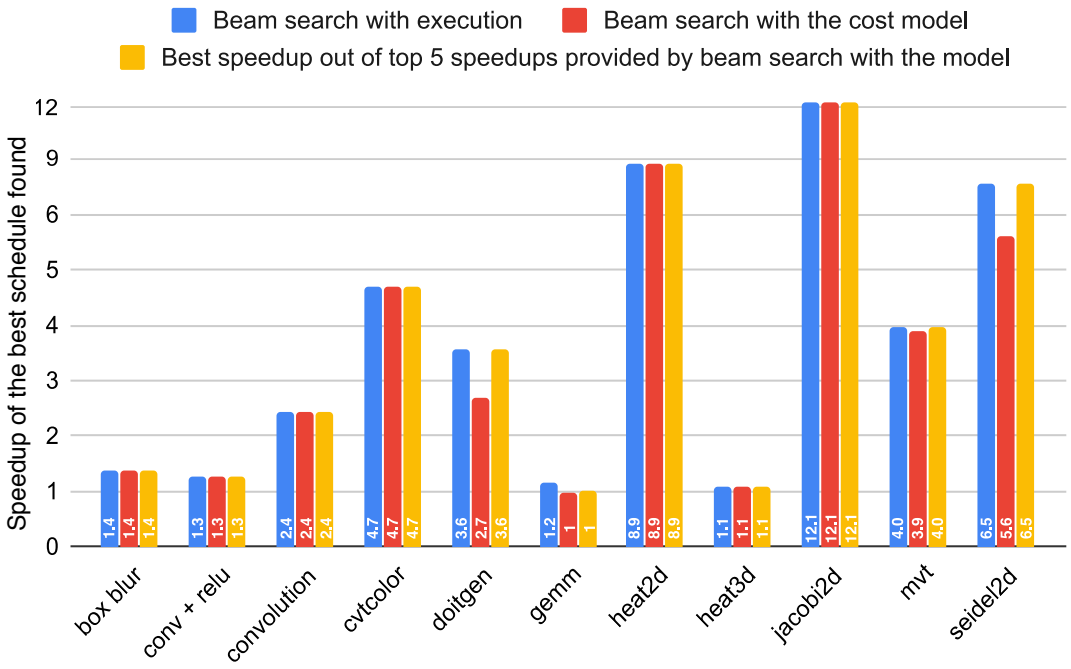


Figure 4.9: Comparison of the auto-scheduling results using Beam Search method with both the cost model and ground truth measurements.

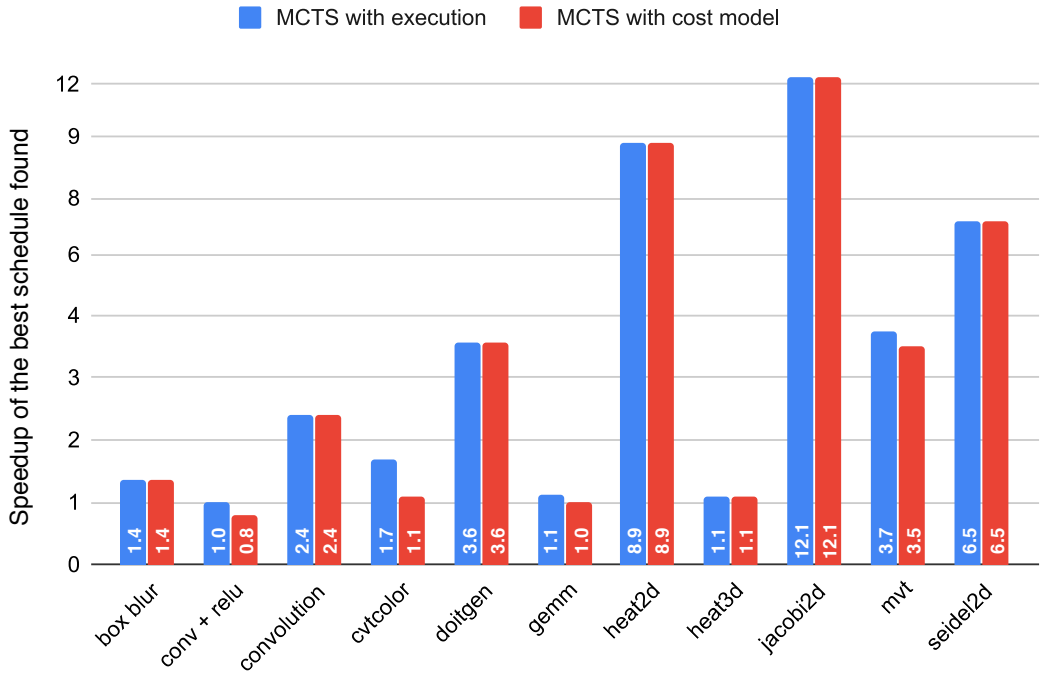


Figure 4.10: Comparison of the auto-scheduling results using MCTS method with both the cost model and ground truth measurements.

The experiment shows that for most benchmarks, the results of relying on our cost model match the baseline results, and although for some benchmarks the baselines are not matched, the results are still very competitive.

Other than the quality of the found schedules, the search time is an important aspect to emphasize on. Table 4.3 details the trade-off between search time and the quality of schedules found for each benchmark.

