Automatic Source-to-Source Optimizations using Machine Learning

Maksim Berezov

PSL UNIVERSITÉ PARIS

THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PSL

Préparée à Mines Paris-PSL

L'automatisation des optimisations source-à -source de programmes en utilisant des techniques de Machine Learning

Automatic Source-to-Source Optimizations Using Machine Learning

Soutenue par

Maksim Berezov

Le 06 Decembre 2022

École doctorale n°621

ISMME

Spécialité

Informatique temps réel, robotique et automatique

Composition du jury :

Frédéric MAGOULÈS Université Paris Saclay – Centrale Supélec	*Présidente du jury, Rapporteur*
Cédric BASTOUL Université de Strasbourg	*Rapporteur*
François IRIGOIN MINES Paris – PSL Université	*Examinateur*
Guillaume IOOSS Inria Grenoble	*Examinateur*
Thiago TEIXEIRA Stanford University: Palo Alto, CA, US	*Examinateur*
Corinne ANCOURT MINES Paris – PSL Université	*Directrice de thèse*

MINES PARIS | PSL

Contents

Acknowledgements

My doctoral thesis was a fascinating journey that took four years. I've met many people I would like to express my deep gratitude to.

I am grateful to my supervisor, Corinne Ancourt. She was always supportive and helpful in all my activities. Her guidance was encouraging and motivating in the scientific field. I am grateful for the warmth, kindness, and care that she had toward me during my Ph.D.

I am beholden to the colleagues I had the luck to work and collaborate with during their period at CRI. Big thanks to Justyna Zawalska and Maryna Savchenko. Their impressive work totally changed the vector of my research. I am thankful to Chermen Tsgoev for productive discussions about tiling and for being an amazing friend.

I am grateful to the members of my jury: Cedric Bastoul for deep and interning discussion on my thesis; Frederic Magoules for being so a great president of my jury; Thiago Teixeira for the tools that I've used during my work and his help and collaboration; François Irigoin for creating so pleasant atmosphere at CRI laboratory, it was a pleasure to work there; Guillaume Iooss for navigating me during the first year and giving me many precious pieces of advice.

I express my gratitude to all the people that I have met at CRI. It was an honor to work together: Fabien Coelho, Laurent Daverio, Emilio Gallego, Olfa Haggui, Olivier Hermant, Pierre Jouvelot, Claire Medrala, Bruno Sguerra, Lucas Sguerra, Laila Bouhouch, Guillaume Genestier, Hassib Abdouli, Adilla Susungi, Claire Medrala, Claude Tadonki. Highly likely, I forgot to mention many people, I'm really sorry.

I say thanks to my dear friends Patryk, Monica, and Gabriella for their support, interesting discussions, and a lot of fun together.

Finally, I send love to my parents: Natalia and Yuri, who did everything for me to get an education. Without their support, I would not finish the Ph.D. They were always near, not physically but morally.

Chapter 1

Introduction

Résumé

Les codes informatiques sont partout, dans nos appareils mobiles, les rovers Mars Perseverance [27], en passant par les jeux et les équipements critiques (médicaux, systèmes autonomes, aérospatial, etc.). Tous les calculs qu'ils impliquent nécessitent de grandes ressources de mémoire, énergétiques et du temps. Pouvoir appliquer des transformations de code qui permettront de réduire ces temps d'exécution, voire de réduire le besoin en ressources mémoire et énergétique, est essentiel, surtout en cette période de consommation raisonnée des ressources.

Cette thèse se place dans le contexte des transformations de programme source-à-source. Cela signifie qu'ayant un programme écrit dans un langage de programmation source (dans notre cas le langage C), nous voulons obtenir un programme transformé dans le même langage de programmation qui est plus efficace. La fonction de coût qui guide l'optimisation peut varier : temps d'exécution, temps de compilation, consommation mémoire, combinaison de différentes métriques, etc. De plus, cette thèse tire parti des techniques d'apprentissage automatique.

L'apprentissage automatique est une classe de méthodes d'intelligence artificielle, dont une caractéristique distinctive n'est pas une solution directe au problème, mais l'apprentissage par évaluation de solutions sur de nombreuses tâches similaires. L'objectif principal de cette thèse est de définir une recette appropriée de transformations source-à-source de programme pour améliorer une fonction de coût choisie (temps d'exécution, empreinte mémoire, etc.) pour une architecture homogène en utilisant le Machine Learning.

Introduction

Computer codes are everywhere, from our mobile devices to Mars Perseverance rovers [27], games, and critical equipment (medical, autonomous systems, aerospace, etc.). All the calculations they involve require execution time and large memory and power resources. Being able to apply code transformations that will reduce

their execution times, or even reduce the need for memory and energy resources, is essential, especially in this period of rational resource consumption.

Code optimizations could be applied at very different levels. For instance, one may propose an algorithm that would have lower asymptotic computational time complexity, and the other - a compiler - might generate code that runs faster. This thesis takes place in the context of source-to-source program transformations. This means that having a program written in a source programming language (in our case C language), we want to obtain a transformed program in the same programming language which is more efficient. By transformation, we mean an action that changes the source code but does not change the semantics of the program after transformation.

The cost function that guides the optimization can vary: execution time, compile time, memory consumption, a combination of different metrics, etc. The most significant and used in the context of this Ph.D. is the execution time and memory consumption.

In particular, we focused on nested loop transformations, because typically loops are the most consuming part of a program. As experiments show, [1], [29], [32] appropriate transformations can greatly improve performance.

The common source-to-source loop transformations targeted in this thesis are:

- Loop unrolling
- Loop tiling
- Loop interchange

Loop tiling is the core transformation in the context of this thesis. We show that this transformation has many parameters to predict that were not properly investigated in literature. Loop interchange is connected with loop tiling, it is necessary to apply it before tiling in order to either to enable the loop parallelism or to improve the locality of array accesses. Loop unrolling is a classical transformation that provides gains in performance for many kernels of our interest. Although, we considered three transformations, our methodology allows the addition of arbitrary transformations (e.g. loop fusion/fission) for Machine Learning modeling. They would follow the same steps that we applied for the other transformations.

It is very difficult to predict the execution time of a program (or any other metric) even after a single transformation. Indeed, many factors (e.g. temporal and spatial locality, computational loop overhead, instruction-level parallelism) underlie performance, and transformations can make each of them worse or better.

The performance also depends on the architecture. Nowadays, more and more new architectures emerge. They are more productive and efficient, but also more complex, which undoubtedly makes program optimization more difficult. Moreover, find a good optimization for one architecture is not enough, since the performance is not portable.

In addition, this thesis takes advantage of machine learning techniques. Machine learning is a class of artificial intelligence methods, a distinctive feature of which is not a direct solution to the problem, but learning in applying solutions to many similar tasks. These methods are widely used in fields such as banking, computer vision, speech recognition, bioinformatics, and many others. The main objective of this thesis is to define an appropriate recipe of source-to-source program transformations to improve a chosen cost function (execution time, memory footprint, etc.) for a homogeneous architecture using Machine Learning.

1.1 Motivation

Choosing the best parameters of loop transformations is not an easy matter. The main difficulty is that the number of possible parameters of the transformations can be very large. For instance, if we would like to apply parallelepiped tiling on a kernel that multiplies 4096×4096 matrices, this gives us 68.7 billion potential partitioning matrices. Adding the fact that we would like to investigate simultaneously the loop unrolling and loop interchange transformations, this gives us 395 billion potential options.

Many heuristics of discrete optimization (e.g. genetic algorithms for autotuning, simulated annealing, or just random sampling) can be integrated into a compiler. They work quite well and help to find acceptable solutions. But they do not generalize the knowledge about previous "bad" and "good" executions. Machine Learning could potentially do that. This is the main motivation for the techniques we used in this thesis: the search in a large optimization space could be accelerated with a smart generalization technique, and it can also be considered as a form of extrapolation.

The other motivation to use Machine Learning is that many factors affect performance. Some optimization combinations can improve one factor and worsen others. As mentioned before, temporal and spatial locality, computational loop overhead, and instruction-level parallelism could be considered. Creating heuristics for each factor does not seem like the best strategy. Moreover, what if we do not know about the existence of some factors that affect performance? The sources of data that can provide information on these factors are of different natures. It can be information obtained during static analyzes or dynamic analyses, or it can be some graphs describing the code. All this interaction cannot be represented as an explicit function. As experiments show, machine learning successfully copes with such tasks.

1.2 Research challenges

The challenges we faced during our study are common for almost all Machine Learning pipelines. We use Machine Learning as a tool, hence we see our main impact

not on improving the ML techniques but on integrating existing techniques for our problems. The most crucial challenges are listed below.

- **Data collection**: Since this thesis takes place in the context of Machine Learning, it is crucial to have appropriate training and validation sets to train properly and draw reasonable conclusions. There are many options on how to mine data for training (e.g. open-source code parsing, synthetic code generation, using existing benchmarks). Data collection becomes the main research challenge before creating the ML itself. We argue that this is the most important step since the ML model just captures patterns observed in the training set. We started our investigation with known existing benchmarks and then shifted to the field of synthetic code generation.

- **Machine Learning modeling**:

  Machine Learning is a very general concept. It aggregates many different ideas and approaches inside. For instance, we can distinguish sub-fields such as supervised learning, unsupervised learning, reinforcement learning, and many many others. On the other hand, the code optimization domain also aggregates a lot of different sub-fields. It was a challenge to find their proper interaction, where the code optimization domain benefits the most from ML techniques.

  To address this challenge, we have proposed solutions to the following problems:

  - Synthetic code generation.
  - Determination of the optimal parameters for certain transformations such as tiling, unrolling, and interchange.
  - Choosing the best way to encode the meaningful code properties in fixed numerical vectors.

  As the most appropriate sub-fields of ML, we used supervised ML (regression methods + classification methods) and Active Learning methods for the code generation. The key difference between supervised and unsupervised learning is that we have training labels for learning in supervised process. It serves better when it is crucial to predict the outcomes for the new data (the goal of this thesis).

- **Feature space design**: When the training set is collected and the problem statement is fixed, it is the step of feature representation of the data that we operate. Chapter 2 highlights the whole evolution process toward the feature space design in code optimization. Shortly speaking, the use of ML started with the use of handcrafted features created by experts. Then the

focus shifted to the automatic collection of specific features that are compiler-related. The current trend is to use embeddings (vector representation of some concepts) which have been obtained during representation learning with some deep neural networks. It is a challenge to define what kind of features fit the most for our problems. In this thesis, we work with handcrafted features mainly, but also endow the search with a more abstract code representation that does not rely on any potentially biased knowledge. In the scenario of more abstract code representation, we allow the system itself to choose which features to take from raw data for better performance.

1.3 Research directions

This thesis targets the applicability of Machine Learning models to the code optimization domain. We investigated four main research questions during this thesis.

- **Question 1**: Can we collect enough training data for Machine Learning models that would be representative of all the transformations that we are targeting?
- **Question 2**: Are we able to make our predictions automatically for the different options of the Tiling transformation?
- **Question 3**: Can we propose alternative ways to encode the code properties that would be invariant for the code transformations?
- **Question 4**: Can our predictions help to accelerate the iterative search of the best parameters?

1.3.1 Data mining

Chapters 3 and Chapter 4 aim to answer **Question 1**. We believe that creating our synthetic code generator is the most reasonable option to solve the problem of lack of data. Chapter 3 provides the main building block and guidelines for the generator. We define the associated domain-specific language to describe the desired properties of the generated code. Chapter 4 provides the way how to generate just the most representative data that maximizes the performance of the ML model. It relies on the idea of Active Learning that not all data samples are equally important, and the model can choose the best match by itself.

1.3.2 Tiling transformation

The state-of-the-art Machine Learning models that target tiling transformation aim to make predictions only with cubic/parallelepiped tiling. To answer **Question 2**, we try to show that other tiling options and hyperparameters impact a lot on performance. Chapter 5 shows that different tiling shapes, scanning directions, and hyperparameters are crucial components of well-tuned code. In Chapter 6, we build the predictions of such tiling options as 1) Inter-tile scanning directions 2) Intra-tile scanning directions 3) Tile shape 4) Tile size. Moreover, we give a comparison of existing pipelines to predict the optimal tile size and define the best way to do that.

1.3.3 Feature space design

The majority of well-performing ML models in the field of code optimization rely on handcrafted code characteristics proposed by an expert. Our research therefore also extends the research in this direction. We propose a representation to encode data dependencies in chapter 6. It allows for characterizing data dependencies with a fixed-size data structure, precise enough to be exploited by ML techniques. However, we do not only consider handcrafted features. Chapter 6 tries to answer **Question 3** by proposing some other alternative feature spaces. We consider the encoding of the data-dependence graph and the control-flow graph into vectors of fixed size and use them as features for the ML model. Close vectors correspond to the graphs which "look similar". For instance, code with many data dependencies will be different from code without dependencies. The main advantage of our features is that they could be used for the prediction of any transformation.

1.3.4 Iterative search acceleration

State-of-the-art Auto-tuners are very sensitive to the initial parameters of the search. Poorly chosen parameters could lead to a longer search and not optimal found configuration of the optimal parameters. This is a very common case for Autotuners since the search is task-agnostic. The search engine knows nothing about the problem of interest and could choose the not optimal first solution. Chapter 7 tries to answer to **Question 4**. We used the predictions of our models as the initial seed for the Autotuner. Moreover, we investigate the ability of the model to perform the iterative search by itself.

1.4 Thesis structure

This thesis is organized as follows. Chapter 2 presents the related work towards 4 different fields: 1) Data collection 2) Prediction of optimization parameters 3) Feature space design 4) Machine Learning methods. Chapter 3 and 4 present our synthetic code generator that would be used for the collection of the training set for each ML model. Chapter 4 presents our strategy on how to select the most representative data generated by our generator given time constraints. Chapter 5 investigates the impact of different tiling parameters on code performance. Chapter 6 attempts to predict all these parameters that are crucial for efficient code generation and evaluates two modeling pipelines to predict the tile size. Chapter 7 evaluates how the one-shot prediction of the tile sizes used as an initial seed for the Autotuner could accelerate the search process. We also evaluate the applicability of our prediction ranking in the autotuning process.

Chapter 2

Related work

Résumé

Cette thèse adresse deux domaines : l'optimisation de code et l'apprentissage automatique. Ce chapitre met en évidence les travaux connexes dans les deux domaines et également leur interaction. Nous considérons l'apprentissage automatique comme un outil pour résoudre certaines tâches concrètes, et l'optimisation du code comme le domaine d'applicabilité de cet outil. Nous pensons qu'il est raisonnable de fournir des travaux connexes pour chaque phase du pipeline d'apprentissage automatique. Le pipeline typique de Machine Learning pourrait être décomposé de plusieurs phases :

- Collecte de données
- Formulation du problème et modélisation ML
- Modélisation de l'espace des caractéristiques
- Machine Learning formation/prédiction/évaluation des résultats

Nous présentons des travaux connexes pour chaque phase. La section 2.2 résume la manière dont les données sont collectées et extraites dans les pipelines d'optimisation de code. La deuxième phase en section 2.3 met en évidence les problèmes d'optimisation du code lorsque l'apprentissage automatique est utilisé. La troisième section 2.4 cible les caractéristiques concrètes du code requises pour la modélisation des problèmes et celles utilisées dans l'état de l'art traitant de la conception de l'espace des fonctionnalités. La dernière section 2.5 fournit des travaux connexes concernant les techniques et les pipelines d'apprentissage automatique existants.

2.1 Introduction

This thesis takes place in two domains: code optimization and machine learning. This chapter highlights related work in both domains and also in their interaction. We consider machine learning to be a tool to solve some concrete tasks, and code optimization to be the domain of applicability of this tool. We believe it is reasonable to provide related work for each phase along the machine learning pipeline. The typical Machine Learning pipeline could be decomposed into several phases:

- Data collection
- Problem formulation and ML modeling
- Feature space modeling
- Machine Learning training/prediction/result evaluation

We provide related work for each phase. Subsection 2.2 summarizes how data is collected and mined in the code optimization pipelines. The second phase in Subsection 2.3 highlights code optimization problems when Machine Learning is used. The third subsection 2.4 targets concrete code characteristics required for problem modeling and how the authors of existing research deal with feature space design. The last subsection 2.5 provides related work in the scope of existing machine learning techniques and pipelines.

2.2 Data collection

Quality of data is a crucial component in ML pipelines. The training data must be as indicative as possible and reflect similar characteristics to those on which ML techniques are applied. This section summarizes common approaches to collecting training and validation sets for ML issues in the code optimization domain.

2.2.1 Existing benchmarks

There are various known benchmarks for the C programming language that address specific aspects. For instance, BEEBS Benchmarks [51], Embench™ [42], MiBench [43] address performance analysis on embedded platforms. PolyBench 4.2 [46], Livermore loops (LFK) [49], LCALS v1.0.2, TSVC, [47], LORE [66] focus mainly on compiler optimizations and performance analysis. However, these benchmarks contain a limited number of typical kernels. For instance, TSVC contains 151 perfectly-nested loops, PolyBench 4.2 contains 30 computational kernels (kernel may contain several loop nests), Livermore loops (LFK) have 30 loop nests, and LCALS v1.0.2

contains 32 loop nests, LORE aggregates loops from other benchmarks and contains 2499 loops in C.

This amount of data may not be enough when the code optimization domain actively integrates with the machine learning domain [36]. The strength of Machine Learning techniques often comes from the use of a large training set. For instance, MNIST [34] a benchmark for image processing contains 70.000 images, LibriSpeech [39] for speech recognition includes 1000 hours of speech, Enron corpus [40] for natural language processing aggregates 500,000 messages.

Therefore, there are much less benchmark data available in the code optimization domain than in fields where ML shows state-of-the-art performance. There is not enough training data to properly cover the feature space of parameters for complex transformations such as loop tiling, loop unrolling, loop interchange, etc. Note further that different transformations have different feature spaces from a machine-learning perspective. One training set could capture better features for one transformation, another - for another. It becomes challenging to create a universal training set. Thus, synthetic code generation is a crucial component of the proper use of Machine Learning for code optimization.

The possible data collection can be organized in two ways: data mining from open sources or synthetic data generation.

2.2.2 Data mining

The advantage of data mining is that we collect programs directly from the real world. These are exactly the programs that we want to optimize. Also, modern hosting services provide countless amounts of data. For instance, there are at least 10 million GitHub repositories. There is a whole line of research in this direction [53], [56], [55], [54]. However, there are some drawbacks, such as the correctness of the data, their completeness, difficulties arising from parsing, the lack of input data necessary for their execution, the diversity of the types of collected programs, and much more.

2.2.3 Synthetic data generation

The alternative direction that tries to get rid of the drawback of the previous approach is synthetic code generation. Some researchers have decided to create or synthesize their own benchmarks.

Deniz et al. [57] propose a MINIME-GPU benchmark synthesis framework for GPU using OpenCL. Joshi et al. [58] propose the BenchMaker framework where microarchitecture-independent characteristics are used to describe the required workload. Code synthesizing for some particular problems was introduced in [59], [60]. However, the closest to our work are projects such as GENESIS and CLgen.

Chiu et al. [37] introduce GENESIS, a language for Generating Synthetic Programs. The main idea is that the user can annotate the code template with the parameters that she/he wants to vary. The generation works based on the known pre-defined statistical distribution of the parameters. It helps to obtain reliable data for known domain-specific problems. Statistical distributions depend on the domain knowledge of the expert who tunes them. It makes it complicated to imitate not investigated domains.

Cummings et al. [38] [21] propose a slightly different approach. Their code generator CLgen has a deep learning model as the core of their pipeline. It helps to capture statistical distributions over training codes and imitate new domains without expert knowledge. The disadvantage of this approach is that it requires a huge training set for the deep learning model.

2.2.4 Conclusion

Our work in Chapter 3 presents our original contribution to this related work. We have developed tools to automatically generate synthetic data because the existing solutions do not fully reflect the patterns (uniform and non-uniform data-dependencies, loop index permutations, different patterns of array accesses) we are interested in.

2.3 Prediction of optimization parameters

This section focuses on how Machine Learning can help in the selection of optimization parameters. By optimization parameters, we mean a huge scope of things. It can be parameters of program transformations, compilation flags, paths, and many other things. One of the main contributions of this thesis targets the loop tiling transformation. We consider it reasonable to provide related work for this program transformation separately.

2.3.1 Loop tiling transformation

Research related to tiling parameter prediction can be classified into three main groups.

- Static analytical models handcrafted by an expert,
- Iterative auto-tuners,
- Static analytical models derived by Machine Learning algorithms.

The earliest approaches [72], [71], [78], [75] [73], [76], [74], [79], [77] [24] related to the tile size selection problem refers to analytical models that capture architectural, program and compiler characteristics. These models attempt to analytically disentangle all the complexity of their interactions and provide the optimal size. However, the interaction of so many concepts can be a very complex problem. Therefore, analytical solutions will lead to not fully satisfactory results. Moreover, new analytical models need to be re-analyzed for every single architecture or compiler and require a huge expert time to maintain the up-to-date versions of the analytical solution. The exact opposite is the approach using auto-tuners [32], [88], [84], [87], [86], [85], [83]. The idea is to explore a grid of transformation parameters iteratively by finding better and better solutions. The majority of existing auto-tuners are limited to cubic tiling, which may not be the best option. This approach provides good results but requires a lot of time to execute potential points, and it does not provide any insights about the final choice.

Our study relates to the third group, we use Machine learning techniques to make predictions and generalizations about the tiling transformation. The most related papers to our study are [1], [45], [81], [44].

Yuki et al. [1] investigate the problem of automatic tiling selection using machine learning approaches. The authors consider cubic tiling on three nested loops with 2D data. The authors describe each loop nest based on the array references inside the loop nest. Each array reference can either take advantage of the spatial locality given by the prefetcher or not, or be constant in the innermost loop. Yuki et al. also mark each reference as a read or write reference.

Liu et al. [45] propose a slightly similar approach. The key difference is that the construction of the feature space can potentially contain loops of any fixed depth and data of any dimensionality.

Malik [44] uses dynamic hardware performance characteristics to determine the optimal tile size. The drawback of this approach is that hardware performance does not provide any insights about the code at the source level. Moreover, Malik considers cache interaction just for the L1 cache.

Rahman et al. [81] propose an alternative to iterative compilation. Their method predicts execution time based on given input tile sizes. This approach helps to learn the distribution of beneficial tile sizes for a given kernel and to determine the limits of convergence for random empirical search. However, this model does not provide any insights into the factors that impact this distribution.

We consider our work to be closest to [1] and [45]. However, all related work focuses on the prediction of just one parameter - the tile size.

The originality of our approach is that we consider several tile shapes and provide a way to predict the optimal scanning directions of tiles and tile elements during their execution. Moreover, we consider data dependencies to be a piece of meaningful information to make this choice and encode them in a feature vector for Machine Learning problems.

2.3.2 Static approach for optimization

This subsection highlights related work for code optimization where parameters are predicted statically. It means that code execution is not required for the optimization decision.

Transformation parameters prediction

Monsifrot et al. [52] make a binary decision on whether to unroll a loop or not. They identify 5 classes of features that have an impact on the result: memory access class, arithmetic operations, size of the loop body, control statements in the loop, and number of iterations. For binary classification, the authors use the Random Forest(+ boosting technique). The above features are the input of the algorithm. The disadvantages of this approach include the fact that the authors do not predict how many times this loop should be unrolled.

Stephenson et al. [134] try to predict the optimal unrolling factor. They consider this problem as a multiclass classification with 8 possible values of the unrolling factor {1,2,3...8}. To solve it, they use NN and SVM. For this classification, the authors used 38 handcrafted features. Stephenson et al. conclude that this technique predicts optimal/ near-optimal unrolling factors in 68% / 79% of cases.

Magin et al. [128] addresses a problem of thread-coarsening transformation.(merging the code that is executed by different threads into one thread). This paper is interesting because the authors used a complex machine-learning model consisting of cascade-connected neural networks to predict the optimal value of thread-coarsening. However, the input for the model was not raw code, but meaningful features. This creates limitations for further generalization of the results to other transformations. Nevertheless, the authors come to the conclusion that this model shows the best results of their popular heuristics. In particular, they achieved an average speed from 1.11x to 1.33x depending on the GPU architecture.

Fursin et al. [132] proposes MILEPOST Framework. This work is the first attempt to create an open-source compiler that uses machine learning.

The authors conclude that this compiler is able to show competitive performance and can improve existing baselines on the benchmarks tested. The main goal of Machine Learning is to predict optimal optimization passes. For ML, the authors use models of two classes:

- Probabilistic Model. The probabilistic distribution for the training set of programs is first studied. Then, for the new program from the test set, we find its closest neighbor from the training set.

- Transductive Model. Supervised Machine Learning problem, decision tree model was used to obtain predictions.

Park et al. [114] propose a new approach to represent programs and capture the most relevant information for Machine Learning models called graph-based characterization (they test on SVM in order to predict optimization sequences). This approach can work in both iterative mode and non-iterative mode. The authors conclude that on the tested benchmarks, this representation shows better performance (73% of the maximum) than the existing concepts (53%).

Compiler flags prediction

Cavazos et al. [109] investigate the possibility of using machine learning tools in order to select optimal compilation flags. They represent each program as a 26-dimensional vector. Then, using a logistic regression-based strategy, the authors obtain the required flag vector for this program. Depending on the benchmark, the authors get a decrease in the execution time by 25% and 51% compared with baselines.

Phase-ordering prediction

Kulkarni et al. [107] investigate the problem of phase-ordering prediction for Jikes RVM JIT compiler. The idea is to predict the best next single optimization (or predict stop of prediction), then re-evaluate the characteristics of the code and predict the best next optimization. The authors use a neuro-evolution approach to derive ANN, which will be used. Kulkarni et al. conclude that this approach is effective and the first of this kind.

Other predictions

Shivan et al. [129] predict which compiler will generate the fastest code for a loop nest. Namely, the optimization of serial code to be auto-parallelized was observed. There are 4 code optimizers: clang (LLVM), GCC (GNU), ICC (Intel), and Polly. The goal is to predict the optimal optimizer for a given code. Performance counters were used as features and for classification, the Random Forest algorithm was applied. The performance gain from the ML predictions is up to 1.42x for the serial code and up to 1.71x for the auto-parallelized code across two multi-core architectures.

2.3.3 Dynamic approach for optimization

The alternative approach to code optimization is a dynamic one. We execute a program to optimize and collect essential information about its performance.

In this subsection, we provide an overview of strategies used in dynamic search space exploration. We will highlight the most influential articles on each approach

and attempt to understand the big picture of how each strategy has evolved and its prospects.

Iterative compilation without Machine Learning

The iterative compilation is a technique for optimizing programs, where each new iteration generates a new version of a program according to a certain criterion. The most popular iterative search techniques are:

- Genetic algorithms
- Random search
- Simulated annealing
- Grid search
- Window search

After several steps, we can get a well-optimized program. In the general case, this approach is very expensive, but for some applications (e.g. embedded applications), this cost is amortized. The opposite direction is a strategy based not on program profiling, but on collecting some of its static indicators and directly predicting the optimization of the target.

Space exploration

Bodin et al. [98], **Kisuki et al. [115] and Fursin et al. [125]** conclude that iterative compilation overperforms existing static methods for program optimization, and it can achieve high optimization level of optimization for a small number of steps. The authors considered the optimization set of loop unrolling and tiling and/or array padding. To focus on iterative search, they applied heuristic strategies.

Cooper et al. [151] consider iterative compilation in terms of reducing code size. The authors translate the source code to ILOC (low-level intermediate language). After that, Cooper et al. explore iloc-to-iloc transformations. The search space contains 10 non-parametrized transformations and allows to the generation of sequences of arbitrary length with repetitions.

Triantafyllis et al. [116] present the Optimization-Space Exploration (OSE) technique based on the search space pruning strategy and use of static heuristic, which reduces the number of iterations.

Pan et al. [113] propose an orchestration heuristic algorithm in order to find effective compilation settings. The key idea is to iteratively eliminate options with a negative effect on a cost function according to a proposed heuristic algorithm. The authors claim that this approach requires less time to tune than existing methods.

Ding et al. [105] present a two-level machine learning approach in order to predict the most optimal optimization strategy for iterative compilation depending on a given input.

Almagor et al. [153], [103] address the problem of order in compilation sequence. This article attempts to answer 3 questions: (1) What percentage of theoretically possible sequences reach a specified level of performance relative to the absolute maximum? (2) Is there a pattern in the distribution of "good" sequences? (3) Understand the distribution of local minimums. Their frequency will influence the choice of the optimal search strategy. The authors consider sequences of length 10 with 5 optimizations: peeling one iteration of a loop, partial redundancy elimination, peephole optimization over logical windows, register coalescing via graph coloring, and dead code elimination. The optimization space has 5^{10} points.

Almagor et al. conclude that (1) about 15% of sequences are in 10% of the maximum, and 30% of all sequences are in 20% of the maximum. (2,3) The authors managed to find one large cluster of good solutions (2.6% of the maximum) and many isolated local minimum clusters. The authors also investigate the possibility of iterative search using genetic algorithms in the search space with 13^9 points, and conclude that this approach overperforms baselines.

Iterative compilation navigated by Machine Learning

Studies based on heuristic search strategies have several shortcomings. The main one is that the knowledge obtained from previous experiments is not generalized for future experiments. This leads to an increase in the number of iterations to achieve the desired level of optimization and to the deterioration of performance. Using machine learning techniques can help overcome these shortcomings.

Agakov et al. [118] propose a methodology using machine learning to identify areas of the transformation space that are most likely to improve performance. The idea is to use an Independent identically distributed model or Markov Model [23] to better target Genetic Algorithms or Random Search. The new program must go through the stage of feature extraction and mapping to the 5-D space (after PCA), where we find its "nearest neighbor" (for which we have learned probability distribution).

The authors consider the following set of transformations: Loop unrolling (1-4), loop flattening, for-loop normalization, non-perfectly nested loop conversion, break load constant instructions, common subexpression elimination, dead code elimination, hoisting of loop invariants, move loop-invariant conditionals, copy propagation. That is, 9 are not parametrized transformations and 1 is parametrized. The output is the sequence of non-fixed length. The authors conclude that this method shows highly efficient results.

Cavazos et al. [130] try to generalize the behavior of programs at a lower level, namely, they profile each program 3 times in order to collect relevant hardware

performance counters. Next, the authors use a model based on logistic regression to find a mapping between performance counters and optimal optimization sequences. For the program for which we want to find the optimal optimization sequence (121 flags were selected in order to construct the search space), the input is only 3 times measured performance counters. The output is a distribution, which **Cavazos et al.** use to draw optimization sequences. The authors claim that by using this technique they are able to achieve a certain level of optimization much faster than the current state-of-the-art techniques.

Park et al. [100] considers three ways of modeling the prediction of optimization sequence: Sequence Predictor, Speedup Predictor, and Tournament Predictor. Tournament Predictor is a novel approach and the authors investigate its applicability. The authors consider 7 optimization phases (45 unique optimizations, there are a lot of optimizations related to loop nest optimization phases). The input of this model is performance counters and two optimization sequences, the output is the prediction of the best sequence. Prediction models are based on SVN or Linear Regression. The authors claim that Tournament Predictor overperforms Sequence and Speedup Predictor and generally shows its viability on a variety of different benchmarks.

Ashouri et al. [108] present COBAYN a compiler autotuning framework based on Bayesian Networks. The authors use hybrid features in order to characterize programs in their model. Bayesian Networks were chosen as an intellectual core because of their capability to capture the probability distribution of available features.

The authors consider 7 optimizations: optimizations for floating-point arithmetic, unrolling of all loops, -O2 optimization level, not guessing of branch probabilities disabling loop optimizations on trees, disabling optimizations that inline all simple functions, and disabling induction variable optimizations on trees. The authors claim that they overperform existing state-of-the-art iterative and non-iterative compilation techniques.

Ogilvie et al. [123] emphasize that not all samples of points in the decision space provide useful information. And if we can use just the most "useful" measurements, we will significantly reduce the iteration compilation overhead. The authors use sequential analysis and active learning to solve this problem.

Ashouri et al. [106], Martins et al. [104] and Nombre et al. [99] focus on Design Space Exploration approaches in order to solve the phase ordering problem.

Ashouri et al. [106] propose a methodology based on predicting modeling. It has classical phases of data collection, training, and prediction. The final model is able to predict immediate speedup for a given optimization. This model implements 2 search strategies for the optimal sequence: DFS Search Heuristic and Exhaustive Search Heuristic.

The authors observe 13 different optimizations for LLVM (such as -loops -loop-simplify -lcssa -branch-prob). These 13 optimizations form 4 different genes (sequences) of compilation. The goal is to predict the next-gen. Ashouri et al. conclude

that these heuristics overperform LLVM by 4% and 2%.

Martins et al. [104] also address the problem of phase ordering for LLVM optimizations. The proposed methodology consists of encoding, clustering, and arrangement of pass stages. In order to generate clusters, Martins et al. use approaches such as Normalized Compression Distance, Neighbor-Joining, and a new ambiguity-based clustering algorithm.

The authors come to the conclusion that this approach achieves around 20x speedup in search space exploration compared to genetic algorithms and a 1.41x speedup over baseline in terms of the execution time.

Dubach et al. [135] propose a methodology based on the predictive model in order to predict optimization sequence. It requires 64 runs (with different optimizations) of a program to be optimized. Then the obtained execution times and the feature representation of the program (4 classes of features: cycles, memory accesses, operations executed, and operations presented in the source code) are used to predict a speedup for each possible optimization. ANN and Linear regression were used to construct a predictor.

The authors consider source-to-source transformations, namely, loop unrolling (with factors 1-4) and 9 non-parameterized optimizations like dead code elimination or move loop-invariant conditionals. This feature space yields 88000 invariants. The authors test their method in larger optimization space: 10^{34} points. Dubach et al. conclude that the proposed methodology manages to reduce the cost of search and gives predictions with a high correlation coefficient.

Frameworks

Many frameworks have been created for iterative compilation issues. For instance, **Chen et al. [112]** introduce their loop transformation framework CHiLL. The authors conclude that this framework finally fills the gap between the best hand-tuned codes and compiler optimizations for loops. Based on CHiLL and Active Harmony **[111] Tiwari et al. [88]** propose a framework that is able to perform both fully automatic code transformations and transformations under user assistance (static sequence defined by the user) and then search for the optimal parameters.

COLE Framework [87] considers multi-objective space exploration while all previous studies on iterative compilation considered only single-objective exploration.

Ansel et al. [86] propose framework for building domain-specific multi-objective program autotuners. This framework contains such search techniques as AUC Bandit Meta Technique, Nelder-Mead search, Torczon hillclimbers, and many others.

Baghdadi et al. proposes a polyhedral compiler framework called TIRAMISU. The distinct features of this work are four-level intermediate representation of a code and special scheduling language allowing targeting different architectures. The authors claim that their approach over performs existing state-of-the-art tools and hand-tuned codes.

As will be indicated in Chapter 4, I collaborated with the author of Locus framework **[32] by Teixeira et al.**, so I pay special attention to its description.

Locus is a system and a language to orchestrate the optimization of applications. In particular, Locus allows us to deal with source-to-source transformations. The distinctive feature is that we can define transformation sequences separately from the source code. This preserves the clarity and readability of the code. The idea behind Locus is that in the source code we specify the sections that we want to optimize. Then an optimization program written in a DSL (domain-specific language) defines transformations over the specified code segment. Parameters of transformations can be specified either explicitly or in the form of intervals. To apply transformations and find their optimal parameters, Locus integrates several modules :

- Transformation modules: Locus integrates multiple transformation models, namely, the source-to-source compilers PIPS and RoseLocus, Pragmas, and BuiltIn,
- Search modules: In order to perform a search in the optimization space, Locus integrates modules such as OpenTuner and HyperOpt.

The authors used various benchmarks to test performance and compare it with Pluto. They examined 856 loop nests with execution longer than 10^4 CPU cycles. The best code generated by Locus archives a 1.15x average speedup, while Pluto (with pre-defined parameters) reaches 1.05x. Moreover, Locus is able to transform 822 loop nests out of 856 (versus 397 for Pluto).

2.3.4 Conclusion

Our results presented in Chapter 6 bring contributions relative to this subsection. Our contributions target static approaches. We have focused on optimizing the loop tiling transformation and defined a set of parameters to predict that has not been covered in previous existing studies.

2.4 Feature space design

The proper choice of features is one of the key points in the performance of almost every machine-learning algorithm. Therefore, it is important to understand what data we can collect and use to train our models. Below we give a classification of possible features based on their essence.

Note that this section does not try to compare and determine which type of features is worse or better, and does not try to bring the most recent research with mentioned features. The main goal is to show the immediate advantage of different features for solving various problems relative to our work. The most illustrative studies are given as examples.

2.4.1 Static code features (without code profiling)

This subsection highlights the features that could be extracted during the phase of static code analysis without compilation and execution.

Handcrafted features. The simplest idea is to let the experts decide which features in the programs are important for a certain optimization and which are not. Relying on their rich experience and intuition, we can get a fairly representative set of values that describe in detail the distinctive characteristic of this program.

Below we give an example of two studies that use handcrafted features in order to predict the unrolling loop factor.

Monsifrot et al. [52] predict a binary decision whether to unroll a loop or not. The authors distinguish 5 classes of handcrafted features that can influence performance. These classes are "Memory access" features, "Arithmetic operations count", "Size of the loop body" and "Number of iterations". The authors claim that this approach overperforms compiler optimization (-O3).

Stephenson et al. [133] solve multi-class classification problems to predict the optimal unrolling loop factor. The authors distinguished 38 handcrafted features for Nearest Neighbours and SVN algorithms. They achieve a 5-9% (depending on the benchmark) improvement over existing methods.

Generated features. Although handcrafted features intuitively well describe the essence of the program, they have a number of shortcomings. The main thing is that the space of possible features is infinite and we have no control if these features are important or not. But experts choose from this infinite set only some limited list based on their intuition. It may potentially be biased and not optimal.

Leather et al. [120] propose an approach based on a combination of genetic algorithms and machine learning. The authors choose productions from the internal representation using genetic algorithms in order to construct feature space. Then the machine learning algorithm is trained on these features (productions) and consistently we get a better set of features. Leather et al. test their approach on loop unrolling. They achieve 76% of the maximum performance available, while state ML (using handcrafted features) achieves only 59%.

Namolaru et al. [121] propose a general method for systematically generating numerical features from a program. The authors view the program as a set of relationships between its entities and infer new ones and extract features from them. The authors evaluate this approach to the optimization flag selection.

They achieve 74% of the potential speedup obtained through iterative compilation on a wide range of benchmarks and four different general-purpose and embedded architectures.

Adams et al. [4] propose a mode to schedule Halide programs. The authors distinguish two types of features: schedule-specific features (either count events of various types, or characterize memory footprints) and algorithm-specific features (histograms of the operations performed). Then these features go through specific architecture of a Neural Network. The Neural Network produces schedules that are on twice faster than existing Halide auto-scheduler.

Representation Learning using Deep Learning. The approaches described above have deeper drawbacks. They, as a rule, are suitable only for predicting one specific heuristic (for example, loop unrolling). But if we want to predict something else, then we need to extract new features by ourselves (for handcrafted features), or apply the whole methodology from scratch and repeat many expensive experiments. In other words, these methodologies cannot generalize to many different optimizations (although they can generalize many different programs across the same optimization). Moreover, they are deeply embedded into the compiler (very dependent on the used AST). The use of neural networks looks very promising and powerful direction to overcome these limitations.

Cummings [126] developed a deep neural network that learns heuristics over the raw code, entirely without using code features. Their model consists of source re-writer, language model, LSTM (long short-term memory) [91] and neural network.

They test their model on 2 tasks: heterogeneous device mapping and thread coarsening. The authors claim that this approach overperforms state-of-the-art ML approaches with handcrafted features. Namely, in 89% of the cases, the quality is not worse than the Machine Learning models and the average speedup is 16% for heterogeneous device mapping and 12% for thread coarsening.

Baghdadi et al. [102] present a new cost model to predict program speedup. This cost model is a regression that takes special code embeddings as input. The authors introduce a program characterization in the form of an ordered tree of computation vectors. This concept includes 1) loop nest representation; 2) assignment representation; 3) loop transformation representation. This essential information helps to create representative code embeddings.

Graph-based Features. Many important relations in a program are usually represented as graphs. So, for example, it can be a dependency graph or a control flow graph. Undoubtedly, this information is important for more accurate

prediction and can be used by machine learning algorithms. The use of information in the form of a graph is reflected in [114]. To represent a program the authors use graph-based intermediate representation, which is based on CFG (Control-Flow Graph).

The authors conclude that such a feature presentation gives good results in the iterative compilation scenario (achieves 88% of maximum speedup in 5 iterations), and in the non-iterative scenario (achieves 74% of maximum speedup and overperforms state-of-the-art techniques).

2.4.2 Dynamic features

Dynamic features cannot be obtained directly without code profiling. As a rule, we can highlight three levels of abstraction to which dynamic features belong: application level, operation system level, and hardware level.

- **Application.** Roughly speaking, this level works at the same level of abstraction as static code analysis. Sometimes the data collected during static code analysis is not enough (some features cannot be counted, for example, the number of iterations may depend on user input). But we can count them after code profiling.
- **Operation System.** As the name implies, at this level we can track features that are defined at the operating system level. This, for example, is all about input/ output or CPU loads.
- **Hardware.** These features work at the lowest level of abstraction. They track the relevant information about the application performance on a given hardware. Only at this level we can, for example, find out the number of different cache misses because the cache sizes are determined by hardware.

An example of using such features is **[130]** where **Cavazos et al.** use performance counters in order to predict appropriate compiler optimizations. The authors conclude that this approach overperforms existing static code methods since it is able to capture a lot of relevant information about hardware performance.

2.4.3 Feature learning approaches

Deep Learning methods have the unique ability to reconstruct data distribution from raw data. This ability allows them to gradually enter into all spheres of our life, like text processing or banking. Not surprisingly, they are gaining more and more popularity in compiler optimizations.

Cumming et al. [126] developed a deep neural network that learns heuristics over raw code, entirely without using code features. The authors evaluated their

approach to heterogeneous device mapping and GPU thread coarsening. The key components of this model are:

- **Source rewriter.** The main tasks are parsing the AST, removing conditional compilation, then rebuilding the input source code using a consistent code style and identifier naming scheme.
- **Sequence encoder.** Code to a sequence of integers
- The order to apply transformations can be 1) not predetermined (we must find it by ourselves) 2) predetermined in accordance with best practices
- **Embedding.** Each token (integer value) to a real 64-dim vector (Similar tokens (int, float) to vectors with small distance)
- **LSTM.** Long short-term memory [91] (special kind of recurrent neural network).
- **Auxiliary inputs.** To maintain system flexibility, it allows you to add features that cannot be obtained from code.
- **Heuristic model** consists of Batch Normalization and Neural Network. The idea is that the neural network, based on the code representation and auxiliary inputs, makes predictions about the optimization parameter values.

The authors conclude that in 89% of the cases, the performance of this approach matches or surpasses the state-of-the-art predictive models that use handcrafted features. The average speedup is 16% for heterogeneous mapping and 12% thread coarsening factor prediction The main contribution of this article is that with the help of this model we can generalize over different transformations (we should not develop everything from scratch) and over different programs.

Chen et al. [127] propose a framework to iteratively optimize tensor operator programs for a given platform. It consists of Exploration Module, Code Generator, Cost Models, and History data

The authors paid great attention to the formalization of the problem, which became the theoretical basis for their invariant code generator. They also proposed two cost models (based on gradient-boosted trees and on TreeGRU), which allow the exploration module to select candidates to query on Hardware. The authors created a transferable representation for both their cost models that is invariant to the source and target domains. It means that transfer learning becomes possible.

Results on deep learning workloads show this framework overperforms TensorFlow, ARMComputeLib, TensorFlow XLA, TensorFlow Lite, and MXNet from 1.2× to 3.8× over tested frameworks. Although this paper contains a methodology of iterative search, it is interesting for us from the point of view of the presented cost models and the proposed data representation.

[38] by Cummings et al. addresses an important problem of generating data using deep learning models. The fact is that any machine learning/deep learning algorithm relies on the data on which it was trained. Deficiencies in the training data, such as their unrepresentativeness can have a negative impact on the training of the model. Hence, we get poor predictions that do not correspond to the real state.

The authors propose a methodology using deep learning architectures to generate a set of representative programs. They conclude that the state-of-the-art predictive model trained on data generated by this method speedups by 27%.

2.4.4 Conclusion

This section gives us a detailed overview of how the information about code characteristics is captured now and what we can improve. This information targets explicitly chapter 6. We see the drawbacks of the existing feature spaces and overcome them.

2.5 Machine Learning methods

Machine learning is a class of artificial intelligence methods. A distinctive feature is that these algorithms can be trained on a set of similar tasks to solve the given problem. Typically, they deal with tasks where it is not possible to establish the mapping function between input and output in an explicit, human-readable form.

Since Machine Learning acts only as a tool for obtaining results in the context of this Ph.D., we will only give a brief description of the classes of possible algorithms to give an idea of what results can be obtained.

2.5.1 Supervised Learning

The basic idea is that we have a labeled data set. Each sample consists of a labeled target variable and a set of independent variables (predictors). Predictors are also called features, they represent some characteristics, some values that describe the essence of a phenomenon being observed. Based on this labeled data set (training set), a machine learning algorithm tries to find an optimal mapping function between features and the target variable. When this function is found, we can use it to predict a target variable for new samples of data.

The values of the target variable can be discrete or continuous. In the first case, we are dealing with a classification problem, and in the second with a regression problem.

- Common classification algorithms are Support Vector Machines (SVM), K-nearest neighbours, Decision trees, and Random Forest, Naive Bayes, and

Log. regression, Gradient Boosting

- Common regression algorithms are Linear Regression, LASSO (least absolute shrinkage and selection operator), Ridge regression, and Elastic Net.

2.5.2 Unsupervised Machine Learning

In unsupervised Learning, we have no target variable to predict. Our data are not designed in such a way as to get directly from them some kind of correct answer. Instead, we can use existing feature representation to solve such problems as

- Clustering - grouping objects into some clusters. Objects within one cluster are more similar than objects outside this cluster. Common algorithms: K-Means, distribution-based clustering, hierarchical clustering,

- Associations - roughly speaking, to find interesting dependencies/relations in our data. Common algorithms are the Apriori algorithm, Eclat algorithm, and FP-growth algorithm.

- Autoencoders - compressing the initial data into some code, and then recovering data from this code only.

2.5.3 Reinforcement learning

Reinforcement learning is one of the methods of machine learning, during which the software agent is trained by interacting with a certain environment and getting rewards for its actions.

Common algorithms are Q -learning, Deep Q-learning, Actor-Critic, Policy Gradients, and Proximal Policy Optimization.

2.5.4 Deep Learning

Deep Learning is a subset of Machine Learning, which deserves special attention. A distinctive feature of deep learning models is that they are able to construct data representations automatically, not relying on extracted features. Thus, these models can be trained entirely on raw data. By raw data, we understand the data without any pretreatment. For example, it may be a set of image pixels or a source program code. The deep learning model usually refers to some subsets of neural networks (deep neural networks), which may have very different architectures and different parameters.

Common neural network architectures are Feed-forward networks, Convolutional NN, Residual NN, Generative Adversarial Networks, Recurrent NN, and many others.

2.5.5 Active Learning

Active learning is a sub-field of Machine Learning. The crucial idea is that the model itself decides which data to use for more effective training. It finds thought in areas where data annotation is relatively expensive or maybe not feasible. We would like to highlight this section because it finds deep reflection in the context of this thesis.

Sampling scenarios

Active learning pipelines [28] work under several scenarios: pool-based scenario [64], stream-based selective sampling [63] and membership query synthesis [62]

The pool-based approach is the most suitable for the code optimization domain. We could obtain more accurate results with this approach since we have time to evaluate all the candidates. The main idea is that we have a pool of candidates. Candidates of this pool are not labeled. For each candidate, we evaluate a measure that corresponds to a performance gain of the ML model after labeling this candidate. It provides the most accurate estimation due to the evaluation of each possible candidate from the pool of candidates. This approach has been used in the context of this thesis.

Classification problem

Classification is one of the two main problems of supervised Machine Learning tasks. The idea is to choose the output value from a finite number of classes, contrary to the regression problem where continuous value is predicted.

1. Uncertainty Sampling

   The idea is to get a measure of how a classifier is sure about its prediction on this data sample. If the classifier is not *sure* then we would like to label this piece of data. We can use the entropy [101] as the required measure.

   More formally, let θ denotes the classifier, x denotes a feature vector, and y_i stands for the label of the i-th sample. Then, the chosen data sample x^* equals

   $$x^* = \underset{x}{\operatorname{argmax}} - \sum_{i=1}^{n} P(y_i|\theta, x) * \log P(y_i|\theta, x)$$

   The idea is to choose the data sample, which features vector gives the highest value of entropy.

   Alternatively, we can judge based on the lowest probability of the most probable class.

$$x^* = \underset{x}{\operatorname{argmin}} P(y_+|\theta, x)$$

Where y_+ stands for the most probable class. The idea is that the classifier does not know exactly where this sample belongs, and we ask to label this piece of data.

2. Query-By-Committee

   The high-level idea of this method is to create a committee of models that introduce the concurrency hypothesis. The goal is to define the measure of their disagreement about a given data sample and to take the label which maximizes this measure. For instance, we have four models and two potential classes. The first two models vote for the first class, and the two second models vote for the second class. We can construct a measure of this disagreement.

   The following metrics could be used for this measure:

$$x^* = \underset{x}{\operatorname{argmax}} - \sum_{i=1}^{n} \frac{V(y_i)}{C} * \log \frac{V(y_i)}{C}$$

   Where $V(y_i)$ gives how many votes this label got from the committee, and C is the number of models in the committee. Alternatively, KL distance could be used.

Regression problem

Wu et al.[65] propose a technique to generate the most representative data for a regression problem. It consists of 3 algorithms: Greedy Sampling on the Inputs, Greedy Sampling on the Outputs, and Greedy Sampling on the Inputs and Outputs.

Let Λ consists of N samples $\{x_n\}_{n=1}^{N}$. The goal is to choose K most representative samples.

1. Greedy Sampling on the Inputs

   The main idea is to choose the initial point as the closest to the centroid of the global pool Λ, and then iteratively choose points farthest from the one already chosen to increase the diversity of the data. If we assume that k samples have already been chosen, then for each remaining sample the method computes the following distances.

$$d_{nm}^x = ||x_n - x_m||, m = 1, ..., k; n = k+1, ..., N$$

$$d_n^x = \min_m d_{nm}^x, n = k+1, ..., N$$

Samples with the highest d_n^x will be chosen.

2. Greedy Sampling on the Outputs

   The key idea is to use greedy sampling on the inputs to build the initial model, then to choose points with the farthest distance but in the output space according to the model prediction. The distance formulation d_n^y follows:

$$d_{nm}^y = ||f(x_n) - y_m||, m = 1, ..., k; n = k+1, ..., N$$

   where f(x) is the regression model and $\{y_m\}_{m=1}^k$ is a pool of already labeled outputs.

$$d_n^y = \min_m d_{nm}^y, n = k+1, ..., N$$

3. Improved Greedy Sampling on both Inputs and Output

   This approach considers the multiplication of the distances in the input and output spaces as the deciding metric. The data sample with the highest value is chosen.

$$d_n^{xy} = \min_m d_{nm}^y d_{nm}^x, n = k+1, ..., N$$

2.5.6 Conclusion

The overview of existing methods that we have just presented in this section highlights the main tools used in this thesis. Machine Learning algorithms are involved in each section of this Ph.D. Active Learning ideas are explicitly used in Section 4 and data collection of all experiments in this thesis.

2.6 Chapter conclusion

A detailed analysis of the related work provides us with valuable insights.

- Generating synthetic data is the suitable approach for our issues. It helps us to define the appropriate high-level specifications of the considered transformations. We can rely on existing benchmarks due to the amount of data and data mining due to the specificity of our optimizations.
- Research on loop tiling transformation is limited to tile size prediction. We see the potential to predict much more parameters.
- The scope of problems to which ML is applied is very heterogeneous. There is no common pipeline or problem formulation. We can formulate the problem of our interest in a very flexible way.

- Much research relies on handcrafted features. There is a trend towards representation learning, but it does not find deep reflection at the moment. We try to fill this gap.
- Machine learning is a powerful tool that finds applicability in many domains. There are prerequisites to successful use in code optimization. Active Learning helps us to accelerate synthetic data collection in such a time-consuming domain.

Chapter 3

Synthetic data generator

Résumé

De nos jours, les algorithmes de Machine Learning atteignent un très haut niveau de performance, mais la qualité des données d'apprentissage disponibles n'est pas au même niveau. La qualité des données est essentielle dans les problèmes modernes. Les ingénieurs en apprentissage automatique passent la majorité de leur temps non pas à améliorer des algorithmes, mais à résoudre des problèmes liés aux données (par exemple, collecte de données, nettoyage, synthèse de données). C'est le principal axe d'amélioration de la qualité des prédictions.

Nous soutenons que ce problème est aussi très présent dans le domaine de l'optimisation de code. Nous sommes confrontés à un manque de données. Pire encore, ces données ne sont pas nécesssairement représentatives des problèmes que nous ciblons. Il existe deux façons de résoudre ce problème : la génération de données synthétiques et la collecte de données à partir de sources publiques. Nous nous sommes concentrés sur la première approche. Cela signifie que nous visons une génération des données d'apprentissage à partir de caractéristiques que nous considérons significatives. Nous présentons notre générateur de code qui permet de cibler ces concepts de performance, la localité des données et le parallélisme.

Introduction

Nowadays, Machine Learning algorithms reach a very high level of performance, but the quality of available training data is not at the same level. Data quality is becoming a bottleneck in modern problems. Machine Learning engineers spend the majority of their time not improving algorithms but solving data-related problems (e.g. data collection, cleaning, data synthesis). This is the main direction of improvement for the quality of predictions.

We argue that this problem is very pronounced in the code optimization domain. We are facing a lack of data. Even worse, if this data is not representative of many of the issues we target. There are two possible ways to solve this problem: synthetic

data generation and data collection from open sources (e.g. GitHub). We focused on the first approach. This means that we would like to generate data with certain characteristics that we consider meaningful. We present our code generator which can target such performance concepts, data locality, and parallelism.

The results of this chapter were obtained with the help of Justyna Zawalska and Maryna Savchenko during their internship at MINES ParisTech. Justyna and Maryna were responsible of the first version of the code generator (generation of the instructions and loop bounds computation). My contribution relates to array initialization and declaration, code infrastructure, and the determination of the appropriate high-level concepts for the generator.

3.1 Motivation

Benchmarking is an essential part of testing code optimization techniques and models. In case of obtaining unsatisfactory results, all the assumptions of the study may be rejected. Therefore, benchmark programs must be the most indicative as possible and reflect similar characteristics to those on which we want to apply our techniques.

Our objective is to have benchmarks adapted to the evaluation of automatic code transformations. These transformations (presented in Chapter 2) make it possible to improve program characteristics such as the spatial and temporal locality of data accesses, the loop iteration order, and the potential parallelism. The execution time gain of a transformation depends on the transformation parameters that have to be defined for each kernel and the target architecture.

Chapter 2 concludes that there are much fewer data in the code optimization domain than in fields where ML shows state-of-the-art performance. There is not enough training data to properly cover the feature space of parameters for complex transformations such as loop tiling, loop unrolling, loop interchange, etc. As presented in Chapter 2 different transformations have different feature spaces from a machine-learning perspective.

One training set could capture better features for one transformation, another - for another. It becomes challenging to create a universal training set. Thus, synthetic code generation is a crucial component of the proper use of Machine Learning for code optimization.

In this chapter, our work proposes a solution to these problems. We introduce a methodology to generate a representative benchmark that captures many computation patterns. We present a code generator that can automatically create synthetic data. Our code generator uses information like array sizes, data dependencies, loop index order, and data access functions as a high-level specification of the generated code. We use a domain-specific language (DSL) to easily manipulate these concepts and generate code in a very parametric and flexible way. This data generation

approach enables the creation of highly representative training sets for program optimization in a machine-learning context. Moreover, we are able to generate our codes for different benchmark distribution styles.

This chapter is structured as follows. Subsection 3.2 introduces the context of our work, pointing out the guidelines we used in our code generator. Sections 3.3 - 3.7 introduce the main constructing blocks of the generation process. Subsection 3.8 gives a detailed description of our domain-specific language.

3.2 Guidelines for the generator

We follow three concepts as guidelines for building an automatic code generator of programs used by machine learning techniques to predict efficient transformation parameters.

First, Machine learning algorithms build prediction models based on training data. The key idea is that the model should capture meaningful patterns in training data and should be able to generalize them for arbitrary input. We consider the training set to be "good", if the model trained on this set is able to generalize data from the test set. However, if the training and the test sets are from different distributions, we can expect poor performance. The model will capture patterns during the training phase that may not be significant during the test phase. However, we do not want to mimic the test set itself. We use such high-level concepts for the code generator to be able to express all the different patterns for the target transformations. The global overview of each building block of this pipeline is given in Section 3.3.

The second important concept is the amount of available data. From a machine learning perspective, the more data available the better, and the more insights we can obtain. However, data labeling can be a very time-consuming process (e.g. computation of speedups for considered loop unrolling factors). Therefore, time also constrains the size of the training set. It is important to build a representative benchmark of the right size.

We solve this problem by using active learning methods. The idea is that not all points in a training set have the same impacts on model training and its final performance. Our goal is to select only the most representative samples from the training set to match all the time constraints. This issue is discussed in detail in Chapter 4.

The third concept concerns code characteristics. Because loop nests are often the time-consuming computation parts in programs, our study focuses on the optimizations conventionally used by compilers such as loop permutation, unrolling, tiling, etc. To optimize their execution time, it is necessary to take into account the spatial and temporal locality of the data accesses and data dependencies to extract the potential parallelism and apply the transformations only when they are legal.

Therefore, the important information to consider is the number of different arrays referenced in the kernel, array sizes, array access functions, data dependency types, the execution order of iterations, and the iteration domain and its shape. It also seems to be a reasonable choice to express any type of access pattern. Section 3.3 presents how this code specification is exploited in our code generator.

3.3 Code Generator Design

In this subsection, we introduce the main components of our automatic generator of C code. For each of them, we precise the type of code generated. Figure 3.1 highlights its main building blocks.

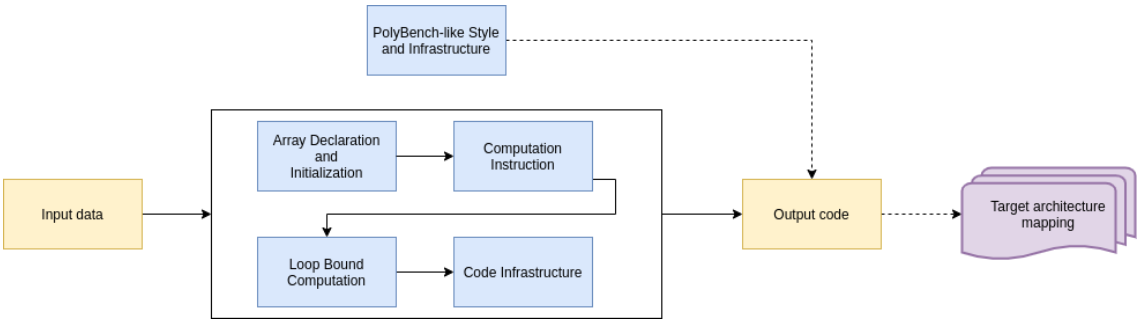

Figure 3.1: Pipeline of the code generator

3.3.1 Output Code and Input Data

The objective of the generator is to automatically produce a code written in C that respects the following hypothesis:

- it is compliant with C language,
- it is correct. The code must not produce any runtime errors such as out-of-bound memory access, etc;
- it meets the code criteria specified by the user in the DSL sample;
- the number of instructions in the generated code that matches the input specification should be minimized;
- it includes the necessary infrastructure to perform performance tests such as header files, *directives/pragmas* and calls to timing reporting functions;
- it can be easily compiled and executed. For instance, the arrays are properly initialized in the code.

However, these requirements do not have much in common with the high-level criteria we want to use for code optimization. These are the number of arrays and their sizes (memory pressure), data dependencies (management of the ordering of iterations and instructions), an order of the loop indices, and array access functions (pressure on spatial and temporal locality). These concepts allow us to explore the legality of potential transformations and optimize the code. It seemed logical to separate the high-level requirements for the code and the implementation of the code generator. We have therefore developed a Data Specific Language (DSL) allowing to define the high-level criteria and semantics of the targeted generated codes. This DSL is introduced in Subsection 3.8. The information described in an instance of the DSL is communicated in a JSON format and given as input to the generator.

We apply the building blocks of Figure 3.1 after parsing the input. After these steps, we obtain the first version of our code. Then there is the option to process the generated code to a PolyBench-like style or another benchmark distribution style.

Note that it is also easy to produce code for an architecture other than CPU such as GPU. Indeed, we can add an additional pass translating our kernels into dedicated GPU code as is the case in the PIPS framework [33].

3.3.2 Array Declaration and Initialization

The code generator takes array sizes from the input file (DSL description) and dynamically or statically (depending on the chosen option) allocates the requested arrays. The code generator may choose array sizes automatically if the user uses the PolyBench-like style of kernels. For instance, the EXTRALARGE_DATASET directive indicates that arrays should not fit the L3 cache. Then the code generator will perform these computations respecting the directive and ignoring any numeric input. The following default options are available for array initialization in the same manner as in PolyBench 4.2:

- **EXTRALARGE**. Around 120MB of memory.
- **LARGE**. Around 32MB of memory. Arrays should not fit within L3 cache.
- **MEDIUM**. Around 1MB of memory. Arrays should not fit within L2 cache but may fit within the L3 cache.
- **SMALL**. Around 128KB of memory. Arrays should not fit within L1 cache but may fit within the L2 cache.
- **MINI**. Around 12KB of memory. Arrays should fit within L1 cache.

3.3.3 The shape of the initialized arrays

Our code generator allows the generation of arrays of arbitrary shapes in manual mode. For example, hyper rectangular shape A[35][10][4] is possible. However, predefined options for array size generate only cubic/square shapes of the arrays with powers of two sizes (e.g. A[16][16][16]).

3.3.4 Array size selection in automatic mode

The main idea is that all requested arrays should take about "the same amount" of memory. We introduce the logic of array size selection in automatic mode with the example below.

Suppose the user asks to generate one 2-D array and two 3-D arrays with the EXTRALARGE option. The type of each array is assumed to be `integer`.

Let x be the size of each dimension for the 2-D array (it has a square shape), y and z are the sizes for the two 3-D arrays. The equation below implies that the sum of the sizes of all arrays should be approximately equal to the size denoted as an EXTRALARGE cache:

4 bytes $\times$ $(x^2 + y^3 + z^3) \approx 1.2 * 10^8$ bytes $\Longrightarrow$ $x^2 + y^3 + z^3 \approx 3 * 10^7$ bytes.

We assume that each array should take about "the same amount" of memory. It means that x^2, respectively y^3 and z^3, should be close to 10^7.

Finding the closet powers of two that respect these constraints, we found that $x = 2048$, $y = z = 256$.

Hence, we generate A[2048][2048], B[256][256][256], C[256][256][256] and these arrays occupy 150 MB of memory which is quite close to the desired conditions.

3.4 Computation Instructions

This component generates the computation instructions included in the loop nest. Each instruction is composed of a reference to a write array and several (at least one) to read arrays. The array access functions are either explicitly given by the user or defined by the generator that respects the data dependencies which have been expressed in the DSL sample.

The main challenge is that when we generate code directly based on what the user has requested, more instructions than necessary can be produced. Indeed, it is possible to generate an instruction for each requested dependence or to generate a single instruction carrying all the dependencies. The goal is to minimize them. We use simple heuristics.

The main idea is to check the set of already generated dependencies after each generated reference. It helps to control the number of dependencies at each step and to take into account already generated ones to optimize the number of generated instructions.

3.5 Loop Bound Computation

The generated code should be correct. To avoid out-of-bound memory accesses to array elements, the generator computes the largest computation iteration domain according to the array declarations and the array access functions. We use linear-programming techniques to compute correct bounds. For constant (uniform) dependencies, we generate numerical values for loop bounds. Non-constant (non-uniform) dependencies require the usage of macros that calculate the *MIN* and *MAX* values of several linear expressions (e.g. Listing 1.2 l.3) in our approach.

Our generator generates kernels made up only of perfectly nested loops. This assumption comes from the loop transformations that we want to use and which only apply to this type of loop nest.

3.6 Code Infrastructure

This component consists of adding all the infrastructure necessary for the execution of a stand-alone C program with time-reporting functions. It includes:

- header files (header from line 9, Listing 3.1)
- variable declaration (lines 70-78)
- initialization, array allocation, and deallocation (lines 11-29, 76-78, 82-84)
- calls to time reporting functions (lines 88, 98, 99)
- pragmas and directive insertion (lines 1-9, 90, 97)
- adjustment of array sizes according to the requested cache size (Section 3.4.4)

This phase is classic and systematic. It is the same whatever the input, except for the variable declarations and array allocation, which considers the given code specifications. It concludes the generation of the basic style code.

3.7 PolyBench-like Style and Infrastructure

PolyBench 4.2 is considered one of the most famous benchmarks in the code optimization domain. We propose a processing pass that transfers our generated kernel to a PolyBench-like style. This pass adds the PolyBench header files, uses its array allocation and deallocation functions, includes calls to execution time reporting, and some *pragmas* marking kernels. Then the processed code could be considered as just one more computational kernel from the PolyBench suite. It allows users to use the known code style and to use the convenient infrastructure of this benchmark.

The example of the generated code is shown on the listing 3.1

Listing 3.1: PolyBench style generated code

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <string.h>
4  #include <math.h>
5  #include <polybench.h>
6  #include <stdio.h>
7  # include < time.h >
8  # include < stdlib.h >
9  #include \"16136950084198.h\"
10
11   static void init_array(int xa, int ya, DATA_TYPE POLYBENCH_2D(A, xA, yA, xa, ya),
         int xb, int yb, DATA_TYPE POLYBENCH_2D(B, xB, yB, xb, yb), int xc, int yc,
         DATA_TYPE POLYBENCH_2D(C, xC, yC, xc, yc)) {
12     srand(time(NULL));
13     int i, j, k, l;
14     for (i = 0; i < xa; i++) {
15       for (j = 0; j < ya; j++) {
16         A[i][j] = rand() % 50;;
17       }
18     }
19     for (i = 0; i < xb; i++) {
20       for (j = 0; j < yb; j++) {
21         B[i][j] = rand() % 50;;
22       }
23     }
24     for (i = 0; i < xc; i++) {
25       for (j = 0; j < yc; j++) {
26         C[i][j] = rand() % 50;;
27       }
28     }
29   }
30 static void print_array(int xa, int ya, DATA_TYPE POLYBENCH_2D(A, xA, yA, xa, ya),
       int xb, int yb, DATA_TYPE POLYBENCH_2D(B, xB, yB, xb, yb), int xc, int yc,
       DATA_TYPE POLYBENCH_2D(C, xC, yC, xc, yc)) {
31   int i, j, k, l;
32   POLYBENCH_DUMP_START;
33   POLYBENCH_DUMP_BEGIN("A");
34   POLYBENCH_DUMP_START;
35   POLYBENCH_DUMP_BEGIN("A");
36   for (i = 0; i < xa; i++) {
37     for (j = 0; j < ya; j++) {
38       fprintf(POLYBENCH_DUMP_TARGET, "\n");
39       fprintf(POLYBENCH_DUMP_TARGET, DATA_PRINTF_MODIFIER, A[i][j]);
40     }
41   }
42   POLYBENCH_DUMP_END("A");
43   POLYBENCH_DUMP_FINISH;
44   POLYBENCH_DUMP_START;
45   POLYBENCH_DUMP_BEGIN("B");
46   POLYBENCH_DUMP_START;
47   POLYBENCH_DUMP_BEGIN("B");
48   for (i = 0; i < xb; i++) {
49     for (j = 0; j < yb; j++) {
50       fprintf(POLYBENCH_DUMP_TARGET, "\n");
51       fprintf(POLYBENCH_DUMP_TARGET, DATA_PRINTF_MODIFIER, B[i][j]);
52     }
53   }
54   POLYBENCH_DUMP_END("B");
55   POLYBENCH_DUMP_FINISH;
56   POLYBENCH_DUMP_START;
57   POLYBENCH_DUMP_BEGIN("C");
58   POLYBENCH_DUMP_START;
59   POLYBENCH_DUMP_BEGIN("C");
60   for (i = 0; i < xc; i++) {
61     for (j = 0; j < yc; j++) {
62       fprintf(POLYBENCH_DUMP_TARGET, "\n");
63       fprintf(POLYBENCH_DUMP_TARGET, DATA_PRINTF_MODIFIER, C[i][j]);
64     }
65   }
66   POLYBENCH_DUMP_END("C");
67   POLYBENCH_DUMP_FINISH;
68 }
69 int main(int argc, char ** argv) {
```

```
70    int xa = xA;
71    int ya = yA;
72    int xb = xB;
73    int yb = yB;
74    int xc = xC;
75    int yc = yC;
76    POLYBENCH_2D_ARRAY_DECL(A, DATA_TYPE, xA, yA, xa, ya);
77    POLYBENCH_2D_ARRAY_DECL(B, DATA_TYPE, xB, yB, xb, yb);
78    POLYBENCH_2D_ARRAY_DECL(C, DATA_TYPE, xC, yC, xc, yc);
79    init_array(xa, ya, POLYBENCH_ARRAY(A), xb, yb, POLYBENCH_ARRAY(B), xc, yc,
          POLYBENCH_ARRAY(C));
80    kernel_16136950084198(xa, ya, POLYBENCH_ARRAY(A), xb, yb, POLYBENCH_ARRAY(B), xc,
          yc, POLYBENCH_ARRAY(C));
81    polybench_prevent_dce(print_array(xa, ya, POLYBENCH_ARRAY(A), xb, yb,
          POLYBENCH_ARRAY(B), xc, yc, POLYBENCH_ARRAY(C)));
82    POLYBENCH_FREE_ARRAY(A);
83    POLYBENCH_FREE_ARRAY(B);
84    POLYBENCH_FREE_ARRAY(C);
85    return 0;
86  }
87  void kernel_16136950084198(int xa, int ya, DATA_TYPE POLYBENCH_2D(A, xA, yA, xa, ya)
        , int xb, int yb, DATA_TYPE POLYBENCH_2D(B, xB, yB, xb, yb), int xc, int yc,
        DATA_TYPE POLYBENCH_2D(C, xC, yC, xc, yc)) {
88    polybench_start_instruments;
89    int i, j, k, l;
90    #pragma scop
91    tiling_3D: for (i = 0; i < 1024; i++)
92      tiling_2D: for (j = 0; j < 1024; j++)
93        for (k = 0; k < 1024; k++) {
94          A[i][j] = A[i][j] - B[j][k] - C[j][k] + 72;
95
96        }
97    #pragma endscop
98    polybench_stop_instruments;
99    polybench_print_instruments;
100 }
```

3.8 Domain-Specific Language

Our domain-specific language makes it easy to express the specifications of the code we want to generate. The code generator is then in charge of producing a code consistent with the specifications and guaranteeing the correction of the generated code. Moreover, it is much easier to generate a JSON file (the format of our DSL) than the entire code.

3.8.1 Grammar

The subsection below presents the grammar of our DSL.

```
<alpha> ::= 'a' .. 'z' | 'A' .. 'Z'
<digit_positif> :: 1|2|3|4|5|6|7|8|9
<digit> ::= 0|<digit_positif>
<integer> ::= <digit_positif> <digit>*

<array_name> ::= <alpha>+ (<digit>* |<alpha>*)
<dim_name>::= <alpha>+ (<digit>* |<alpha>*)
<dependence_name>::= <alpha>+ (<digit>* |<alpha>*)
```

```
<array_type> ::= type :  (float | double | int) ,

#definition of symbolic size  ex: x:32,y:64
<size> ::= <dim_name> : <integer>
#sizes :  x:32,y:64
<sizes> ::=  sizes : { (<size> ,)* <size> },

<distance> ::= <dependence_name> : <integer>
#distances:  d1:2,d2:6
<distances> ::=  distances : { ((<distance> ,)* <distance>) } ,

# x | 32
<array_dimension> ::= <dim_name> | <integer>

#init_with : random
<array_init_value> ::=  init_with : (zeros | ones | random) ,

#loop_nest_level : 3
<loop_nest_level> ::= loop_nest_level : <digit_positif> ,

# A[x,y] | A[32,64]
<array> ::= <array_name> [ (<array_dimension> , )*  <array_dimension> ]
<array_declaration> ::= array_declaration : { (<array> , )* <array> } ,

# index_permutation has as many digits as the number of loops.
#The order corresponds to the array access function of the elements written.
#It will be used for the array access function of the read array elements,
#  in case of constant dependencies
<index_permutation> ::= index_permutation : [ (<digit> , )* <digit> ]

<constant_dim_dependence> ::= <dependence_name> | <integer>
# constant_dependence has as many constant_dim_dependence as the
#  array dimension.
#ex: constant_dependencies :[d1,d2] | [-2,1]
<constant_dependence> ::=  [ (<constant_dim_dependence> , )*
                             <constant_dim_dependence> ] ,

# there can be several references to the same array with different
#  constant dependencies.
#Here the array name is the referenced array of the instruction.
<constant_dependencies> ::= constant_dependencies :
                        { (<constant_dependence> , )*
                                <constant_dependence> } ,
```

```
<referenced_array> ::= array_name : <array_name> ,
# array_dim_access_function has as many digits as the loop indices +1.
#The last dimension corresponds to the constant offset.
#It corresponds to one row of the array_access_function matrix.
<array_dim_access_function> ::= [ (<digit> ,)* <digit> ] ,

# as many array_dim_access_function as the array dimension.
#Here the array name is the referenced array of the additional computation.
<array_access_function> ::= array_access_function :
                          [ <array_dim_access_function> , )*
                            <array_dim_access_function> ] ,

# there can be additional computations with several references to the same
#  array with different array access functions
# Here, the elements of the referenced array are read
<additional_computations> ::=  additional_computations :
                     {
                     ({ <referenced_array> <array_access_function> } ,)*
                      { <referenced_array> <array_access_function> }
                     } ,
# here, the elements of the referenced array are written
<instruction> ::= { <referenced_array>
                     <index_permutation> <constant_dependencies>
                     <additional_computations>}

# there can be several instructions in the loop nest body.
<instructions> ::= instructions : { (<instruction> ,)* <instruction> } ,

<kernel> ::= <sizes> <distances>
             <array_type> <array_init_value> <loop_nest_level>
             <array_declaration> <instructions>
```

3.8.2 High-Level Specification

We use the number of arrays, data dependencies, loop index order, and array access functions as high-level code specifications for our code. The main motivation to use them is that these concepts have a critical impact on the performance criteria of many code transformations we want to apply. For instance, the number of arrays referenced in the kernel and their layout has an impact on memory pressure. The data dependencies define the legality of the transformations that can be applied. Loop interchange, loop distribution, and loop tiling are not always legal whereas loop unrolling can always be applied. They impact the iteration and instruction

orders as the potential parallelism that can be extracted from the computations. The loop index order and array access functions define the spatial and temporal locality of the computations that we seek to increase in order to reduce the pressure on caches.

3.8.3 DSL Concept

In this subsection, we introduce briefly the code specification that is given as input to generate the code. An example of a DSL sample is represented in Listing 1.

The main structures of our DSL, with their default values, follow:

- *Array name, type, and size*: array type could be float, double, or int. Array size is numeric or symbolic depending on variables defined in a previous variable-declaration code section.
- *Array initialization value* could be one, zero, or random.
- *Loop nest level* defines the number of nested loops.

For each *instruction* included in the loop nest, the written array reference is set first. Then, in the case of dependencies, for each one, a read reference to the same array verifying the dependency characteristics is added to the instruction. The *instruction-block* components are:

- the *Array name* which precises the array to be referenced in the written part, and in the reading part if constant dependencies are requested,
- the *Index permutation* that precises the permutation of loop indices to consider in the array access function, and
- the *Constant dependencies* (if any) which are expressed as dependence vectors whose size depends on the array dimension (e.g. Listing 3.2 l.5). There could be a list of dependencies.

The previous instruction block enables only to the addition of constant dependencies, so *additional-computation blocks* could be added for each instruction. These blocks are used to add references to new arrays or to new references to the same array with more complex linear access functions. This block is composed with:

- *Array name* precises the additional array to be referenced in the reading part of the instruction.
- *Array access function* is expressed as a numerical matrix `(nxm)` (e.g. Listing 3.2 l.7). `m` is the size of the loop index vector plus 1 component for a potential constant value. `n` is the array dimension. This access function may imply complex dependencies with the other references.

Note that additional dependencies can be introduced via successive instructions. They can come from the complex array access functions given in the DSL sample. Our generator therefore can only guarantee that at least the set of dependencies requested by the user is included in the nest of loops.

Listing 3.2 illustrates an example of the input file, giving the requested code specification using our DSL. Listing 3.3 expresses the computation kernel generated by our generator.

Listing 3.2: Input JSON file

```
1 [{"array_sizes": {"xA": 64, "yA": 32, "zA": 128}, "type": "int", "init_with": "
      random", "loop_nest_level": 3,
2     "arrays": ["A[xA,yA,zA]", "B[256,256]"],
3     "instructions": [{"array_name": "A",
4         "index_permutation": "(1,0,2)",
5         "dependencies": {"distance": "[(1,2,3)]"},
6         "additional_computation": [{"array_name": "B",
7             "array_access_function": "[[0,2,0,8], [1,1,1,8]]"}]}]}]
```

Listing 3.3: Generated code