	Beam Search		MCTS	
Benchmarks	Search time improvement	Performances degradation	Search time improvement	Performances degradation
box blur	80.79x	0%	46.38x	0%
conv + relu	34.60x	0%	37.63x	20.0%
convolution	5.05x	0%	19.38x	0%
cvtcolor	50.39x	0%	38.86x	35.29%
doitgen	73.52x	24.44%	158.60x	0%
gemm	94.30x	14.78%	18.18x	11.50%
heat2d	94.73x	0%	32.77x	0%
heat3d	220.96x	0%	587.48x	1.82%
jacobi2d	136.82x	0%	7.37x	0%
mvt	115.62x	1.27%	39.17x	6.42%
seidel2d	138.89x	14.37%	22.77x	0%
Average	**95.15x**	**4.98%**	**90.87x**	**6.82%**

Table 4.3: Search time improvement compared to performance degradation when using the cost model for auto-scheduling

Search time improvement is the ratio of the time consumed by the auto-scheduler when it uses the cost model as evaluation module versus when it relies on ground truth measurements

$$\textit{Search time improvement} = \frac{\textit{Search time with execution}}{\textit{Search time with cost model}} \tag{4.8}$$

Performances degradation is the relative degradation of the best speedup found by the auto-scheduler when it uses ground truth measurements for evaluation versus when it relies on the cost model.

$$\textit{Performances degradation} = \frac{\textit{Best speedup found by } AS_e - \textit{Best speedup found by } AS_m}{\textit{Best speedup found by } AS_e} \times 100 \tag{4.9}$$

where AS_e stands for auto-scheduler with execution (executes schedules to evaluate them) and AS_m stands for auto-scheduler with cost model (uses the cost model to evaluate schedules).

For Beam Search, the exploration with the cost model is, on average, 95 times faster than the exploration with execution, while incurring 5% of loss in schedule quality. Similarly, the quality degradation of schedules is 7% when using MCTS with the cost model, but the search is 91 times faster. The quality degradation is due to some imperfection of the model's prediction. This degradation is tolerable in favor of the speed that the cost model offers during space exploration.

More details about the parameters used for the search techniques and the best schedules found by this experiment can be found in Appendix C.

4.3 Design Alternatives Comparison

We have designed multiple architectures for the cost models, under the hypothesis that they support different algorithm structures, computation types, code transformations, etc. At that stage, we mainly relied upon the recursive nature of the architectures and their ability to incorporate information from the tree-structured program representation. Validation and test procedures provide more color to our hypothesis. In Table 4.4, we provide a detailed comparison between the explored cost model architectures introduced in Section 3.3.4. The Recursive LSTM is proven superior compared to all other architectures on both metrics. The Recursive LSTM has less parameters and it is twice faster than Tree-LSTM that have the same degree of generality (supports the general form of TIRAMISU programs).

Architectures		FeedForward NN	Simple RNN	Tree-LSTM	Recursive LSTM
Mean Absolote percentage Error		26.89%	21.74%	18.10%	16.32%
MAPE above measurements-induced error		16.72%	11.57%	7.93%	6.15%
Program-wise Sprearman's correlation		0.8587	0.8759	0.9094	0.9558
Average Inference time for a single program on CPU[3]		0.5269 ms	1.2270 ms	7.1009 ms	3.3846 ms
Number of learnable parameters		6397781	2563701	3371101	2538221
Program representation size [4]		10136	$2534 \times N_{comp}$	$1235 \times (N_{comp} + N_{loop})$	$2534 \times N_{comp}$
Design limitations	Supports unbounded number of computations	No	Yes	Yes	Yes
	Supports general program's structures	No	No	Yes	Yes

Table 4.4: Comparison between the explored model architectures.

[3] On an Intel Xeon CPU E5-2680 v3 @ 2.50GHz

[4] N_{comp} : the number of computations in the program, N_{loop} : the number of loops in the program

4.4 Conclusion

In this chapter, we demonstrated the quality of our model's predictions and its influence on the performance of the overall auto-scheduler. We evaluated our cost model through various experiments against various settings on both synthetic programs and real-world benchmarks.

In the first section of this chapter, we evaluated our cost model independently from the auto-scheduler. This first evaluation was performed on a 276 thousand point test set. at predicting speedups, our cost model achieved a low error rate of 16% $MAPE$. At ranking different schedules of a program, our model archives a high $nDCG$ score of 98% and an $nDCG_1$ of 0.94. The second section consists in a real-world use case evaluation. After setting the Tiramisu auto-scheduler to use our cost model as an objective function, we compared the results of the exploration against those where the auto-scheduler relies on ground truth measurements. This evaluation showed that using the cost model makes the auto-scheduler run, on average, 95× faster on the benchmark set in exchange for 5% performance degradation. In the third section of this chapter, we compared some major conceptual alternatives that we explored when designing the cost-models architecture. This evaluation showed that the proposed architecture (the Recursive LSTM architecture) achieves lower error rates compared to the other explored architectures.

Conclusion

The rising need for data processing in modern applications such as deep learning requires building efficient software. Writing highly optimized codes implies applying many code transformations that improve parallelism, temporal locality, spatial locality, etc. Alongside with code optimization expertise, the programmers must be mindful of the target machine architecture. This is due to the fact that the same combination of code transformations can have discrepant effects from one architecture to another.

Predicting the effect of a sequence of code transformations is a challenging task. Consequently, experts go through an experimental approach to evaluate the suggested solutions during exploring the possible code transformations. Auto-schedulers aim to automate this time-consuming and error-prone process in order to explore more combinations with minimal user intervention. However, evaluating each sequence of code transformations by compilation and execution remains a serious bottleneck to cover substantial portions of such a combinatorial search space. In order to solve this difficulty, statistical models can be involved to approximate the cost function (speedup or execution time). Hand-designed cost models have a limited accuracy and they encounter difficulty to generalize for more architectures.