```
1 int A[64][32][128],B[256][256];
2   ....
3 for (int i = 0; i < 30; i++)
4 for (int j = 0; j < 63; j++)
5 for (int k = max(-i-j-8, 0); k < min(248-i-j, 125); k++)
6     A[j][i][k]=A[j+1][i+2][k+3]+B[2*j+8][i+j+k+8];
```

3.9 Conclusion

This chapter presents a tool for efficiently generating benchmarks for the code optimization domain. It includes

- An automatic code generator enabling to imitate of some existing benchmark styles
- An associated DSL handling the high-level specification of the code to be generated

The concepts used to specify the generated code are simple: the number of arrays, loop index order, array access functions, and data dependencies, but enough to create highly representative codes for code optimizations. The state-of-the-art has proven these concepts have a strong impact on the performance criteria of the code transformations commonly applied to kernels for efficient parallel execution. Hence, our generator can produce representative code for training Machine Learning models in the context of the optimization problems mentioned above. Machine Learning algorithms would be not able to draw reasonable conclusions and make fair generalizations without representative data. This synthetic code generator solves this problem.

The number of potential codes that we can generate is not limited, but bound by time constraints for data labeling. In our experiments, we create several thousands of programs for each task. The aspects of proper generation strategy and how well it covers a feature space will be discussed in the next chapter.

Potential future work related to this generator concerns the extension of our benchmark criteria with arbitrary loop nests, not necessarily perfectly nested. This will extend the type of code transformations that can be tested.

Our generator can be extended to many programming languages (not only C) because the principal concepts we used are language-agnostic. It only requires some changes in the syntax and code routines for successful translation into the target language.

Chapter 4

Data augmentation and optimal experimental design

Résumé

Ce chapitre fournit un pipeline sur la façon d'organiser l'ensemble des données nécessaires à la phase d'apprentissage et engendrer une stratégie de génération efficace en terme de performance du modèle. Du point de vue du ML, plus il y a de données disponibles, mieux c'est et plus nous pouvons obtenir d'information. Cependant, l'étiquetage des données peut être un processus très long. Par conséquent, le temps limite également la taille de l'ensemble d'apprentissage. Il est important de construire un benchmark représentatif avec une taille raisonnable. Aussi, l'optimisation de la phase d'apprentissage permet de réduire la consommation d'énergie nécessaire à l'utilisation des techniques ML dont on sait qu'elle est importante.

Nous introduisons des techniques d'apprentissage *actif* pour former l'ensemble d'apprentissage et proposons un pipeline de l'ensemble du processus d'optimisation, utilisant ces données, pour la sélection efficace de la taille des tuiles 3D. Nous évaluons l'impact de la nouvelle stratégie et la comparons aux pipelines classiques d'apprentissage *passif*.

Dans cette section, nous évaluons notre méthodologie complète sur la transformation de tuilage, mais les idées principales utilisées pour obtenir toutes les données d'apprentissage sont applicables à d'autres transformations.

Introduction

The previous chapter presents the generation and underlying concepts of code generation, but it says nothing about how it should be used to generate data to solve a concrete problem.

Moreover, there are many ways to define the generation strategy, but how to select the optimal one? This section tries to answer these questions.

4.1 Motivation

Chapter 3 presents the high-level parameters we have chosen to characterize our generated code and its associated DSL to express them. We need to generate many files written in our DSL, pass them to the code generator, collect outputs and thus obtain a training set. The way how we generate the input files is called the generating strategy.

There is a bunch of requirements for the generating strategy. On one hand, the input files should capture meaningful properties of the problem posed. On the other hand, these properties should have a great diversity. Intuitively it means that we would like to explore all possible values of the properties that we are exploring. We can draw reasonable conclusions based on that.

This chapter provides a pipeline on how to organize the training set/choose the generation strategy in the best way in terms of model performance. From an ML perspective, the more data available the better, and the more insights we can obtain. However, data labeling can be a very time-consuming process. Therefore, the time also constrains the size of the training set. It is important to build a representative benchmark with a reasonable size. Also, optimization of the learning phase helps to reduce the energy consumption necessary for the use of ML techniques which is known to be important.

We introduce Active Learning techniques to form the training set and show the pipeline of the whole process for the 3-D tile size selection problem. We evaluate the impact of the new strategy and compare it to the classical Passive Learning pipeline.

In this section, we evaluate our complete methodology on the tiling transformation, but the main ideas used to obtain all the training data are applicable to other transformations.

This section is organized as follows. Section 4.2 introduces the ML pipeline of our experiments and highlights metrics, and concrete ML models. Section 4.3 presents the basic steps of the Active Learning pipeline and shows the results in the context of different ML models and different feature spaces. Section 4.4 draws the conclusion about the applicability of our methods.

4.2 Machine Learning modeling

Our objective is to show that our approach can accelerate the techniques of code optimization using ML in the context of a given feature space. In this section, we describe the pipeline that we would like to accelerate using Active Learning techniques. By accelerating, we mean the need for less training data to achieve good performance (in terms of obtained speedups after our predictions) compare to the original pipeline. Alternatively, we can achieve the same level of performance

using less amount of data compared to the original pipeline.

4.2.1 Machine Learning pipeline

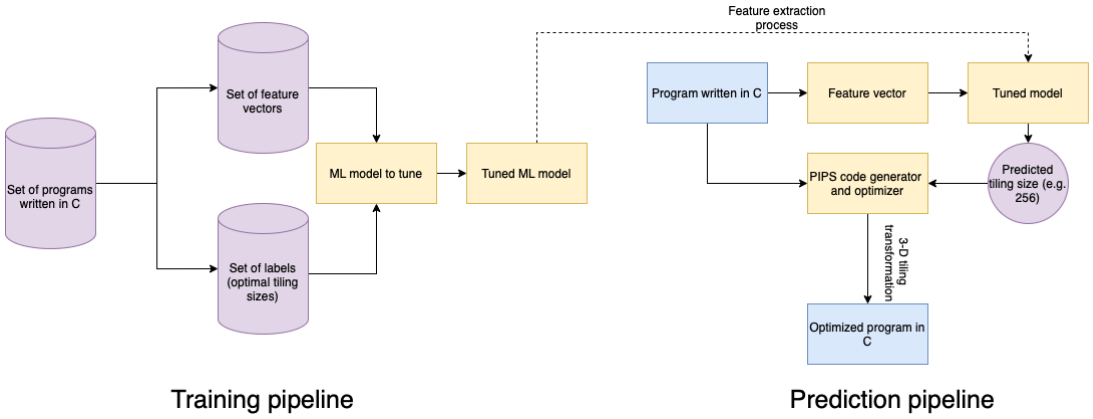

Figure 4.1: Training and prediction pipeline

We investigate the problem of loop tiling size prediction for 3-D cubic tiles to validate our Machine Learning model. We consider tile sizes from 2 to 512 for the experiments and predictions. As features, we take the code characteristics proposed by a) Yuku et al. [1], b) Liu et al. [45] and c) one-hot encoding [8] of array references.

We consider this problem a regression problem. The model takes the features mentioned above as input and predicts the values of the tile sizes in the real domain. A heuristic of rounding the tile size to the nearest divisor of the loop bound could be applied and was used in our experiments. It impacts a lot on the performance, this heuristic is discussed in detail in Section 5.2.2.

Then we generate the code based on the predicted tile size. The training and prediction pipelines are shown in Figure 4.1. The main idea is that a set of feature vectors and corresponding labels are used to tune the ML model. Then this model is used to make the predictions (tile size) for a given code. Predicted tile size is used for the code generation step. After we get the optimized code.

Note that once the training pipeline phase is complete, the parameters of the prediction model are fixed. It is possible to predict with this tuned model the best parameters of the program transformation we want to apply in one shot. This model can then be integrated into a compiler.

A program Autotuner, such as LOCUS [32], typically uses several techniques to traverse a solution space and find an optimal version of a program. But the time needed to reach this solution for a program is not comparable to that of a single one-shot prediction of a tuned machine learning model. For this reason, we use the result of the Autotuner only as a reference of the optimal version to compare with our best-predicted version of the program.

4.2.2 Machine Learning models

Machine Learning models could be classified as linear ones and nonlinears ones. The main difference is that linear models assume the linear relationship between input features and the output. We argue that non-linear machine learning models are more appropriate for our problem. There exist cases where the model must solve the following dilemma: to optimize the level of parallelism or to optimize data locality? Maximization of one factor could negatively impact on the other factor. It can be seen as a decision tree. This is the main prerequisite for using nonlinear models for this task. Random Forest regressor [93] showed the best results in terms of metrics considered in our experiments.

4.2.3 Metrics

The mean squared error (MSE) or mean average error (MAE) losses were used as Loss Functions for regression.

$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y_i} - y_i)^2$, where y_i is the ground-truth value of the optimal tile size of the i-th data sample, and $\hat{y_i}$ was predicted by our ML model. We use this metric for ML modeling since optimal tile sizes are distributed near the same neighborhood, and we want to penalize our model if it predicts tile sizes that are far from the global optimum.

$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y_i}|$

It handles local minimum situations better than MSE (mean square error), but penalizes less for the predictions that are far from the global minimum.

These cost functions have several drawbacks. They do not provide explicit information about our target goal - fast code execution. The losses provide no information to the programmer on how the generated code would perform in terms of execution time. Moreover, they do not provide insights into architecture parallelism and the profitability that we can gain from the transformation. The illustration of the previous statement can be observed in Figures 5.5 and 5.6 in the next chapter. The y-axis (execution time) gives insights into the profitability of the tiling transformation for given tile sizes.

That is why we introduce the second-step metric showing how far we are from the most efficient generated code. We use the following relative speedup metric.

$RS_i = \frac{speedup(\hat{y_i})}{speedup(y*_i)}$, where $speedup(\hat{y_i})$ gives the speedup obtained after tiling the code with the predicted parameter. And $speedup(y*_i)$ gives the speedup found by the Autotuner. An average relative speedup can be computed with $RS = \frac{1}{n}\sum_{i=1}^{n} RS_i$.

The drawbacks of this function are that it is very sensitive to outliers. RS of a tile in the same neighborhood may be different due to factors that are not possible to take into account using existing feature spaces. For instance, information about divisors of the iteration space is not included in the mentioned feature spaces. Imagine, we

have a cubic iteration space with 1024 iterations per loop, tile sizes 32 and 33 would be very similar from the point of view of MAE and MSE metrics. But RS could be significantly different due to the non-uniform balancing factor of computations between threads.

Moreover, it does not have derivatives; it is a piecewise-defined function. Hence, it is not applicable to be used for the training of many ML models. Thus, each metric is more appropriate for the stage where it is used. The combination of both provides a more correct way to navigate the training process and evaluate the results.

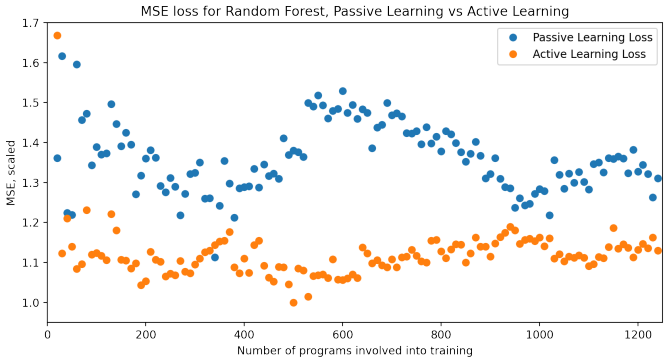

Figure 4.2: Yuki et al., MSE on validation set

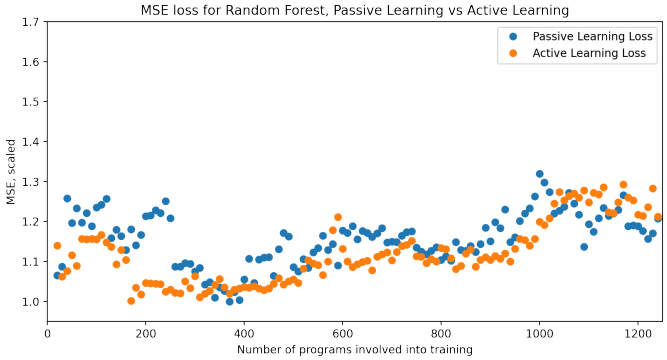

Figure 4.3: Liu et al., MSE on validation set

4.3 Active Learning

Data labeling is the calculation of a value of the target variable for a data sample of the training set. This step can be very time-consuming in traditional ML pipelines. It makes sense to find a trade-off between how quickly we collect data and the

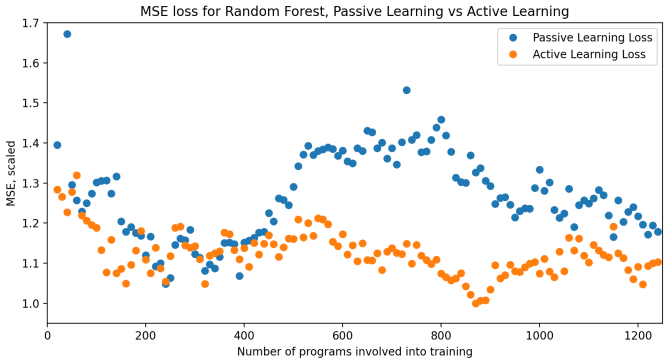

Figure 4.4: One-hot encoding, MSE on validation set

accuracy of the final model. The issue of optimal experimental design arises. How to construct our training set to get the maximal possible gain? The techniques used in the active learning domain seem a promising direction to answer this question. A detailed overview of Active Learning methods was presented in Chapter 1.

4.3.1 Experimental statement

The learning process goes more efficiently for data generated with active learning, especially when we do not have expert knowledge about the given domain. We expose this statement to demonstrate the applicability of active learning techniques for the code optimization domain. While any handwritten strategy brings some bias to data, especially in case the expert knows which benchmarks will be used for testing, active learning appears to be the approach to facilitate representative data generation without introducing significant bias.

The pipeline for training the model is shown in Figure 4.1. The set of C programs, used as inputs, could be obtained using naive sampling (passive learning) or more sophisticated strategies (active learning). The quality of the predictions and the speed of convergence of the models depends on this set.

4.3.2 Generating strategy

Training, test, and validation sets are required to properly tune the model and evaluate its applicability for real problems.

We train the ML model on the training set. The validation set is needed to evaluate the model performance (MSE) and determine its parameters based on that. The test set represents real-world data. We use our generator to sample data for the training and validation sets. We use a simple generation strategy that does not require any expert knowledge about the feature space for the loop tile size prediction.

The most important parameters that we vary are the existence of data dependencies, the number of statements and arrays involved in the computations, and loop index permutations.

10000 kernels were randomly sampled to obtain a pool of not annotated data. Then, the Active Learning phase chooses 1250 most suitable kernels (training set for Active Learning). We do the labeling of the chosen samples and train the model on them. 300 kernels were sampled (from the same distribution as 10k kernels) and labeled for the validation set. There are not involved in model training but are used as intermediate evaluation of the performance.

9 well-known computational kernels were taken to form our test set. We compute the average relative speedup for them after tiling to assess the quality of the generated code.

Statistical characteristics

We provide mean, variance, and standard deviation for the features of programs chosen by Active and Passive learning. They are shown in Table 1 and Table 2. These features introduce the count of array references that match different patterns, 6 different patterns were provided. The statistical characteristics show how many these patterns we can observe for the average kernel in the training sets and the variance of these counts.

Active Learning tends to suggest data with higher variance and larger absolute values of features. It means that Active Learning takes - more diverse programs in terms of these counts, - programs with more array references inside of any pattern.

	Invariant Read	Locality Read	No locality Read	Invariant Write	Locality Write	No locality Write
Mean	3.30	3.25	3.5	0.98	0.95	1.01
Std. Dev	2	1.96	2.05	0.75	0.76	0.74
Variance	4.00	3.86	4.22	0.56	0.57	0.55

Table 4.1: Statistical characteristics, Active Learning candidates

	Invariant Read	Locality Read	No locality Read	Invariant Write	Locality Write	No locality Write
Mean	3.04	2.98	3.04	0.92	0.91	0.92
Std. Dev	1.79	1.8	1.83	0.71	0.70	0.71
Variance	3.23	3.24	3.37	0.51	0.49	0.51

Table 4.2: Statistical characteristics, whole set of candidates

4.3.3 Passive Learning Training Set

We sample the same amount (1250) of kernels with random sampling to compare the performance of the model trained on the training set obtained with Active Learning.

These 1250 kernels were chosen randomly also from the 10000 samples of not labeled data.

We investigate the possibility of Active Learning to shift the distribution to meaningful patterns in a given distribution.

4.3.4 Data labelling

The data labeling process begins after the choice of the kernels of the training set. This process is very time-consuming. For each kernel, we generate about 300 code variants (tiled codes with different tile sizes) and execute them to assign labels for the regression problem. The time to propose a variant plus its execution time varies from 0.1s to 50s, the median value is about 2s.

The whole process is equal to the number of repetitions $\times$ number of variants $\times$ number of kernels $\times$ (the time to generate a variant + the time to execute the variant). For us, it took around 30 days to label all the required data. This estimation illustrates that the data labeling process time can be significant. When time is limited, data quality becomes crucial. This is the main motivation for using the Active Learning approach.

4.3.5 Experimental results

The objective of this chapter is not to find the best ML algorithm to perform tiling but to propose efficient techniques to automatically generate benchmarks suitable for the evaluation of code transformations and used as input for the ML techniques. In this section, we compare the results obtained with the Active and Passive Learning approaches.

The experiments were run on Intel® Core™ i7-8650U 4C/4T @1.90GHz with capacity caches of L1: 32KB, L2: 256KB, L3: 8192KB, and 32GB DDR4 DIMM RAM, Phys. cores: 4, Compiler: GCC 5.4.0, Number of Threads: 4, Opt. level: -O3

Loss on the validation set

Figures 4.2, 4.3, 4.4 compare the MSE on the validation set for the Active Learning approach and the Passive Learning approach for 3 different feature spaces. X-axis corresponds to the number of training kernels involved into the training process, Y-axis shows the MSE loss on the validation set. The lower loss the better. It shows the fact that the square of the difference between the true tile size and the predicted tile size is less for lower points than for higher points. Blue line shows the losses for the Passive Learning strategy and the orange line for Active Learning approach.

MSE was scaled by the minimum value found for the Active Learning strategy for the corresponding feature space. We can conclude that the larger the feature space

we have, the more impact Active Learning techniques could have. We observe the biggest gap between Active and Passive Learning performance for 4.4. This feature space contains 18 numerical values that capture array accesses. On the other hand, the training set gain for 4.3 is not obvious since it has just 4 features related to array access encoding.

At some point, the losses for both strategies converge. But Active Learning significantly overperforms (for Yuki et al. and One-hot encoding) Passive learning under current settings due to the choice of the most diverse data. This fact could be used for problems where we have time constraints for data labeling and we need a faster-converged ML model.

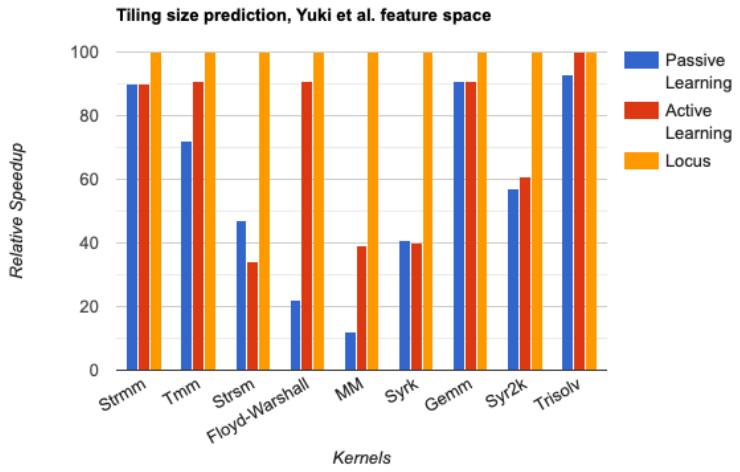

Figure 4.5: Yuki et al. features, Average relative speedups

Losses on the test set

We measure the average relative speedup for the test set to evaluate the quality of the generated code. Figures 4.5, 4.6, 4.7 show the results for nine well-known computational kernels after applying loop tiling and for three different feature spaces. The blue columns correspond to the training process based on passive learning settings, and the red ones - are based on Active Learning.

The other columns correspond to speedups obtained with the state-of-the-art LOCUS auto-tuner [32] when loop tiling is applied. The autotuner's search space is made up of the same points as for our ML model (integer values from 2 to 512 for 3-D cubic tiling). LOCUS was asked to execute 300 points out of the search grid to find its best solution. These last results are used as references to know how far we are from the optimum. The higher the bar, the better. It shows the fact

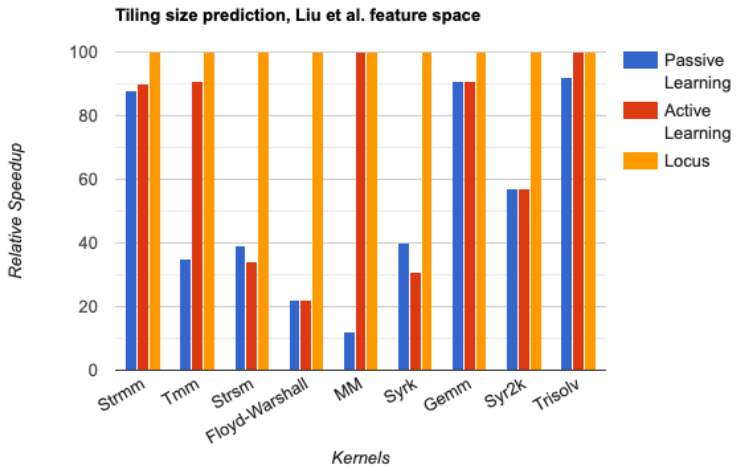

Figure 4.6: Liu et al. features, Average relative speedups

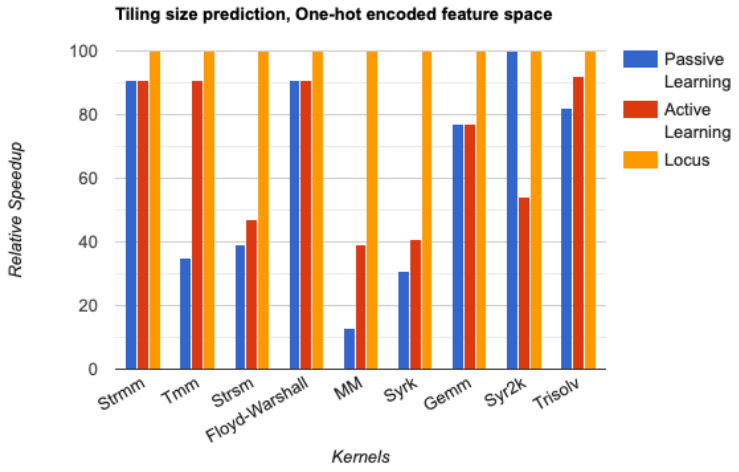

Figure 4.7: One-hot encoding, Average relative speedups

that the found speedup the closer to the speed that the autotuner found during the exhaustive search.

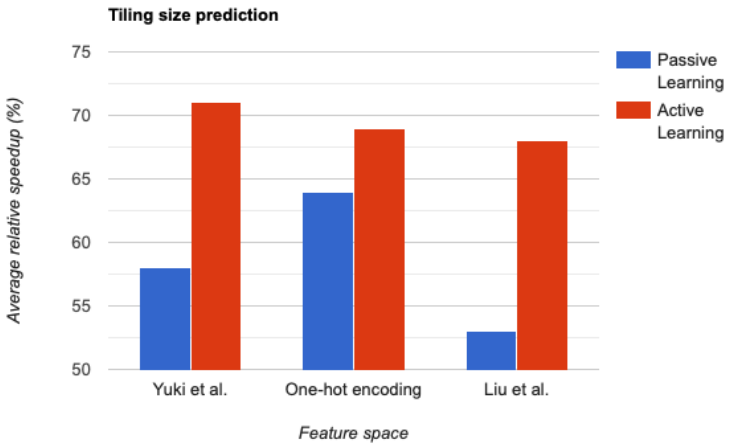

Figure 4.8: Average relative speedups

Figure 4.8 introduces the relative average speedups for the three different feature spaces. The average relative speedup with the Yuki et al.[1] features obtained by Active Learning is 71% out of the speedup found by LOCUS autotuner. The average speedup obtained by Passive Learning is 58%. The same result is observed for the one-hot encoded features [8]. The corresponding values for Liu et al.[45] features are 68% and 53%. The average speedup with Active Learning is 69% compared to 64 % without it.

Predictions made by our models were not robust for kernels TMM and Floyd-Warshall. It means that each new piece of data could dramatically change the performance. We connect it with the factors that impact performance but are not taken into account in the used feature spaces. This is a non-rectangular iteration space for the TMM kernel and a conditional statement for Floyd-Warshall. Our training data had only rectangular iteration spaces and did not contain conditional statements. For other kernels, we observed more or less the same behavior not depending on the feature space. The closer kernel is to the center of the data points used - the better predictions we have.

Active Learning performs better than passive learning on average and for the majority of kernels. The average speedup along feature spaces is 1.11x higher with the use of Active Learning. The results obtained show that the active learning approach can traverse the learning process more efficiently and shift the distribution of chosen kernels toward important patterns. For the results shown in this chapter, we used Greedy Sampling on both Inputs and Outputs since it achieved the best quality.

Losses on the test set. Alternative Machine Learning models

Figures 4.9, 4.10, 4.11, 4.12, 4.13 show the loss function on the test set for the Active Learning and the Passive Learning approaches. Each point was obtained by training the model on the exact number of training samples and measuring MAE. MAE was scaled on the minimal value found for the Active Learning strategy for each classifier. K-top ranked points were taken to compute the function at point k (at the x-axis). We trained these models on a restricted training set (top-300 programs) due to time constraints.

At some point, the losses for both strategies converge. But Active Learning significantly over-performs Passive Learning under current settings due to the choice of the most diverse data. This fact could be used for problems where we have time constraints for data labeling. We can observe that at some point MSE/MAE loss stops to decrease for the Active Learning approach. Firstly, we investigated just 300 kernels for the training. Perhaps, it may go down investigating more kernels. Secondly, we considered the relatively simple transformation and used feature spaces with not high cardinality. For the experiments in the next chapters, the saturation point would be shifted much righter in terms of the kernels used, since we need to

cover more patterns in data.

We provide the results of the experiments for five ML models: Random Forest, Gradient Boosting, Decision Tree, SVR, and Lasso. Due to computational non-stability, results for the Decision Tree cannot be interpreted, but they are stable for the other classifiers and Active Learning overperforms Passive Learning for each case. We can observe more pronounced results for non-linear models (Random Forest and Gradient Boosting) with a gain in terms of loss up to 100%, for linear models it is up to 20%.

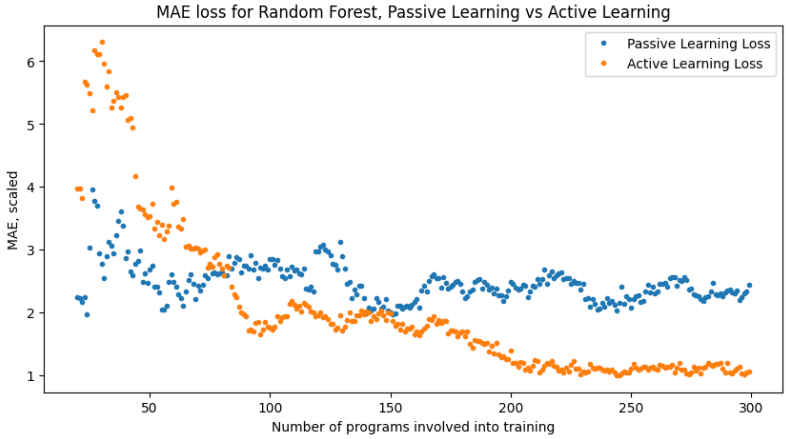

Figure 4.9: Random Forest, MAE (scaled) on the test set

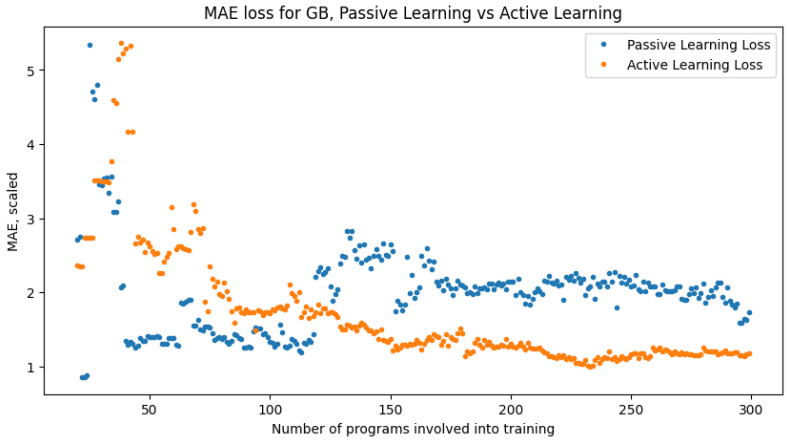

Figure 4.10: Gradient boosting, MAE (scaled) on the test set

Figure 4.14 introduces the MAE loss for different ML models. Non-linear models (Random Forest and Gradient Boosting) perform much better than linear ones. MAE loss is more than two times less.

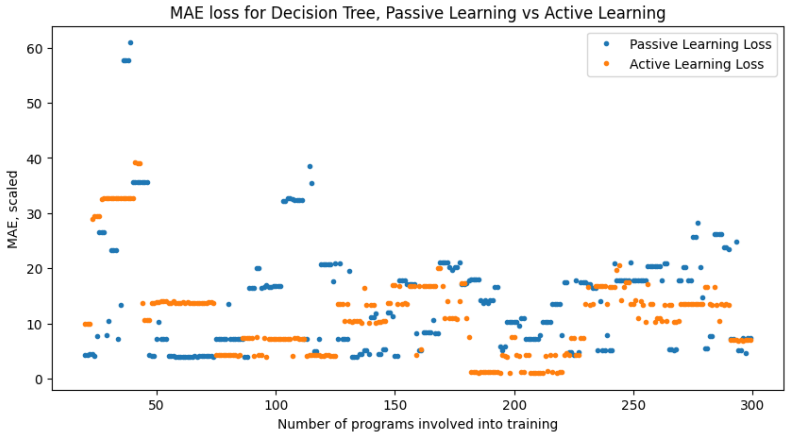

Figure 4.11: Decision Tree, MAE (scaled) on the test set

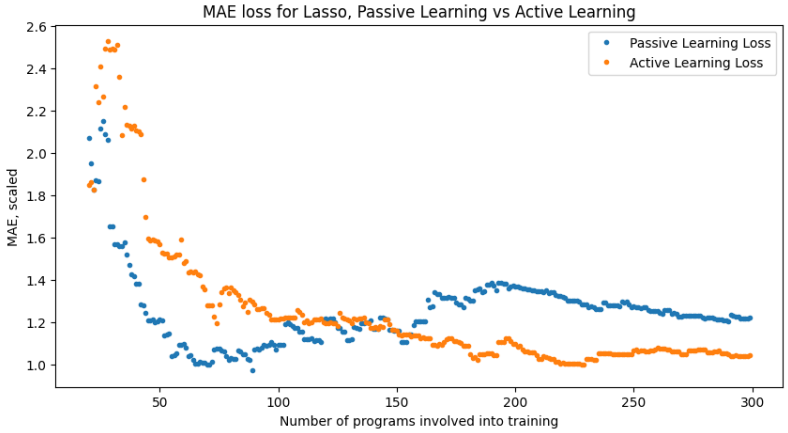

Figure 4.12: Lasso, MAE (scaled) on the test set

Sampling strategies comparison

We also provide a comparison of different sampling strategies in figure 4.15.

The greedy Sampling strategy on Inputs and Outputs performs the best. These results demonstrate that diversity in each concept matters and their combination can bring significant gain. However, diversity in inputs brings more profit.

4.4 Conclusion

This chapter presents a methodology for efficiently generating benchmarks for code optimization using ML techniques. It presents a smart strategy with active learning for extending the benchmark as needed.

We have proposed a strategy to increase the amount of data in a limited time. In

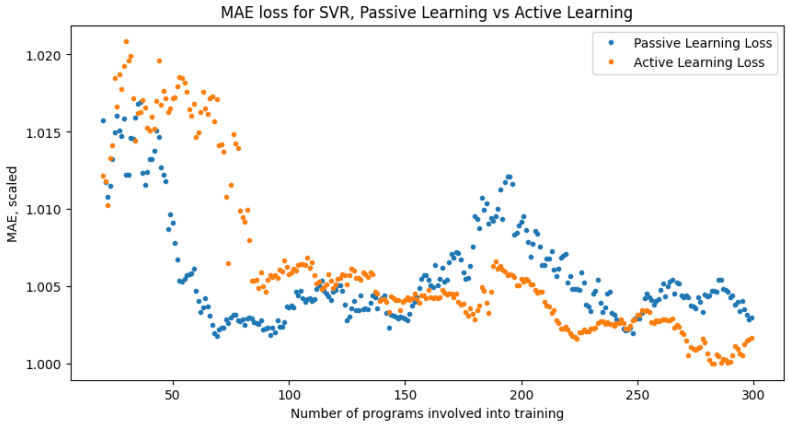

Figure 4.13: SVR, MAE (scaled) on the test set

this way, we only generate the most useful inputs. This approach allows us to select the best data for analysis and generate the most representative machine-learning models if we do not have enough expert knowledge about the domain or do not want to introduce bias in the selection. The speedup gain for our strategy is up to 15% higher depending on the feature space and 11% higher on average.

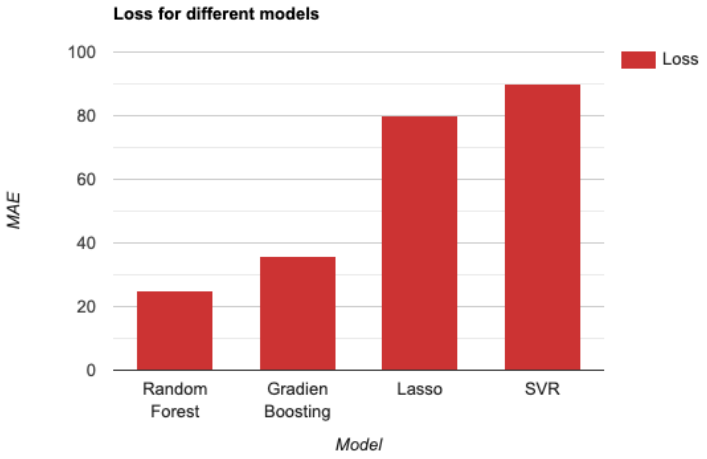

Figure 4.14: MAE loss for different models

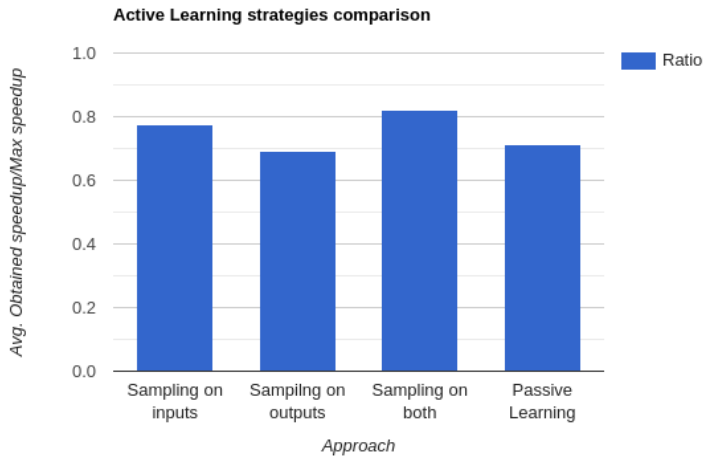

Figure 4.15: Comparison of different sampling strategies

Chapter 5

Loop Tiling transformation. Its parameters and experimental insights

Résumé

La transformation de tuilage est l'une des techniques d'optimisation de code essentielle pour optimiser la localité et le parallélisme des données. L'idée principale est de diviser l'espace d'itération initial en blocs et de les parcourir dans un ordre adéquate. Cette transformation est paramétrique et très sensible au réglage des paramètres. Une mauvaise sélection des paramètres peut entraîner des performances bien inférieures à celles du code initial. Les solutions performantes existantes considèrent une liste restreinte de paramètres pour gérer ce problème et garantir des solutions sûres.

Cette thèse propose des solutions qui vont au-delà des techniques de pointe actuelles et gagnent en accélération supplémentaire compte tenu d'un plus grand nombre d'options de tuilage. Notre approche est basée sur des méthodes d'apprentissage automatique et dérive automatiquement des heuristiques pour ajuster les paramètres de tuilage.

Ce chapitre donne le contexte nécessaire à cette transformation et met en évidence les points clés qui impactent le plus les performances et celles que nous essayons de prévoir.

Introduction

The tiling transformation is one of the most crucial code optimization techniques to expose data locality and parallelism. The main idea is to split the initial iteration space into blocks and traverse them in a special order. This transformation is parametric and very sensitive to parameter tuning. Poor parameter tuning can lead to much lower performance than the initial code. Existing state-of-the-art solutions consider a restricted list of parameters to handle this issue and guarantee safe solutions.

This thesis proposes solutions that go beyond current state-of-the-art techniques and gain additional speedup considering a larger set of options for tiling. Our approach is based on Machine Learning methods and automatically derives heuristics to tune tiling parameters.

This chapter brings the necessary context for this transformation and highlights the key points that impact the most on the performance and what we are trying to predict.

This chapter is organized as follows. Section 5.1 presents the main parameters of our tiling transformation. An adequate setting of them determines the performance of this transformation. Section 5.2 presents interesting insights found during our experimental phase. This knowledge helps us to better understand which loop tiling parameters are necessary for a good prediction during the machine learning phase, and what must be taken into account.

Finally, Section 5.3 summarizes our experimental knowledge gained in this chapter and draws a conclusion on which hyperparameters to be taken into account for ML modeling.

5.1 Loop Tiling Parameters

The potential performance that you can get by applying loop tiling depends on the choice of the loop tiling parameters but also on the quality of the code generated after tiling (in the context of this thesis, we use only PIPS as code generation tool). Indeed, once the tile parameters have been chosen, there are several possible execution paths to generate the tiled code which respect the semantics of the initial program. The questions of the choice of the directions for scanning, the tiles, and the ordering of the computational iterations in the tiles arise.

This section introduces important criteria: parameters and code generation options that have an impact on the loop tiling transformation performance.

5.1.1 Tile partitioning

Loop tiling transformation implies the partitioning of the iteration space into blocks according to the chosen partitioning matrix [67].

Figure 5.1 shows an example of diamond tiling of the iteration space. Partitioning vectors **p1**,**p2** on the figure form the partitioning matrix $\begin{pmatrix} p1 \\ p2 \end{pmatrix}$

In the context of this thesis, we explore the following techniques of tile partitioning.

- Cubic partitioning applied on 3D nested loops.
- Parallelepiped partitioning applied on 3D nested loops.

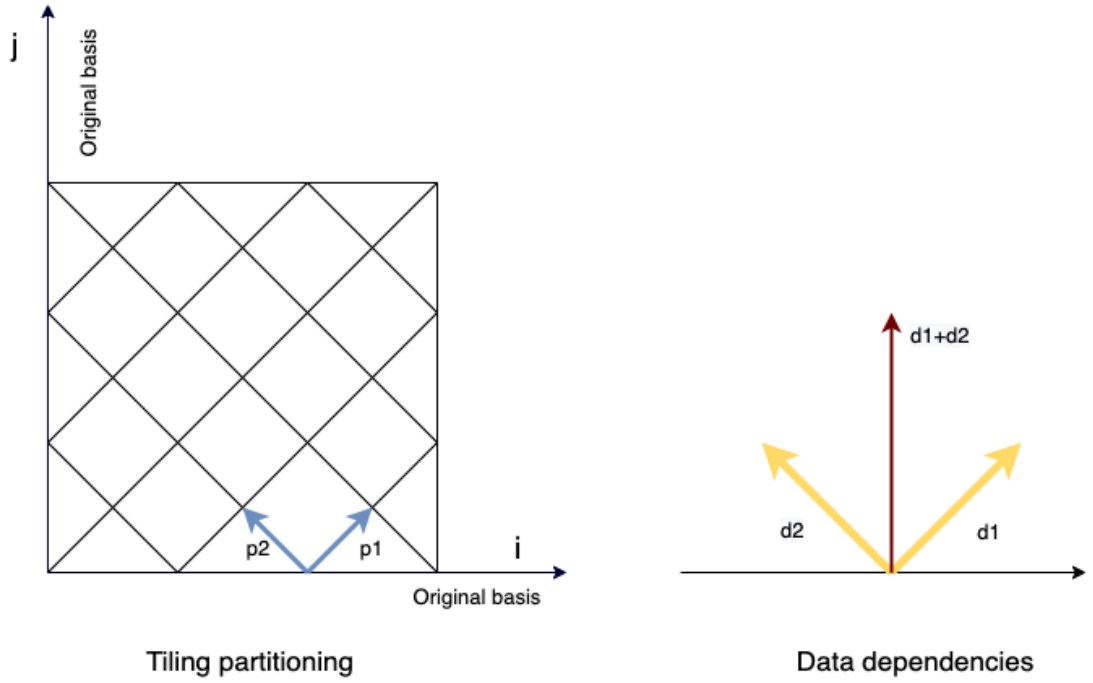

Figure 5.1: Tile partitioning

- Square partitioning applied on 2D nested loops.
- Rectangular partitioning applied on 2D nested loops.
- Diamond partitioning applied on 2D nested loops.

Tile partitioning is a general concept, it leaves a lot of freedom for researchers to define it. These options help us to consider a huge set of tile partitioning matrices, and to investigate more general or more narrow cases to draw conclusions about their applicability. We tried to find a trade-off between the complexity of the partitioning matrix, the quality of an ML prediction, and the gain we can obtain. Obviously, the more general case we consider, the more potential gain we can achieve, but on the other hand, a Machine Learning model can perform much worse due to the complexity of the output and more factors to capture. However, our methodology is not limited with these tiling partitionings. Potentially, partitions of any depths could be applied.

5.1.2 Tile sizes

The selection of tile sizes is a crucial step of the tiling transformation. Large tile sizes can cause a lot of slow high-level cache usage. In contrast, small tiles may not fully benefit from improved data locality.

Also, tile sizes that are not divisors of the loop bounds can lead to restricted tile partitioning on the borders of the iteration space. It can lead to poor load balancing between threads and unsatisfactory performance.

The tile size selection depends on many criteria such as the application, the architecture, and the cache hierarchy. In our study, we only vary the application characteristics such as the size of the iteration domain, the access functions for array elements, and the data dependencies. The architecture is fixed.

5.1.3 Scanning directions for tiles and its elements

The loop tiling transformation implies the partitioning of the iteration space into blocks according to the chosen partitioning matrix [67]. However, the order in which the blocks are traversed is not unique. Data dependencies define the correctness of execution (block traversal). Likewise, the order of traversal of the points inside the block is not unique. Since several possibilities are possible, we can consider this choice as a parameter to predict.

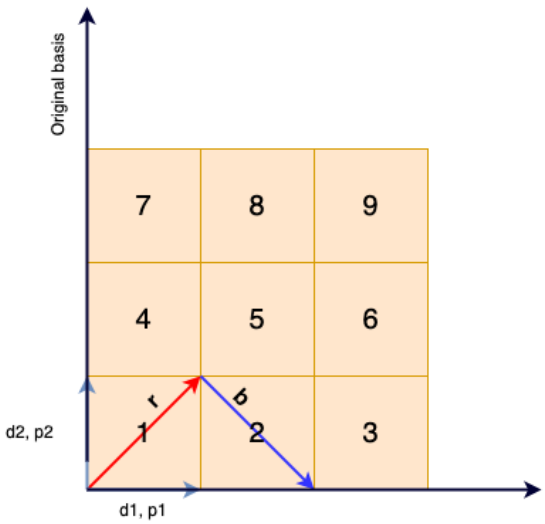

Figure 5.2: Inter-tile scanning

Our compiler PIPS [33] offers nine possible choices for generating tiled code. It is a combination of three possible choices for scanning the tiles and three possible choices for scanning the elements inside the tiles.

Figure 5.2 illustrates one possible way of scanning the tiles of the computational domain according to its data dependency vectors. Let us define **d1** and **d2** as the extreme rays [2] of the dependence cone which summarizes the set of dependencies of the computational kernel [69]. The concept of extreme rays and dependence cone is introduces in details in 6.2.1.

The red vector **r** represents their sum (the sequential hyperplane direction), the blue vector **b** is perpendicular to the red vector and represents a potential parallel direction [157]. The red and blue vectors describe a possible basis for scanning the tiles that we refer to as $\mathtt{T}_{Parallel}$ tile direction. The sequential hyperplane direction carries all dependencies, while blocks along the parallel direction can be executed concurrently. So, the sequence of blocks 1-4-2-7-5-3-8-6-9 represents a legal execution schedule if we apply $\mathtt{T}_{Parallel}$ scanning; among others, blocks 3-5-7 can be run in parallel.

The second possible direction to scan tiles is named $\mathtt{T}_{Shape}$. Its scanning directions are parallel to the partitioning vectors of the tiles (rectangular shape in this case), so **p1** and **p2** become a possible tile scanning basis. Thus the sequence of blocks 1-2-3-4-5-6-7-8-9 is an example of legal execution in this case. The third possible scanning directions $\mathtt{T}_{Initial}$ are parallel to the initial iteration basis and match the second case here.

We use the same strategy to define possible scanning directions for the elements inside each tile:

- $\mathtt{L}_{Initial}$: relative to the initial iteration basis,
- $\mathtt{L}_{Parallel}$: the hyperplane sequential direction and its relative orthogonal vector,
- $\mathtt{L}_{Shape}$: relative to the partitioning vector directions.

The same possible scanning vectors exist to traverse intra-tile elements.

In the general case, we have nine combinations of scanning directions. However, some of them may not be legal due to data dependencies. PIPS is responsible for verifying that the requested transformation is legal.

5.1.4 Kernels of interest

Loop tiling transformation can only be applied to parts of kernels whose loops are perfectly nested.

We distinguish two classes of these kernels which benefit the most from the tiling. First, kernels with data dependencies (read-read or read-write dependencies). It is intuitive because if dependencies exist, some data is referenced several times and it is interesting to exploit the locality of their use. Therefore, if tiling is not applied first, we have a more restricted set of optimizations and less potential benefit. Second, kernels that have large array references. The tiling transformation effectively helps to reduce the number of living variables needed when computing the tiles.

5.1.5 Parallel code generation

After applying tiling and exploiting parallelism in the nest of loops, several loops can be parallelized and vectorized.

In OpenMP, it is not beneficial to generate several levels of parallel loops because the creation of new parallel threads in threads has an additional cost. It is more efficient to keep a single parallel loop externally and a vectorial loop internally.

The choice of which parallel loops to keep when there are several possible is tricky. We must take into account the number of iterations of these loops; it must be greater than the number of processors and be a multiple of it, if possible, to take full advantage of the potential number of architectural parallelism. The outermost loop should be preferred for GPU or coarse-grain architectures.

In the framework of this study, we exploit the parallelism of loops by generating OpenMP parallel directives for the outermost parallel loop and OpenMP vectorial directives for the innermost loop. The number of threads has been chosen to use the maximum number of parallel cores available and multi-threading when possible. The example of the original code and the tiled code is shown in listings 5.1 and 5.2. A cubic tiling with tile size 8 was applied to the initial code.

Listing 5.1: MM original code

```
1 for ( i = 0; i < 1024; i++)
2     for ( j = 0; j < 1024; j++)
3         for ( k = 0; k < 1024; k++)
4                 C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

Listing 5.2: MM with the best cubic tile size

```
1  #pragma omp parallel for private(i_t, k_t,j_t,i_l,k_l,j_l,i,k,j)
2  for (i_t = 0; i_t <= 127; i_t += 1)
3      for (j_t = 0; j_t <= 127; j_t += 1)
4      for (k_t = 0; k_t <= 127; k_t += 1)
5      for (i_l = 8 * i_t; i_l <= ((8 * i_t) + 7); i_l += 1)
6      for (j_l = 8 * j_t; j_l <= ((8 * j_t) + 7); j_l += 1)
7    {
8    #pragma  vector always
9      for (k_l = 8 * k_t; k_l <= ((8 * k_t) + 7); k_l += 1)
10       C[i_l][j_l] = C[i_l][j_l] + (A[i_l][k_l] * B[k_l][j_l]);
11   }
```

5.1.6 Conclusion

This section summarizes all the crucial components that we need for efficient code. First, we need programs that potentially benefit from the tiling transformation. Then, we need a set of options for the transformation to be tuned. And the final step is the proper parallel code generation and its benefits.

5.2 Experimental insights about tiling transformation

This section provides some experimental answers to some questions related to tiling that could be answered only in a practical way. In this section we compare different tiling options, choose hyperparameters and measure their profitability. This information is required to define the proper settings of Machine Learning models, it helps us to understand what to predict exactly and which parameters/hyperparameters to use.

The fixed hyperparameters that we used were:

- Architecture: Intel® Core™ i7-8650U 4C/4T @1.90GHz; Caches: L1: 32KB, L2: 256KB, L3: 8192KB; 32GB DDR4 DIMM RAM; Phys. cores: 4
- Compilation sequence: -O3 -march=native -mtune=native -ftree-vectorize -fopenmp / 4 Threads
- Baseline sequence: -O3 -march=native -mtune=native -ftree-vectorize -fopenmp / 1 Thread.

We investigate the impact of the following choices on the performance:

- Strategy to label data: only divisors of loop bounds vs. any integer in the domain
- Size of the iteration domain: whether it fits L3 cache or not
- Strategy to choose dimensions for tiling: 2D or 3D tiling

- Shape and complexity of the partitioning matrix: Cubic or parallelepiped tiling
- Number of threads: 2, 4, 8 threads
- Tile size
- Scanning directions
- Tiling matrices: Cubic, Parallelepiped, Square, Rectangular, Diamond matrices

5.2.1 Loop tiling speedups

The previous sections have presented the theoretical components of a loop tiling transformation. This subsection experimentally addresses the real benefits that can be achieved just with loop tiling without any other transformations.

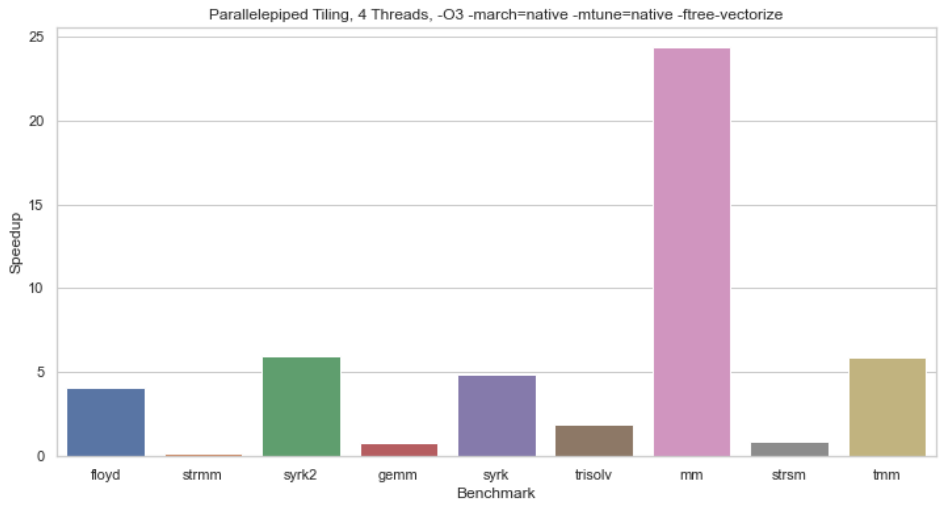

Figure 5.3: Tiling Speedups

Figure 5.3 gives the speedups obtained by applying the loop tiling transformation to 9 famous kernels. We applied a parallelepiped tiling on a cubic iteration domain with 4096 iterations per dimension and executed it with 4 threads. The optimal tile sizes were found with the LOCUS Autotuner [32] after an iterative search.

6 out of 8 benchmarks benefited from tiling transformation, the average speedup is 6.07x, the median is 4.46x and we achieved 24.33x speedup for mm kernel. Two kernels (gemm and strsm) did not benefit from the transformation.

Listing 5.3 gives the code of gemm-kernel. It contains three different array references: C[i][k], A[i][j], and B[j][k]. We notice that because of the order of execution of the iterations, tiling (i,j,k) provides no performance gain or better locality to the references of arrays C (the second dimension is fully accessed for each iteration j) and B (fully accessed for each iteration i). Only the reference on A is invariant for the last innermost loop.

Strsm-kernel 5.5 has a non-typical iteration domain shape (triangular on i,k dimensions), hence some problems with "small tiles" on the borders of the iteration space which can affect and downgrade performance.

On the other hand, listing 5.4 contains the reference B[k][j] which basically does not have good locality property and makes this kernel a point of interest for the tiling transformation. Note, that Gemm and MM are variants of the same benchmark but with interchanged loops.