In this project, we introduce a data-driven cost model for TIRAMISU. This novel regression model accurately predicts the speedups of full programs. It does not rely on complex manually-driven features, but it leverages a deep neural network designed to capture the effect of code transformations on the performance. The model operates over AST-based program characterization that includes simple source code features mirroring algorithms and code transformations. Both the model and the characterization are general enough to handle any TIRAMISU program. The training data set is constructed by our synthetic program generator. In addition to evaluating the model on synthetic programs where it has a low error rate of 16%, we show that the model enables TIRAMISU to automatically optimize real-work benchmarks. In most instances, the found sequences of code transformations are competitive to the optimal solutions.

Our cost model powers TIRAMISU's auto-scheduler, a system that automatically optimizes data-parallel algorithms in reasonable search time. Image processing and deep learning programmers can now spend less time optimizing code and more time expressing novel algorithms. Aside from increasing the productivity of experienced users, non-computer scientists can also achieve efficient code only by specifying abstract algorithms and letting the compiler take over the burden of finding a suitable sequence of code transformations and inserting the corresponding scheduling commands on their behalf.

The evolutionary perspectives that can further improve this work include:

Improving data quality. Generating synthetic programs has been proven efficient for training. The trained cost models are able to generalize to unseen real-world benchmarks. The next step is to endow the training dataset with real-world programs. This can be achieved by simply collecting programs from TIRAMISU free repositories. Apart from that, we can ask the auto-scheduler users to donate their algorithms during production. The explored code transformations by the search method can be included to form new batches to apply transfer learning, or even to re-train the model, in case the new dataset size allows it.

Expending the supported optimizations. We considered only code transformations targeting multicore CPUs. Since the model relies on high-level features, it is easily extensible for other hardware having different optimization types. For example, to support the command that maps loop levels to GPU, we need just to add the block and thread IDs in the feature vector of the concerned computation. Another possible improvement is to include code transformations operating on non-original loop levels. In particular, support interchanging two loop levels yield from a previous tiling.

Integrating the model in Q-learning algorithm. Currently, our model is pre-trained, then it is used by a search algorithm to evaluate the explored code transformations. The weights of the model are not updated during space exploration. The model can be integrated into a Q-learning method to store Q-factors of state-action pairs. In this case, the state is the algorithm and the previously applied code transformations, while the action is the next code transformation to apply. When a Q-factor is needed, it is fetched from the model by a forward pass. When a Q-factor is to be updated, the new Q-factor is used to update the model weights by backward pass. So the model is trained during space exploration.

Appendices

A

Assessing the Execution Time's Imprecision: Experiment Details

This experiment aims to empirically estimate the variability in the measured speedups and hence deduce what is the minimum prediction error a machine learning model can achieve when trained on our dataset.

To estimate the speedup's variability, we need to compare a sample of measurements from our dataset to a reference that we can assume to be stable. To get such a reference we randomly selected a sample of 70000 scheduled programs and run each program 120 times, then we calculate the relative change of the measured speedup after each run and plotted the result in Figure A.1

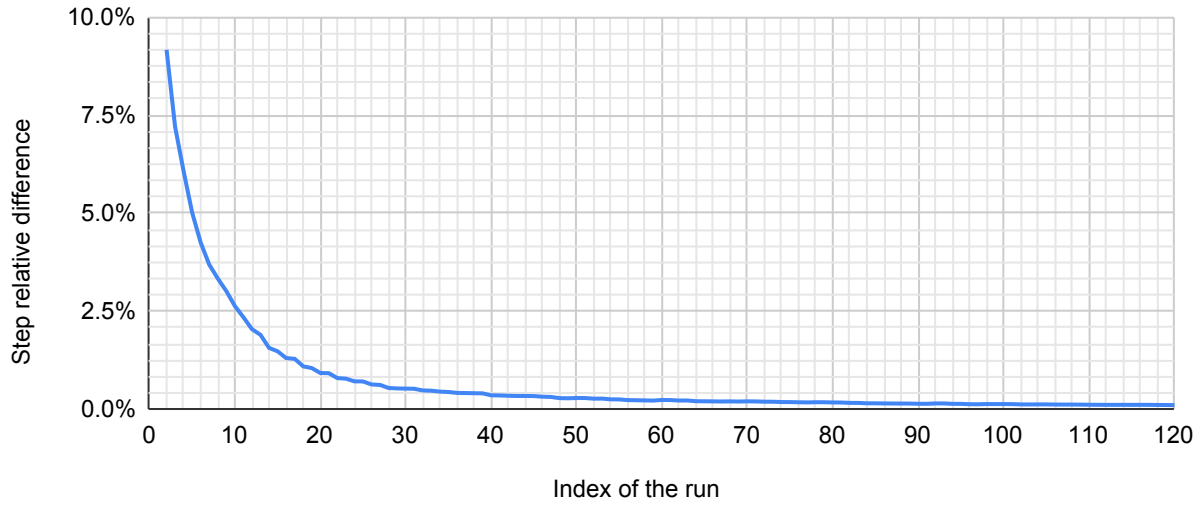


Figure A.1: Relative change of the measured speedups after each run.

$$StepRelativeChange_i = \frac{|Speedup_i - Speedup_{i-1}|}{Speedup_{i-1}} \times 100$$

where $Speedup_i$ is the measured speedup after the i^{th} run. As the chart shows that curve significantly flat after the 100^{th} run, meaning that the speedup won't change considerably

after 100^{th} run, we can safely assume that the measurements are stable at the 120^{th} run and we can take the speedups measured at the 120^{th} run as references.

Then, to effectively estimate the variability, we started a new set of measurements on the same sample, and we compared the speedups measured at the n^{th} run (varying the n from 1 to 120) against the reference speedups in terms of relative change,

$$RelativeChangeToRef_n = \frac{|Speedup_n - Speedup_{ref}|}{Speedup_{ref}} \times 100$$

where $Speedup_n$ is the measured speedup after the n^{th} run and $Speedup_{ref}$ is the reference speedup.