Listing 5.3: Gemm example

```
for ( i = 0; i < n; i++)
    for ( j = 0; j < n; j++)
        for ( k = 0; k < n; k++)
                C[i][k] = C[i][k] + A[i][j] * B[j][k];
```

Listing 5.4: MM example

```
for ( i = 0; i < n; i++)
    for ( j = 0; j < n; j++)
        for ( k = 0; k < n; k++)
                C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

Listing 5.5: Strsm example

```
for ( i = 0; i < 1024; i++)
    for ( j = 0; j < 1024; j++)
        for ( k = i+1; k < 1024; k++)
        {
                if (k == i+1) B[j][i] = B[j][i]/ A[i][i];
                B[j][k] = B[j][k]  - A[i][k] * B[j][i];
}
```

We can conclude that the experimental results of tiling transformation are very promising, this transformation can accelerate code up to 14x for some known benchmarks. However, not all kernels benefit from it. We show the results of this transformation with basic parameters, different options and trade-offs will be shown in the following subsections.

5.2.2 Loop bound divisors vs. all integers in the domain as tile sizes

This subsection provides insights on which values to predict. Can we restrict our predictions just to the divisors of the loop bounds (in this subsection they are powers of two) or do we need to consider any integer of the domain to archive the maximum gain?

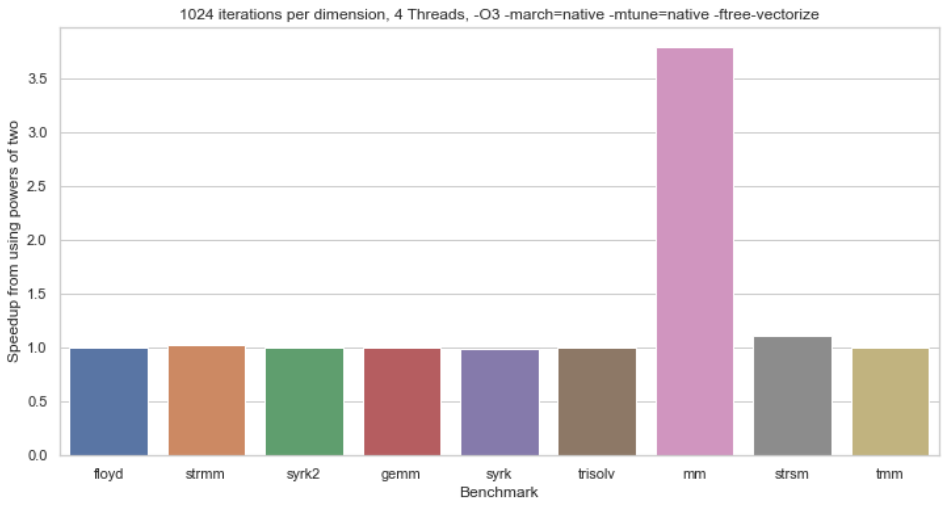

Figure 5.4: Speedups from using powers of two versus integers for cubic tiling

Figure 5.4 gives a comparison of speedups found when we asked autotuner to consider just loop-bound-divisors as tile sizes versus all-integers for **cubic** tiling. Y-axis corresponds to the ratio of execution time with the best integer solution found by the Autotuner [32] to the execution time found with the loop-bound-divisors search strategy.

As we can see, for 7 out of 8 benchmarks, there is no gain in performance. However, the loop-bound-divisor solution for the mm-benchmark was 3.78x higher than the integer one. Figure 5.5 shows the autotuning results for mm-kernel. We can see that the loop-bound-divisor solutions overperform all possible solutions out of this domain. The all-integers strategy marked tile size 512 as the best one but did not investigate the area of small tile sizes properly, marking it as not potentially profitable. It leads to poor performance compared to the divisor strategy that easily found the best tile size 8.

However, loop-bound-divisor solutions do not always overperform the other solutions. Figure 5.6 demonstrates that divisors do not tend to be outliers during autotuning. The autotuning results of other kernels are presented in appendix A

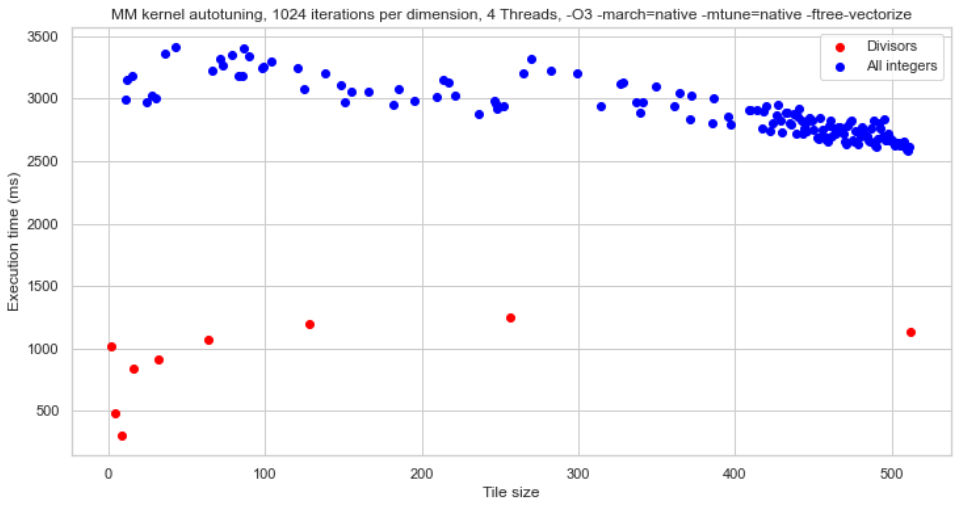

Figure 5.5: Autotuning of kernel MM

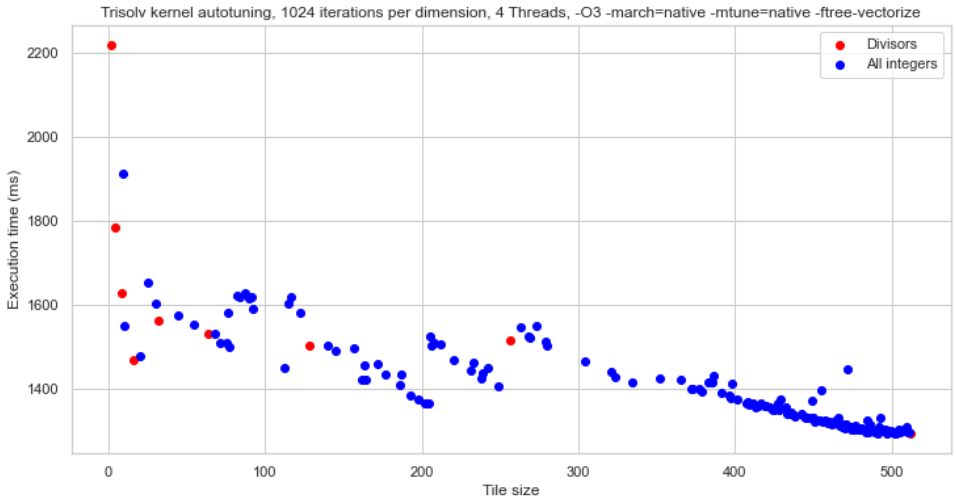

Figure 5.6: Autotuning of kernel Trisolv

Figure 5.7 shows the difference between the best tile size and the one found with the all-integers strategy. The x-axis corresponds to the studied kernel and the y-axis corresponds to its optimal tile size. In two cases the misprediction was significant in absolute value. For these cases, the best tile size is a "power of two" and corresponds to a loop-bound-divisor since we considered 1024 as loop bound for all the loops.

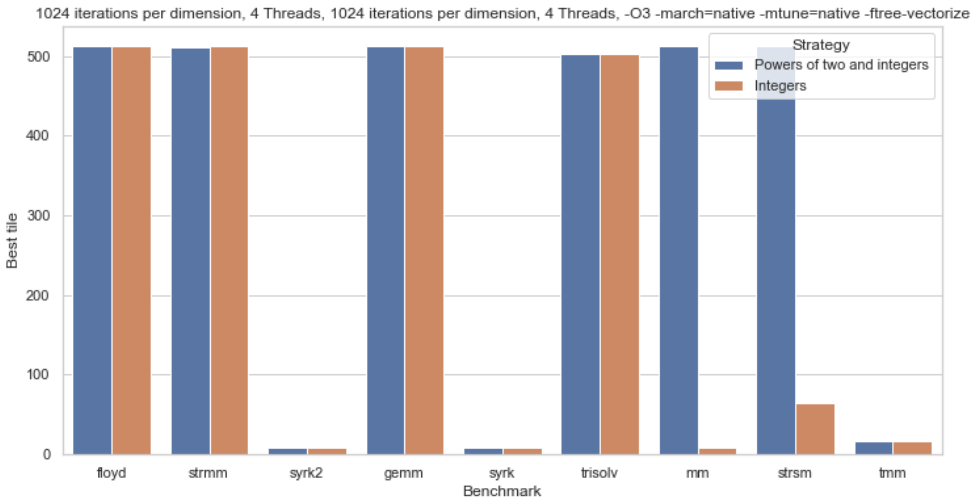

Figure 5.7: Difference between best tile and the best "integer" one

The main point is that even if the search domain is relatively small, then considering all integers worsen the performance sometimes. What are the results in the opposite situation when we increase the search domain?

Now we consider parallelepiped tiling instead of cubic. Figure 5.8 presents a comparison of speedups found when we asked the autotuner to consider just loop-bound-divisors versus all-integers for **parallelepiped** tiling. Our search domain has been increased in cubic progression. We increased the autotuner search time by a factor of 5, but this did not help to achieve the performance of tiled codes with loop-bound-divisors (powers of two) for 6 out of 8 codes. The average result obtained with loop-bound-divisors is 1.10x higher than the result found considering all integers. The further increase in autotuner search time shows weak convergence compared to the loop-bound-divisors results.

The search space is complex and nonlinear, so Autotuner does not aim to generalize information about previous executions to find the appropriate solution in a reasonable amount of time.

This fact will be used in our further models. Actually, we predict integer solutions all the time, but we introduce the heuristic to round the prediction to the nearest divisor of the corresponding loop bound. It helps to achieve better code in terms of execution time in many cases.

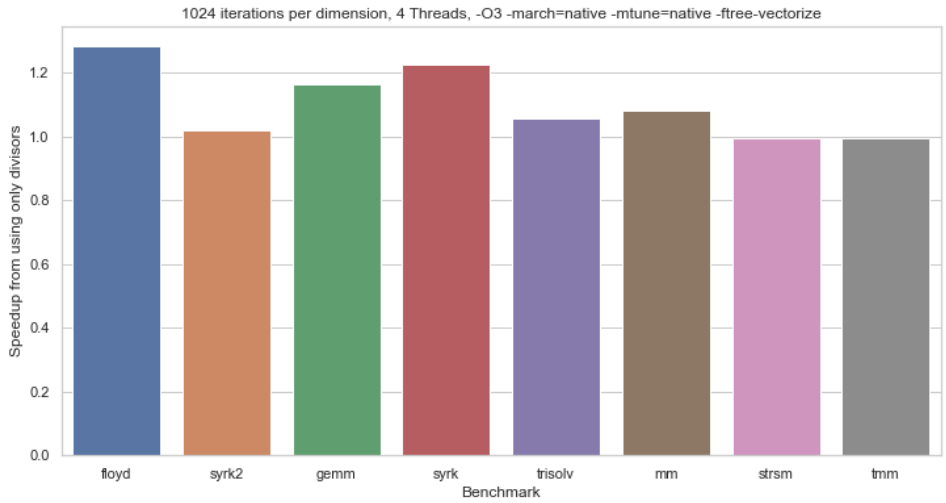

Figure 5.8: Speedups from using loop-bound-divisors vs. integers for parallelepiped tiling

5.2.3 Impact of the size of the iteration domain

This subsection investigates the impact of the number of iterations of the computational domain on the achieved speedups after tiling. We consider two sizes of the iteration space: the first one that does not fit to the L3 cache (4096x4096x4096 iterations per loop) versus 1024x1024x1024 that fits the L3 cache on the mentioned above architecture.

Figure 5.9 provides a comparison of the speedups that were achieved with a domain of 1024 iterations per loop dimension and 4096 iterations per loop dimension. Figure 5.10 shows the ratio of how the speedup increases for each benchmark. On average, we can expect 90% in speedup if we increase the number of iterations from 1024 to 4096. The maximum gain was achieved for tmm kernel. The speedup for 1024 iterations was 1.29x but increased by 350% and became 5.85x.

As we can see, for the majority of the kernels, the increase in the number of iterations to 4096 per dimension implies a huge gain in performance. It can be easily explained by the amortization of the computation of complex loop bounds derived from tiling transformation and the execution of not full tiles at the border of the iteration space.

We do not consider it reasonable to predict the speedup itself to evaluate the quality of our predictions, as it is an iteration-dependent metric. We observe the more stable behavior of the relative speedup metric. The relative speedup is the ratio of the speedup that gives the tile size that we predicted to the speedup found by the autotuner. This metric is not iteration dependent and can help us to generalize

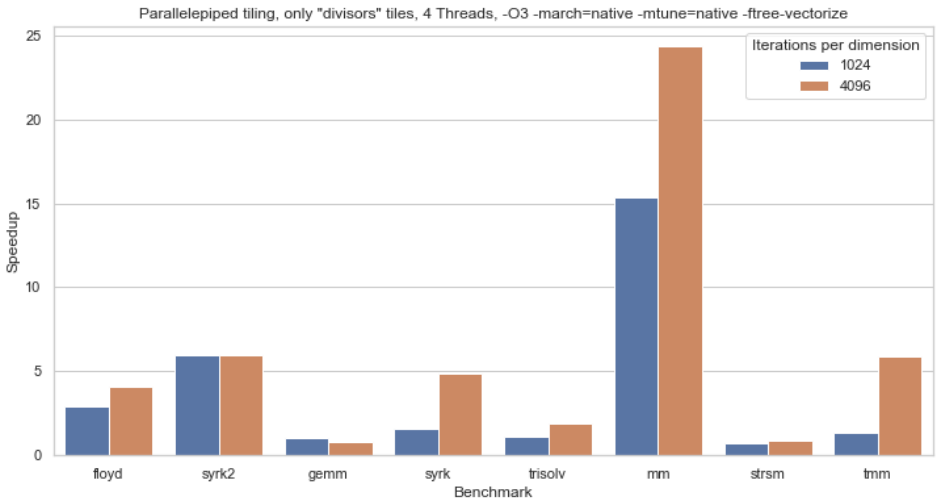

Figure 5.9: Impact of the number of iterations on the speedups

and evaluate our predictions if we consider loop nests with different numbers of iterations. We consider that if the data was labeled taking into account relative speedup as the labels then it maximizes the relative speedup on the training set. Of course, relative speedup has a correlation with absolute one but if the absolute speedup labels were used then it would maximize absolute speedup greater.

5.2.4 Cubic or Parallelepiped Tiling?

In the previous sections, we showed examples of cubic and parallelepiped tiling. Obviously, cubic tiling has fewer parameters to tune. Assuming that the size of each dimension of the iteration domain is `n` then we have only `n` possible values for the tile size. In opposite, we have n^3 values for these parameters in the case of parallelepiped tiling because the tile size for each dimension should be chosen with respect to those of the other dimensions. Does this increase the performance or just make the search process longer for the Autotuner without significant gain?

Figure 5.11 shows the experimental results. Parallelepiped tiling increased the performance for 8 kernels out of 8. It seems to be a very powerful tiling setting that worth to be investigating. The average speedup with parallelepiped tiling is 3.74x, with cubic - 1.72x. We can reach 2.17x higher performance considering parallelepiped tiling settings.

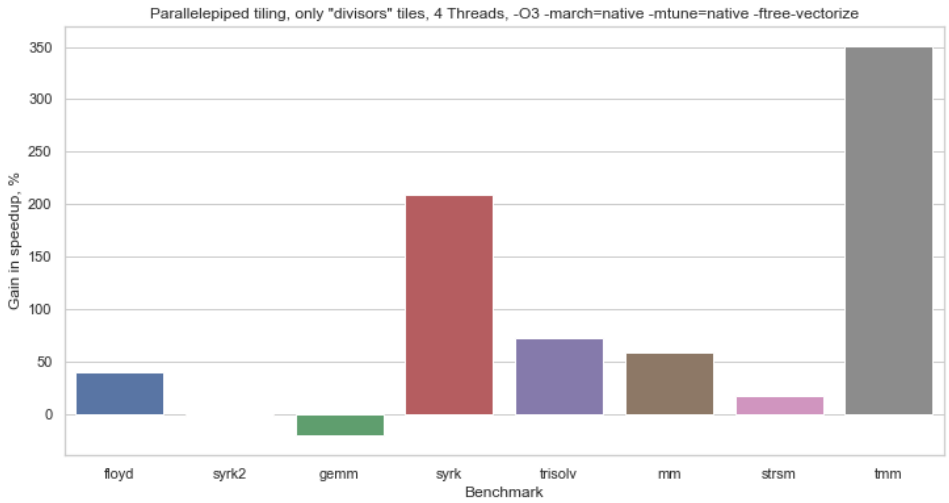

Figure 5.10: Relative gain from the increase of the iteration domain

5.2.5 2-D or 3-D Tiling?

Loop Tiling is a very flexible transformation. It leaves a lot of freedom to define its parameters. For instance, one of the most arguable questions is to use tiling on which levels. We investigate the question of profitability of 3-D tiling on three nested loops and 2-D tiling on two, the innermost loops of a three-loop nest.

Figure 5.12 gives the results of this comparison. 3D tiling is a much more powerful code transformation. On average we can expect a 3.42x average gain in performance if we consider three nested tiling. The gain in performance is crucial, we see it reasonable to consider 3-D tiling only.

5.2.6 Number of threads

This subsection demonstrates the impact of the number of threads on the speedup that we can achieve.

Figure 5.13 gives the speedups that we can achieve if we execute the code on the different numbers of threads 2/4/8 vs sequential baseline. In our experiments, we consider the number of threads to be a hyperparameter. We do not vary it and consider it to be fixed. We have chosen a safe choice of the number of physical cores. In our case, this is 4 threads. We can see the degradation of performance in the case of 2 threads for trisolv-kernel and in the case of 8 threads for mm-kernel. Four threads (the number of cores) seems a safe choice for this hyper-parameter.

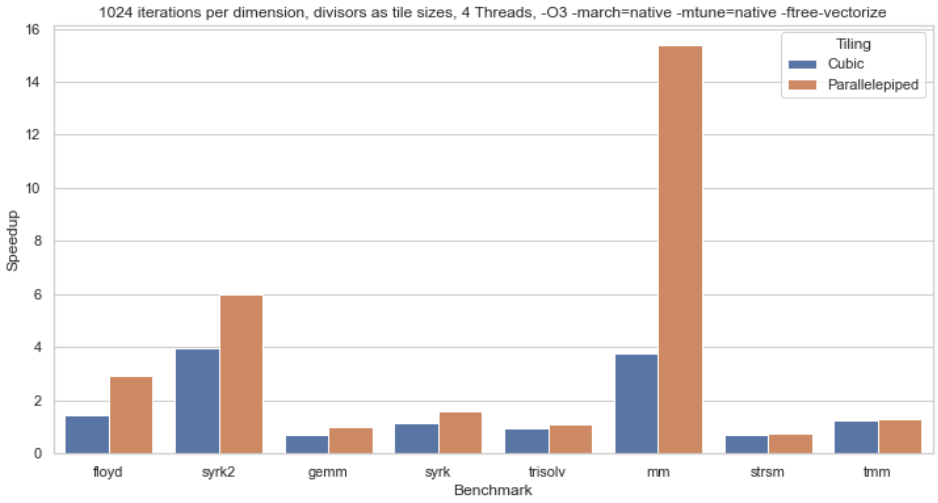

Figure 5.11: Comparison of cubic and parallelepiped tiling

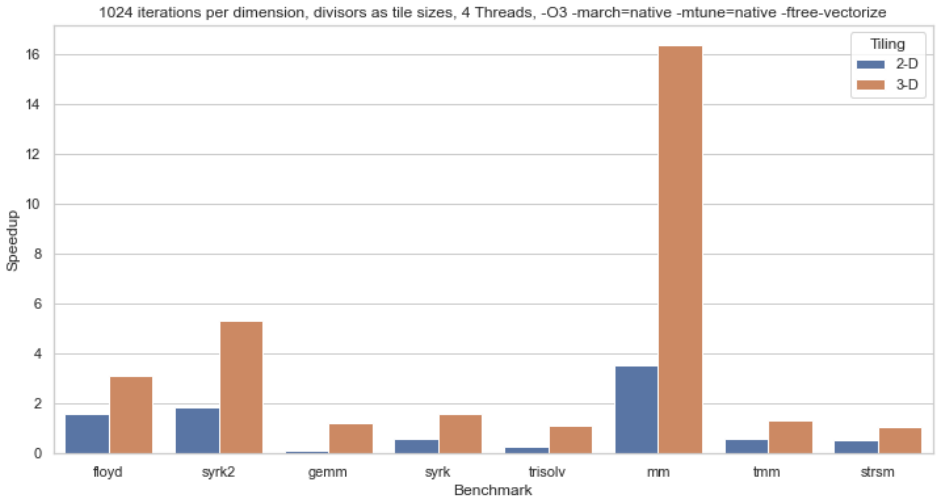

Figure 5.12: Comparison of 2-D and 3-D Tiling

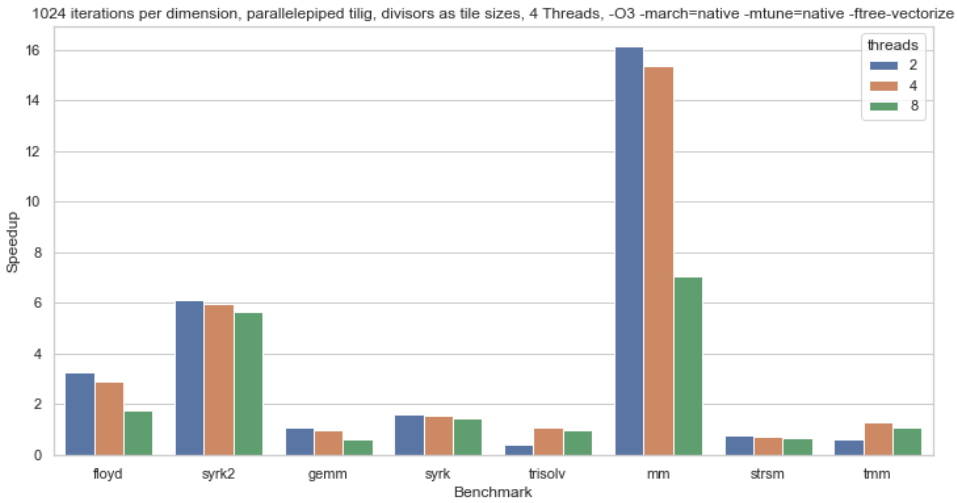

Figure 5.13: Comparison of results on different number of threads

5.2.7 Tile size selection

In this subsection, we show that the selection of the tile size is a crucial element. The distribution of the optimal tile sizes is not regular and requires the development of complex, nonlinear models. This is the reason why we have developed a powerful ML model.

Figures 5.14 and 5.15 show the speedup distribution for different tile sizes on two different kernels (4096 iterations per dimension). The speedup distribution of other kernels is presented in appendix B. We illustrate this distribution on examples of parallelepiped tiling. The X-axis corresponds to the chosen tile size on the outermost loop, Y-axis on the second loop, and the Z on the innermost loop.

The points of this space were visited with the Autotuner. Of course, it is not possible to visit all 4096^3 combinations in a reasonable amount of time. We shrink the search space to the powers of two and ask the Autotuner to visit all of them. Autotuner could stop the search earlier if it reaches some inner criterion and is sure that the best combination was found. As we can see in Figure 5.17, kernel Syr2k has a relatively large amount of "good executions". They are concentrated around small values of the X-axis. The opposite case is the MM kernel. The amount of "good executions" is relatively small, we see just several points that overperform the others.

Figures 5.16 and 5.17 present the results for rectangular tiling (2-D case) for the same kernels (1024 iterations per dimension). The speedup distribution of other kernels (2D-case) is presented in appendix C

We see that there is no unique rendering of how the tile sizes are distributed.

The distribution is very different from kernel to kernel, non-linear, with different sizes of "good executions" clusters, and there are many outlier values.

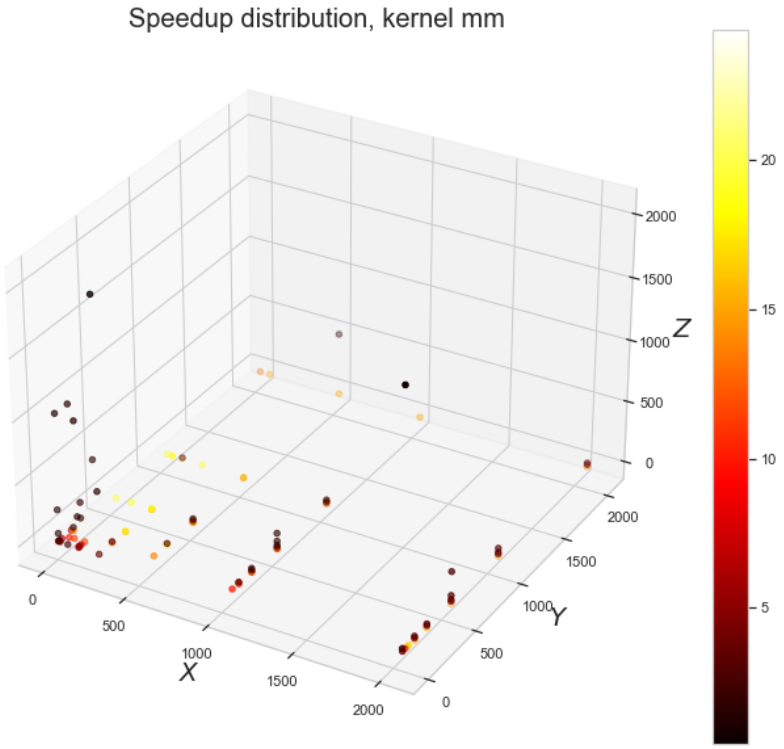

Figure 5.14: Speedup distribution, MM kernel

5.2.8 Tile matrices and scanning directions

Scanning directions

To illustrate the impact of different scanning directions, we use handcrafted kernel (Listing 5.6).

Listing 5.6: Handcrafted kernel

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        A[i][j] = A[i-1][j] + A[i][j-1] + A[i-1][j-1];
```

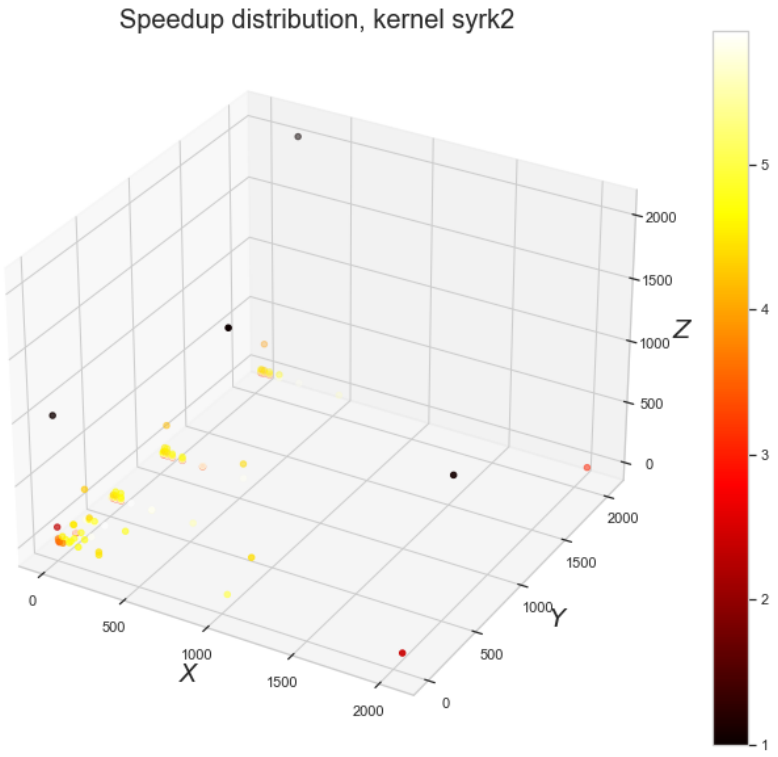

Figure 5.15: Speedup distribution, Syr2k kernel

There are two different ways to scan -intra and -inter tiles for this kernel: $\mathtt{T/L}_{Initial}$ - relative to the initial iteration basis and $\mathtt{T/L}_{Parallel}$ - hyperplane sequential direction and its relative orthogonal vector. The second outermost loop after tiling is parallel.

Figures 5.18, 5.19, 5.20, 5.21 present the performances for each possible combination of scanning directions for this kernel. The inter-tile scanning directions $\mathtt{T}_{Parallel}$ (TP), $\mathtt{T}_{Initial}$ (TI) and the intra-tile directions $\mathtt{L}_{Parallel}$ (LP), $\mathtt{L}_{Initial}$ (LI) are described in section 5.1.3. The execution was scaled based on the minimum execution time of the TI-LI scanning directions to give an intuition of what we might get compared to the initial settings. The X-axis corresponds to the tile size along the i dimension in the original code, the Y-axis corresponds to the j dimension.

Our experiments show that, for this kernel, we can achieve 1.21x better performance compared to the original scanning directions, if we used the TI-LP directions ($\mathtt{T}_{Initial}$ for -inter and $\mathtt{L}_{Parallel}$ for -intra). If we were using TP-LI directions, we can

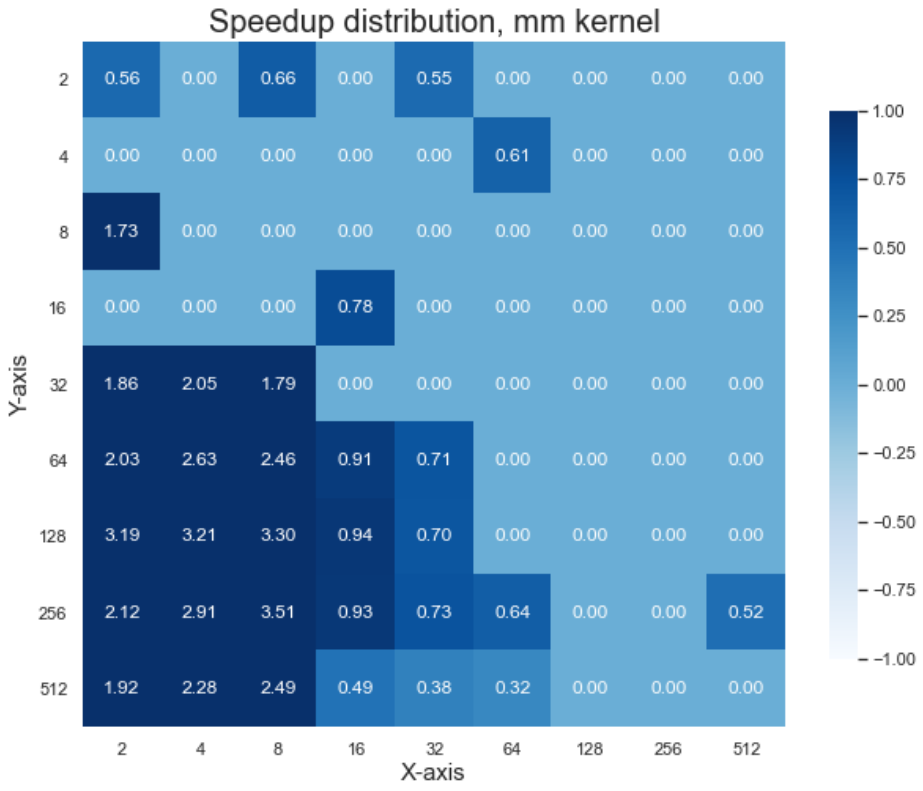

Figure 5.16: Speedup distribution, MM kernel

only get 0.71x the performance of the original settings.

This example shows that scanning directions are a crucial part of a successful model. This should certainly be taken into account if a kernel has data dependencies.

Tiling matrices

In the same way, we compare the diamond matrix and the rectangular one for the kernel 5.7. The results are shown in Figures 5.22 and 5.23. Our experiment shows that the proper matrix selection could bring up to 15% gain. This choice may be essential for kernels that have data dependencies.

Listing 5.7: Handcrafted kernel

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        A[i][j] = A[i+1][j-1] + A[i-1][j-1] + A[i][j-1];
```

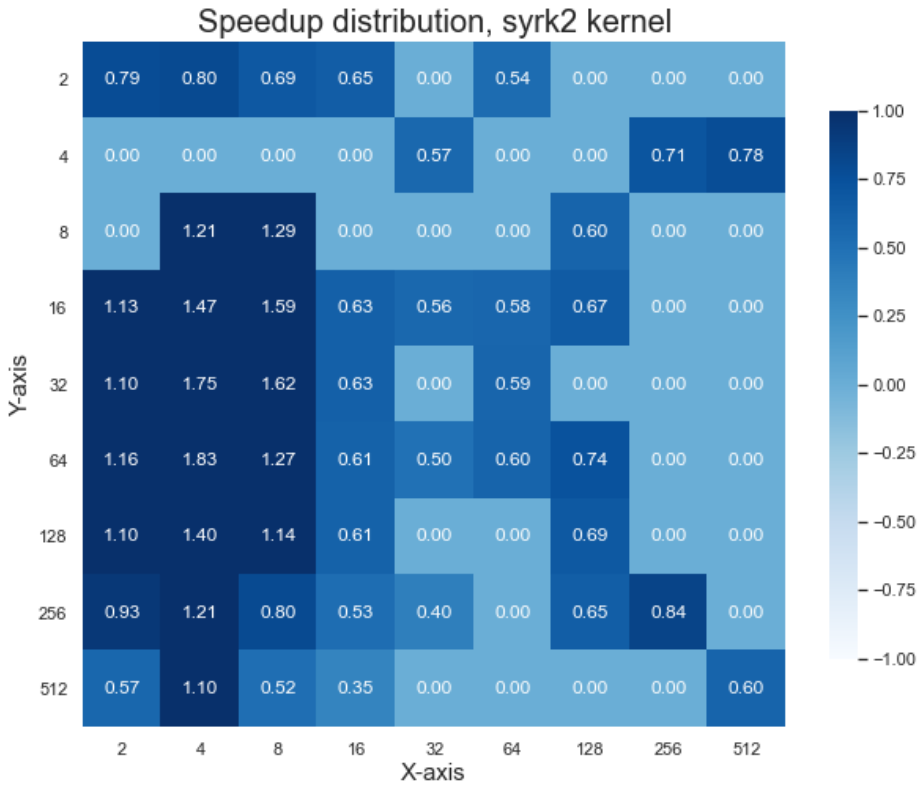

Figure 5.17: Speedup distribution, Syr2k kernel

5.3 Conclusion

Our experiments help us understand which tiling parameters are needed to predict and derive maximum benefit from the tiling transformation. Our experiments imply the following conclusions:

- Using loop-bound-divisors for the data labeling seems a reasonable choice. Heuristic to round the prediction to the nearest divisor may improve the performance.
- Size of the iteration domain matters the absolute speedup. The more iterations the more speedup. We propose an iteration-independent metric - relative speedup. We would like to evaluate not absolute values but to understand how far we are from the best solution.
- 3D tiling is a much more powerful transformation than 2D tiling. We see it reasonable to concentrate on 3D case for ML modeling.

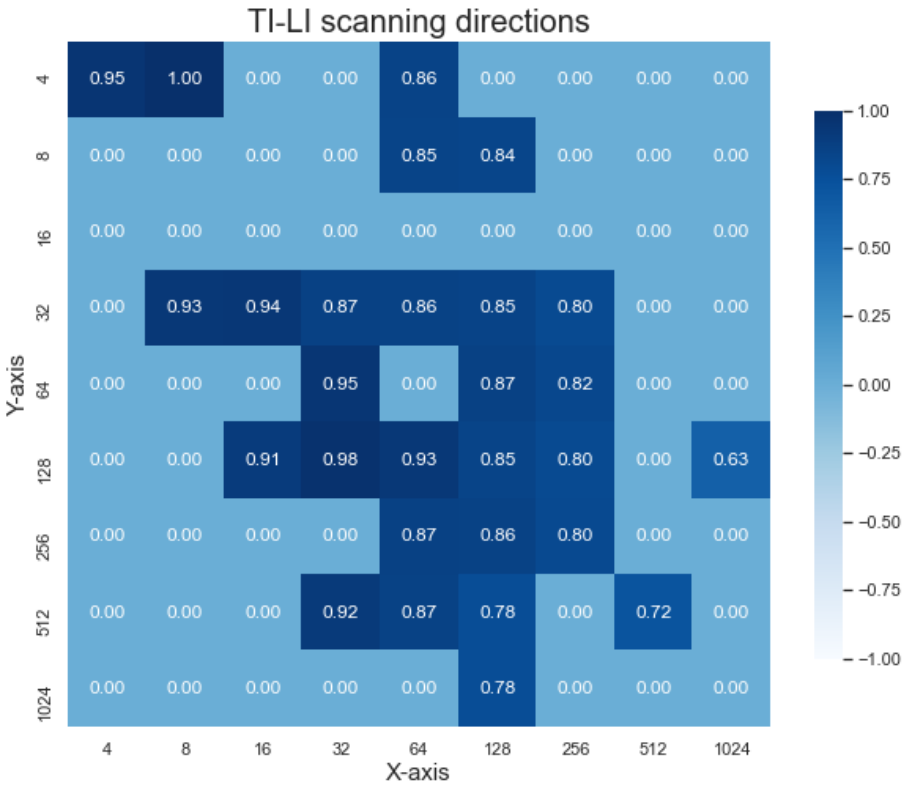

Figure 5.18: Normalized performance, TI-LI scanning directions

- Parallelepiped is a more complex partitioning than the cubic one but potentially can bring a significant gain. We see it reasonable to capture this partitioning in ML modeling.
- We keep the number of threads to be a hyperparameter. It means we do not vary it in our experiments. The number of physical cores seems the safest choice for the number of threads.
- Tile size is crucial. It is the most significant parameter in terms of impact on performance. The goal of our ML model is to predict the best possible tile size.
- Scanning directions and different tile partitioning may be very useful for kernels, where we have data dependencies. Our model should capture this information and provide the most reasonable way of scanning and way of partitioning.

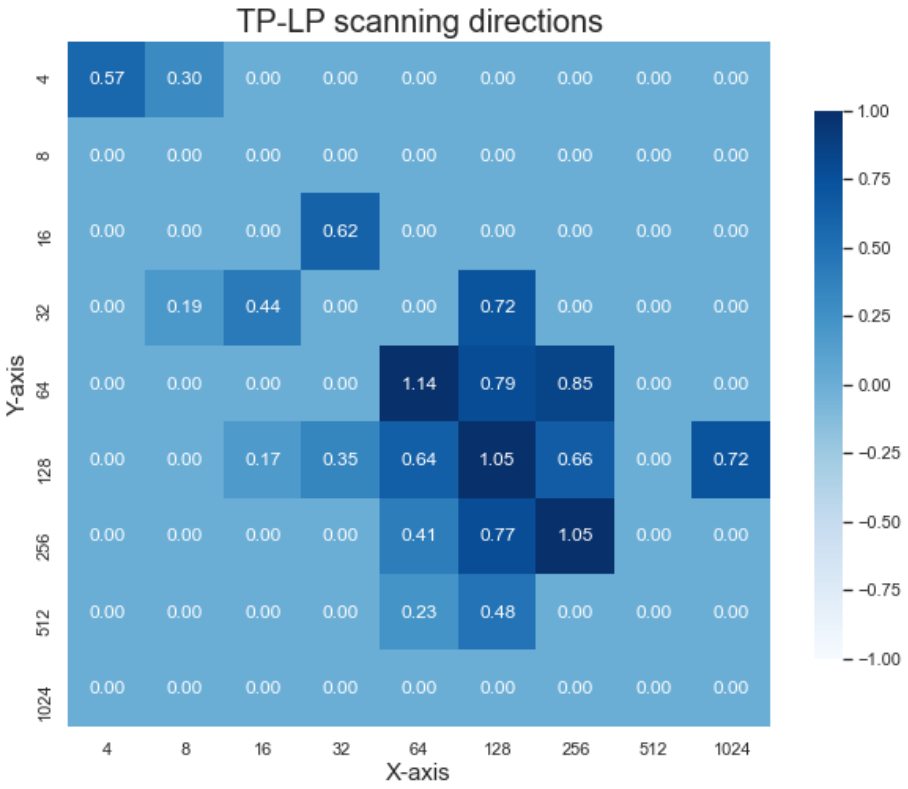

Figure 5.19: Normalized performance, TP-LP scanning directions

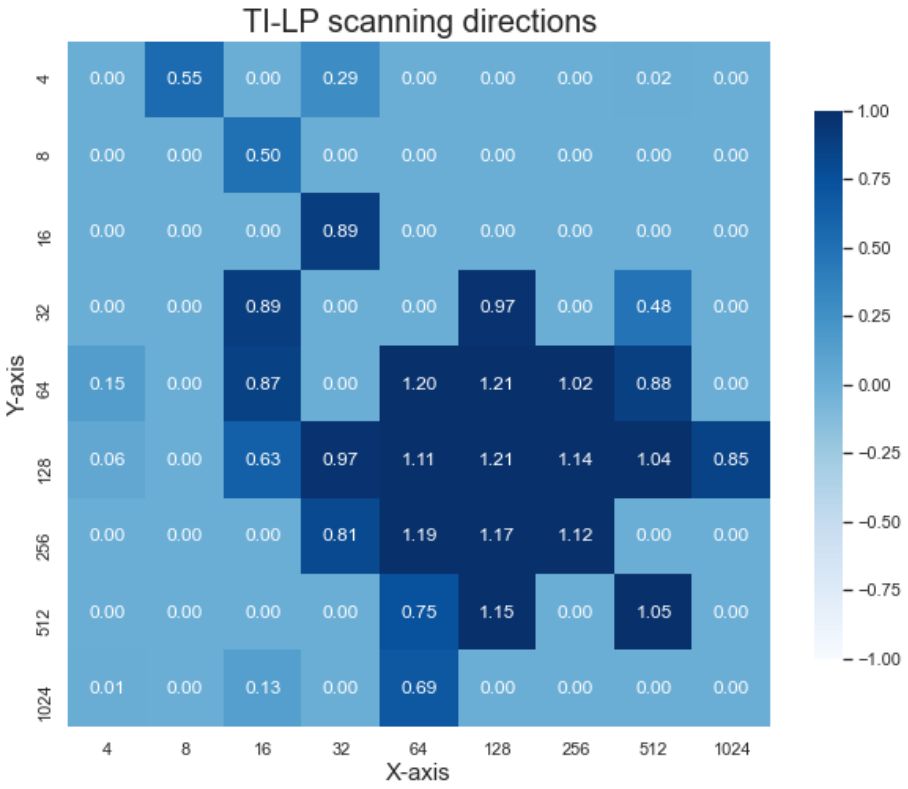

Figure 5.20: Normalized performance, TI-LP scanning directions

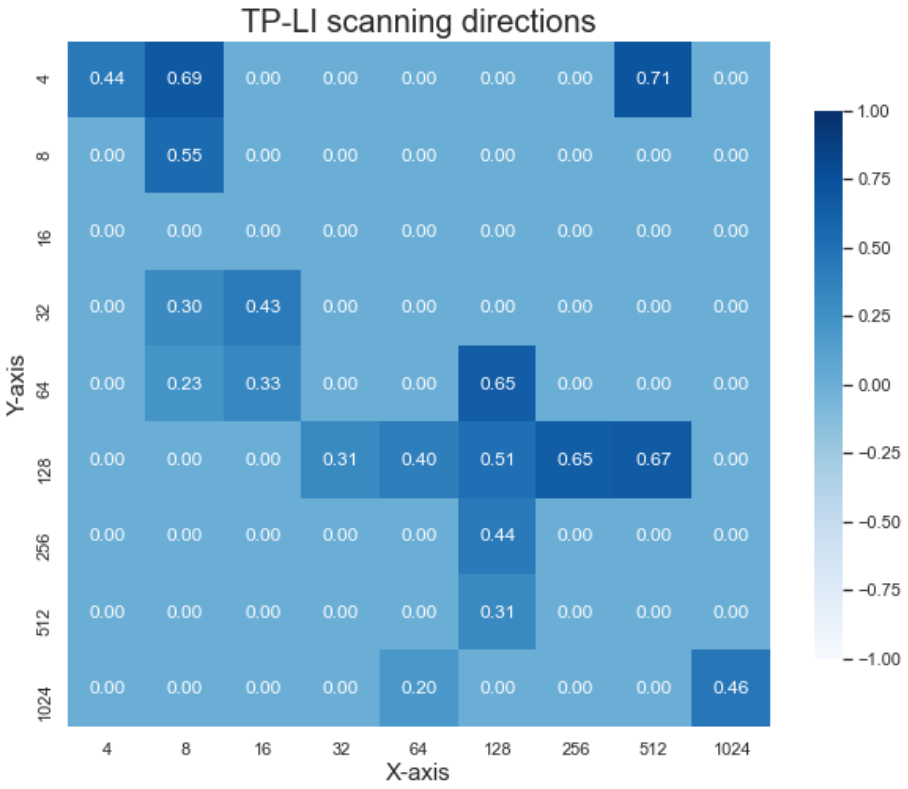

Figure 5.21: Normalized performance, TP-LI scanning directions

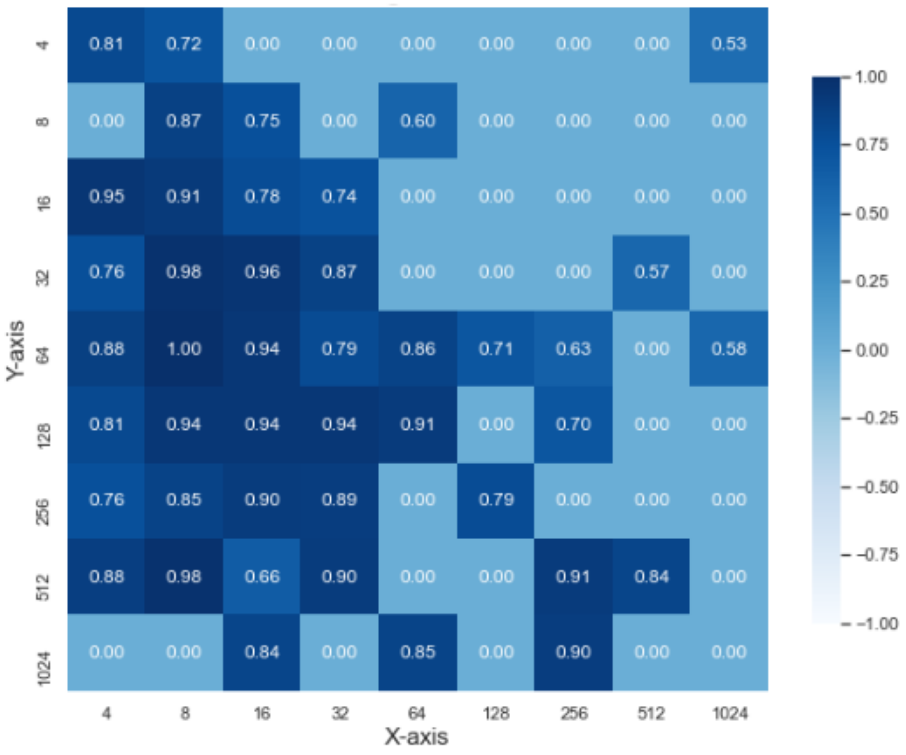

Figure 5.22: Normalized performance, Diamond tiling

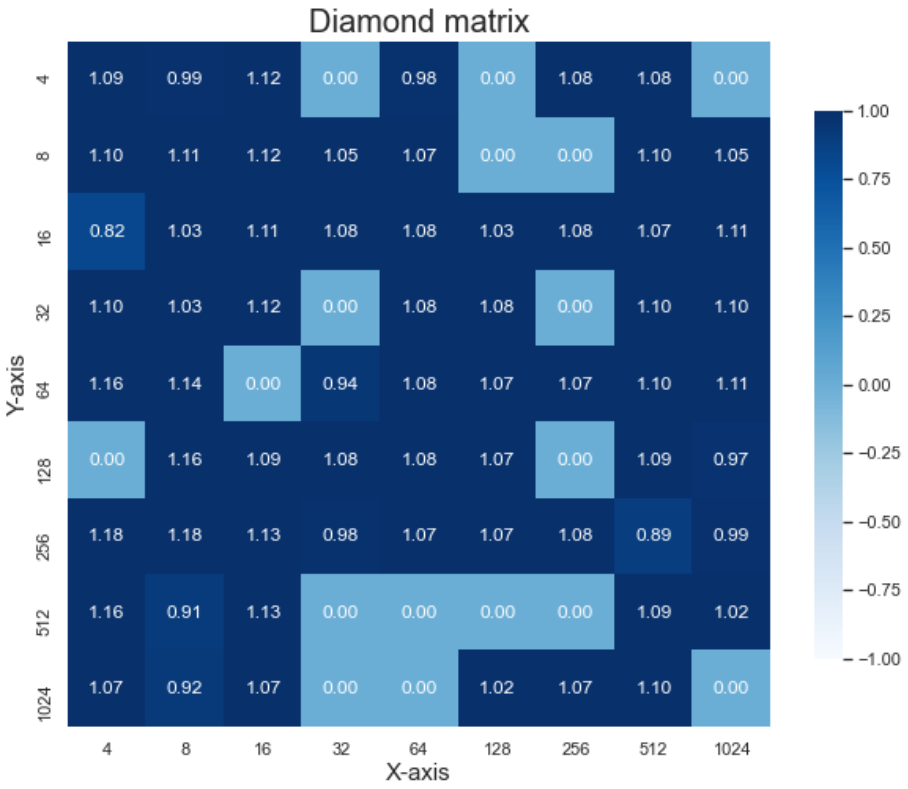

Figure 5.23: Normalized performance, Rectangular tiling

Chapter 6

Tiling parameter prediction

Résumé

La transformation de tuilage de boucles est une technique d'optimisation de code puissante pour augmenter le niveau de localité et de parallélisme des données dans un code donné. Comme il a été mentionné dans le chapitre 5, cette transformation a de nombreux paramètres (taille des tuiles, forme, directions de balayage) et hyper-paramètres (nombre de cœurs de l'architecture cible, caractéristiques dépendantes de l'architecture). Le choix inadapté d'un seul d'entre eux peut entraîner de mauvaises performances. Cela rend le problème de la sélection des paramètres de tuilage très compliqué.

Ce chapitre se concentre sur deux contributions importantes. Tout d'abord, nous étudions le compromis entre la complexité du modèle et le bénéfice potentiel que nous pouvons en tirer. On pourrait s'attendre à ce que plus le modèle est complexe, meilleur est le code optimisé que l'on peut obtenir, mais en fait la qualité des prédictions peut être dégradée par rapport à un modèle relativement simple.

Notre deuxième contribution concerne la prédiction d'un ensemble plus avancé de paramètres pour la transformation de tuilage. Nous ne limitons pas nos prédictions uniquement à la taille des tuiles. Notre pipeline d'apprentissage automatique est capable de prédire 1) la forme des tuiles 2) les directions de balayage intra-tuiles 3) les directions de balayage inter-tuiles 4) la taille des tuiles pour un code donné.

Introduction

Loop tiling transformation is a powerful code optimization technique to increase the level of data locality and parallelism in a given code. As it was mentioned in chapter 5, this transformation has many parameters (tile size, shape, scanning directions) and hyperparameters (number of cores of the target architecture, architecture-dependent characteristics). The poor choice of even one of them can lead to poor performance. This makes the problem of selection tiling parameters very complicated.

This chapter focuses on two significant contributions. Firstly, we investigate the trade-off between the model complexity and the potential benefit we can gain. One would expect that the more complex the model, the more optimized code we could obtain but the quality of the predictions could be worse compared to a relatively simple model.

Our second contribution relates to the prediction of a more advanced set of parameters for tiling transformation. We do not restrict our predictions just to the tile size. Our Machine Learning pipeline is able to predict 1) tiling shape 2) intra-tile scanning directions 3) inter-tile scanning directions 4) tile size for a given code.

This chapter is organized as follows. Section formulates which models would be used in this chapter and for which tasks. Section 6.2 introduces the features that would be used for each model presented in Section . Section 6.3.1 details the process of each model from an ML perspective. Sections 6.3 and 6.4 present the results for the two mentioned contributions. Section 6.3 presents the differences between the quality of the predictions for classical pipelines. Section 6.4 highlights our progress on the prediction of advanced tiling options (tiling shapes, scanning directions).

6.1 Problems of interest

In the context of this thesis, we build Machine Learning models for the following cases:

- 3D Cubic partitioning applied on 3D nested loops.
- 3D Parallelepiped partitioning applied on 3D nested loops.

Existing research [1], [45] does not give any insight into which tile partitioning to use for ML predictions. It gives general recommendations on how to build a pipeline for each of them. Our work investigates each pipeline and gives a recommendation on the partitioning to choose for ML modeling. The difficulty lies in the trade-off between model complexity and its potential benefit. For instance, on one hand, the 3D parallelepiped model is potentially the most beneficial as was stated in the previous chapter, but also the most complicated from a Machine Learning point of view, since we need to predict three values (higher probability for misprediction). The complexity of the model could have a negative impact on the quality of the predictions, and hence this pipeline might not be the most suitable for code optimization. Moreover, there are many ways how to model this problem from a Machine Learning perspective. We analyze each way of modeling and discuss the most appropriate.

Our second contribution towards tiling prediction targets the prediction of an extended set of parameters - 1) Tile size 2) Intra-tile scanning directions 3) Inter-tile scanning directions 4) Partitioning shape. Our target model can predict all these

parameters. We argue that the consideration of these parameters for kernels that have uniform data dependencies could help the optimization. For this contribution we considered the following partitioning matrices:

- 2D square partitioning applied on 2D nested loops.
- 2D diamond partitioning applied on 2D nested loops.

It should be noted that our methodology allows adding other types of tiling partitionings. We stopped just on mentioned above in the context of this thesis.

6.2 Feature space design

Machine Learning techniques draw conclusions based on some characteristics of a phenomenon under consideration. Usually, it is a vector of fixed size, and each value represents relevant properties of the phenomenon [90]. A choice of irrelevant features implies that the characteristics of the phenomenon are not sufficiently represented. Machine Learning algorithms cannot generalize based on them. It would lead to unsatisfactory results and poor performance. Therefore, the design of a suitable function space comes to the fore.

We distinguish two main concepts which are crucial for the selection of tiling parameters. These are 1) the data dependencies, 2) the array access functions, and information about the iteration domain. Data dependencies define the legality of the tiling and motivate the use of different scanning directions [68].

Array access functions associated with the iteration domain characterize the spatial and temporal data locality and the parallelization/vectorization opportunities [1], [45]. Array access functions are important criteria to choose the correct tile sizes.

6.2.1 Encoding of dependencies

Encoding dependencies is not an easy task. The main problem is that although the concept of data dependencies is well-defined, they can be represented or approximated in many ways. Moreover, the information about data dependencies needs to be transferred to a vector of a fixed size for ML purposes. To find the proper level of abstraction, we investigate research about the suitable abstractions for data dependencies to apply tiling.

Data Dependence Abstraction for Tiling

Yang et al. [69] defined the minimal abstraction for a loop transformation. The dependence cone turned out to be the minimal abstraction to legally apply the tiling

transformation, which means that among the studied abstractions, it is the smallest abstraction that is enough precise to check whether the transformation is legal. In our study, we encode this abstraction into a feature vector for Machine Learning issues. The dependence cone is the convex hull of the set of points that are a positive linear combination of dependence distance vectors. More formally, the dependence cone is defined as $DC(L) = \{v = \sum_{i=1}^{k} \lambda_i d_i \in Z^n | d_i \in D(L), \lambda_i \geq 0, \sum_{i=1}^{k} \lambda_i \geq 1\}$, where $D(L)$ is the set of all distance vectors d_i in loop nest L.

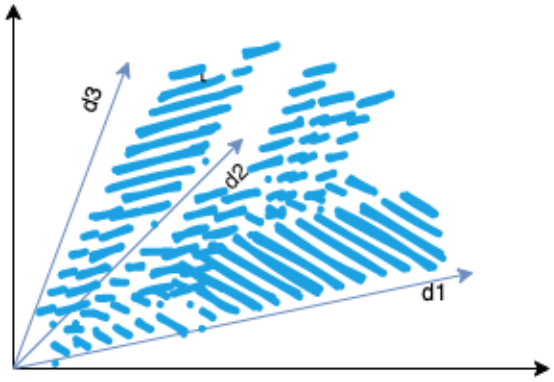

Figure 6.1: Dependence cone

Figure 6.1 shows an example of dependence cone for a code with **d1**, **d2**, **d3** dependence vectors. Note, that vectors **d3** and **d1** form the extreme rays of this cone.

Abstractions to encode data dependencies in our ML model

We use three abstractions to encode the data dependency information. We illustrate them in the example of Listing 6.1. The tiled version of this code with TP=LP scanning directions is presented in appendix D.

Listing 6.1: Original code

```
1 for (int i = 2; i < 1022; i++)
2   for (int j = 2; j < 510; j++)
3     A[i][j] =  A[i-2][j-1] + A[i-1][j-1] + A[i-1][j-2];
```

1 - Dependence cone

The dependence cone is the minimal abstraction to verify the legality of the loop tiling application. The dependence cone can be represented as a set of extreme rays. The first abstraction encodes the areas of the iteration space where we observe these extreme rays. For each extreme ray, we compute its angle with the x-axis in degrees. Here, x represents the first dimension of the dependency vector.

Our abstraction to encode the information on the extreme rays is a vector of fixed size (8 for 2-dimensional iteration domain). Let's describe the elements of the vector from 0 to 7. The i-th element of this vector gives how many extreme rays of the dependence cone form the angle θ with the x-axis, such as $\theta \in [45\times$ i; $45 \times$ (i+1)).

The pseudo-code of this mapping function is given in 6.2. The illustration is given in Figure 6.2. The blue vectors correspond to the extreme rays and are located in the first and second sub-areas.

Listing 6.2: Encoding of the extreme rays. Python-like pseudo code.

```
dependence_cone_encodings = [0,0,0,0,0,0,0,0]
dependence_cone = [22.5, 67.5] #angles in degrees

for extreme_ray in dependence_cone:
    encoding_index = math.floor(extreme_ray/45)
    dependence_cone_encodings[encoding_index] += 1