As shown in figure A.2 the measured speedups after running each program 30 times varies 10.52% on average from the reference one, furthermore the speedup acquired by our policy (30 runs per program plus an additional 15 runs for the base programs, not visible in the chart) varies 10.17% from the references one. As depicted in the chart, this variability can't be reduced much further even by considerably increasing the number of runs, in fact even after running each program 120 times the speedups varies by about 9.14% from the reference ones.

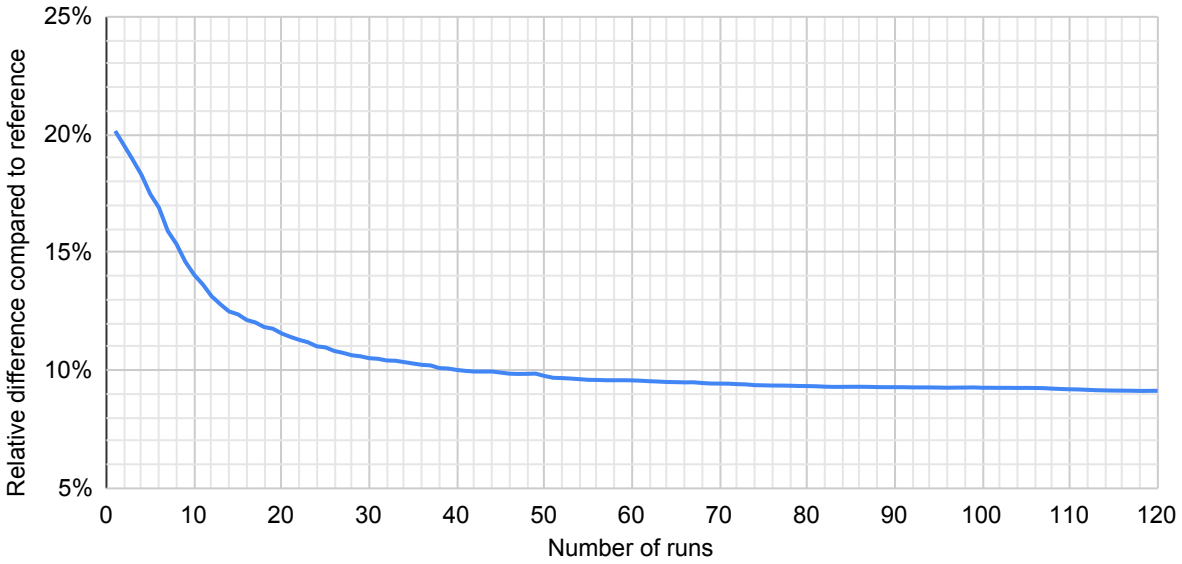


Figure A.2: Relative change of the measured speedups after each run in comparison to the reference speedups.

B

Normalized Discounted Cumulative Gain: Further Explanation and Examples

To prove the relevance of the $nDGC$ metric over the other regular regression metrics in our case, let's take an example of four schedules for an arbitrary algorithm and two cost models m_1 and m_2 (Table B.1). At a first glance, the model m_1 appears to be better than m_2. Nevertheless, when the two models are required to select the best schedule based on predicted values, the model m_1 fails lamentably against m_2. In view of the fact that, according to m_1, the schedule s_b has the best speedup ($y_b^1 = 4$) while its actual value is ($y_b = 2.5$). It is true that the speedups predicted by m_2 are far from being perfectly accurate and they have bigger MAPE, but they result in a perfect schedule ranking

Schedules	s_a	s_b	s_c	s_d
Measured speedups (y_i)	5.0	2.5	1.5	0.8
Speedups predicted by m_1 (y_i^1)	3.5	4.0	1.5	0.9
Speedups predicted by m_2 (y_i^2)	3.1	1.9	1.5	1.2

Metrics	MAPE	$nDCG_1$
model m_1	22.86%	0.5
model m_2	31.55%	1.0

Table B.1: Example of measured and predicted speedups by two cost models and their evaluation

Schedules	s_a	s_b	s_c	s_d
Measured speedups	3.5	2.1	1.4	0.5
Predicted speedups	2.0	2.4	0.8	1.2

Table B.2: Example of measured and predicted speedups

For the sake of calculation example, we consider the measured and the predicted speed up in Table B.2. According to the predicted speedups, the schedules are ranked as follows : (s_b, s_a, s_d, s_c). In this order, we partially calculate $nDGC_4$ (Equation B.1). $iDCG_4$ is calculated by Equation B.1 according to the ideal order : (s_a, s_b, s_c, s_d). We finally obtain a score of 0.8973.

$$nDCG_4 = \frac{1}{iDCG_4}\left[\frac{2.1}{log_2(1+1)} + \frac{3.5}{log_2(1+2)} + \frac{0.5}{log_2(1+3)} + \frac{1.4}{log_2(1+4)}\right] \approx \frac{5.16}{iDCG_4} \tag{B.1}$$

$$iDCG_4 = \frac{3.5}{log_2(1+1)} + \frac{2.1}{log_2(1+2)} + \frac{1.4}{log_2(1+3)} + \frac{0.5}{log_2(1+4)} \approx 5.75 \tag{B.2}$$

C

Results Details of the Model-Guided Exploration

Table C.1 show the schedules found by the auto-scheduler by the different search techniques for each benchmark, the notation used to describe schedules is the following:

- $F(i)$: indicates that the computations are not fused at the same loop nest and the innermost fused loop is level i (fusion transformation).
- $I(i_1, i_2)$: indicates that the loop levels i_1 and i_2 are interchanged (loop interchange transformation).
- $T_2(i_1, i_2, f_1, f_2)$: indicates that the loop iterators at levels i_1 and i_2 are tiled with tile factors f_1 and f_2 respectively (2D loop tiling transformation).
- $T_3(i_1, i_2, i_3, f_1, f_2, f_3)$: indicates that the loop iterators at levels i_1, i_2, and i_3 are tiled with tile factors f_1, f_2 and f_3 respectively (3D loop tiling transformation).
- $U(f)$: indicates that the innermost loop levels are unrolled with the factor f.
- / : indicates that the schedule consists only in parallelizing the outermost loops.