# dependence_cone_encodings = [1,1,0,0,0,0,0,0]
```

The mapping function for 3-dimensional iteration domains is pretty similar, except that it goes through the coordinates of two polar angles for the dependency vectors instead of one. The vector summing these types of vectors is of size 32 (c.f. section 6.2.3).

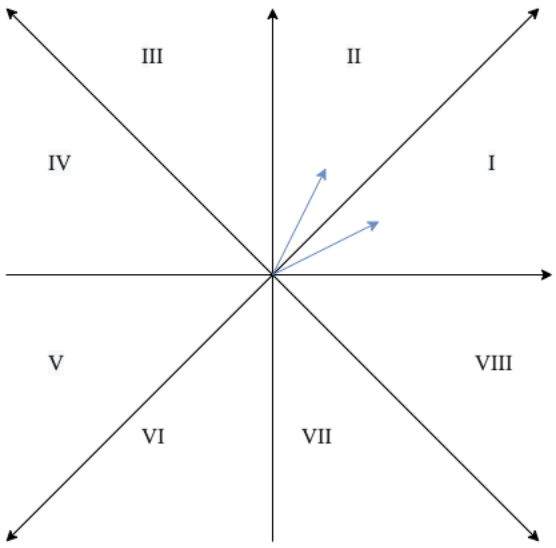

Figure 6.2: Encoding of the extreme rays

2 - Summation vector

The second abstraction related to the data dependencies is a vector which is the sum of all data dependency vectors. We extract a unique value - the angle of this vector with the x-axis. This information is required because this vector is used as a basis vector for some scanning directions ($\mathtt{T}_{Parallel}$ and $\mathtt{L}_{Parallel}$ directions). In the example 6.1, the sum of three dependence vectors results in vector (4,4), which makes an angle of 45 degrees with the x-axis.

3 - Encoding of uniform data-dependencies

In the same way, we encode the extreme rays 6.2.1, we encode the uniform data dependence vectors. Our abstraction to encode the information on the uniform dependencies is a vector of fixed size (8 for 2 dimension iteration domain). The i-th element of this vector gives how many uniform data dependencies form the angle θ with the x-axis, such as $\theta \in [45\times$ i; $45 \times$ (i+1)).

This vector for Example 6.2 equals [1,2,0,0,0,0,0,0].

6.2.2 Encoding of the iteration domain

The iteration domain can have an impact on the prediction of all the parameters considered. We propose to encode it with two simple features: 1) the number of iterations for each loop and their total 2) the presence of constant loop bounds for all loops.

The first feature vector is a vector of the size that equals the number of nested loops + 1. The first value represents the number of iterations for the first nested loop (NaN, if it is unknown, e.g. variables of functions), the second value represents the number of iterations for the second nested loop, etc. The last value represents the total number of iterations for the loop nest. This vector, for example, 6.2 equals [1020, 508, 518160].

The second feature vector is a vector of the size that equals the number of nested loops. The first value represents the presence of constant loop bound (1) or their absence (0) for the first nested loop, etc. This vector, for example, 6.2 equals [1,1] since the loop bounds are constant.

6.2.3 Generalization for 3-D case

In the proposed encoding of the dependence cone, data dependencies can be easily generalized for 3-D tiling and for higher dimensions. Let θ be the angle between a data-dependence vector/extreme ray of a dependence cone and the Z-axis. And ϕ is the angle between the projection of the same data-dependence vector/extreme ray to the XY plane and X-axis. Let V denotes the vector, which contains all the subareas of 3-D space to be encoded. $\mathrm{V} = \langle v_1 \quad v_2 \quad ... \quad v_K \rangle$

where v_i is a particular subarea of 3-D space, $i \leq K$, K is the number of chosen subareas of 3-D space (e.g. we have experimented that 32 is a good choice). Then we can define the mapping function that assigns the subarea of the 3-D space to each value of θ and ϕ angles. Actually, this mapping function could be considered as the mapping from polar coordinates to the subareas of 3-D space.

$f(x) :< \phi, \theta > \rightarrow v_i \in V$

In the 3-D case, we need also two angles θ and ϕ to encode the information about the summation vector.

6.2.4 Encoding of array accesses

There is a lot of research investigating the proper encoding of array accesses.

Yuki et al. [1] investigate the problem of automatic tiling selection using machine learning approaches. The authors consider cubic tiling on three nested loops with 2D data. They describe each loop according to the references inside the arrays. Each array reference can either benefit from the advantages of spatial locality or not, or be constant in the innermost loop. Yuki et al. also mark each reference as a read or write reference. However, this approach could be easily applied to loops of any dimensionality. For instance, if we consider MM kernel from example 5.1, then from Yuki perspective it has the write and write references C[i][j] that are invariant for the innermost loop. This kernel also contains A[i][k] reference that benefits from spatial locality and B[k][j] that does not.

Liu et al. [45] propose a slightly similar approach. The key difference is that the construction of the feature space can potentially contain loops of any fixed depth and data of any dimensionality.

In the context of this Ph.D., we use the concatenation of two vectors to properly encode the array accesses. For the 2D case, Yuki et al. [1] features contain 4 values, and Liu et al. [45] contain 3 values. The concatenation vector contains 7 values. Our experiments showed that this approach gives the best results.

6.2.5 Note on the feature space design

Table 6.1 summarizes the features appropriate to each model. Our experiments show that data-dependency information is not useful for the rectangular/parallelepiped-shaped partitioning for the original scanning directions. However, this information could be crucial for shape and direction predictions.

6.2.6 Encoding CFG et DDG

We presented the features that were used to achieve the best results in this Ph.D. in 6.2.1 - 6.2.4. However, we have also conducted research on other feature representations that have not achieved the same level of performance but are worth mentioning

	Encoding of array accesses	Encoding of data dependencies	Information about iteration domain
3D Cubic Tiling	+	-	+
3D Parallelepiped Tiling	+	-	+
Shape + direction + tile size prediction model	+	+	+

Table 6.1: Feature spaces

as they appear to be a promising direction for extending our work. Machine Learning projects tend towards models that are as general as possible and do not focus on a very narrow issue. This is reflected in the feature space construction. The trend is to use representative learning from raw data. It enables us to find automatically the appropriate feature representation of a concept under observation. The emerging direction that finds applicability in many domains is the use of embeddings.

Our point of interest in this subsection is the embedding of graphs. We introduce a method to embed the Data-Dependence graph (DDG) and the Control-Flow Graph (CFG) for a given code. We consider the concepts they integrate (data dependencies and control flow) as crucial for many code transformations.

We believe that the representation of graphs is a proper level of abstraction to describe important code properties and extract their features from them. Embedding raw code can be too general and capture a lot of unnecessary properties (e.g. everything related to coding writing style). Obviously, this information does not bring clarification on the prediction but weakens the model with useless inputs.

On the other hand, the "concentration" of useful information in the graphs describing the code is very high. It also gives the flexibility to construct graphs that concern different code aspects (e.g. data-dependencies of a given code or control flow between basic blocks) and to embed information from them.

Graph construction

We use PIPS Framework [33] to extract information about dependencies between array references and control flow between basic blocks or any required abstraction (e.g. statement in a code that is not a basic block in classical definition). This information is provided in explicit text format by PIPS. We construct the graph using NetworkX library [18] based on the text output. The main idea is to map text information from PIPS output to NetworkX objects (vertices, edges, labels).

Control-Flow Graph

In our study we consider the CFG to be a directed and unweighted graph with labels on the nodes and we focus on the flow of control just inside a loop nest.

More formally, let $G = (V, E, \phi, \rho)$ be a graph, where V is a finite set of vertices, E a set of edges. V corresponds to the set of basic blocks in the code, E illustrates the existence of control flow between blocks.

$\phi : E \to \{(x, y)|(x, y) \in V^2 \wedge x \neq y\}$ is a mapping of each edge to an ordered pair of vertices. $\rho : V \to \{origin, sink, if, else, statement, for\}$ is a mapping of each vertex to a set of possible labels. We take into account each statement independently, that slightly changes the classic definition of a basic block. We consider each statement as a basic block.

Thus, we have two artificial concepts *origin* and *sink* in our graph that correspond to the entry and exit points of the loop execution. Note, that we consider for-loops with only one exit point (no return, break, or function call in the loop nest). Hence, all other concepts that we can observe are labeled *for*, *statement* or $< if, else >$ in case of a conditional statement.

Data-Dependence Graph

Our construction of the Data-Dependence Graph describes the dependencies between the data references in the code. We define DDG as a directed, unweighted graph with labels on edges.

More formally, let $F = (V, E, \phi, \nu)$ denote a DDG, where V is a finite set of vertices and E is a set of edges. V corresponds to the data references. E illustrates the existence of data dependency between references. As for CFG, ϕ is a mapping of each edge to an ordered pair of vertices, and $\nu : E \to DDV$ is a mapping of each edge to a corresponding dependence-distance vector.

Generation of synthetic data from CFG and DDG

The Control-Flow Graph is a relatively simple object to synthesize. We managed to create a synthetic CFG generator. Any generated CFG can correspond to a legal, compilable, executable program. The main steps are:

- Synthetically generate a training set λ_1 that contains 10.000 CGFs
- Train embedding model M_1 using training set λ_1. M_1 takes an arbitrary CFG and returns an embedding vector.

The Data-Dependence Graph is a much more complex object. There is no guarantee that there is a potential program corresponding to a randomly generated dependence graph. Hence, we used a synthetic code generator [70] that uses data

dependencies as a high-level abstraction for the code, to get valid programs. The main steps are:

- Synthetically generate training set Λ_2 that contains 10.000 kernels
- Extract DDG to λ_2 from each kernel in Λ_2.
- Train embedding model M_2 using training set λ_2. M_2 takes an arbitrary DDG and returns an embedding vector.

We investigate techniques proposed by library NetworkX [18] to embed these graphs.

Conclusion on the encoding of CGF and DFG

The proposed features seem to be an interesting concept to capture code characteristics regarding its control flow and data-dependence graphs. They could be used for any code transformation, not only tiling. However, they are not precise enough as hand-crafted features to provide a satisfying level of performance.

6.3 Tiling predictions

Existing ML tile size selection models [1], [45] propose many different ways to predict tile sizes. Both papers mention that modeling the tile size selection could be done by predictions of tile size directly, or by prediction of speedup/execution time. Some papers such as [102] also consider the prediction of the speedup for a given sequence of transformations. It gives us the intuition that modeling could be performed in many different ways. Moreover, the papers that target tiling transformation consider different partitioning matrices, for instance, [1] considers only cubic, [45] considers parallelepiped tiling. To our best knowledge, there is no generalization among the different approaches for the tile size selection problem. This section overcomes these limitations. We do a generalization through the cubic and parallelepiped tiling, considering two techniques to model this issue in an ML pipeline. The first technique predicts the tile size directly, the second predicts the speedup based on program characteristics and potential tile size. The first approach tends to memorize the best patterns of tiling and choose them as labels, the second approach learns to memorize all possible tiling patterns. We consider it important to investigate which one is more applicable for this issue. The tile size that maximizes the speedup is taken as the optimal one. Kernels of interest for this section do not have uniform data dependencies. Kernels under investigation represent a permutation of the vector of indices in the array access functions. We could have dependencies, such as a reduction in one or two dimensions.

6.3.1 Machine Learning modeling

We investigate two approaches to choose the best tile sizes: direct tile size prediction and tile size prediction based on potential **absolute** speedup prediction.

The pipeline of our first approach is shown in Figure 6.3. It is done in a straightforward manner. We learn the regressor to predict just the optimal tile directly. Only optimal tile sizes are taken for each execution.

The second approach is a bit more tricky and is illustrated in Figure 6.4. The idea is to predict the speedup for a given code based on tile size and a feature vector from section 6.1. We sample all possible tile sizes for a given kernel and predict the speedup for each tile size. We select the tile size with the highest predicted speedup. It should be noted that the speedup we predict **cannot be used as an accurate estimate of the real speedup**. There are many different factors that impact the speedup and we do not take them into account, but the predicted "speedup" helps differentiate the impact of the tiling parameters on real speedup. This approximation helps to choose the more appropriate combinations of parameters.

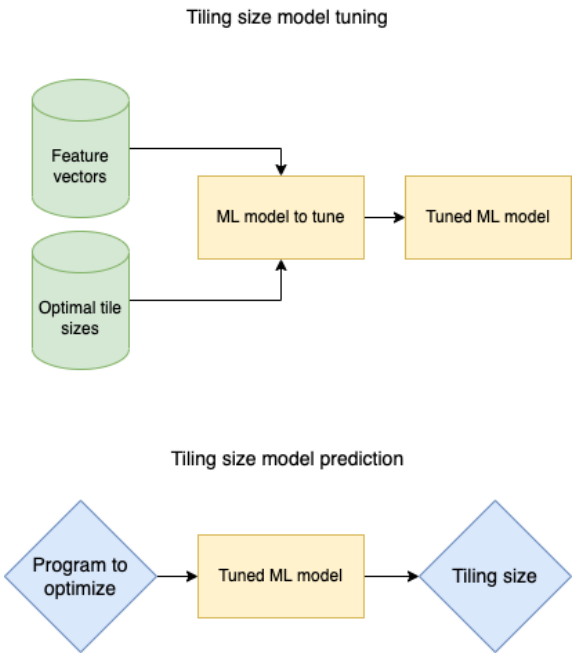

Figure 6.3: Tile size prediction pipeline

6.3.2 Machine Learning models

The problems that we want to model relate to the regression problem. The idea is to predict continuous numerical value.

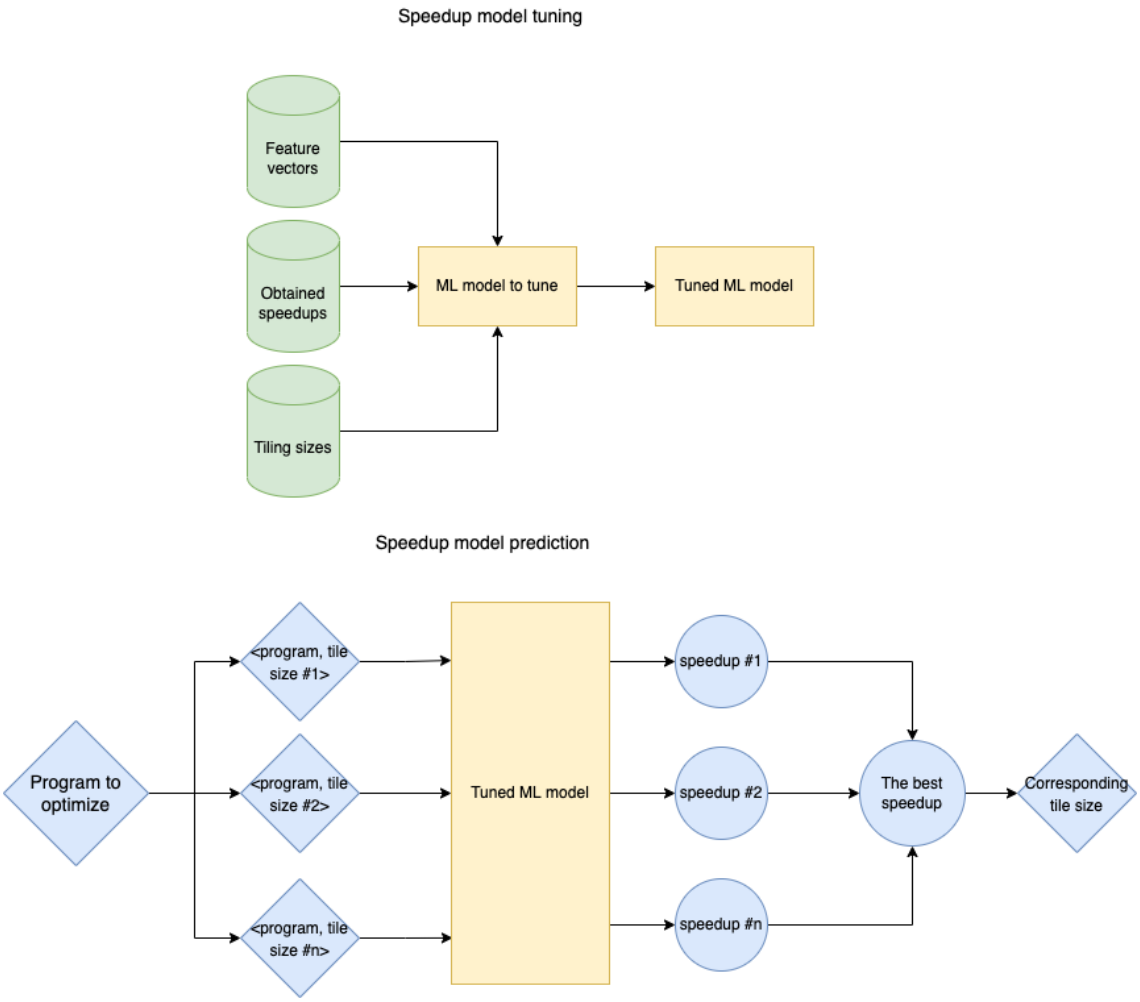

Figure 6.4: Tile size prediction based on speedup prediction pipeline

The description of the ML models was given in chapter 2. In the context of this thesis, we consider the following algorithms for our experiments.

- Gradient boosting regression [143]
- Random Forest regression [162]
- Linear regression [159]
- Regularized regressions [158]
- Linear support vector regression [22]

We can distinguish two groups of models: linear ones and non-linear ones. The first class models the relationships between input and output based on the linear predictor function. The function used for the second class might be non-linear.

Non-linear machine learning models (such as Gradient Boosting, and Random Forest) are more appropriate to our problems. They show better performance than linear ones since the distribution of the targeted outputs is not a linear function of the inputs. For instance, we can easily demonstrate that the tile size or the speedup is not linear. For instance, for some kernels we can apply vectorization, it makes sense to increase the tile size of the innermost dimension until some limit. On the other side, it may be the case that the increase of the innermost dimension leads to the degradation of data locality. The model should find the trade-off between all the factors that impact the performance, these factors may contradict each other.

Note, that we do not apply Deep Learning models for this task. We consider that we have a relatively small dataset and we can easily overfit our model. The number of samples is hundred of thousands but the number of unique programs is just 1250. We argue that there are prerequisites for over-fitting [7].

Stacked classifier

The best single regressor (e.g. Random Forest or Gradient Boosting) may lead to acceptable performance but could be easily improved by stacking different regressors [20]. The idea is to have a set of independent first-step regressors that outputs the predictions of a given kernel independently. The second-step regressor takes the outputs of the first-step regressors as features and produces the final prediction. Although some weak regressors might be included in the first-step regressors the main idea is to use regressors that have not correlated predictions. It will lead to the decrease of variance of the model [17] and hence better predictions.

In our experiments, SVR regression [22] and Gradient boosting regression [143] showed less correlated predictions. We decided to use them as first-step models. Another Boosting model aggregates their predictions. It was chosen based on an empirical search among all potential candidates. The pipeline of the final regressor is shown in Figure 6.5.

6.3.3 ML metrics

Two classes of metrics could be distinguished: proxy metrics and business metrics [19]. Proxy metrics are used during the process of learning, while business metrics show the correspondence of our target predictions to the target goal.

Proxy metrics

Mean squared error (MSE) is used as a Loss Function for the regression.

$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i^* - y_i)^2$, where y_i is the ground-truth value of the optimal tile size of the i-th data sample, and y_i^* was predicted by our ML model. Where n is the number of data samples.

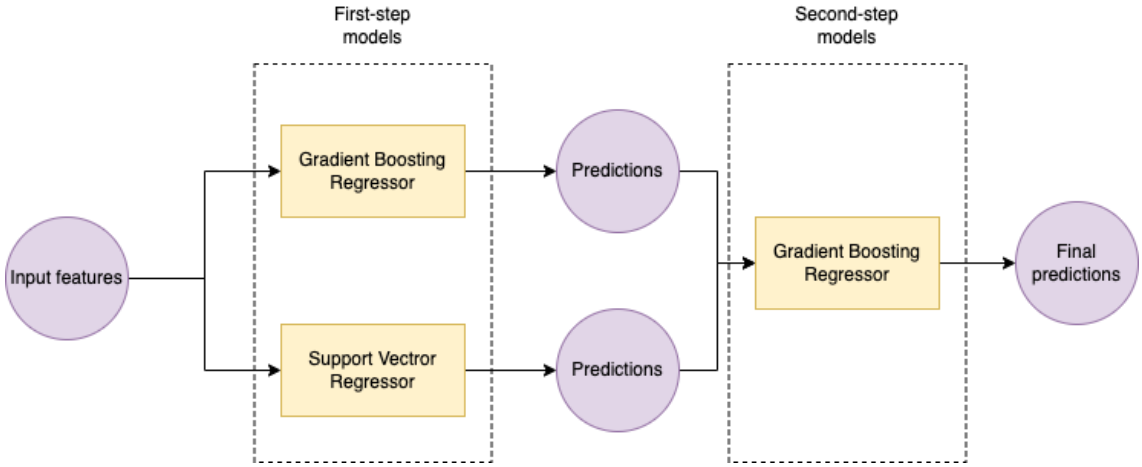

Figure 6.5: Stacked regressor

We use this metric for ML modeling since optimal tile sizes are distributed near the same neighborhood (empirical conclusion, check Appendix A), and we want to penalize our model if it predicts tile sizes that are far from the global optimum.

This cost function has several drawbacks. It does not provide explicit information about our target goal - fast code execution. The losses provide no information to the programmer on how the generated code would perform in terms of execution time. Moreover, it does not provide insights into the parallelism of the architecture and the profitability that we can gain from the transformation.

That is why we introduce the second-step metric showing how far we are from the most efficient generated code. We use the relative speedup metric described in the next section.

Business metrics

Business metrics should reflect the quality of the code generated on real examples. They should not be proxy metrics but actually correspond to our target. We use two metrics to evaluate the quality of the generated code: Absolute speedup and Relative speedup.

We introduce two definitions:

$RS_i = \frac{speedup(\hat{y}_i)}{speedup(y*_i)}$, where $speedup(\hat{y}_i)$ gives the speedup obtained after tiling the code with the predicted parameter. And $speedup(y*_i)$ gives the speedup found by the Autotuner [32] An average relative speedup can be computed with

$RS = \frac{1}{n}\sum_{i=1}^{n} RS_i$, where n is the number of samples to evaluate.

$AS_i = \frac{speedup(\hat{y}_i)}{speedup(y+_i)}$, where $speedup(\hat{y}_i)$ gives the speedup obtained after tiling the code with the predicted parameter. And $speedup(y+_i)$ gives the speedup of the initial code without optimization. An average absolute speedup can be computed with $AS = \frac{1}{n}\sum_{i=1}^{n} AS_i$, , where n is the number of samples to evaluate.

The drawbacks of these functions are that they are very sensitive to outliers. RS and AS of a tiled code with tile sizes in the same neighborhood may be different due to factors that are not possible to take into account using existing feature spaces (e.g. load balancing among threads).

Moreover, they do not have derivatives; they are piecewise-defined functions. Hence, it is not applicable to be used for training our ML models. Thus, each metric is more appropriate for the stage where it is used. The combination of the two provides a more correct way to implement the training process and evaluate the results.

6.3.4 Experimental setup

The experiments were run on Intel® Core™ i7-8650U 4C/4T @1.90GHz with capacity caches of L1: 32KB, L2: 256KB, L3: 8192KB and 32GB DDR4 DIMM RAM, Phys. cores: 4, Compiler: GCC 5.4.0, Number of iterations per dimension in the test set: 1024, Number of Threads: 4, Opt. level: -O3

6.3.5 Training/Validation/Test sets

We generated about 1500 unique programs with our generator [70] and labeled them with LOCUS [32]. Firstly, we labeled them with parallelepiped and cubic partitioning. In total, this generates around 155.000 executions for parallelepiped tiling and around 12.000 for cubic one. We removed duplicates in terms of Yuki et al. [1] feature space and got eventually around 1000 unique programs that have different features. Executions of 900 of them were taken to form our training sets. Executions related to the other 100 programs were taken for validation. 9 well-known kernels were taken for the test set to evaluate the quality of our predictions on not generated data.

Data labelling

Data labeling is a very time-consuming process. For each kernel, we generate up to 1000 code variants (tiled codes with different tile sizes and options) and execute them to assign labels for the regression problem. The overall time required for this process is equal to the number of repetitions $\times$ number of variants $\times$ number of kernels $\times$ (the time to generate a variant + the time to execute the variant). For us, it took around 2 months to label all the required data for the experiments in this chapter.

Speedup interpretation

There are several factors that impact the performance and could be worsened or improved by tiling transformation. They are data locality, load balancing across

threads, level of parallelism, and some others. The ML model tries to entangle their complex interaction and chooses the tile size that improves the most factors that affect the performance.

The other factor that worsens the performance is complex loop bounds after tiling transformation. The loop bounds might be computationally expensive and the optimized kernel would perform worse than the baseline version. This factor could be amortized by the number of iterations.

6.3.6 Validation set results

This subsection provides experimental results for the validation set that consists of executions from 100 kernels. The results are presented in figures 6.6 and 6.7 and in the table 6.2.

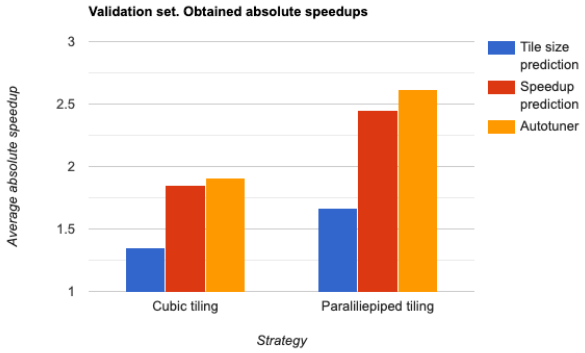

Figure 6.6: Absolute speedup

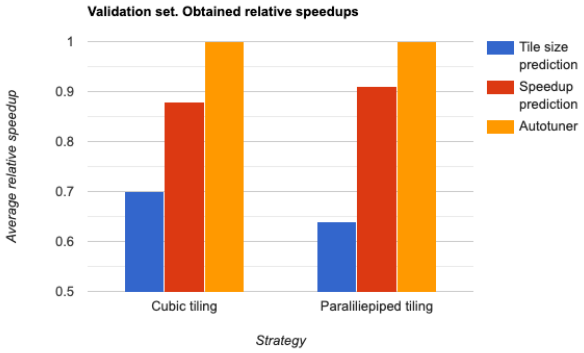

Figure 6.7: Relative speedup

Results on the test set show that

- The parallelepiped partitioning is a more profitable transformation for the targeted kernels. The average maximal speedup is 2.62x versus 1.91x for cubic partitioning.

- The strategy to select tile size based on the predicted speedups overperforms the strategy to predict tile size directly. It achieves 0.91 and 0.88 of average relative speedup versus 0.7 and 0.64 for the direct predictions. Absolute speedups are also significantly higher.

- The strategy to predict parallelepiped partitioning based on the predicted speedups seems the best choice on the validation set. It overperforms all other strategies.

Strategy	Absolute speedup	Relative speedup
Cubic tile size prediction	1.35	0.7
Parallelepiped tile size prediction	1.67	0.64
Cubic tile size selection based on speedup prediction	1.85	0.88
Parallelepiped tile size selection based on speedup prediction	2.45	0.916
Cubic tile size selection with Autotuner	1.91	1
Parallelepiped tile size selection with Autotuner	2.62	1

Table 6.2: Validation set. Results

- The strategy to predict parallelepiped partitioning based on the predicted speedups seems the best choice on the validation set. It overperforms all other strategies.

- The strategy to predict based on the predicted speedups is more applicable for this issue. The models take data from all tile size (not just the best ones) for the training. Models behave in more robust way when they do not know how to optimize given kernel in the best.

6.3.7 Test set results

This subsection provides experimental results for the 9 known kernels that were not generated with our code generator. The results are presented in Figures 6.8 and 6.9 and in table 6.3. The relative speedups for the test set could be improved if we used relative speedup (not absolute) as a target value for the Machine Learning model that implicitly predicts the tile sizes (Section 6.3.1). Obviously, if we used RS as labels for our data, then the RS speedup on the test set would be greater. But our experiments show that choosing RS/AS as the label does not change (just on several percent) the results on average.

6.3.8 Test set representativeness

We consider that the programs we used in the training set are more representative than the kernels that we observe in the test set. We used a much wider range of parameters for the training set. They are presented in Table 6.4.

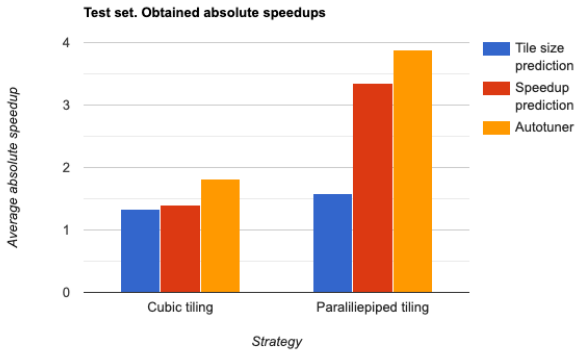

Figure 6.8: Absolute speedup

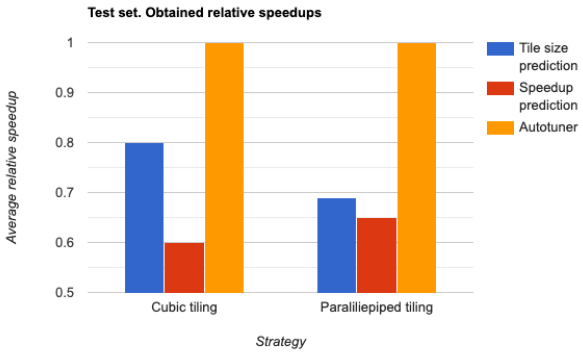

Figure 6.9: Relative speedup

Strategy	Absolute speedup	Relative speedup
Cubic tile size prediction	1.34	0.8
Parallelepiped tile size prediction	1.57	0.67
Cubic tile size selection based on speedup prediction	1.58	0.6
Parallelepiped tile size selection based on speedup prediction	3.35	0.64
Cubic tile size selection with Autotuner	1.81	1
Parallelepiped tile size selection with Autotuner	3.85	1

Table 6.3: Test set. Results

We compare the ranges of Yuki's features [1] in the two sets. We could mimic the training set to increase the performance on the test significantly but the value of the speedups was not the focus of this study.

Conclusion on test set results

Results on the test set shows that

- The parallelepiped partitioning is a more profitable transformation for the test set.
- Absolute speedups of parallelepiped predictions overperform the cubic ones. 3.35x and 1.57x versus 1.58x and 1.34x.

Ranges of features	Test set	Training set
# of writes with locality	0-1	0-2
# of writes without locality	0-0	0-2
# of reads with locality	1-4	0-10
# of reads without locality	0-2	0-11
# of invariant writes	0-2	0-2
# of invariant reads	1-3	0-11

Table 6.4: Test set vs Training set. Feature ranges

- Relative speedups are higher for direct predictions. Direct tile size selection achieves a good level of relative speedups on kernels that could not be improved a lot by tiling (e.g. gemm). The other strategy shows worse results on this type of kernel but maximizes the performance on kernels that are improved by tiling.

Figure 6.10 shows the values of relative speedups for the proposed strategy. It should be mentioned that the proposed model is able to capture the profitable patterns for kernels that benefit a lot from tiling (syr2k, syrk, mm) and does not capture properly for non-beneficial kernels (gemm, strsm).

6.3.9 Conclusion on our tiling predictions

In this section, we do a comparison of two groups of approaches. Firstly, we compare the predictions made for cubic vs parallelepiped tiling. Cubic tiling is a less beneficial transformation but could be easy to model and predict. Hence, we could potentially get better predictions due to the low complexity of the problem.

Secondly, we do a comparison of strategies on how to model the problem of tile size selection from a machine learning point of view. Both approaches are regression techniques, the first one predicts tile sizes directly [1], and the second approach predicts speedups [102] and chooses the tile size that maximizes the speedup.

We conclude that

- The parallelepiped partitioning is the most reasonable concept for ML modeling in our context. Predictions made on parallelepiped are up to 32% better on the validation set and up to 112% better on the test set in terms of absolute speedup.

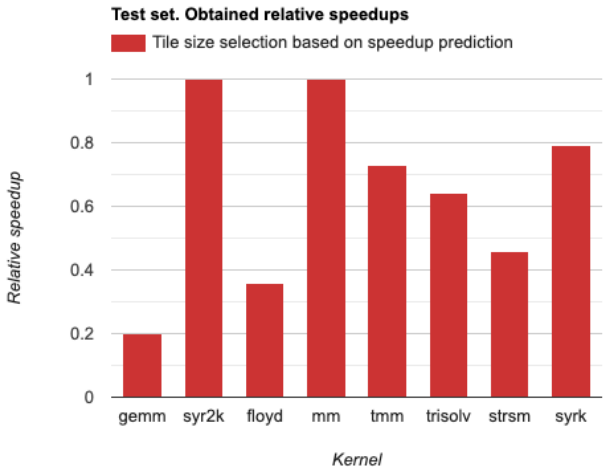

Figure 6.10: Test benchmark

- The tile size selection based on speedup predictions is the best strategy. We can improve predictions up to 47% on the validation set and up to 113% on the test set.

- Our predictions reach up to 91.6% out of the optimal ones found by autotuner on the validation set and up to 70% on the test set formed with real kernels.

6.4 Advanced tiling

State-of-the-art research for tile size selection using Machine Learning considers mainly kernels that do not have data dependencies [1], [45]. So the feature space design in the previous articles does not reflect the fact that the program may have data dependencies. Moreover, Machine Learning predictions are limited only to tile sizes for cubic [1] or parallelepiped [45] tile shapes. This chapter attempts to overcome these limitations. We show that taking into account scanning directions and tile shapes can improve the performance of kernels that have uniform data dependencies. We validate our methodology on 2-dimensional nested loops and show that it could improve our predictions for the kernels that have data dependencies.

6.4.1 Data collection

We used the code generator [70] to generate around 1000 synthetic kernels. The main concept that we targeted during the generation was the creation of uniform data dependencies. The generated programs had from 1 to 5 true uniform data

dependencies (constant write-before-read dependencies). We tried to generate kernels with the most different possible dependence cones. The methodology was to generate kernels whose second outermost loop could be parallel after the tiling transformation.

Autotuning process

We used LOCUS Autotuner [32] to label the data for two different settings. First, we asked the Autotuner to find the optimal combination of parameters for the square tiling considering 4 different scanning directions: TI-LI, TI-LP, TP-LI, and TP-LP. TS and LS scanning directions are respectively identical to TI and LI in the case of square tiling. Second, we asked the Autotuner to tune the same kernel with diamond tiling considering 9 possible scanning directions. Scanning directions related to the partitioning shape (TS, LS) have been added for -Intra and -inter tiles. That gives 9 combinations in total. All the collected executions and corresponding execution times were collected for the learning process.

Legality checks

We used a 4-step verification system to validate the correctness of our experiments. First of all, PIPS compiler [33] is used with the options allowing the generation of tiled codes only in cases where the transformation is legal. If it is the desired scanning directions that are not legal, PIPS returns a tiling with the initial scanning directions. Secondly, we always verified that the scanning directions of the generated code correspond to those requested. Third, we performed data dependence analysis to validate that the second outermost loop could be parallelized after the tiling transformation. Finally, we compare the checksum of the initial sequential code compiled without OpenMP and the aggressive compiler optimizations with the optimized tiled parallel version. If one of the kernels fails any of these tests, it is not taken as input to the learning process.

Data analysis

To our best knowledge, the consideration of additional tiling parameters has not been thoroughly analyzed in the literature. Moreover, it has not been shown that it could have a performance impact on a large set of programs. We try to overcome these issues in this subsection.

Figure 6.11 shows what percentage of generated kernels benefit from extended scanning directions for cubic or diamond tiling (any scanning directions). Our analysis shows that around 21% of the generated kernels benefit from this extended setting. Figure 6.12 shows the speedup distribution of the generated kernel compared to the same kernels auto-tuned with the original tiling settings. The mean

speedup across all beneficial kernels is around 6.5% compared to the original settings. We consider these results meaningful. First, we managed to generate around 200 kernels which perform better with the extended tile settings. This research topic has not been sufficiently addressed. Some kernels have been used as a reference of profit for extended tiling settings, such as diamond tiling code, but performance has rarely been measured. The high code generation overheads, due to more complex loop bounds, are often not mentioned. However, we show that despite the overhead, we can generate kernels that benefit from these parameters. We used the COLA-Gen autotuner [70] for that. These results justify that our generator is capable to generate the code of our interest.

Figure 6.13 shows the distribution of the best scanning directions. The initial basis was the best in 79% of cases. Unfortunately, the scanning directions related to the partitioning of the diamond tiling shape have never been the best in our experiments. It could be explained as not optimal generation strategy of the synthetic data or by large overhead to compute these scanning directions. The TP-LI scanning direction takes second place in terms of frequency of the best scanning.

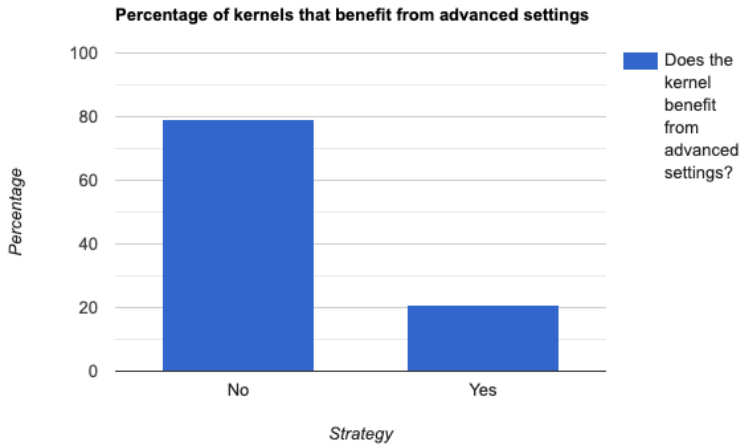

Figure 6.11: Amount of kernels that benefit from Advanced settings

6.4.2 Machine Learning modeling

Figure 6.11 concludes that we have a huge imbalance in our classes. The original tiling shape and scanning are the best options in almost 80% of cases. Hence, we think it reasonable to create a two-stage ML model that would consist of two parts. The first model will predict whether we consider only the original tile settings or the extended settings, the best option for the given code. The second model will predict the tile sizes if the answer is positive (not profitable for extended settings),

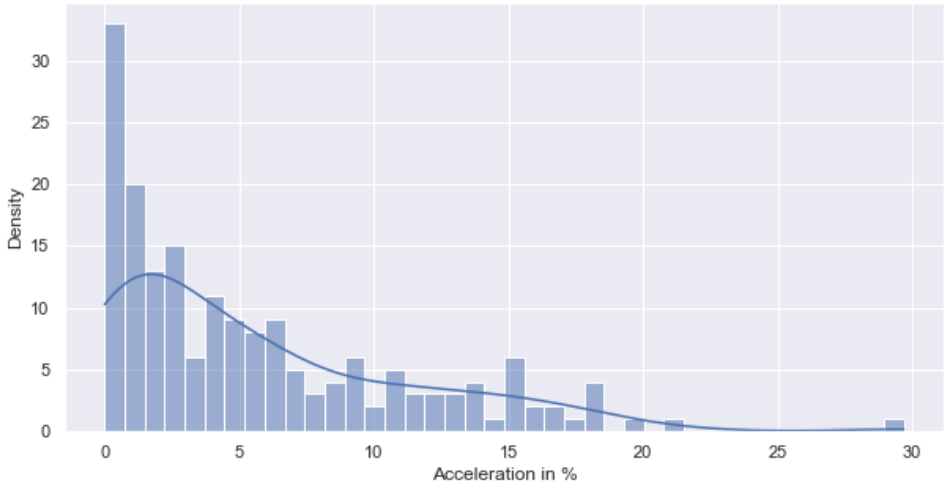

Figure 6.12: Acceleration distribution of kernels that benefit from advanced settings. Training set

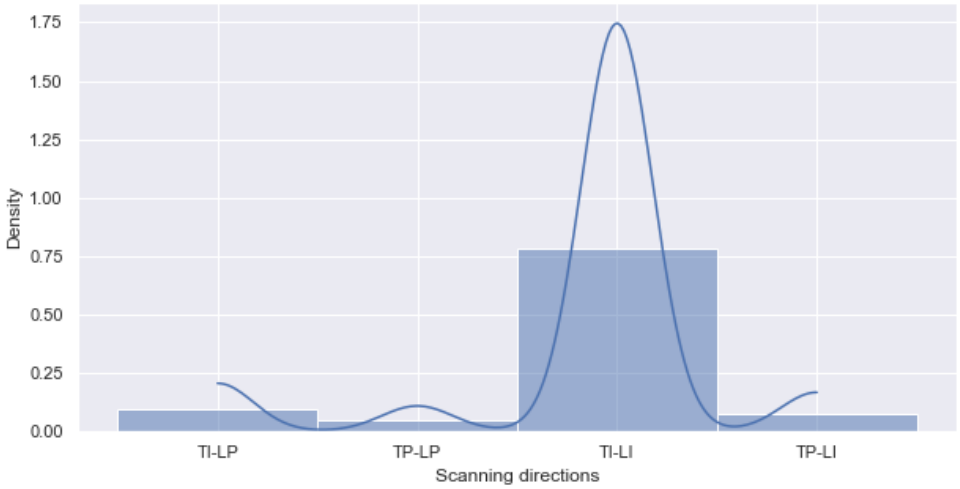

Figure 6.13: Distribution of the best scanning directions

or all the parameters for a given kernel (tiling shape+scanning directions+ tile size) otherwise. The pipeline is illustrated in Figure 6.14.

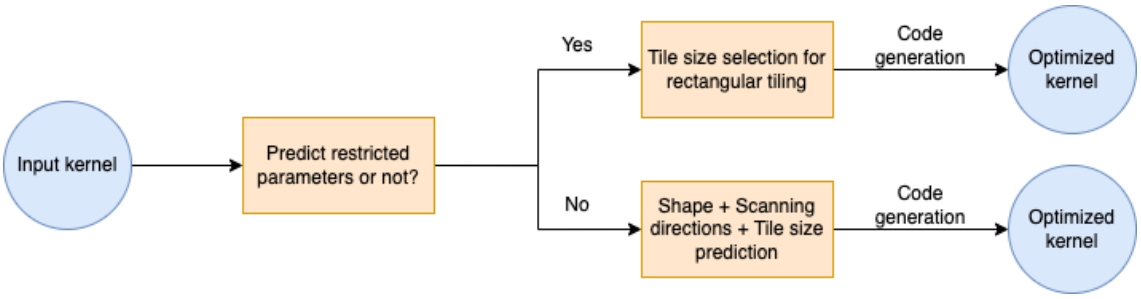

Figure 6.14: Prediction pipeline

Classifiers	ROC-AUC score
Random Forest	0.83
Gradient Boosting	0.84
Logistic Regression	0.82
Stacking	0.89
Majority class prediction	0.79
Random guess	0.50

Table 6.5: Comparison of different models

First-step model. Binary Classification

Our first-step model is a binary classification model. It takes the input features presented in Table 6.1 and predicts whether this kernel is potentially a good candidate for restricted parameters of tiling or not (extended parameters). We do modeling using the stacked classifier. The base models are: Random Forest Classifier [30], Gradient Boosting Classifier [16], SVM [161] and Logistic Regression [160]. We use Logistic regression as a meta-classifier to aggregate the predictions The pipeline is shown in Figure 6.15. The performance of different classifiers is shown in table 6.5.

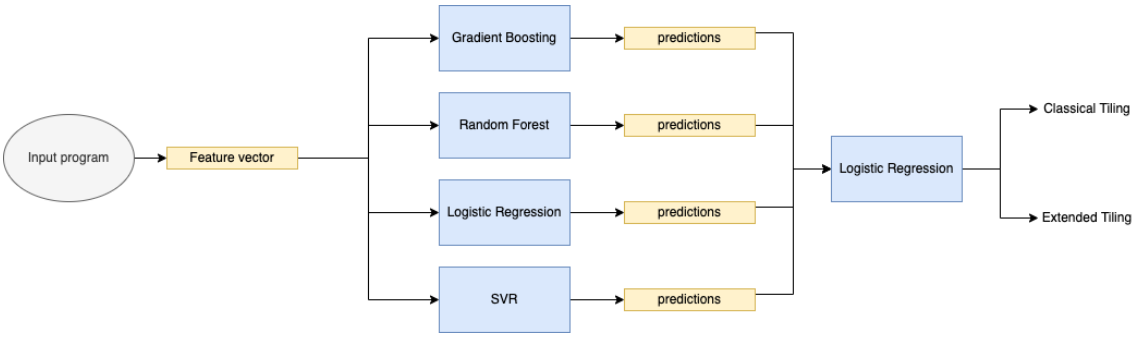

Figure 6.15: First-stage model

Train/test split

We evaluated the model on the test set. These kernels were not involved in the training phase. Kernels from the train set were involved in the training of all models in this subsection. The train/Test split ratio is 0.85/0.15 with the same class balance of the minority class in both splits.

Model evaluation

We face a problem of imbalanced classification. We consider it reasonable to use the ROC-AUC score to evaluate the quality of the predictions. The ROC-AUC curve is a graph showing the performance of a classification model at all classification thresholds. The y-axis corresponds to the recall of the model and the y-axis corresponds to the False Positive Rate. We compute the ROC-AUC score, the area under its curve. The best value of 1 would correspond to the ideal classifier, the value of 0.5 corresponds to the random guess, and the value of 0.79 would correspond to the constant classifier which always predicts the majority class. We achieved a ROC-AUC score of 0.89, we conclude that our model has a good generalization ability to distinguish whether a kernel benefits from extended tiling parameters or not.

Feature importance

Figure 6.16 presents the importance of the features of the encoding that we used. We used the single CatBoost [16] classifier to measure the feature importance. The importance was calculated based on model performance without considering particular features. If we ignore some features and train models without them, there are two cases. The model performance is degrading, which means the information captured by this feature was crucial. If performance does not change or even improve, then this feature was meaningless. The feature important algorithm implemented in CatBoost also uses feature interaction. The metric of performance is normalized to have a sum of 100 if we would add all the importance. The summation vector is the crucial feature of our model, it impacts the performance the most. The iteration domain is also crucial. We can conclude that all our encodings are meaningful and provide useful information for the model.

Second-step models

We distinguish two models for the second step prediction. If the data sample was marked as beneficial for the extended tiling parameters then we predict the tile shape, scanning directions, and tile size. Otherwise, we only predict tile sizes. We consider it as a regression problem that predicts the speedup and we sample the best parameters that maximize the speedup for the prediction. We used single CatBoost [16] to handle this task.

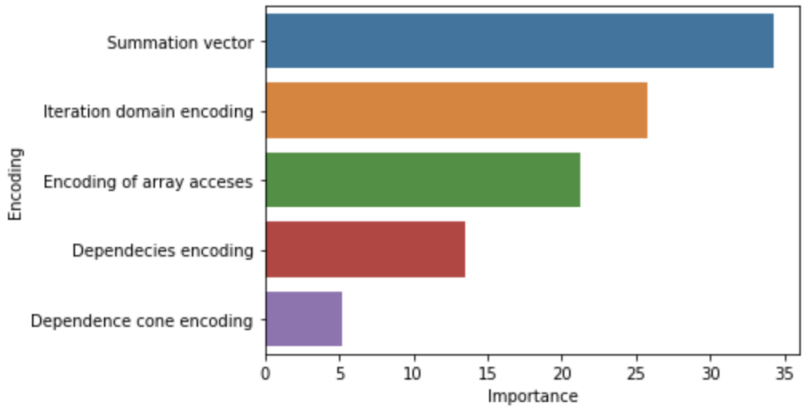

Figure 6.16: Feature importance

Evaluation of the results

Figure 6.17 shows the comparison of our approach to predict the extended tiling parameters by applying the first and second steps of our model compared to the model that predicts cubic size and initial scanning directions. Our methodology could bring up to 5% of absolute speedup for the kernels that benefit from it. The relative speedup we get is about 84% for the results of the two concatenated step models.

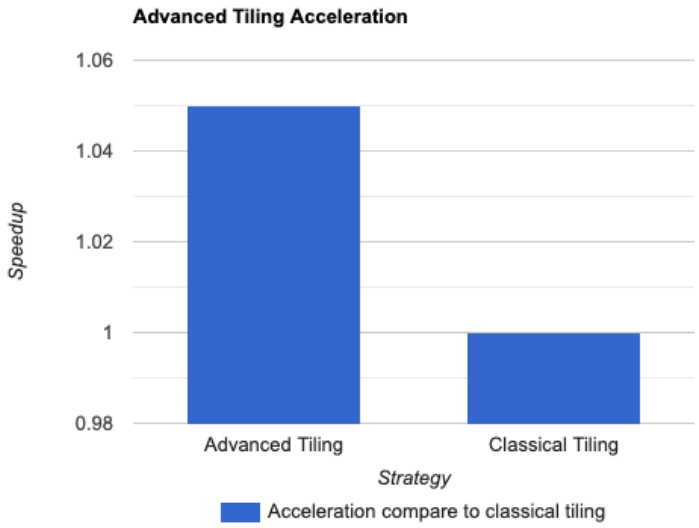

Figure 6.17: Acceleration of classical pipeline

6.4.3 Conclusion on predictions for Advanced Tiling transformation

In this subsection, we perform Machine Learning modeling to predict the extended set of tiling parameters. We do not restrict our predictions to tile size but predict 1) tile shape 2) -intra tile scanning directions 3) -inter tile scanning directions 4) tile size.

Our pipeline consists of two consecutive Machine Learning models. The first one predicts whether or not we should apply tiling with extended parameters for this kernel. The second model applies tiling with advanced parameters if it is necessary.

We conclude that

- We managed to generate around 200 kernels that benefit from extended parameters of tiling compared to tiling with the original parameters. It shows the ability of our COLA-gen generator [70] to generate very diverse codes. To our knowledge, the profitability of tiling with extended parameters has not been properly addressed in the literature, we show that it could bring up to 30% performance for 2-dimensional nested loops. Our preliminary experiments on 3-dimensional nested loops show that the acceleration can be up to 75% on some particular examples.

- We manage to create the Machine Learning model that predicts if this kernel potentially benefits from extended tiling parameters. It has shown its efficiency and achieved a 0.89 ROC-AUC score.

- Our second-step Machine Learning model predicts tiling parameters for kernels that benefit from advanced tiling. We managed to accelerate these kernels by 5% on average compared to the case when we only considered the initial scanning directions and the initial shape. It is about 84% of what Autotuner could find.

- All features that we used have a significant impact on the model performance. Their elimination would lead to model performance degradation.

6.4.4 Technological stack

For the experiments in this chapter, I used the following tools

- Python: Programming language

- Sklearn: Machine learning modeling, data prepossessing.

- NetworkX: Work with graphs, graphs visualization, graphs embeddings.

- PyTorch: Experiments with Deep Learning models

- Numpy, Pandas: Data processing
- CatBoost: Implementation of gradient boosting

6.4.5 Conclusion

This section brings two important contributions to this thesis.

- We proposed the encoding of the following concepts: data dependencies, dependence cone, iteration domain, and the vector which is the sum of all the dependencies. We show that these concepts have an important impact on the performance of the model and help to capture the problem being observed.
- We do a generalization through existing methods to model the problem of tile size predictions and generalization between different shapes for tiling transformation. We conclude that the best way to model tiling from an ML perspective is to implicitly predict tile sizes of parallelepiped shape. First, predict the potential speedup that we could get with each tile size, then choose the tile size that maximizes the speedup. We can improve predictions up to 47% on the validation set and up to 113% on the test set.
- Our predictions can reach up to 91.6% out of the optimal ones found by autotuner on the validation set and up to 70% on the test set formed with real well-known kernels. We did not try to mimic the distribution of real kernels but used a very general training set with a very wide range of parameters.
- We do not restrict our model predictions to tile sizes only. We add information about scanning directions (for -inter and -intra tiles) and tile shapes (square or diamond). We introduce a two-stage ML pipeline to model that. The first model predicts whether or not we apply extended tiling for a given kernel and the second model chooses the appropriate parameters. We show that the extended tiling could bring up to 30% compared to the classical tiling. Our predictions for the kernels that benefit from the extended tiling achieved a speedup of about 5% over tiling where we only consider original tiling parameters. We reach about 84% of what the autotuner could find.
- We used about 1000 perfectly nested 3-D loops generated by COLA-gen [70] for the prediction of the best tile sizes in section 6.3. We used about 1000 perfectly nested 2-D loops for section 6.4 also generated by our generator.

Chapter 7

Autotuning acceleration using Machine Learning

Résumé

L'autotuning est une technique puissante pour trouver les paramètres performants des transformations de code. L'idée est d'explorer itérativement différents points de l'espace de recherche en recherchant des solutions de plus en plus proches de l'optimal. Cette idée est assez différente de ce sur quoi nous nous sommes concentrés auparavant. Nous avons ciblé des prédictions ponctuelles, cela signifie que nous n'avons qu'une seule tentative pour estimer la meilleure solution. Cette approche est beaucoup plus rapide mais moins précise que la recherche exhaustive. Dans le chapitre 6, nous montrons que nous visons à obtenir environ 60%-80% de ce que l'Autotuner a trouvé pour la transformation de tuilage.

Ce chapitre combine les deux approches précedentes. Nous essayons de comprendre si les prédictions uniques pourraient être utiles ou non pour améliorer l'Autotuning et si cela pourrait être compatible.

Introduction

Autotuning is a powerful technique to find beneficial parameters of code transformations. The idea is to iteratively explore different points of the search space by looking for better and better solutions. This idea is quite different from what we focused on before. We targeted one-shot predictions, it means we have just one attempt to guess the best solution, this approach is much faster but less accurate than the exhaustive search. In chapter 6, we show that we aim to get around 60%-80% out of what the state-of-the-art Autotuner found for the tiling transformation.

This chapter combines these two different approaches. We try to understand if the one-shot predictions could be useful for Autotuning improvements or not and if could it be compatible.

This chapter is organized as follows. Section 7.1 presents the autotuner we used in the context of this thesis and introduces its search modules. Section 7.2 shows our preliminary results on the autotuning strategy based on predicted speedup estimation of the search domain. Section 7.3 present our investigation on how the choice of the initial point could accelerate the iterative search.

7.1 Locus Autotuner

Locus [32] is a system and a language to express and optimize complex kernels for different architectures. It uses optimization techniques and guides an empirical search for the best solutions using several complex search space traversal techniques. This tool was used to label all the data in this thesis and we use it as a reference for this chapter. The key part of the autotuner is its empirical search modules.

Locus integrates two empirical search modules inside [86] and [5].

OpenTuner [86] has a hierarchical structure of search techniques. There is a root technique that chooses sub-techniques to perform. AUC Bandit technique with greedy mutation, differential evolution, and two hill climber instances were used as the default root technique for this search engine.

Bergstra et al. [5] presents hand-crafted algorithm Hyperparameter Optimization algorithm (HOA) to focus the iterative search.

7.2 Point Ranking Strategy

The previous section highlights the search engines that Locus uses. This section presents preliminary results showing that Machine Learning can accelerate the iterative search. We plot the results of our autotuning strategy based on Machine Learning. The strategy is very simple and naive. The idea is to use the model 6.3 described in the previous chapter in order to evaluate the speedup of the potential candidates. We rank all candidates according to their speedup and take one after the other. Obviously, this strategy is not optimal since it does not even have a feedback loop that changes actions based on observations. The goal is to show that, even in this context, Machine Learning ranking can sometimes be competitive with Autotuner.