Table C.2 shows the parameters used for each benchmark by the search methods during the experiments.

	Beam Search			MCTS	
	Execution	Model	Model + top 5	Execution	Model
box blur	$I(0,1)$				
conv + relu	$F(3) + I(0,1)$			/	$F(3)$
convolution	$I(0,2)$				
cvtcolor	$T(0,1,32,128)$			$I(0,1)+$ $T(1,0,32,64)$	$I(0,1)$
doitgen	$I(0,1)$	$I(0,1) + U(16)$	$I(0,1)$	$I(0,1)$	$I(0,1)$
gemm	$I(0,1) + T(1,0,32,32)$ $+U(8)$	$U(4)$	/	$F(1)$	/
heat2d	$T(0,1,64,128) + U(16)$				
heat3d	$I(0,1)$				
jacobi2d	$T(0,1,64,128) + U(16)$				
mvt	$I(0,1) + T(1,0,64,32)$ $+U(16)$	$T(1,0,32,32)$ $+U(16)$	$I(0,1)+$ $T(1,0,64,32)$ $+U(16)$	$T(1,0,64,64)$	$T(1,0,32,32)$
seidel2d	$T(0,1,32,32)$	$T(0,1,32,32)$ $+U(16)$	$T(0,1,32,32)$		

Table C.1: Schedules found by the different search methods.

	Beam Search	MCTS	
	Beam size	Number of samples	Number of executions
box blur	2	1	1
conv + relu	2	1	1
convolution	2	1	1
cvtcolor	2	10	5
doitgen	2	20	5
gemm	2	10	5
heat2d	2	5	1
heat3d	2	5	5
jacobi2d	2	10	5
mvt	2	5	5
seidel2d	2	10	5

Table C.2: The parameters for the search methods for each benchmark

Bibliography

[Abdous, 2020] Abdous, K. (2020). Automatic code optimization with machine learning and combinatorial optimization. Master's thesis, École nationale supérieure d'informatique, Algiers, Algeria. Promotion : 2019 / 2020.

[Adams et al., 2019] Adams, A., Ma, K., Anderson, L., Baghdadi, R., Li, T.-M., Gharbi, M., Steiner, B., Johnson, S., Fatahalian, K., Durand, F., and Ragan-Kelley, J. (2019). Learning to optimize halide with tree search and random programs. *ACM Trans. Graph.*, 38(4).

[Aho, 2006] Aho, A. V. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley.

[Altman, 1992] Altman, N. S. (1992). An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 46(3):175–185.

[Ashouri et al., 2016] Ashouri, A. H., Bignoli, A., Palermo, G., and Silvano, C. (2016). Predictive modeling methodology for compiler phase-ordering. In *Proceedings of the 7th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and the 5th Workshop on Design Tools and Architectures For Multicore Embedded Computing Platforms*, PARMA-DITAM '16, page 7–12, New York, NY, USA. Association for Computing Machinery.

[Ashouri et al., 2018] Ashouri, A. H., Killian, W., Cavazos, J., Palermo, G., and Silvano, C. (2018). A survey on compiler autotuning using machine learning. *ACM Comput. Surv.*, 51(5).

[Bachir et al., 2013] Bachir, M., Brault, F., Gregg, D., Cohen, A., et al. (2013). Minimal unroll factor for code generation of software pipelining. *International Journal of Parallel Programming*, 41(1):1–58.

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

[Baldominos et al., 2018] Baldominos, A., Saez, Y., and Isasi, P. (2018). Evolutionary convolutional neural networks: An application to handwriting recognition. *Neurocomputing*, 283:38 – 52.

[Banerjee, 2011] Banerjee, U. (2011). Loop nest parallelization. In Padua, D., editor, *Encyclopedia of Parallel Computing*, pages 1068–1079. Springer US, Boston, MA.

[Belson, 1959] Belson, W. A. (1959). Matching and prediction on the principle of biological classification. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 8(2):65–75.

[Benabderrahmane et al., 2010] Benabderrahmane, M.-W., Pouchet, L.-N., Cohen, A., and Bastoul, C. (2010). The polyhedral model is more widely applicable than you think. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, page 283–303, Berlin, Heidelberg. Springer-Verlag.

[Bengio et al., 2020] Bengio, Y., Lodi, A., and Prouvost, A. (2020). Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*.

[Botchkarev, 2019] Botchkarev, A. (2019). A new typology design of performance metrics to measure errors in machine learning regression algorithms. *Interdisciplinary Journal of Information, Knowledge, and Management*, 14:045–076.

[Breiman, 2001] Breiman, L. (2001). Random forests. *Machine learning*, 45(1):5–32.

[Brown, 1998] Brown, C. E. (1998). Coefficient of variation. In *Applied Multivariate Statistics in Geohydrology and Related Sciences*, pages 155–157. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Burges et al., 2005] Burges, C., Shaked, T., Renshaw, E., Lazier, A., Deeds, M., Hamilton, N., and Hullender, G. (2005). Learning to rank using gradient descent. In *Proceedings of the 22nd international conference on Machine learning*, pages 89–96.

[Chen et al., 2006] Chen, C., Chame, J., Hall, M., and Lerman, K. (2006). A systematic approach to model-guided empirical search for memory hierarchy optimization. In Ayguadé, E., Baumgartner, G., Ramanujam, J., and Sadayappan, P., editors, *Languages and Compilers for Parallel Computing*, pages 433–440, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Chen and Revels, 2016] Chen, J. and Revels, J. (2016). Robust benchmarking in noisy environments. *arXiv preprint arXiv:1608.04295*.

[Chen et al., 2018] Chen, T., Zheng, L., Yan, E., Jiang, Z., Moreau, T., Ceze, L., Guestrin, C., and Krishnamurthy, A. (2018). Learning to optimize tensor programs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 3393–3404, Red Hook, NY, USA. Curran Associates Inc.

[Cho et al., 2014] Cho, K., van Merrienboer, B., Gulcehre, C., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. In *Conference on Empirical Methods in Natural Language Processing (EMNLP 2014)*.

[Choi et al., 2017] Choi, M.-j., Jeong, S., Oh, H., and Choo, J. (2017). End-to-end prediction of buffer overruns from raw source code via neural memory networks. *arXiv preprint arXiv:1703.02458*.

[Chollet, 2017] Chollet, F. (2017). Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[Chung et al., 2014] Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*.

[Cortes and Vapnik, 1995] Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Machine learning*, 20(3):273–297.

[Crockford, 2006] Crockford, D. (2006). The application/json media type for javascript object notation (json). *RFC 4627*.

[Cummins et al., 2020] Cummins, C., Fisches, Z. V., Ben-Nun, T., Hoefler, T., and Leather, H. (2020). Programl: Graph-based deep learning for program optimization and analysis. *arXiv preprint arXiv:2003.10536*.

[Cybenko, 1989] Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314.

[Deng et al., 2013] Deng, L., Hinton, G., and Kingsbury, B. (2013). New types of deep neural network learning for speech recognition and related applications: an overview. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 8599–8603.

[Di Biagio and Davis, 2018] Di Biagio, A. and Davis, M. (2018). [RFC] llvm-mca: a static performance analysis tool. https://lists.llvm.org/pipermail/llvm-dev/2018-March/121490.html. Accessed: 2020-08-04.

[Dongarra et al., 1990] Dongarra, J. J., Du Croz, J., Hammarling, S., and Duff, I. S. (1990). A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17.

[Dubach et al., 2007] Dubach, C., Cavazos, J., Franke, B., Fursin, G., O'Boyle, M. F., and Temam, O. (2007). Fast compiler optimisation evaluation using code-feature based performance prediction. In *Proceedings of the 4th International Conference on Computing Frontiers*, CF '07, page 131–142, New York, NY, USA. Association for Computing Machinery.

[Epshteyn et al., 2006] Epshteyn, A., Garzaran, M. J., DeJong, G., Padua, D., Ren, G., Li, X., Yotov, K., and Pingali, K. (2006). Analytic models and empirical search: A hybrid approach to code optimization. In Ayguadé, E., Baumgartner, G., Ramanujam, J., and Sadayappan, P., editors, *Languages and Compilers for Parallel Computing*, pages 259–273, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Ester et al., 1996] Ester, M., Kriegel, H.-P., Sander, J., Xu, X., et al. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231.

[Friedman, 2001] Friedman, J. H. (2001). Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232.

[Fursin, 2010] Fursin, G. (2010). Collective benchmark (cbench), a collection of open-source programs with multiple datasets assembled by the community to enable realistic benchmarking and research on program and architecture optimization.

[Girshick, 2015] Girshick, R. (2015). Fast r-cnn. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pages 1440–1448.

[Glorot and Bengio, 2010] Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In Teh, Y. W. and Titterington, M., editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy. PMLR.

[Goldberg, 2016] Goldberg, Y. (2016). A primer on neural network models for natural language processing. *Journal of Artificial Intelligence Research*, 57:345–420.

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep learning*. MIT press.

[Goss, 2013] Goss, C. F. (2013). Machine code optimization - improving executable object code. *ArXiv*, abs/1308.4815.

[Harrison et al., 2016] Harrison, R. J., Beylkin, G., Bischoff, F. A., Calvin, J. A., Fann, G. I., Fosso-Tande, J., Galindo, D., Hammond, J. R., Hartman-Baker, R., Hill, J. C., Jia, J., Kottmann, J. S., Yvonne Ou, M.-J., Pei, J., Ratcliff, L. E., Reuter, M. G., Richie-Halford, A. C., Romero, N. A., Sekino, H., Shelton, W. A., Sundahl, B. E., Thornton, W. S., Valeev, E. F., Vázquez-Mayagoitia, A., Vence, N., Yanai, T., and Yokoi, Y. (2016). Madness: A multiresolution, adaptive numerical environment for scientific simulation. *SIAM Journal on Scientific Computing*, 38(5).

[He et al., 2017] He, K., Gkioxari, G., Dollar, P., and Girshick, R. (2017). Mask r-cnn. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*.

[He et al., 2016] He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

[Henni and Mekki, 2019] Henni, M. and Mekki, I. I. (2019). A deep learning approach for automatic code optimization in the tiramisu compiler. Master's thesis, École nationale supérieure d'informatique, Algiers, Algeria. Promotion : 2018 / 2019.

[Herruzo et al., 2004] Herruzo, E., Bandera, G., and Plata, O. (2004). Applying loop tiling and unrolling to a sparse kernel code. In Bubak, M., van Albada, G. D., Sloot, P. M. A., and Dongarra, J., editors, *Computational Science - ICCS 2004*, pages 409–412, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Hijazi et al., 2015] Hijazi, S., Kumar, R., and Rowen, C. (2015). Using convolutional neural networks for image recognition. *Cadence Design Systems Inc.: San Jose, CA, USA*, pages 1–12.