Perfect predictions

We guessed the best tile size in one shot for two benchmarks: mm and syr2k-kernels. As it was mentioned before, our model could efficiently capture the patterns for kernels, which benefit a lot from tiling. It took 29 and 33 iterations correspondingly for the autotuner to figure out what tile size is the best for these kernels.

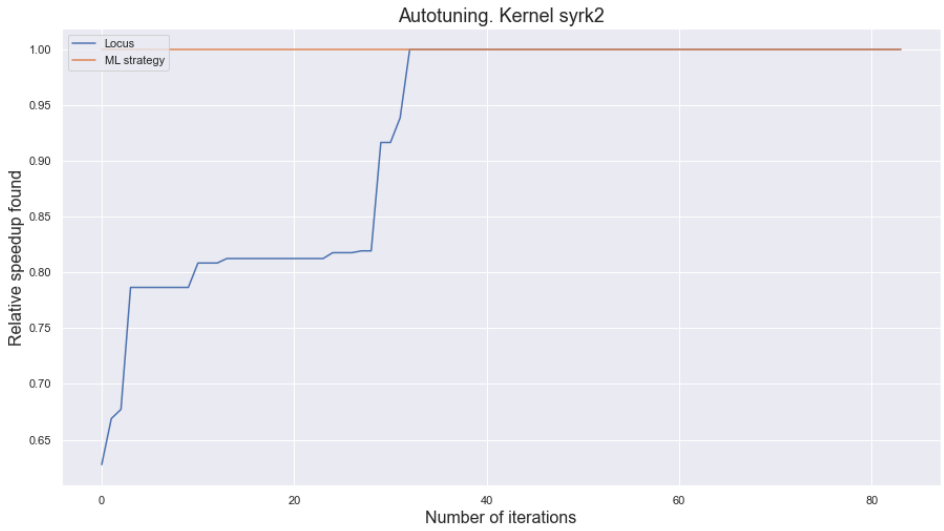

Figure 7.1: Syr2k kernel

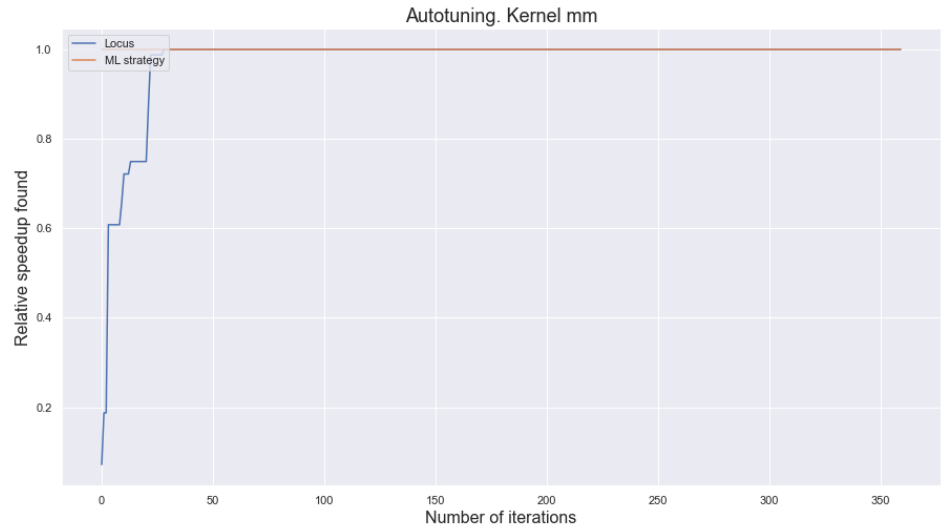

Figure 7.2: MM kernel

Good predictions

There are cases, where we take fewer attempts than the autotuner to guess the best tile sizes. It took 19 versus 30 iterations for trisolv-kernel and 31 versus 64 iterations for syrk-kernel to find the best tile size.

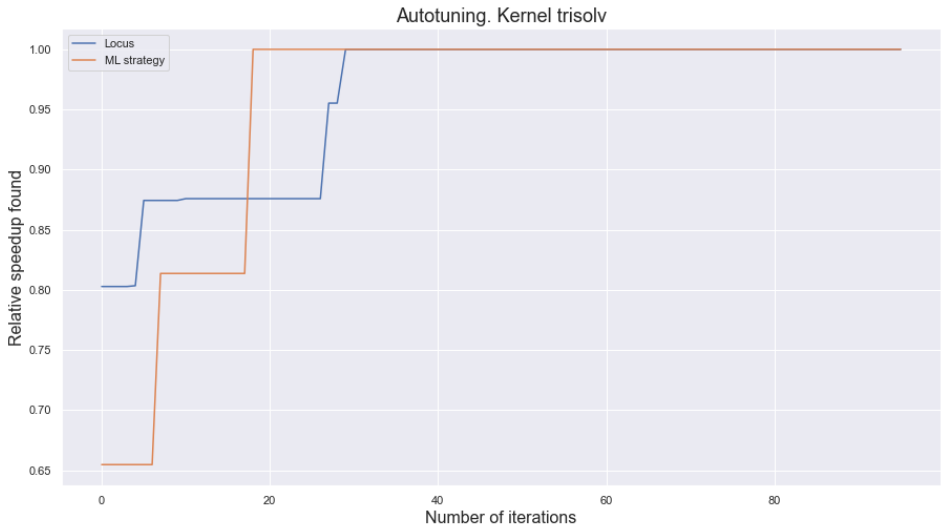

Figure 7.3: Trisolv kernel

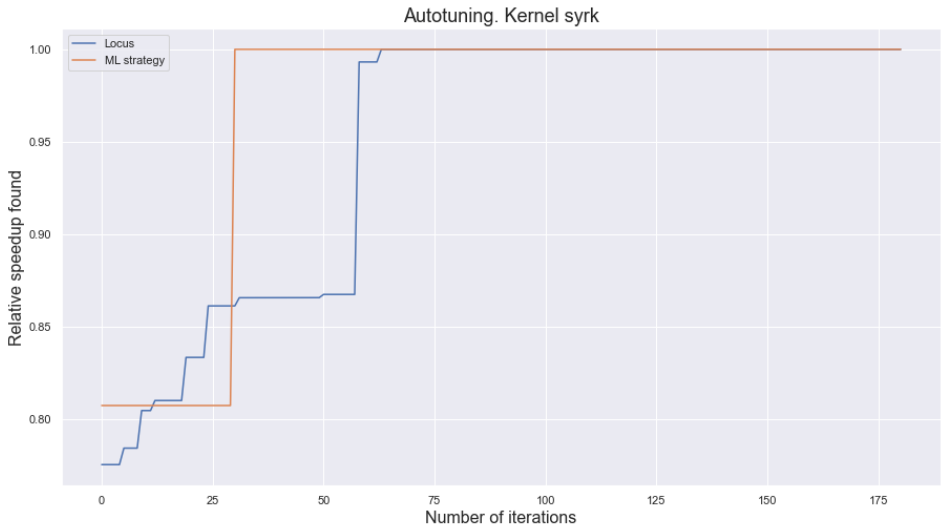

Figure 7.4: Syrk kernel

Bad predictions

Unfortunately, our model cannot capture patterns for cases when the kernel does not benefit a lot from tiling. This is the case for gemm-kernel. We spend 192 iterations versus 24.

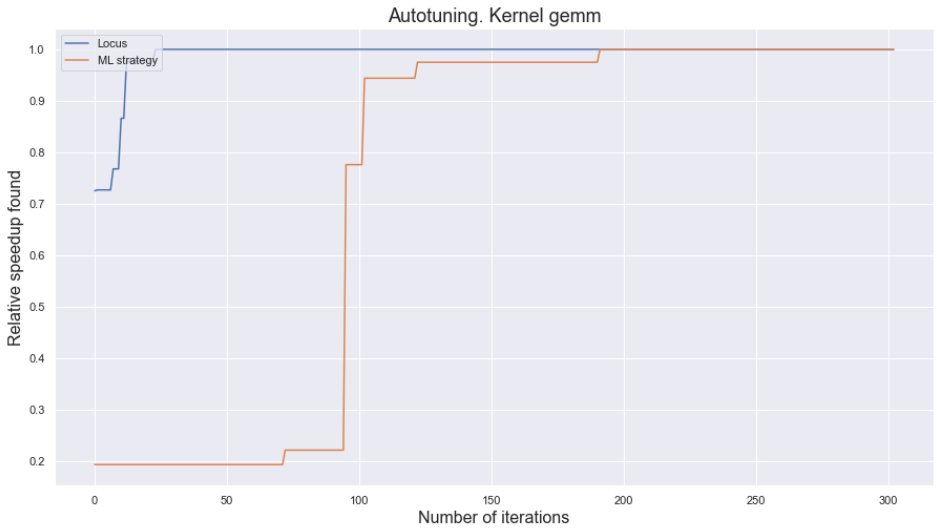

Figure 7.5: Gemm kernel

Conclusion on the naive point ranking strategy

This section shows the ability of our model to overperform the state-of-the-art autotuner for some well-known kernels in predicting the best parameters of the tiling transformation. Obviously, the strategy could be improved by adding a feedback loop. But even in this context, we may capture some crucial proprieties about speedup distribution for the tiling transformation. It is a prerequisite for further generalization.

7.3 Acceleration of more complex search spaces

This section evaluates our approach to the search space that consists of a combination of hierarchical loop tiling (two tilings consequently) and then loop unrolling. The search space has more possible combinations than the search space just for the tiling transformation.

We investigate the hypothesis that the iterative search process is sensitive to the initial point selected. The idea is to investigate some search spaces and evaluate the convergence properties if 1) We let the autotuner choose it by itself and 2) We predict it with Machine Learning.

The target optimization sequence is Paralelepiped loop tiling - parallelepiped loop tiling - loop unrolling. It gives 2657205 different code variants. We predict the parameters of the tiling transformation that is applied first. We do not predict the parameters of the second tiling.

Figure 7.6 shows the comparison of the different initialization strategies for mm-kernel. The strategy based on the initialization with Machine Learning prediction overperforms the default strategy. It only took 20 iterations to find high-quality solutions. On the other hand, the default strategy made about 75 attempts to find the same quality solution.

Figure 7.7 shows that the ML strategy found the q95 solution almost immediately. When default strategy spends around 140 iterations for that.

Figure 7.8 presents the autotuning process for syr2k-kernel. The results are comparable but the ML strategy managed to find a solution that was not found by the default strategy.

This section shows that initializing the Autotuner with Machine learning prediction seeds helps the model to find efficient solutions faster. We validate our methodology on 3 well-known benchmarks: mm, syr2k and syrk.

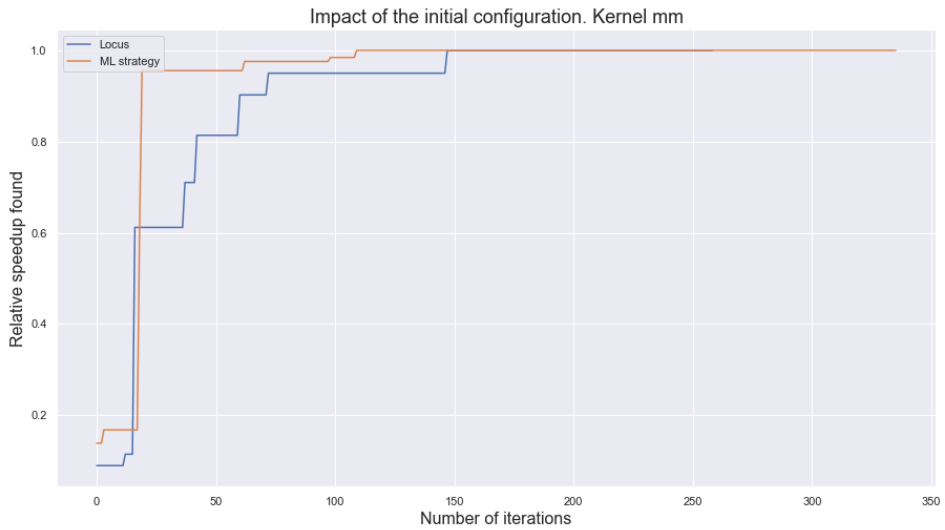

Figure 7.6: Mm. Initial seed

7.4 Conclusion

In this section, we discussed the ability of Machine Learning to accelerate the iterative search for the state-of-the-art Autotuner. The first results show that our ranking ML model could autotuner some well-known kernels faster than Autotuner.

Our results were conducted with parallelepiped and cubic tiling transformations. Our second hypothesis shows that ML predictions used as the initial point for the iterative search could accelerate the search process. We used up to 100 iterations less

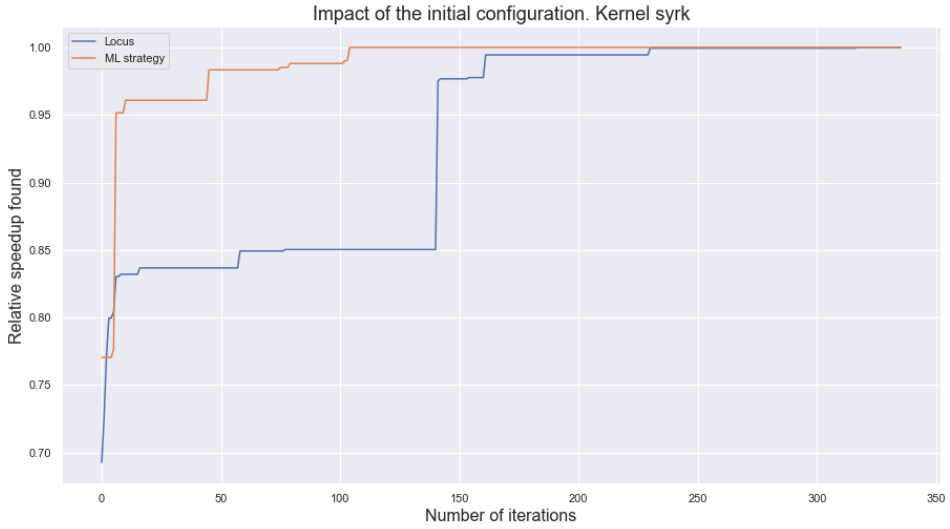

Figure 7.7: Syrk. Initial seed

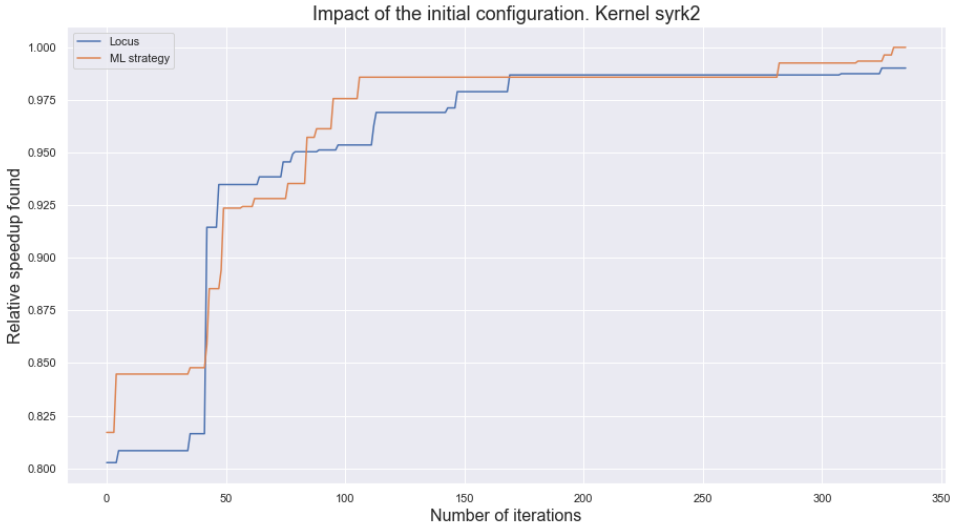

Figure 7.8: Syr2k. Initial seed

to find the same quality of solutions for well-known benchmarks. All these results derive a positive discussion about the applicability of ML methods for Autotuning, especially for large search spaces encountered in program optimizations.

Chapter 8

Conclusion

Résumé

Cette thèse présente les résultats de mes recherches sur l'optimisation du code des noyaux à forte intensité de calcul, basée sur des transformations source-à-source, à l'aide de techniques d'apprentissage automatique.

Nous adressons les deux domaines scientifiques: Machine Learning et optimisation de code. Nous considérons les techniques ML comme un outil pour prédire les paramètres d'optimisation. Par conséquent, mes contributions concernent la formulation de problèmes d'optimisation de code dans le cadre du ML. En particulier, nous proposons des techniques pour prédire efficacement les paramètres étendus d'une transformation de tuilage de boucles efficace permettant de se rapprocher des performances maximales du code.

Nos autres contributions consistent à créer les conditions préalables essentielles à l'application des techniques de ML dans ce domaine. Par exemple, nous avons travaillé sur les données collectées et le design expérimental optimal afin de fournir les données les plus représentatives pour nos modèles, puisque la qualité des données est la principale exigence du Machine Learning. Cela inclut également de travailler sur la façon dont nous représentons ces données dans les modèles, car la représentation d'un concept d'intérêt est un élément crucial d'un modèle réussi.

8.1 Introduction

This thesis presents the results of my research on code optimization of compute-intensive kernels, based on source-to-source transformations, using machine learning techniques.

It takes place in two scientific domains - Machine Learning and Code Optimization. We consider ML techniques as a tool to predict optimization parameters. Therefore, my contributions concern the formulation of code optimization problems within the framework of ML. In particular, we propose techniques to efficiently pre-

dict the extended parameters of an efficient loop tiling transformation to come close to the peak code performance.

Our other contributions are in creating the essential prerequisites for the application of ML techniques in this field. For example, we worked on the collected data and the optimal experimental design in order to provide the most representative data for our models, since the quality of data is the main requirement of Machine Learning. This also includes working on how we represent our data in models as representing a concept of interest is a crucial part of a successful model.

8.2 Thesis Contributions

Our contributions fall into two groups: 1) data collection and its optimal experimental design and 2) application of the loop tiling transformation. The first group targets aspects of the synthetic loop generator used to collect all the data for this thesis. It also targets the techniques to select the most representative data for code generation. The contributions of the second group concern the choice of the best way to model the tiling transformation from the ML perspective, the modeling of the code properties from the tiling perspective, and the consideration of different tiling parameters for the predictions.

Data collection and optimal experimental design

We propose a synthetic code generator that can generate representative data patterns for code transformations such as loop tiling, loop interchange, and loop unrolling. This generator can produce training and validation sets of arbitrary sizes for Machine Learning needs.

Moreover, we propose a strategy based on Active Learning to generate the most representative data samples. It helps either to reduce the training time to reach a certain level of performance or to obtain a more efficient model under the same time constraints. This technique achieves up to 15% more speed-up using the same amount of data. Active learning techniques reduce the number of input programs needed and the time during the training phase. Active learning can easily be integrated when developing ML models of program transformations, because thanks to the generator, we can provide representative kernels in all desired program classes. Contribution related to this section was published and presented in the 13th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and the 11th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2022).

Tiling transformation

Loop tiling has been studied for many years, it was ambitious to start new research on this code optimization. We justify our research by the fact that among the optimizations, loop tiling is essential (Chapter 5), and there is no technique that yields its optimal partitioning parameters in a reasonable time. The main difficulty is that the space of tiling parameters is very large, and a proof of concept of the usefulness of machine learning techniques in a compiler environment needed to be investigated

Firstly, we provide analyses of common ML pipelines to predict the optimal tile size and choose the best one. Our training set was generated with our automatic code generator, totally independent of the benchmarks chosen for the evaluation phase of our results. The performances obtained are remarkable in this context since 80% of the optimal performances are reached for the cubic tiling and 70% for the parallelepiped tiling. We conclude that the best way to predict the efficient tiling is 1) to predict the speedup for a given kernel using potential tile size as the input feature and then 2) for the prediction to sample all possible tile sizes and choose the one that maximizes the predicted speedup. Parallelepiped tiling seems a more reasonable abstraction for ML modeling, we manage to obtain 2.45x speedups on the validation set and 3.35x on the test set. On the other hand, our predictions for cubic tiling reached 1.85x and 1.58x respectively.

Secondly, we investigate new tiling criteria to consider, such as the quality of the generated code, in addition to its natural properties of increasing in data locality and extracting potential parallelism. We do not limit our predictions to tile sizes for a fixed shape. We propose an approach to simultaneously predict 1) tile size, 2) intra-tile scanning directions, 3) inter-tile scanning directions, and 4) tile shape for kernels that have uniform data dependencies. We show that these parameters that could bring up to 30% of additional speedups for some kernels and up to 7% of additional speedups for the kernels that we investigated on average if we compare just with the tile size prediction. Moreover, it eases the derivation of heuristics independent from our ML models. Noting that in 80 % of cases the TI-LI scanning direction setting is beneficial for programs with data dependencies, this setting can be systematically applied to these codes. Our Machine Learning model managed to predict up to 5% of additional speedup for the kernels that benefit from more extended tiling parameters. Our pipeline for extended tile settings for 2-D loops reaches up to 84% of what the Autotuner found.

An essential part of our results is the proper choice of the features of the programs to be considered and the proposal of their representations (fixed size) in an ML context. The most original representation is the one proposed to encode data dependencies. It allows for characterizing data dependencies with a fixed-size data structure, precise enough to be exploited by ML techniques. We also investigated alternative feature spaces such as graph embedding techniques to embed control-flow

and data-dependence graphs. For control-flow graphs, we created a synthetic CFG generator that could produce millions of valid CFGs corresponding to programs we are interested in. For DDG embeddings we use kernels produced by a synthetic code generator [25].

We also provide an analysis of how our ML model could be used for kernel Autotuning. We conclude that it could be beneficial in some cases, especially when the Autotuner needs the initial seed to start the search process.

It should be mentioned that all the results were obtained by applying the tiling transformation alone. It is therefore difficult to compare our gains with other optimizers that apply combinations of transformations including tiling in its simplified version. Our ML approach allows us to obtain even greater gains if we combine the ML models of each of the transformations. However, the preliminary studies that we have carried out on the prediction of the parameters of a single model coupling several transformations (tiling + unrolling + permutation) show that the more the search space for the parameters of an ML model is increased, the less it is precise, compared to the results that can be obtained with separate models for each of the transformations.

8.3 Future work and improvements

This manuscript opens the way for potential improvements that can be made, and for other research of interest.

Generalization for different architectures

The results of this thesis are those obtained for a particular architecture. It would be interesting to validate them for architectures with the same characteristics: memory size, number of processors, number of threads, etc. and to establish a classification of architectures.

Code generator

The prerequisites for extending our methods are as follows: to have a set of representative and executable kernels and to have a tool capable of applying the source-to-source transformation, in a systematic and legal way.

In chapter 3, we present the code generator for the collection of training and validation sets for ML issues. The generated codes are representative of transformations such as loop tiling, loop unrolling, and loop interchange.

- Our generator can be extended to many programming languages (not only C) because the main concepts we used are language-agnostic. It only requires

a few modifications to the syntax and code routines to achieve a successful translation into the target language.

- Our code generation pipeline has been extended to CUDA but not been exploited yet.

- The generator can be extended and generate more diverse programs (triangular iteration domain, conditional iterations, etc.), which will improve the results for less regular calculation kernels.

- The code generation for different architectures is the point for improvement. We investigated the quality of the code generated only on a particular CPU. The analysis of the code generated on different architectures of CPU, GPU and on heterogeneous architectures is the key direction for the research.

Optimal experimental design

In chapter 4, we present techniques to select the most representative data that would be taken as input for the training phase in the ML pipeline. We evaluate our methodology on loop tiling transformation and later use this idea to collect all the data for the experiments in this thesis.

- The future improvements may target the validation on a greater number of transformations than just loop tiling.

- The investigation of better generation strategies. For instance, scanning directions associated with diamond tiling were never labeled as optimal ones in 6.4. It could be also explained by high computational overhead of this scanning.

- More Active Learning techniques could be investigated such as those proposed by Wu et al. [65]. Future improvement targets the applicability of Active Learning not only for the regression task but for a large bunch of ML problems in code optimization.

- The hyperparameters (number of chosen programs to label at each step, distance metrics of programs, etc.) tuning of the active learning techniques is the point for improvement. It may improve the convergence of the models.

Machine Learning part

In the context of this thesis, we used classical Machine Learning models such as classification and regression. Moreover, we investigated Active Learning techniques.

- The state-of-the-art research in ML operates with millions of data samples. However, we work with only thousands. Our results could be improved if much more data were collected. It opens the opportunity of applying Neural Networks for our tasks such as in the work of Baghdadi et al. [102]. Deep Learning models tend to perform better and extract more sophisticated patterns with the increase of the training set and this could potentially improve our prediction pipeline.

Alternative feature spaces

As previously stated, in chapter 6, we investigated the embedding of different code concepts and their impact on the performance of the models.

- We worked on the integration of Control Flow and Data Dependence Graphs. The potential integration of broader concepts (such as whole code) might be beneficial.

Bibliography

[1] Tomofumi Yuki et al. "Automatic creation of tile size selection models". In: *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. 2010, pp. 190–199.

[2] David Avis. "On the extreme rays of the metric cone". In: *Canadian Journal of Mathematics* 32.1 (1980), pp. 126–144.

[3] Riyadh Baghdadi et al. "Tiramisu: A polyhedral compiler for expressing fast and portable code". In: *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2019, pp. 193–205.

[4] Andrew Adams et al. "Learning to optimize halide with tree search and random programs". In: *ACM Transactions on Graphics (TOG)* 38.4 (2019), pp. 1–12.

[5] James Bergstra, Daniel Yamins, and David Cox. "Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures". In: *International conference on machine learning*. PMLR. 2013, pp. 115–123.

[6] Jonathan Ragan-Kelley et al. "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines". In: *Acm Sigplan Notices* 48.6 (2013), pp. 519–530.

[7] Tom Dietterich. "Overfitting and undercomputing in machine learning". In: *ACM computing surveys (CSUR)* 27.3 (1995), pp. 326–327.

[8] Cedric Seger. *An investigation of categorical variable encoding techniques in machine learning: binary versus one-hot and feature hashing*. 2018.

[9] Feature (machine learning). *Feature (machine learning) — Wikipedia, The Free Encyclopedia*. [Online; accessed 11-September-2022]. 2022. URL: https://en.wikipedia.org/wiki/Feature_(machine_learning).

[10] Feature (machine learning). *Machine Learning Crash Course with TensorFlow APIs*. [Online; accessed 31-October-2022]. 2022. URL: https://developers.google.com/machine-learning/crash-course/embeddings/video-lecture?hl=en.

[11] Training, validation, and test data sets. *Training, validation, and test data sets — Wikipedia, The Free Encyclopedia*. [Online; accessed 11-September-2022]. 2022. URL: https://en.wikipedia.org/wiki/Training,_validation,_and_test_data_sets.

[12] Loss function. *Loss function — Wikipedia, The Free Encyclopedia*. [Online; accessed 11-September-2022]. 2022. URL: https://en.wikipedia.org/wiki/Loss_function.

[13] Labeled data. *Labeled data — Wikipedia, The Free Encyclopedia*. [Online; accessed 11-September-2022]. 2022. URL: https://en.wikipedia.org/wiki/Labeled_data.

[14] Optimal design. *Optimal design — Wikipedia, The Free Encyclopedia*. [Online; accessed 11-September-2022]. 2022. URL: https://en.wikipedia.org/wiki/Optimal_design.

[15] Ground truth. *Ground truth — Wikipedia, The Free Encyclopedia*. [Online; accessed 11-September-2022]. 2022. URL: https://en.wikipedia.org/wiki/Ground_truth.

[16] Anna Veronika Dorogush, Vasily Ershov, and Andrey Gulin. "CatBoost: gradient boosting with categorical features support". In: *arXiv preprint arXiv:1810.11363* (2018).

[17] Pedro Domingos. "A unified bias-variance decomposition". In: *Proceedings of 17th international conference on machine learning*. Morgan Kaufmann Stanford. 2000, pp. 231–238.

[18] Aric Hagberg, Pieter Swart, and Daniel S Chult. *Exploring network structure, dynamics, and function using NetworkX*. Tech. rep. Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[19] Diogo V Carvalho, Eduardo M Pereira, and Jaime S Cardoso. "Machine learning interpretability: A survey on methods and metrics". In: *Electronics* 8.8 (2019), p. 832.

[20] David H Wolpert. "Stacked generalization". In: *Neural networks* 5.2 (1992), pp. 241–259.

[21] Chris Cummins et al. *Deepsmith: Compiler fuzzing through deep learning*. 2018.

[22] John Platt et al. "Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods". In: *Advances in large margin classifiers* 10.3 (1999), pp. 61–74.

[23] Leonard E Baum and Ted Petrie. "Statistical inference for probabilistic functions of finite state Markov chains". In: *The annals of mathematical statistics* 37.6 (1966), pp. 1554–1563.

[24] Kumudha Narasimhan et al. "A practical tile size selection model for affine loop nests". In: *Proceedings of the ACM International Conference on Supercomputing*. 2021, pp. 27–39.

[25] Maksim Berezov et al. "COLA-Gen: Active Learning Techniques for Automatic Code Generation of Benchmarks". In: *13th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 11th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2022.

[26] Jeffrey Van der Gucht et al. "Deep Horizon: A machine learning network that recovers accreting black hole parameters". In: *Astronomy & Astrophysics* 636 (2020), A94.

[27] S Parker. "Risk-Constrained Dynamic Programming for Optimal Mars Entry, Descent, and Landing". In: *NASA Tech Briefs* (2013), p. 33.

[28] Burr Settles. *Active Learning Literature Survey*. Computer Sciences Technical Report 1648. University of Wisconsin–Madison, 2009.

[29] Uday Bondhugula et al. "Pluto: A practical and fully automatic polyhedral program optimization system". In: *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*. Citeseer. 2008.

[30] Mahesh Pal. "Random forest classifier for remote sensing classification". In: *International journal of remote sensing* 26.1 (2005), pp. 217–222.

[31] Hervé Abdi and Lynne J Williams. "Principal component analysis". In: *Wiley interdisciplinary reviews: computational statistics* 2.4 (2010), pp. 433–459.

[32] SFX Thiago Teixeira et al. "Locus: a system and a language for program optimization". In: *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2019, pp. 217–228.

[33] Mines ParisTech. *Pips: Automatic parallelizer and code transformation framework*. 2013.

[34] Han Xiao, Kashif Rasul, and Roland Vollgraf. "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms". In: *arXiv preprint arXiv:1708.07747* (2017).

[35] David G Woodward. "Life cycle costing—Theory, information acquisition and application". In: *International journal of project management* 15.6 (1997), pp. 335–344.

[36] Zheng Wang and Michael O'Boyle. "Machine learning in compiler optimization". In: *Proceedings of the IEEE* 106.11 (2018), pp. 1879–1901.

[37] Alton Chiu, Joseph Garvey, and Tarek S Abdelrahman. "Genesis: a language for generating synthetic training programs for machine learning". In: *Proceedings of the 12th ACM International Conference on Computing Frontiers*. 2015, pp. 1–8.

[38] Chris Cummins et al. "Synthesizing benchmarks for predictive modeling". In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2017, pp. 86–99.

[39] Vassil Panayotov et al. "Librispeech: an asr corpus based on public domain audio books". In: *2015 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. IEEE. 2015, pp. 5206–5210.

[40] Bryan Klimt and Yiming Yang. "Introducing the Enron corpus." In: *CEAS*. 2004.

[41] Lakshminarayanan Renganarayanan et al. "Parameterized tiled loops for free". In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2007, pp. 405–414.

[42] J Bennett et al. *Embench: An evolving benchmark suite for embedded iot computers from an academic-industrial cooperative*.

[43] Matthew R Guthaus et al. "MiBench: A free, commercially representative embedded benchmark suite". In: *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*. IEEE. 2001, pp. 3–14.

[44] Abid M Malik. "Optimal tile size selection problem using machine learning". In: *2012 11th International Conference on Machine Learning and Applications*. Vol. 2. IEEE. 2012, pp. 275–280.

[45] Song Liu et al. "An efficient tile size selection model based on machine learning". In: *Journal of Parallel and Distributed Computing* 121 (2018), pp. 27–41.

[46] T. Yuki and L. Pouchet. "PolyBench 4.2". In: *https://sourceforge.net/projects/polybench/* (Jan 26, 2021).

[47] Saeed Maleki et al. "An evaluation of vectorizing compilers". In: *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE. 2011, pp. 372–382.

[48] David Callahan, Jack J Dongarra, and David Levine. *Vectorizing compilers: A test suite and results*. Argonne National Laboratory, 1988.

[49] FH McMahon. "Livermore Fortran Kernels: A computer test of numerical performance range UCRL-53745". In: *LLNL, CA., USA* (1986).

[50] Jan Gustafsson et al. "The Mälardalen WCET benchmarks: Past, present and future". In: *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2010.

[51] James Pallister, Simon Hollis, and Jeremy Bennett. "BEEBS: Open benchmarks for energy measurements on embedded platforms". In: *arXiv preprint arXiv:1308.5174* (2013).

[52] Antoine Monsifrot, François Bodin, and Rene Quiniou. "A machine learning approach to automatic production of compiler heuristics". In: *International conference on artificial intelligence: methodology, systems, and applications*. Springer. 2002, pp. 41–50.

[53] Eirini Kalliamvakou et al. "The promises and perils of mining github". In: *Proceedings of the 11th working conference on mining software repositories*. 2014, pp. 92–101.

[54] Georgios Gousios and Diomidis Spinellis. "Mining software engineering data from GitHub". In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE. 2017, pp. 501–502.

[55] Matthew A Russell. *Mining the social web: data mining Facebook, Twitter, LinkedIn, Google+, GitHub, and more.* " O'Reilly Media, Inc.", 2013.

[56] Jaroslav Fowkes and Charles Sutton. "Parameter-free probabilistic API mining across GitHub". In: *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. 2016, pp. 254–265.

[57] Etem Deniz and Alper Sen. "Minime-gpu: Multicore benchmark synthesizer for gpus". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 12.4 (2015), pp. 1–25.

[58] Ajay Joshi, Lieven Eeckhout, and Lizy John. "The return of synthetic benchmarks". In: *2008 SPEC Benchmark Workshop*. 2008, pp. 1–11.

[59] Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. "Synthesizing geometry constructions". In: *ACM SIGPLAN Notices* 46.6 (2011), pp. 50–61.

[60] Calvin Loncaric, Emina Torlak, and Michael D Ernst. "Fast synthesis of fast collections". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2016, pp. 355–368.

[61] Ross D King et al. "Functional genomic hypothesis generation and experimentation by a robot scientist". In: *Nature* 427.6971 (2004), pp. 247–252.

[62] Dana Angluin. "Queries revisited". In: *International Conference on Algorithmic Learning Theory*. Springer. 2001, pp. 12–31.

[63] David Cohn, Les Atlas, and Richard Ladner. "Improving generalization with active learning". In: *Machine learning* 15.2 (1994), pp. 201–221.

[64] David D Lewis and William A Gale. "A sequential algorithm for training text classifiers". In: *SIGIR'94*. Springer. 1994, pp. 3–12.

[65] Dongrui Wu, Chin-Teng Lin, and Jian Huang. "Active learning for regression using greedy sampling". In: *Information Sciences* 474 (2019), pp. 90–105.

[66] Zhi Chen et al. "Lore: A loop repository for the evaluation of compilers". In: *2017 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2017, pp. 219–228.

[67] François Irigoin and Rémi Triolet. "Supernode partitioning". In: *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1988, pp. 319–329.

[68] Corinne Ancourt and François Irigoin. "Scanning polyhedra with DO loops". In: *Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*. 1991, pp. 39–50.

[69] Yi-Qing Yang, Corinne Ancourt, and François Irigoin. "Minimal data dependence abstractions for loop transformations". In: *International Workshop on Languages and Compilers for Parallel Computing*. Springer. 1994, pp. 201–216.

[70] Maksim Berezov et al. "COLA-Gen: Automatic code generator for program optimization using active learning techniques". In: *CRI report* 1.1 (2021), pp. 1–10.

[71] Stephanie Coleman and Kathryn S McKinley. "Tile size selection using cache organization and data layout". In: *ACM SIGPLAN Notices* 30.6 (1995), pp. 279–290.

[72] Jacqueline Chame and Sungdo Moon. "A tile selection algorithm for data locality and cache interference". In: *Proceedings of the 13th international conference on Supercomputing*. 1999, pp. 492–499.

[73] Monica D Lam, Edward E Rothberg, and Michael E Wolf. "The cache performance and optimizations of blocked algorithms". In: *ACM SIGOPS Operating Systems Review* 25.Special Issue (1991), pp. 63–74.

[74] Chung-Hsing Hsu and Ulrich Kremer. "A quantitative analysis of tile size selection algorithms". In: *The Journal of Supercomputing* 27.3 (2004), pp. 279–294.

[75] Jun Shirako et al. "Analytical bounds for optimal tile size selection". In: *International Conference on Compiler Construction*. Springer. 2012, pp. 101–121.

[76] Sanyam Mehta, Gautham Beeraka, and Pen-Chung Yew. "Tile size selection revisited". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 10.4 (2013), pp. 1–27.

[77] Tobias Gysi, Tobias Grosser, and Torsten Hoefler. "Absinthe: learning an analytical performance model to fuse and tile stencil codes in one shot". In: *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE. 2019, pp. 370–382.

[78] Gabriel Rivera and Chau-Wen Tseng. "A comparison of compiler tiling algorithms". In: *International Conference on Compiler Construction*. Springer. 1999, pp. 168–182.

[79] Kamen Yotov et al. "Is search really necessary to generate high-performance BLAS?" In: *Proceedings of the IEEE* 93.2 (2005), pp. 358–386.

[80] Song Liu et al. "An efficient tile size selection model based on machine learning". In: *Journal of Parallel and Distributed Computing* 121 (2018), pp. 27–41.

[81] Mohammed Rahman, Louis-Noël Pouchet, and P Sadayappan. "Neural network assisted tile size selection". In: *International Workshop on Automatic Performance Tuning (IWAPT'2010). Berkeley, CA: Springer Verlag*. 2010.

[82] Uday Bondhugula et al. "Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model". In: *International Conference on Compiler Construction*. Springer. 2008, pp. 132–146.

[83] Peter MW Knijnenburg et al. "The effect of cache models on iterative compilation for combined tiling and unrolling". In: *Concurrency and Computation: Practice and Experience* 16.2-3 (2004), pp. 247–270.

[84] Louis-Noël Pouchet et al. "Combined iterative and model-driven optimization in an automatic parallelization framework". In: *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2010, pp. 1–11.

[85] Apan Qasem and Ken Kennedy. "Profitable loop fusion and tiling using model-driven empirical search". In: *Proceedings of the 20th annual international conference on Supercomputing*. 2006, pp. 249–258.

[86] Jason Ansel et al. "Opentuner: An extensible framework for program autotuning". In: *Proceedings of the 23rd international conference on Parallel architectures and compilation*. 2014, pp. 303–316.

[87] Kenneth Hoste and Lieven Eeckhout. "Cole: compiler optimization level exploration". In: *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. 2008, pp. 165–174.

[88] Ananta Tiwari et al. "A scalable auto-tuning framework for compiler optimization". In: *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE. 2009, pp. 1–12.

[89] Maksim Berezov et al. "COLA-Gen: Automatic code generator for program optimization using active learning techniques". In: *CRI report* 1.1 (2021), pp. 1–10.

[90] Avrim L Blum and Pat Langley. "Selection of relevant features and examples in machine learning". In: *Artificial intelligence* 97.1-2 (1997), pp. 245–271.

[91] Klaus Greff et al. "LSTM: A search space odyssey". In: *IEEE transactions on neural networks and learning systems* 28.10 (2016), pp. 2222–2232.

[92] Peter J Huber. "Robust estimation of a location parameter". In: *Breakthroughs in statistics*. Springer, 1992, pp. 492–518.

[93] Andy Liaw, Matthew Wiener, et al. "Classification and regression by randomForest". In: *R news* 2.3 (2002), pp. 18–22.

[94] Mohamed Laib and Mikhail Kanevski. "Unsupervised Feature Selection Based on Space Filling Concept". In: *arXiv preprint arXiv:1706.08894* (2017).

[95] George R Terrell and David W Scott. "Variable kernel density estimation". In: *The Annals of Statistics* (1992), pp. 1236–1265.

[96] Michael C Shewry and Henry P Wynn. "Maximum entropy sampling". In: *Journal of applied statistics* 14.2 (1987), pp. 165–170.

[97] Jesse Read et al. "Classifier chains for multi-label classification". In: *Machine learning* 85.3 (2011), pp. 333–359.

[98] François Bodin et al. "Iterative compilation in a non-linear optimisation space". In: *Workshop on Profile and Feedback-Directed Compilation*. 1998.

[99] Ricardo Nobre, Luis Reis, and Joao MP Cardoso. "Compiler phase ordering as an orthogonal approach for reducing energy consumption". In: *arXiv preprint arXiv:1807.00638* (2018).

[100] Eunjung Park, Sameer Kulkarni, and John Cavazos. "An evaluation of different modeling techniques for iterative compilation". In: *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*. ACM. 2011, pp. 65–74.

[101] Alfréd Rényi et al. "On measures of entropy and information". In: *Proceedings of the fourth Berkeley symposium on mathematical statistics and probability*. Vol. 1. 547-561. Berkeley, California, USA. 1961.

[102] Riyadh Baghdadi et al. "A deep learning based cost model for automatic code optimization". In: *Proceedings of Machine Learning and Systems* 3 (2021), pp. 181–193.

[103] L Almagor et al. "Compilation order matters: Exploring the structure of the space of compilation sequences using randomized search algorithms". In: *In Proceedings of the ACM SIGPLAN Symposium on Languages, Compilers, and Tools for Embedded Systems (LCTES, pages 231–239, 2004. 1.1. 2.* 2003.

[104] Luiz GA Martins et al. "Clustering-based selection for the exploration of compiler optimization sequences". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 13.1 (2016), p. 8.

[105] Yufei Ding et al. "Autotuning algorithmic choice for input sensitivity". In: *ACM SIGPLAN Notices*. Vol. 50. 6. ACM. 2015, pp. 379–390.

[106] Amir Hossein Ashouri et al. "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*. ACM. 2016, pp. 7–12.

[107] Sameer Kulkarni and John Cavazos. "Mitigating the compiler optimization phase-ordering problem using machine learning". In: *ACM SIGPLAN Notices*. Vol. 47. 10. ACM. 2012, pp. 147–162.

[108] Amir Hossein Ashouri et al. "Cobayn: Compiler autotuning framework using bayesian networks". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 13.2 (2016), p. 21.

[109] John Cavazos and Michael FP O'boyle. "Method-specific dynamic compilation using logistic regression". In: *ACM SIGPLAN Notices*. Vol. 41. 10. ACM. 2006, pp. 229–240.

[110] Georgios Tournavitis et al. "Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping". In: *ACM Sigplan Notices*. Vol. 44. 6. ACM. 2009, pp. 177–187.

[111] I-Hsin Chung, Jeffrey K Hollingsworth, et al. "Using information from prior runs to improve automated tuning systems". In: *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. IEEE Computer Society. 2004, p. 30.

[112] Chun Chen, Jacqueline Chame, and Mary Hall. *CHiLL: A framework for composing high-level loop transformations*. Tech. rep. Citeseer, 2008.

[113] Zhelong Pan and Rudolf Eigenmann. "Fast and effective orchestration of compiler optimizations for automatic performance tuning". In: *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society. 2006, pp. 319–332.

[114] Eunjung Park, John Cavazos, and Marco A Alvarez. "Using graph-based program characterization for predictive modeling". In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. ACM. 2012, pp. 196–206.

[115] Toru Kisuki, Peter MW Knijnenburg, and Michael FP O'Boyle. "Combined selection of tile sizes and unroll factors using iterative compilation". In: *Proceedings 2000 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. PR00622)*. IEEE. 2000, pp. 237–246.

[116] Spyridon Triantafyllis et al. "Compiler optimization-space exploration". In: *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society. 2003, pp. 204–215.

[117] M O'Boyle, P Knijnenburg, and G Fursin. "Feedback assisted iterative compilation". In: *Preprint* (2000).

[118] Felix Agakov et al. "Using machine learning to focus iterative optimization". In: *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society. 2006, pp. 295–305.

[119] Ricardo Nobre, Luiz GA Martins, and João MP Cardoso. "A graph-based iterative compiler pass selection and phase ordering approach". In: *ACM SIGPLAN Notices* 51.5 (2016), pp. 21–30.

[120] Hugh Leather, Edwin Bonilla, and Michael O'Boyle. "Automatic feature generation for machine learning based optimizing compilation". In: *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society. 2009, pp. 81–91.

[121] Mircea Namolaru et al. "Practical aggregation of semantical program properties for machine learning based optimization". In: *Proceedings of the 2010 international conference on Compilers, architectures and synthesis for embedded systems*. ACM. 2010, pp. 197–206.

[122] Kaushik Datta et al. "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures". In: *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press. 2008, p. 4.

[123] William F Ogilvie et al. "Minimizing the cost of iterative compilation with active learning". In: *Proceedings of the 2017 International Symposium on Code Generation and Optimization*. IEEE Press. 2017, pp. 245–256.

[124] Grigori Fursin et al. "MILEPOST GCC: machine learning based research compiler". In: *GCC summit*. 2008.

[125] GG Fursin, Michael FP O'Boyle, and Peter MW Knijnenburg. "Evaluating iterative compilation". In: *International Workshop on Languages and Compilers for Parallel Computing*. Springer. 2002, pp. 362–376.

[126] Chris Cummins et al. "End-to-end deep learning of optimization heuristics". In: *Parallel Architectures and Compilation Techniques (PACT), 2017 26th International Conference on*. IEEE. 2017, pp. 219–232.

[127] Tianqi Chen et al. "Learning to Optimize Tensor Programs". In: *arXiv preprint arXiv:1805.08166* (2018).

[128] Alberto Magni, Christophe Dubach, and Michael O'Boyle. "Automatic optimization of thread-coarsening for graphics processors". In: *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM. 2014, pp. 455–466.

[129] Aniket Shivam et al. "Towards an Achievable Performance for the Loop Nests". In: *arXiv preprint arXiv:1902.00603* (2019).

[130] John Cavazos et al. "Rapidly selecting good compiler optimizations using performance counters". In: *Code Generation and Optimization, 2007. CGO'07. International Symposium on*. IEEE. 2007, pp. 185–197.

[131] Amir H Ashouri et al. "Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 14.3 (2017), p. 29.

[132] Grigori Fursin et al. "Milepost gcc: Machine learning enabled self-tuning compiler". In: *International journal of parallel programming* 39.3 (2011), pp. 296–327.

[133] Mark Stephenson et al. "Meta optimization: improving compiler heuristics with machine learning". In: *ACM SIGPLAN Notices*. Vol. 38. 5. ACM. 2003, pp. 77–90.

[134] Mark Stephenson and Saman Amarasinghe. "Predicting unroll factors using supervised classification". In: *Proceedings of the international symposium on Code generation and optimization*. IEEE Computer Society. 2005, pp. 123–134.

[135] Christophe Dubach et al. "Fast compiler optimisation evaluation using code-feature based performance prediction". In: *Proceedings of the 4th international conference on Computing frontiers*. ACM. 2007, pp. 131–142.

[136] Eunjung Park et al. "Predictive modeling in a polyhedral optimization space". In: *International journal of parallel programming* 41.5 (2013), pp. 704–750.

[137] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. "Polly—performing polyhedral optimizations on a low-level intermediate representation". In: *Parallel Processing Letters* 22.04 (2012), p. 1250010.

[138] Cedric Bastoul. "Code generation in the polyhedral model is easier than you think". In: *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society. 2004, pp. 7–16.

[139] Konrad Trifunovic et al. "Polyhedral-model guided loop-nest auto-vectorization". In: *2009 18th International Conference on Parallel Architectures and Compilation Techniques*. IEEE. 2009, pp. 327–337.

[140] Uday Bondhugula et al. "A practical automatic polyhedral parallelizer and locality optimizer". In: *Acm Sigplan Notices*. Vol. 43. 6. ACM. 2008, pp. 101–113.

[141] Mohamed-Walid Benabderrahmane et al. "The polyhedral model is more widely applicable than you think". In: *International Conference on Compiler Construction*. Springer. 2010, pp. 283–303.

[142] Kai Sheng Tai, Richard Socher, and Christopher D Manning. "Improved semantic representations from tree-structured long short-term memory networks". In: *arXiv preprint arXiv:1503.00075* (2015).

[143] Jerome H Friedman. "Greedy function approximation: a gradient boosting machine". In: *Annals of statistics* (2001), pp. 1189–1232.

[144] Tianqi Chen and Carlos Guestrin. "Xgboost: A scalable tree boosting system". In: *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. ACM. 2016, pp. 785–794.

[145] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. "Learning to forget: Continual prediction with LSTM". In: (1999).

[146] François Irigoin, Pierre Jouvelot, and Rémi Triolet. "Semantical interprocedural parallelization: An overview of the PIPS project". In: *ICS*. Vol. 91. 1991, pp. 244–251.

[147] Zheng Wang and Michael O'Boyle. "Machine Learning in Compiler Optimisation". In: *CoRR* abs/1805.03441 (2018). arXiv: 1805.03441. URL: http://arxiv.org/abs/1805.03441.

[148] Amir H Ashouri et al. "A survey on compiler autotuning using machine learning". In: *ACM Computing Surveys (CSUR)* 51.5 (2018), p. 96.

[149] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. "Compilers, principles, techniques". In: *Addison wesley* 7.8 (1986), p. 9.

[150] K Cooper et al. "Compilation order matters". In: ().

[151] Keith D Cooper, Philip J Schielke, and Devika Subramanian. "Optimizing for reduced code space using genetic algorithms". In: *ACM SIGPLAN Notices*. Vol. 34. 7. ACM. 1999, pp. 1–9.

[152] John Cavazos and Michael FP O'Boyle. "Automatic tuning of inlining heuristics". In: *SC'05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. IEEE. 2005, pp. 14–14.

[153] Lelac Almagor et al. "Finding effective compilation sequences". In: *ACM SIGPLAN Notices* 39.7 (2004), pp. 231–239.

[154] Pascal Raymond et al. "Improving WCET Evaluation using Linear Relation Analysis". In: *Leibniz Transactions on Embedded Systems* 6.1 (2019), pp. 02–1.

[155] Weiwei Liu and Ivor W Tsang. "On the optimality of classifier chain for multi-label classification". In: *Advances in Neural Information Processing Systems* (2015).

[156] Jin Huang and Charles X Ling. "Using AUC and accuracy in evaluating learning algorithms". In: *IEEE Transactions on knowledge and Data Engineering* 17.3 (2005), pp. 299–310.

[157] Leslie Lamport. "The parallel execution of DO loops". In: *Communications of the ACM* 17.2 (1974), pp. 83–93.

[158] Robert Tibshirani. "Regression shrinkage and selection via the lasso". In: *Journal of the Royal Statistical Society: Series B (Methodological)* 58.1 (1996), pp. 267–288.

[159] Sanford Weisberg. *Applied linear regression*. Vol. 528. John Wiley & Sons, 2005.

[160] Raymond E Wright. "Logistic regression." In: (1995).

[161] William S Noble. "What is a support vector machine?" In: *Nature biotechnology* 24.12 (2006), pp. 1565–1567.

[162] Leo Breiman. "Random forests". In: *Machine learning* 45.1 (2001), pp. 5–32.

Appendix A

Kernel Autotuning

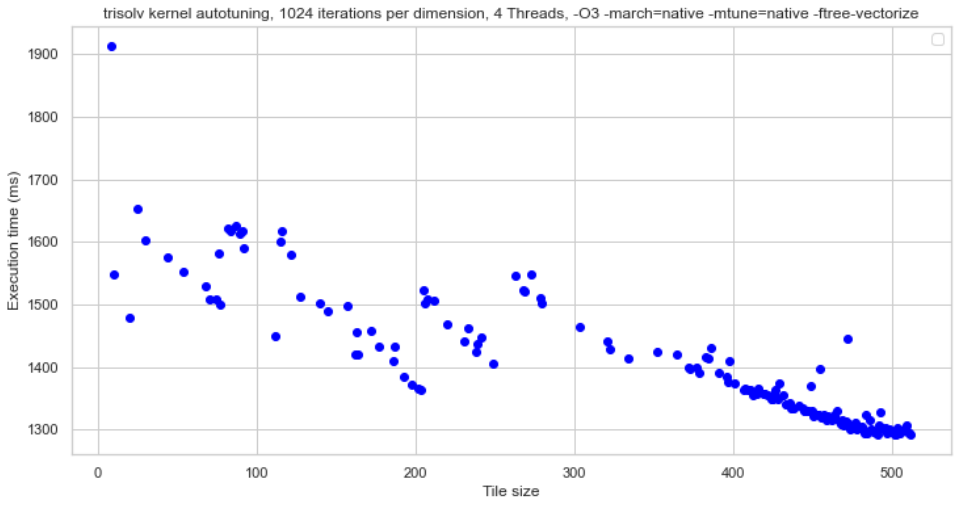

Figure A.1: Trisolv kernel Autotuning

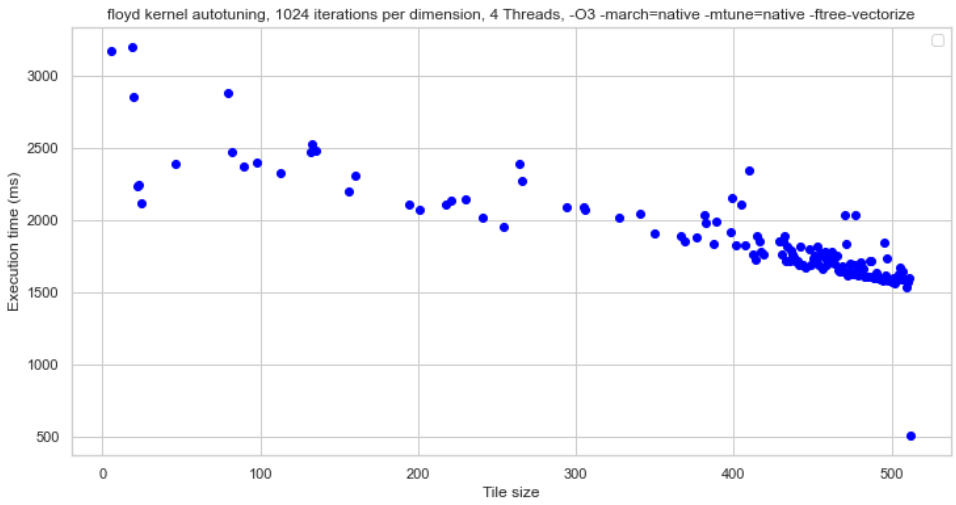

Figure A.2: Floyd kernel Autotuning

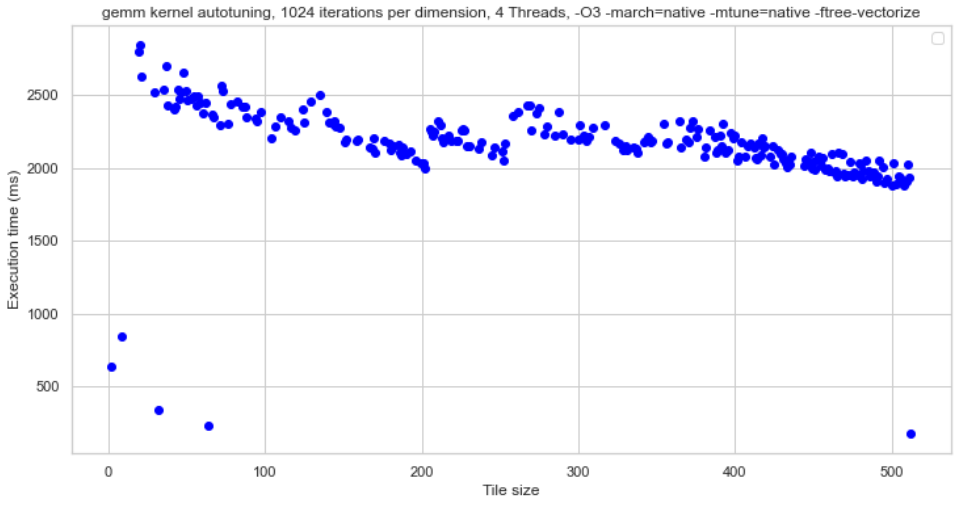

Figure A.3: Gemm kernel Autotuning

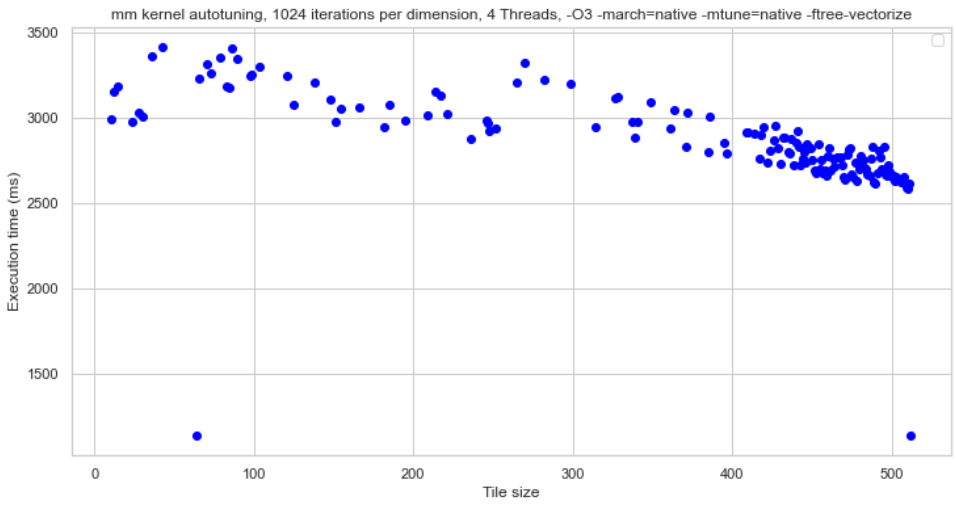

Figure A.4: MM kernel Autotuning

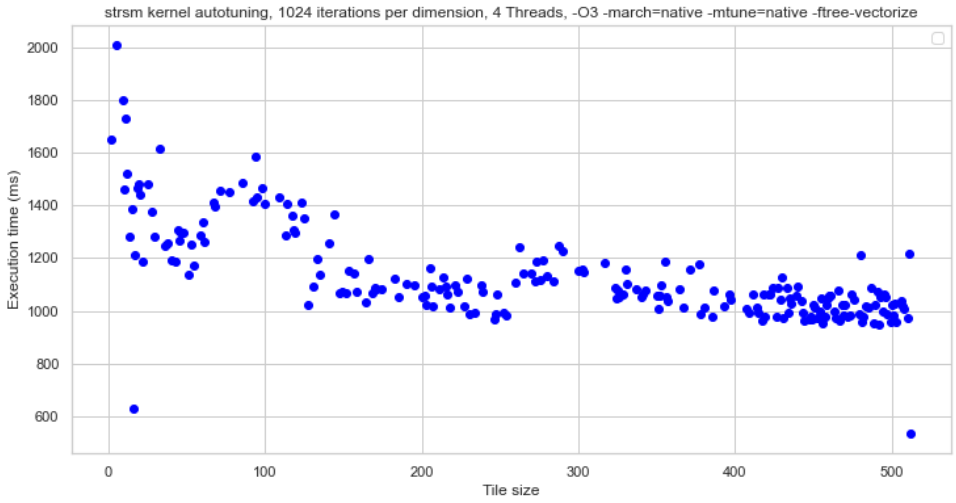

Figure A.5: Strsm kernel Autotuning

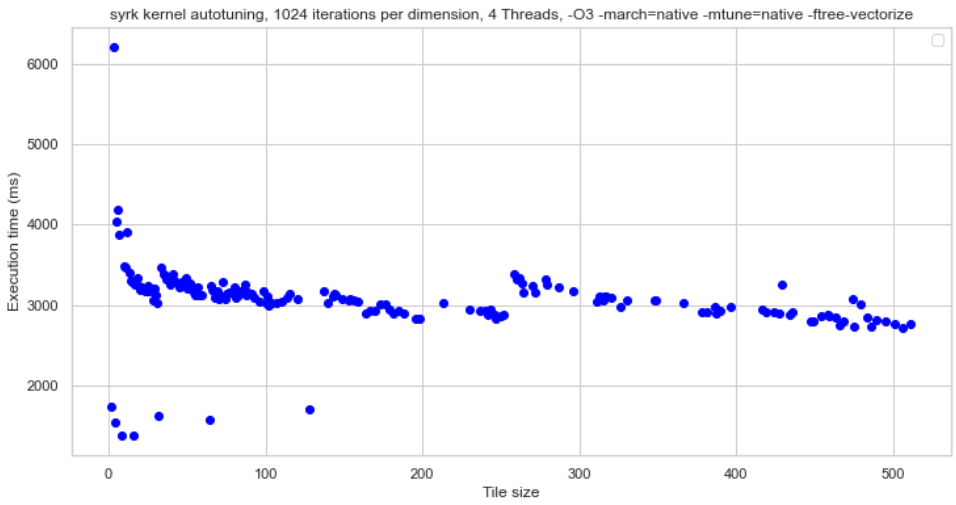

Figure A.6: Syrk kernel Autotuning

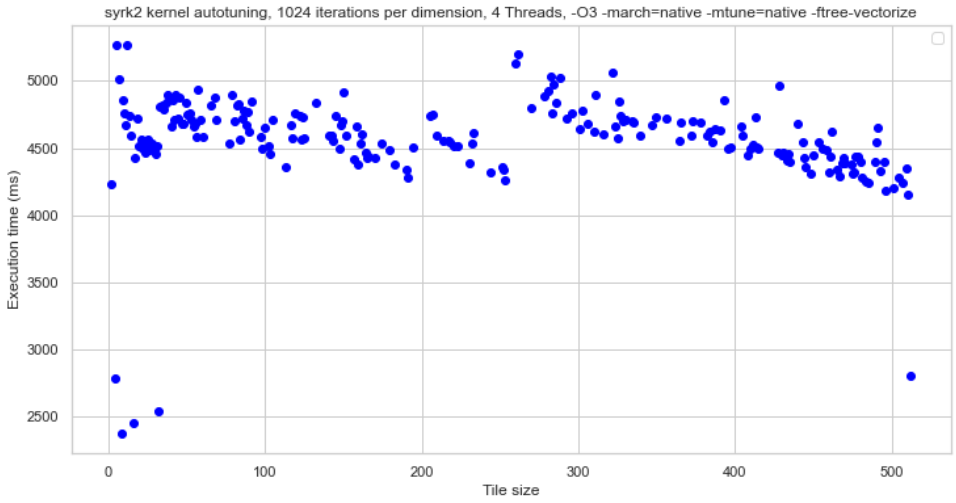

Figure A.7: Syr2k kernel Autotuning

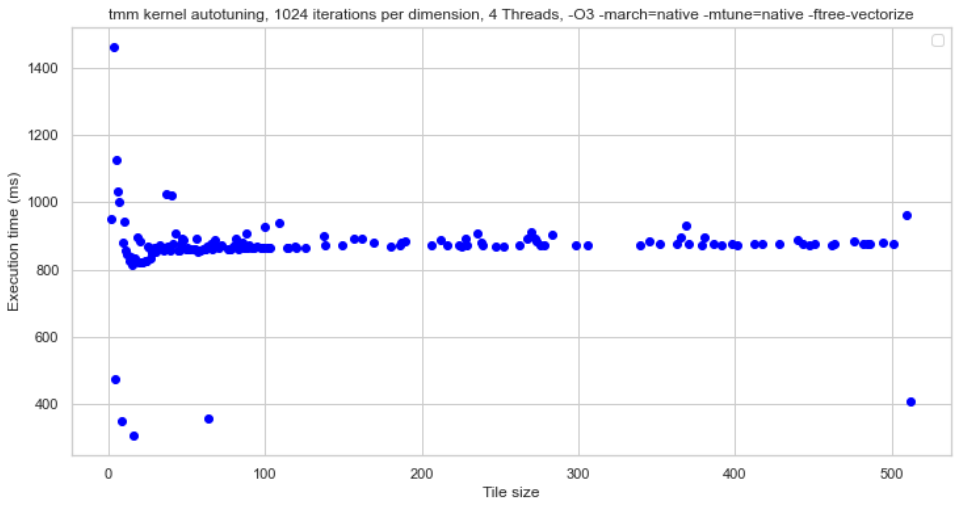

Figure A.8: Tmm kernel Autotuning

Appendix B

Paralelipiped tiling. Speedup distribution

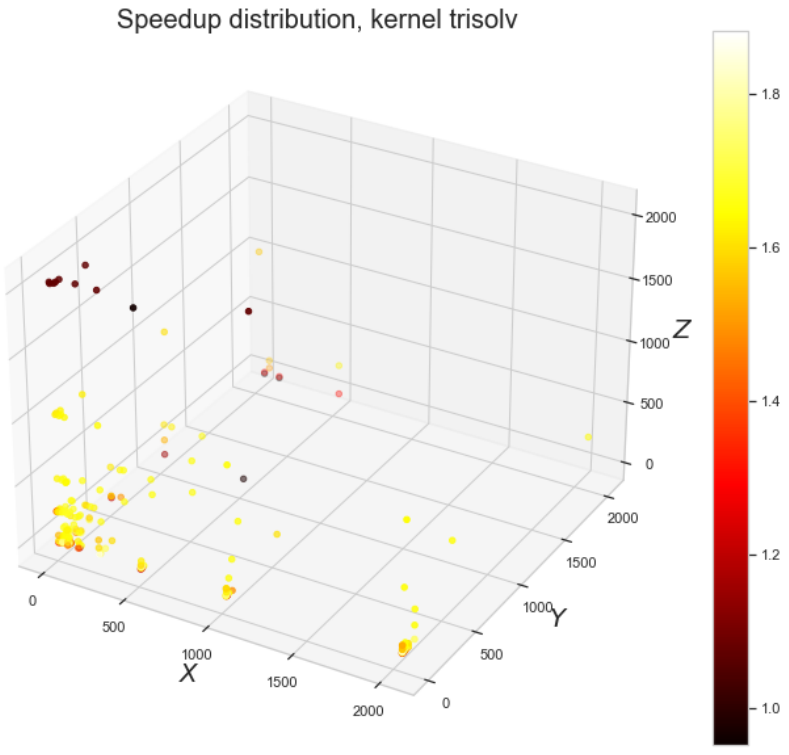

Figure B.1: Trisolv kernel. Speedup distribution

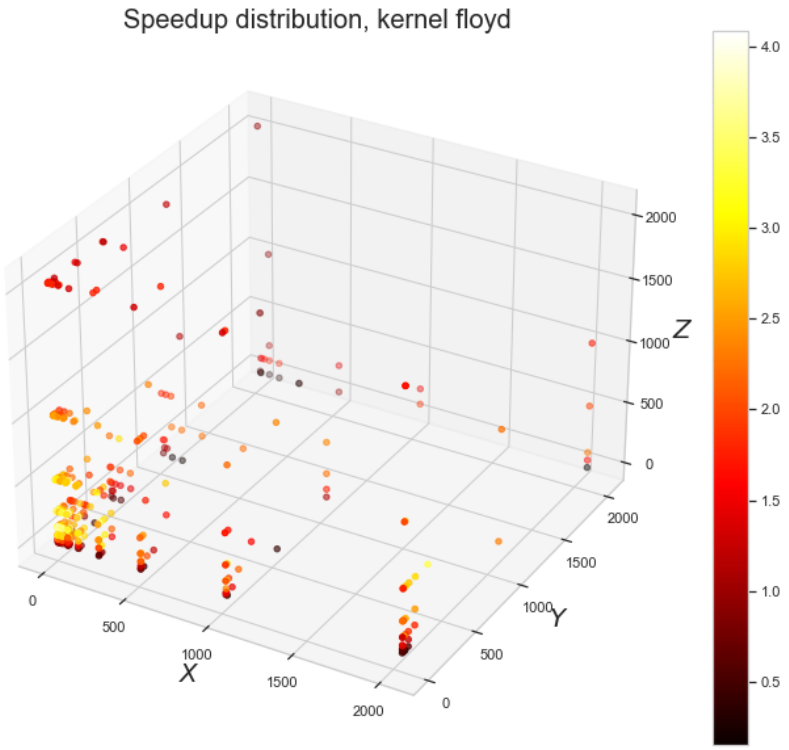

Figure B.2: Floyd kernel. Speedup distribution

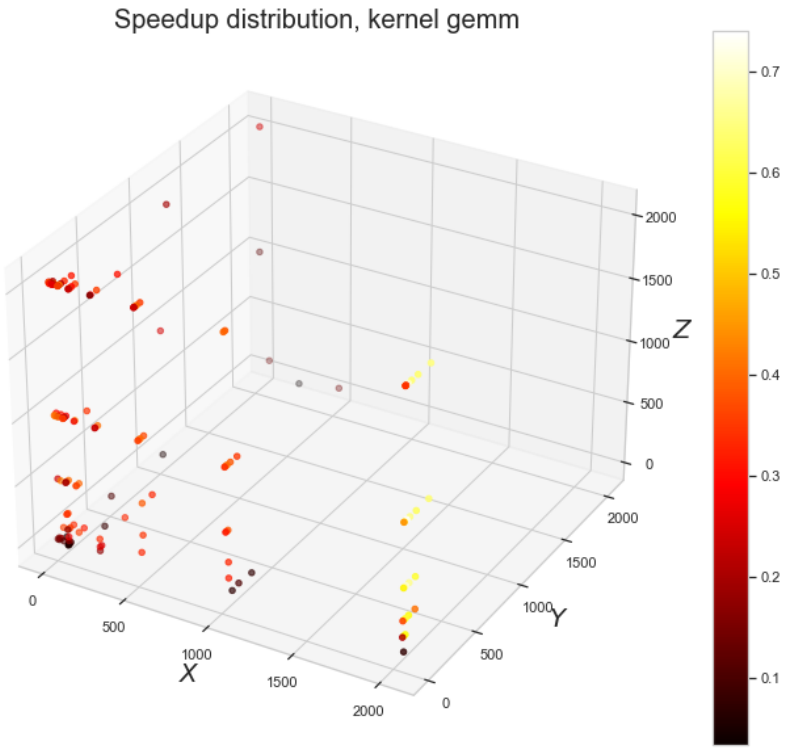

Figure B.3: Gemm kernel. Speedup distribution

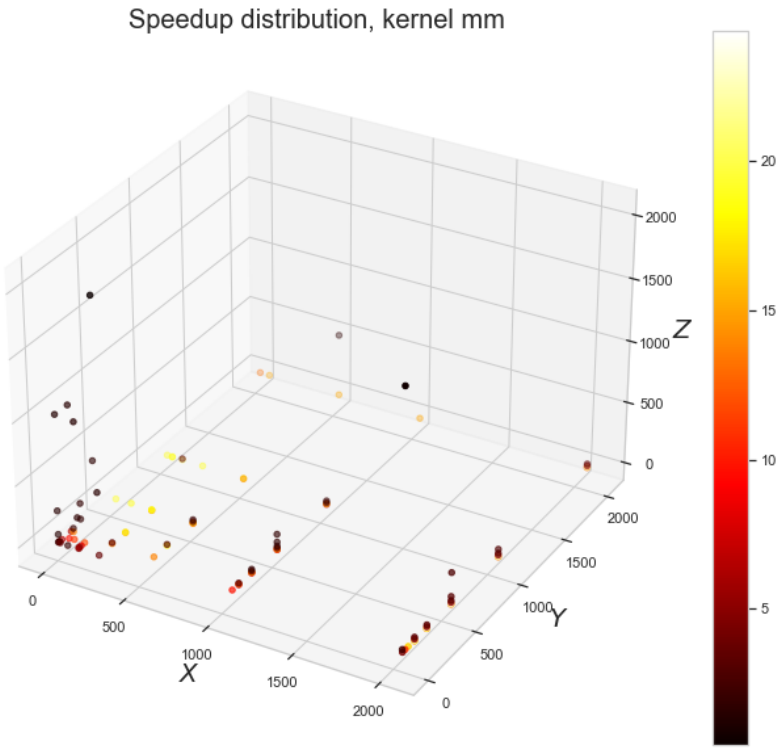

Figure B.4: MM kernel. Speedup distribution

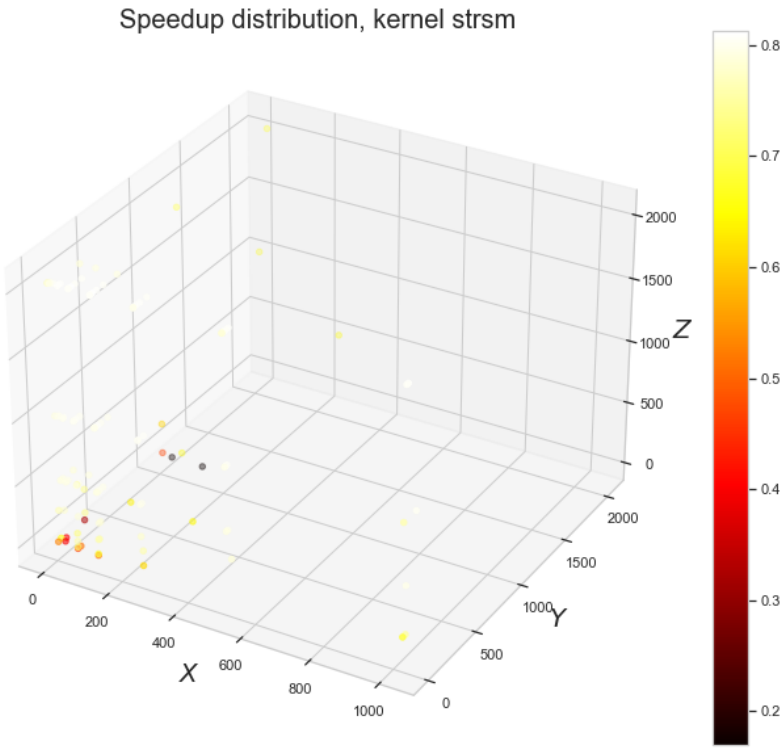

Figure B.5: Strsm kernel. Speedup distribution

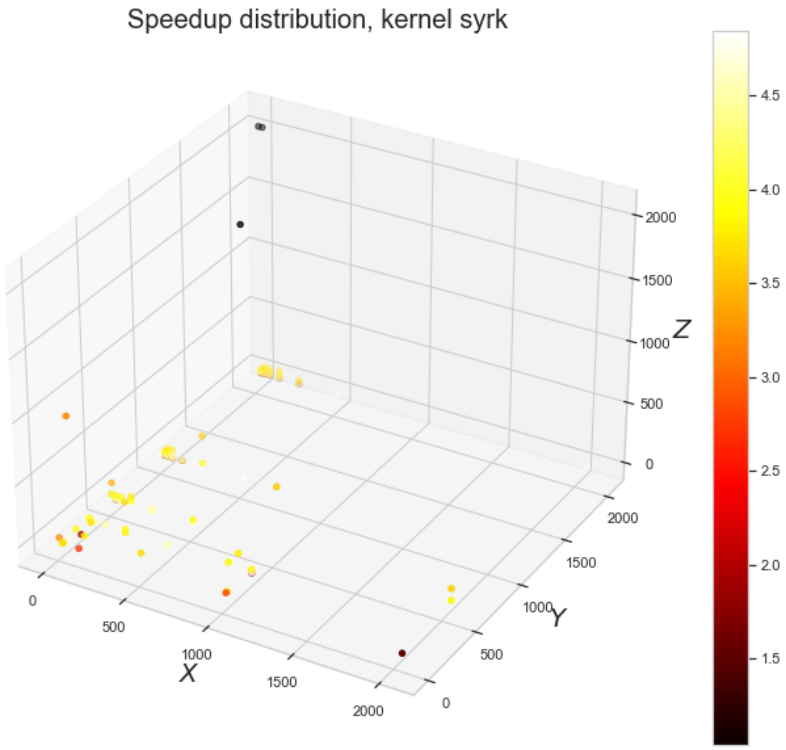

Figure B.6: Syrk kernel. Speedup distribution

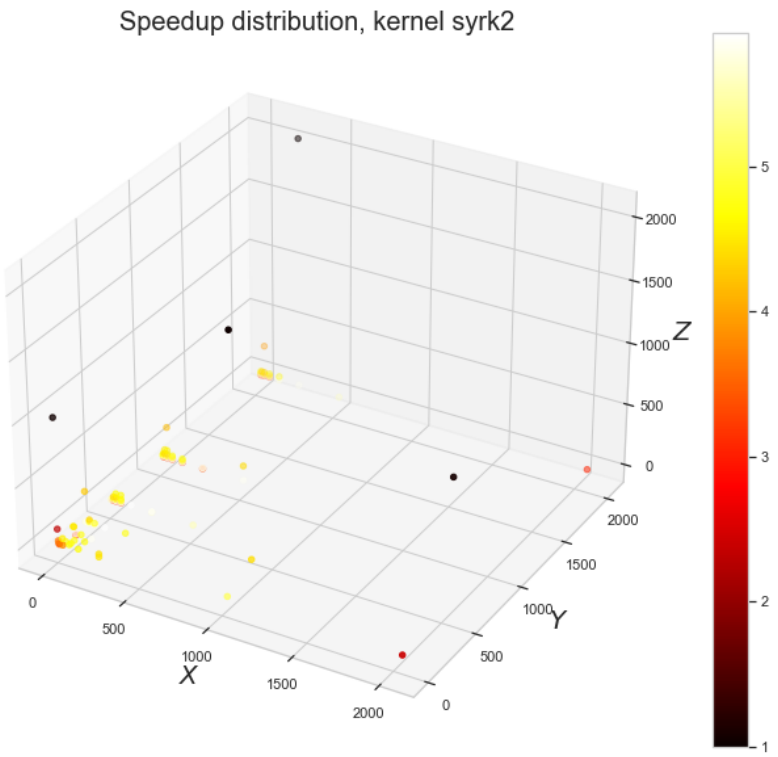

Figure B.7: Syr2k kernel. Speedup distribution

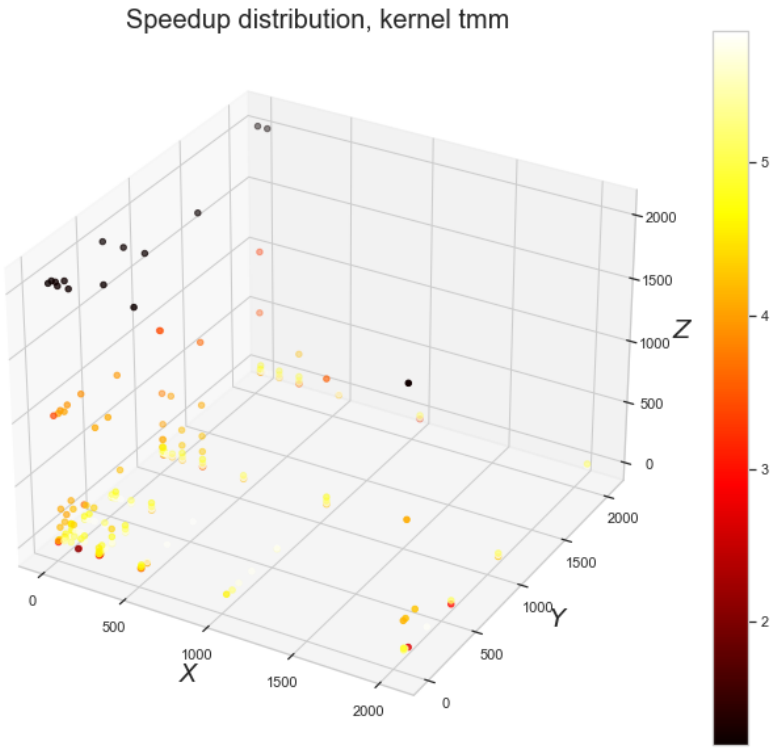

Figure B.8: Tmm kernel. Speedup distribution

Appendix C

Rectangular tiling. Speedup distribution

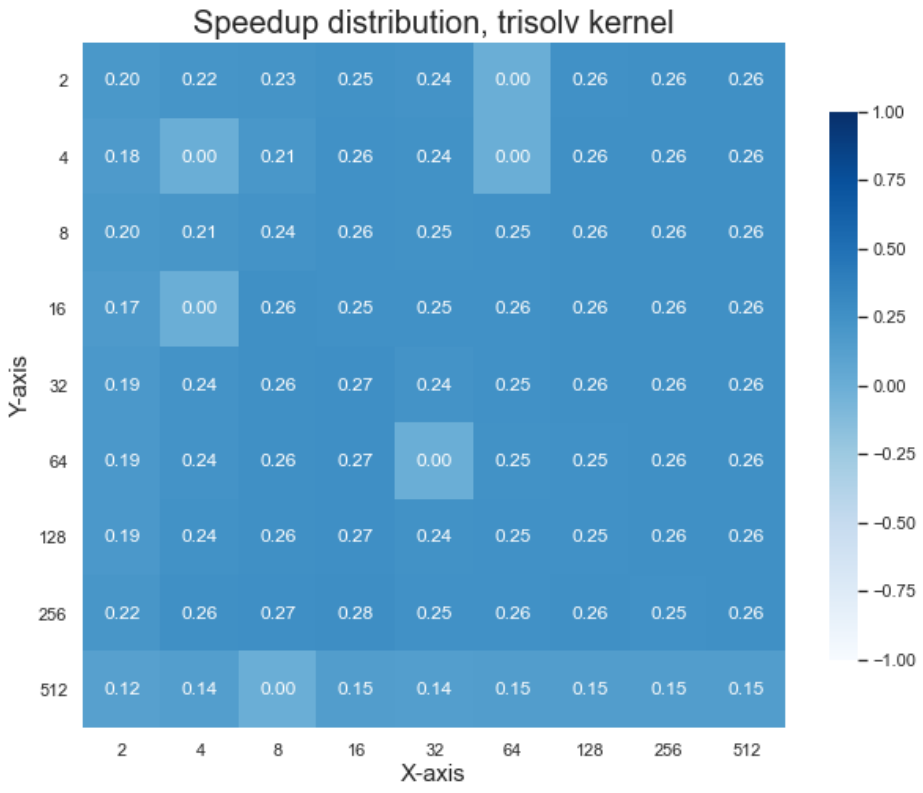

Figure C.1: Trisolv kernel. Speedup distribution

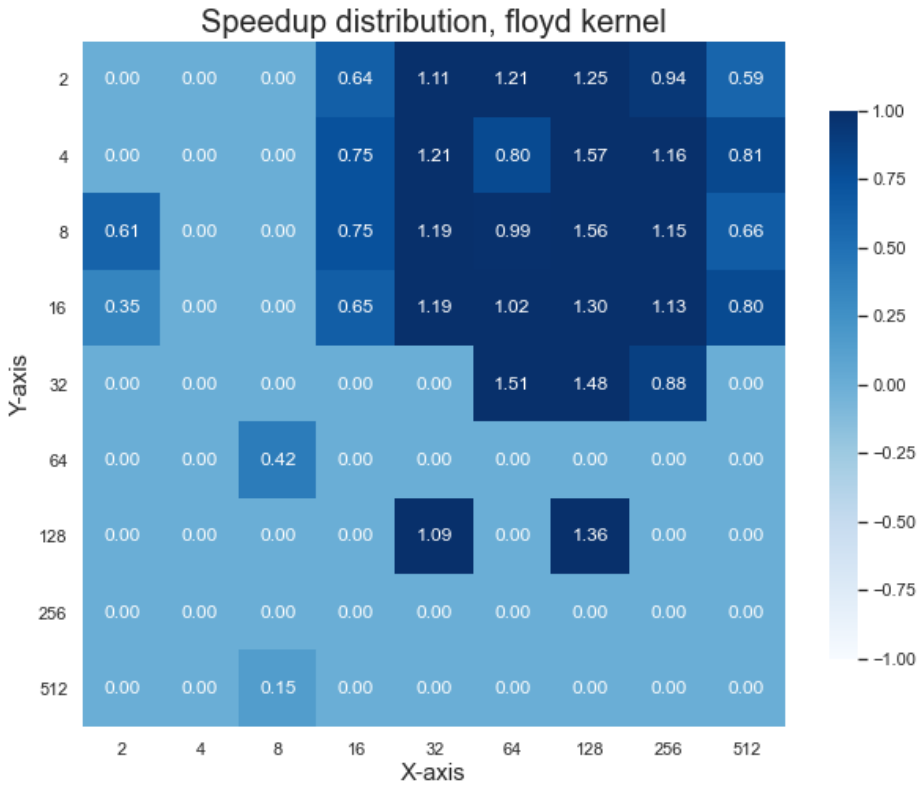

Figure C.2: Floyd kernel. Speedup distribution

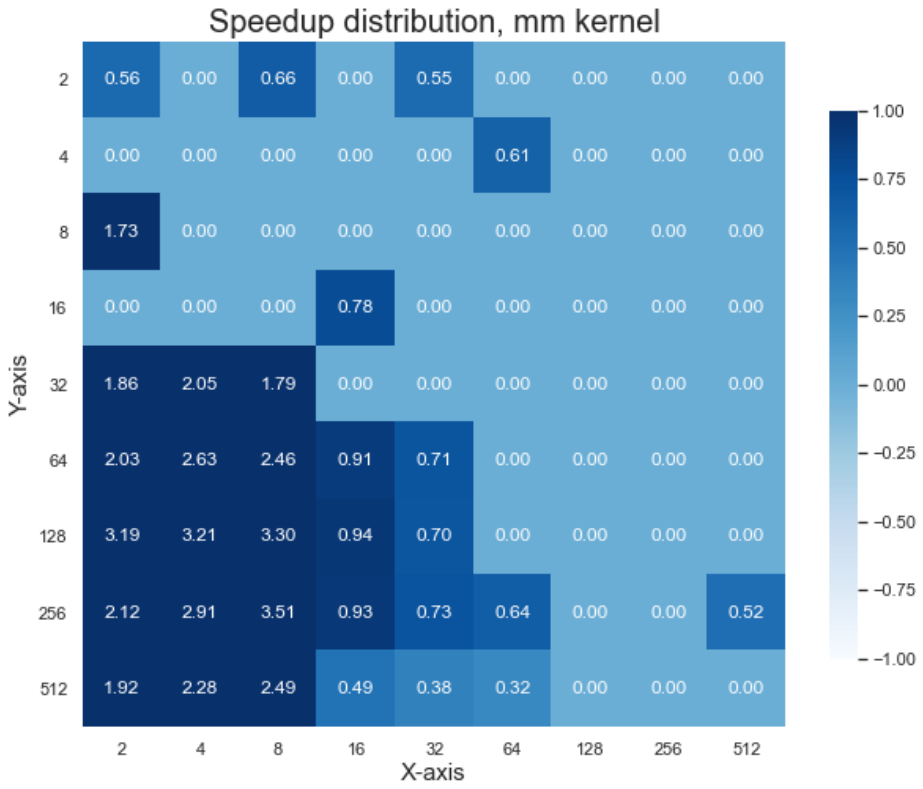

Figure C.3: MM kernel. Speedup distribution

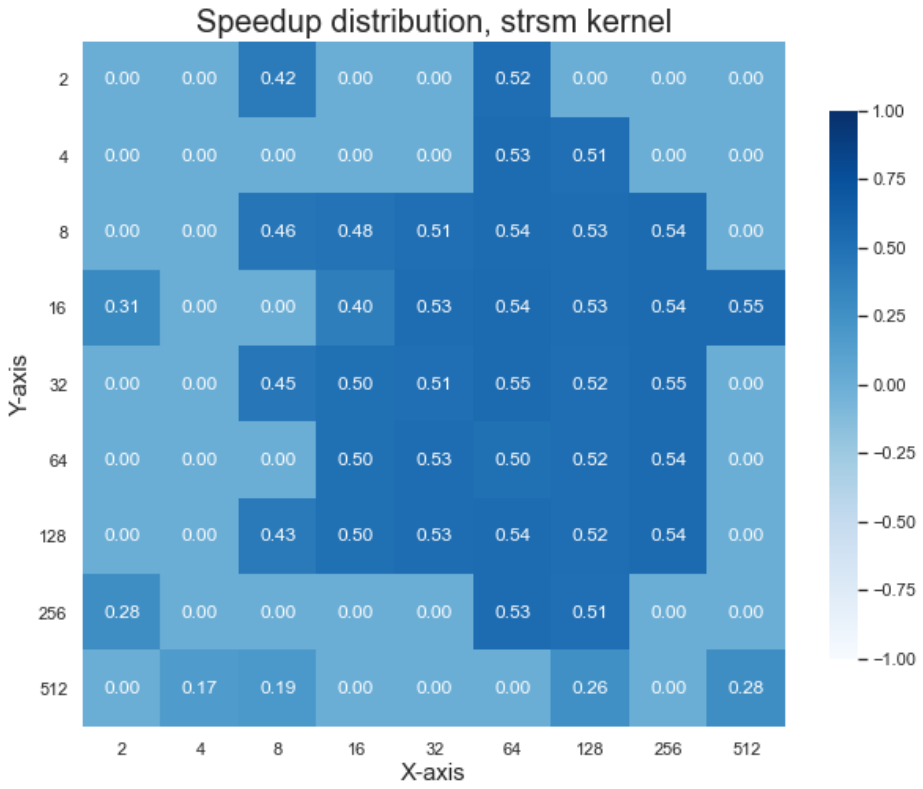

Figure C.4: Strsm kernel. Speedup distribution

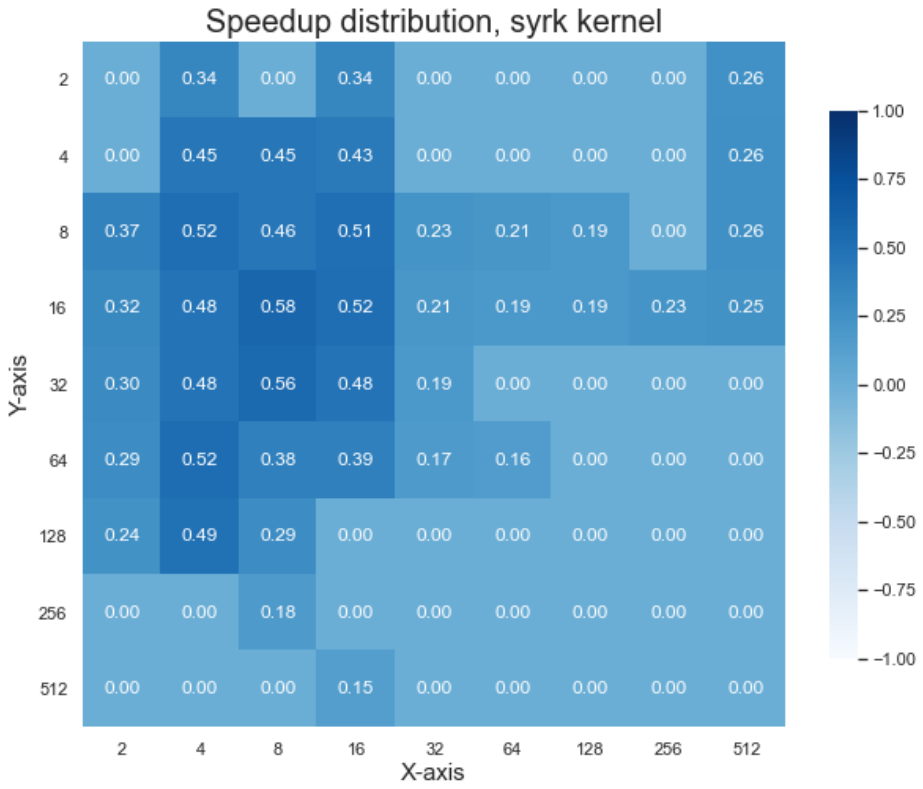

Figure C.5: Syrk kernel. Speedup distribution

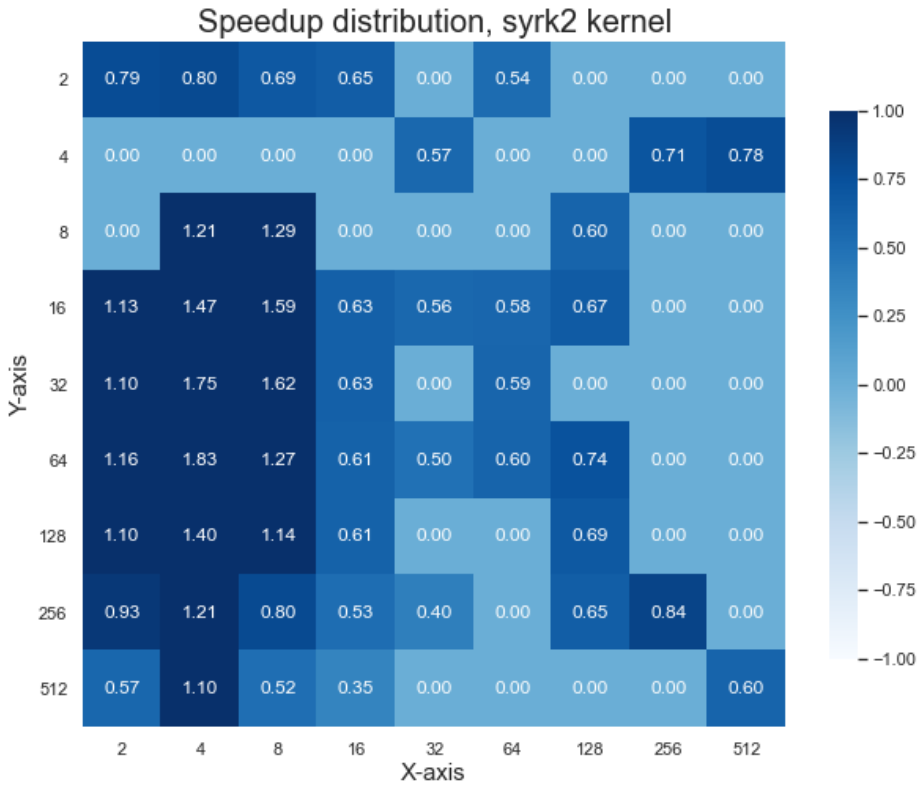

Figure C.6: Syr2k kernel. Speedup distribution

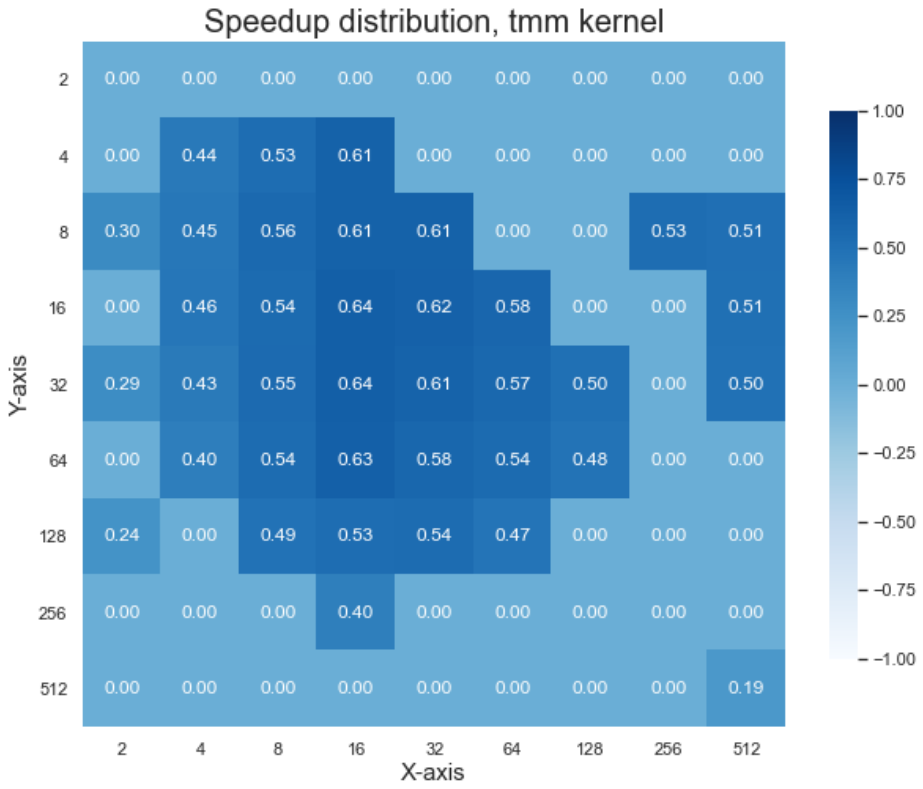

Figure C.7: Tmm kernel. Speedup distribution

Appendix D

Advanced tiling example. TP-LP scanning

Listing D.1: Tiled kernel