[Hines, 1996] Hines, J. W. (1996). A logarithmic neural network architecture for unbounded non-linear function approximation. In *Proceedings of International Conference on Neural Networks (ICNN'96)*, volume 2, pages 1245–1250. IEEE.

[Hochreiter et al., 2001] Hochreiter, S., Bengio, Y., Frasconi, P., Schmidhuber, J., et al. (2001). Gradient flow in recurrent nets: the difficulty of learning long-term dependencies.

[Hochreiter and Schmidhuber, 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8):1735–1780.

[Hornik et al., 1989] Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359 – 366.

[Hoste and Eeckhout, 2007] Hoste, K. and Eeckhout, L. (2007). Microarchitecture-independent workload characterization. *IEEE micro*, 27(3):63–72.

[Hubel and Wiesel, 1968] Hubel, D. H. and Wiesel, T. N. (1968). Receptive fields and functional architecture of monkey striate cortex. *The Journal of physiology*, 195(1):215–243.

[Irigoin, 2011] Irigoin, F. (2011). Tiling. In Padua, D., editor, *Encyclopedia of Parallel Computing*, pages 2040–2049. Springer US, Boston, MA.

[Kalibera et al., 2005] Kalibera, T., Bulej, L., and Tuma, P. (2005). Benchmark precision and random initial state. In *Proceedings of the 2005 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2005)*, pages 484–490.

[Kalibera and Jones, 2013] Kalibera, T. and Jones, R. (2013). Rigorous benchmarking in reasonable time. In *Proceedings of the 2013 International Symposium on Memory Management*, ISMM '13, page 63–74, New York, NY, USA. Association for Computing Machinery.

[Kawaguchi et al., 2017] Kawaguchi, K., Kaelbling, L. P., and Bengio, Y. (2017). Generalization in deep learning. *arXiv preprint arXiv:1710.05468*.

[Kennedy and McKinley, 1992] Kennedy, K. and McKinley, K. S. (1992). Optimizing for parallelism and data locality. In *Proceedings of the 6th International Conference on Supercomputing*, ICS '92, page 323–334, New York, NY, USA. Association for Computing Machinery.

[Keogh and Mueen, 2017] Keogh, E. and Mueen, A. (2017). Curse of dimensionality. In *Encyclopedia of Machine Learning and Data Mining*, pages 314–315. Springer US, Boston, MA.

[Kohonen, 1982] Kohonen, T. (1982). Self-organized formation of topologically correct feature maps. *Biological cybernetics*, 43(1):59–69.

[Lattner and Adve, 2004] Lattner, C. and Adve, V. (2004). Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE.

[Laukemann et al., 2019] Laukemann, J., Hammer, J., Hager, G., and Wellein, G. (2019). Automatic throughput and critical path analysis of x86 and arm assembly kernels. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 1–6.

[LeCun et al., 1999] LeCun, Y., Haffner, P., Bottou, L., and Bengio, Y. (1999). Object recognition with gradient-based learning. In *Shape, contour and grouping in computer vision*, pages 319–345. Springer.

[Lee and Stoodley, 1998] Lee, C. and Stoodley, M. (1998). UTDSP benchmark suite, 1998. http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html.

[Liu et al., 2019] Liu, S., Cui, Y.-Z., Zou, N.-J., Zhu, W.-H., Zhang, D., and Wu, W.-G. (2019). Revisiting the parallel strategy for DOACROSS loops. *Journal of Computer Science and Technology*, 34(2):456–475.

[Loshchilov and Hutter, 2017] Loshchilov, I. and Hutter, F. (2017). Fixing weight decay regularization in adam. *CoRR*, abs/1711.05101.

[MacQueen et al., 1967] MacQueen, J. et al. (1967). Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA.

[Manseri, 2018] Manseri, I. (2018). Méthodes d'optimisation automatique dans le compilateur halide. Master's thesis, École nationale supérieure d'informatique, Algiers, Algeria. Promotion : 2017 / 2018.

[Memeti et al., 2018] Memeti, S., Pllana, S., Binotto, A., Kołodziej, J., and Brandic, I. (2018). A review of machine learning and meta-heuristic methods for scheduling parallel computing systems. In *Proceedings of the International Conference on Learning and Optimization Algorithms: Theory and Applications*, LOPAL '18, New York, NY, USA. Association for Computing Machinery.

[Mendis et al., 2019] Mendis, C., Renda, A., Amarasinghe, S., and Carbin, M. (2019). Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In *International Conference on Machine Learning*, pages 4505–4515.

[Mitchell et al., 1997] Mitchell, T. M. et al. (1997). Machine learning. *Burr Ridge, IL: McGraw Hill*, 45(37):870–877.

[Mnih et al., 2013] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

[Mullapudi et al., 2016] Mullapudi, R. T., Adams, A., Sharlet, D., Ragan-Kelley, J., and Fatahalian, K. (2016). Automatically scheduling halide image processing pipelines. *ACM Trans. Graph.*, 35(4).

[Mytkowicz et al., 2009] Mytkowicz, T., Diwan, A., Hauswirth, M., and Sweeney, P. F. (2009). Producing wrong data without doing anything obviously wrong! *SIGPLAN Not.*, 44(3):265–276.

[Paszke et al., 2019] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R., editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.

[Pouchet et al., 2012] Pouchet, L.-N. et al. (2012). Polybench: The polyhedral benchmark suite. http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/.

[Ragan-Kelley et al., 2013] Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. (2013). Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 519–530, New York, NY, USA. Association for Computing Machinery.

[Rahman et al., 2010] Rahman, M., Pouchet, L.-N., and Sadayappan, P. (2010). Neural network assisted tile size selection. In *International Workshop on Automatic Performance Tuning (IWAPT'2010). Berkeley, CA: Springer Verlag*.