```
1 for (i_t = 0; i_t <= 254; i_t += 1)
2   { #pragma omp parallel for private(j_t,i_l,j_l)
3     for (j_t = pips_max_2(-i_t, -127); j_t <= pips_min_2((-i_t) + 127, 0); j_t += 1)
4       for (i_l = (16 * i_t) + 10; i_l <= pips_min_3((16 * i_t) + 40, ((16 * i_t) + (
            16 * j_t)) + 2062, ((-16) * j_t) + 2062); i_l += 1)
5       for (j_l = pips_max_3((((-i_l) + (16 * i_t)) + (16 * j_t)) + 5, -2042, (16 *
            j_t) - 20); j_l <= pips_min_3((((-i_l) + (16 * i_t)) + (16 * j_t)) + 20,
            (-i_l) + 2042, (16 * j_t) - 5); j_l += 1)
6       A[-j_l][i_l + j_l] = (A[(-j_l) - 2][(i_l + j_l) - 1] + A[(-j_l) - 1][(i_l +
            j_l) - 1]) + A[(-j_l) - 1][(i_l + j_l) - 2];}
```

Appendix E

Glossary

- **Features**: In machine learning and pattern recognition, a feature is an individual measurable property or characteristic of a phenomenon. Features are usually numeric, but structural features such as strings and graphs are used in syntactic pattern recognition. The concept of "feature" is related to that of the explanatory variable used in statistical techniques. [9]

- **Training set**: A training data set is a data set of examples used during the learning process and is used to fit the parameters (e.g., weights) of, for example, a classifier. [11]

- **Validation set**: A validation data set is a data set of examples used to tune the hyperparameters (i.e. the architecture) of a classifier. [11]

- **Test set**: A test data set is a data set that is independent of the training data set, but that follows the same probability distribution as the training data set. If a model fit to the training data set also fits the test data set well, minimal overfitting has taken place (see figure below). A better fitting of the training data set as opposed to the test data set usually points to over-fitting. [11]

- **Loss function**: In mathematical optimization and decision theory, a loss function or cost function is a function that maps an event or values of one or more variables onto a real number intuitively representing some "cost" associated with the event. [12]

- **Labeled data**: is a group of samples that have been tagged with one or more labels. [13]

- **Optimal experimental design**: are a class of experimental designs that are optimal with respect to some statistical criterion. [14]

- **Ground-truth value**: is information that is known to be real or true, provided by direct observation and measurement (i.e. empirical evidence) as opposed to information provided by inference. [15]

- **Embedding**: is a relatively low-dimensional space into which you can translate high-dimensional vectors. Embeddings make it easier to do machine learning on large inputs like sparse vectors representing words. Ideally, an embedding captures some of the semantics of the input by placing semantically similar inputs close together in the embedding space. An embedding can be learned and reused across models. [10]

RÉSUMÉ

Les optimisations de code peuvent être appliquées à des niveaux très différents. Par exemple, un programmeur peut optimiser un algorithme afin qu'il ait une complexité de temps de calcul asymptotique plus faible, et un compilateur peut transformer du code afin qu'il s'exécute plus rapidement. Cette thèse se place dans le contexte des transformations de programme source à source. Cela signifie qu'à partir d'un programme écrit dans un langage de programmation source, nous voulons obtenir un programme transformé dans le même langage de programmation qui est plus efficace. En particulier, nous nous sommes concentrés sur les transformations de nids de boucles imbriquées, car elles représentent les parties de code les plus consommatrices. Les transformations de boucles source-à-source ciblées sont le tuilage de boucle, le déroulement et l'échange de boucles. Cette thèse utilise des techniques d'apprentissage automatique. L'objectif principal de cette thèse est de définir une méthode appropriée pour les transformations de programme source-à-source afin d'améliorer une fonction de coût choisie pour une architecture homogène utilisant l'apprentissage automatique. Mes principales contributions se répartissent en deux groupes 1) la collecte d'un ensemble de codes représentatifs pour le ML et sa conception expérimentale optimale, 2) l'application d'une transformation de tuilage de boucle efficace. Le premier groupe se concentre sur les aspects de génération automatique de codes. Nous proposons le générateur COLAGEN pour former des ensembles de données qui capturent des modèles pertinents pour les transformations de boucles. Ce générateur peut produire des ensembles pour l'apprentissage et la validation de tailles arbitraires pour les besoins du Machine Learning. De plus, nous proposons une stratégie basée sur l'Active Learning pour générer des échantillons de données les plus représentatifs. Elle permet soit de réduire le temps d'apprentissage pour atteindre un certain niveau de performance, soit d'obtenir un modèle plus performant sous les mêmes contraintes de temps. Cette technique peut atteindre une vitesse jusqu'à 15% plus rapide en utilisant la même quantité de données. Le deuxième groupe cible la détermination des paramètres optimaux pour le tuilage de boucle. Deux types de modèles ont été développés. Le premier modèle cible la prédiction des tailles de tuiles optimales. Nous montrons que le meilleur pipeline consiste à prédire 1) l'accélération pour un noyau donné en utilisant des tailles de tuiles potentielles comme caractéristique d'entrée, puis 2) la taille de tuile qui maximise l'accélération prévue. De plus, nous montrons que le pavage parallélépipédique est la forme de tuile la plus appropriée pour la modélisation ML, il atteint des accélérations de 2,45x sur l'ensemble de validation et de 3,35x sur l'ensemble de test. En revanche, les prédictions pour le pavage cubique atteignent respectivement 1,85x et 1,58x. Les performances de notre modèle sont remarquables puisque 80% des gains optimaux sont atteints pour un pavage cubique et 70% pour un pavage parallélépipédique par rapport à ceux obtenus par un autotuneur. Le deuxième modèle cible l'ensemble étendu de paramètres de la transformation de tuilage. Nous proposons une approche pour prédire simultanément 1) la taille des tuiles, 2) les directions de balayage intra-tuiles, 3) les directions de balayage inter-tuiles et 4) la forme des tuiles pour les noyaux qui ont des dépendances de données uniformes. Nous montrons que ce type de paramètres pourrait apporter jusqu'à 30% d'accélération supplémentaire pour les nids de boucles 2D. Il atteint 7% d'accélération supplémentaire en moyenne pour les nids de boucles où les paramètres de direction de balayage sont avantageux. Notre modèle pour les paramètres de tuilage étendus pour les boucles 2D atteint jusqu'à 84% des performances maximales trouvées par un autotuneur.

MOTS CLÉS

Machine Learning, Optimisation, Compilation

ABSTRACT

Code optimizations could be applied at very different levels. For instance, one could propose an algorithm that would have lower asymptotic computation time complexity, and another - a compiler - could generate code that runs faster. This thesis takes place in the context of source-to-source program transformations. This means that having a program written in a source programming language (in our case C language), we want to obtain a transformed program in the same programming language which is more efficient. The cost function that guides the optimization can vary: execution time, compile time, memory consumption, a combination of different metrics, etc. The most significant and used in the context of this Ph.D. is the execution time and memory consumption. In particular, we focused on nested loop transformations, because typically they are the most consuming part of a program. The common source-to-source loop transformations targeted in this thesis are loop tiling, loop unrolling and loop interchange. This thesis uses machine learning techniques. Machine learning is a class of artificial intelligence methods, a distinctive feature of which is not a direct solution to the problem, but which learns by applying solutions to many similar tasks. The main objective of this thesis is to define an appropriate recipe for source-to-source program transformations to improve a chosen cost function for a homogeneous architecture using Machine Learning. My main contributions fall into two main groups 1) Data collection of representative code patterns for the ML purposes and optimal experimental design, 2) Application of an efficient loop tiling transformation. The first group focuses on aspects of synthetic code generation. We propose the COLAGEN generator to collect data that captures relevant patterns for loop transformations. This generator can produce training and validation sets of arbitrary sizes for Machine Learning needs. In addition, we propose a strategy based on Active Learning to generate the most representative data samples. It helps either to reduce the training time to reach a certain level of performance or to obtain a more efficient model under the same time constraints. This technique achieves up to 15% more speed-up using the same amount of data. The second group targets the determination of the optimal parameters for the loop tiling transformation. Two types of models have been developed. The first model targets the prediction of optimal tile sizes. We show that the best pipeline is to predict 1) the speedup for a given kernel using potential tile sizes as input feature, and then 2) the tile size that maximizes the predicted speedup. The parallelepiped tiling seems the most appropriate tile shape for ML modeling, it achieves 2.45x speedups on the validation set and 3.35x on the test set. In contrast, the predictions for cubic tiling reached 1.85x and 1.58x respectively. Our model performances are remarkable since 80% of the optimal gains are reached for cubic tiling and 70% for parallelepiped tiling compared to those obtained by an autotuner. The second model targets the extended set of parameters of the loop tiling transformation. We propose an approach to simultaneously predict 1) tile size, 2) intra-tile scanning directions, 3) inter-tile scanning directions, and 4) tile shape for kernels that have uniform data dependencies. We show that this kind of parameters could bring up to 30% additional speedups for 2D loop nests. It achieves 7% more speedups on average for kernels where the scanning direction settings are beneficial. Our model for extended tile settings for 2D loops achieves up to 84% of the peak performance found by an autotuner.

KEYWORDS

Machine Learning, Optimisation, Compilation