[Reed et al., 2002] Reed, G. F., Lynn, F., and Meade, B. D. (2002). Use of coefficient of variation in assessing variability of quantitative assays. *Clinical and Vaccine Immunology*, 9(6):1235–1239.

[Renganarayana et al., 2006] Renganarayana, L., Ramakrishna, U., and Rajopadhye, S. (2006). Combined ilp and register tiling: Analytical model and optimization framework. In Ayguadé, E., Baumgartner, G., Ramanujam, J., and Sadayappan, P., editors, *Languages and Compilers for Parallel Computing*, pages 244–258, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Rumelhart et al., 1985] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1985). Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science.

[Rummery and Niranjan, 1994] Rummery, G. A. and Niranjan, M. (1994). *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering Cambridge, UK.

[Shuai et al., 2015] Shuai, B., Zuo, Z., Wang, G., and Wang, B. (2015). Dag-recurrent neural networks for scene labeling. *CoRR*, abs/1509.00552.

[Smith and Topin, 2017] Smith, L. N. and Topin, N. (2017). Super-convergence: Very fast training of residual networks using large learning rates. *CoRR*, abs/1708.07120.

[SPEC, 2017] SPEC (2017). Standard Performance Evaluation Corporation CPU Benchmark Suite. https://www.spec.org/cpu2017.

[Srikant and Shankar, 2002] Srikant, Y. N. and Shankar, P., editors (2002). *The Compiler Design Handbook: Optimizations and Machine Code Generation*. CRC Press.

[Stephenson et al., 2003] Stephenson, M., Amarasinghe, S., Martin, M., and O'Reilly, U.-M. (2003). Meta optimization: Improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, page 77–90, New York, NY, USA. Association for Computing Machinery.

[Sutton and Barto, 2018] Sutton, R. and Barto, A. (2018). *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning series. MIT Press.

[Tai et al., 2015] Tai, K. S., Socher, R., and Manning, C. D. (2015). Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1556–1566, Beijing, China. Association for Computational Linguistics.

[Tavarageri et al., 2020] Tavarageri, S., Heinecke, A., Avancha, S., Goyal, G., Upadrasta, R., and Kaul, B. (2020). Polyscientist: Automatic loop transformations combined with microkernels for optimization of deep learning primitives. *arXiv preprint arXiv:2002.02145*.

[Trifunovic et al., 2009] Trifunovic, K., Nuzman, D., Cohen, A., Zaks, A., and Rosen, I. (2009). Polyhedral-model guided loop-nest auto-vectorization. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 327–337.

[von Praun et al., 2011] von Praun, C., von Praun, C., Fineman, J. T., Leiserson, C. E., Gallopoulos, E., Snir, M., Heath, M., Grice, D., White, A. B., López, P., Meister, B., Vasilache, N., Wohlford, D., Baskaran, M. M., Leung, A., Lethin, R., Saltz, J. H., and Das, R. (2011). Reduce and scan. In *Encyclopedia of Parallel Computing*, pages 1728–1736. Springer US.

[Wang et al., 2018] Wang, X., Li, C., Golbandi, N., Bendersky, M., and Najork, M. (2018). The lambdaloss framework for ranking metric optimization. In *Proceedings of The 27th ACM International Conference on Information and Knowledge Management (CIKM '18)*, CIKM '18, page 1313–1322, New York, NY, USA. Association for Computing Machinery.

[Wang et al., 2013] Wang, Y., Wang, L., Li, Y., He, D., Chen, W., and Liu, T.-Y. (2013). A theoretical analysis of ndcg ranking measures. In *Proceedings of the 26th annual conference on learning theory (COLT 2013)*, volume 8, page 6.

[Watkins and Dayan, 1992] Watkins, C. J. and Dayan, P. (1992). Q-learning. *Machine learning*, 8(3-4):279–292.

[Weiss et al., 2018] Weiss, G., Goldberg, Y., and Yahav, E. (2018). On the practical computational power of finite precision rnns for language recognition. *arXiv preprint arXiv:1805.04908*.

[Wismüller, 2011] Wismüller, R. (2011). Parallel loops. In Padua, D., editor, *Encyclopedia of Parallel Computing*, pages 1079–1087. Springer US, Boston, MA.

[Wolf et al., 1996] Wolf, M. E., Maydan, D. E., and Ding-Kai Chen (1996). Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, pages 274–286.

[Xhafa and Dika, 2016] Xhafa, V. and Dika, F. (2016). Parameters that affect the parallel execution speed of programs in multi-core processor computers. *ResearchGate*.

[Zaparanuks et al., 2009] Zaparanuks, D., Jovic, M., and Hauswirth, M. (2009). Accuracy of performance counter measurements. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 23–32.

[Zar, 1972] Zar, J. H. (1972). Significance testing of the spearman rank correlation coefficient. *Journal of the American Statistical Association*, 67(339):578–580.

[Zheng et al., 2020] Zheng, S., Liang, Y., Wang, S., Chen, R., and Sheng, K. (2020). Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 859–873, New York, NY, USA. Association for Computing Machinery.

[Zimmerman et al., 2020] Zimmerman, M. I., Porter, J. R., Ward, M. D., Singh, S., Vithani, N., Meller, A., Mallimadugula, U. L., Kuhn, C. E., Borowsky, J. H., Wiewiora, R. P., Hurley, M. F. D., Harbison, A. M., Fogarty, C. A., Coffland, J. E., Fadda, E., Voelz, V. A., Chodera, J. D., and Bowman, G. R. (2020). Citizen scientists create an exascale computer to combat covid-19. *bioRxiv*.