

## MASTER

### Application of machine learning for polyhedral optimizations

Zhang, I.

*Award date:*  
2019

[Link to publication](#)

#### **Disclaimer**

This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

#### **General rights**

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

# Application of Machine Learning for Polyhedral Optimizations

Graduation Thesis

I. Zhang, 1032404, TU Eindhoven

Supervised by:

Prof. Dr. H. Corporaal, TU Eindhoven

Dr. R. Jordans, TU Eindhoven

Prof. Dr. B. Juurlink, TU Berlin

Dr. B. Cosenza, TU Berlin

December 12, 2018

## Abstract

More and more computers are getting high-end architectures, including multicore processors and SIMD units to vectorize programs. In order to actually make use of these, a method needs to be found to conveniently optimize sequential code to parallel code.

The Polyhedral Model is a useful tool for automating Loop optimizations. By providing a numerical abstraction of the loops, they can easily be transformed, or used for other calculations.

In this thesis, we will use the Polyhedral Model to provide us with Polyhedral Features of code. These Features will be used to train a Machine Learning Model to predict which combination of Polyhedral Optimizations will be optimal.

State of the Art Polyhedral Optimizers that use Machine Learning so far, required the unoptimized code to compile and run once, in order to extract hardware counters to use as features that represent the code. Our approach using Polyhedral Features will improve on that method, by being 72.8x faster in terms of Selection Time due to not requiring a prior compile and run, while also being faster in terms of Run Time.

Additionally, we provide a multimodel method that will sacrifice a bit of Selection Time, in order to get the Run Time Speedup as high as 92.6% of the Run Time of Iterative approaches, while still being 135x faster in terms of Selection Time.

**Keywords**— Loop Optimization, Polyhedral Model, Machine Learning

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Motivation . . . . .	3
1.2	Background Information . . . . .	3
1.2.1	Loop Optimizations . . . . .	3
1.2.2	Polyhedral Model . . . . .	4
1.2.3	Machine Learning . . . . .	5
1.3	State of the Art . . . . .	6
1.4	Problem Statement . . . . .	7
1.5	Contributions . . . . .	7
1.6	Outline . . . . .	7
<b>2</b>	<b>Related Work</b>	<b>10</b>
2.1	Conclusions . . . . .	11
<b>3</b>	<b>Methodology</b>	<b>12</b>
3.1	Park et al. . . . .	12
3.1.1	Training Data . . . . .	13
3.1.2	Features . . . . .	14
3.1.3	Modeling . . . . .	14
3.1.4	Step by Step . . . . .	14
3.1.5	Conclusions . . . . .	15
3.2	Improvements . . . . .	15
3.2.1	Features . . . . .	15
3.2.2	Optimization Space . . . . .	16
3.2.3	Conclusions . . . . .	17
3.3	Minimal Approach . . . . .	17
3.4	Complete Approach . . . . .	17
3.5	Conclusions . . . . .	17
<b>4</b>	<b>Experiment Setup</b>	<b>20</b>
4.1	Setup . . . . .	20
4.1.1	Hardware . . . . .	20
4.1.2	Software . . . . .	20
4.2	Reproduction Park et al. . . . .	20
4.3	Selection Time . . . . .	21
4.4	Performance Metrics . . . . .	21
4.5	Models . . . . .	23
4.6	Conclusions . . . . .	23
<b>5</b>	<b>Results</b>	<b>24</b>
5.1	Research Questions . . . . .	24
5.2	Polyhedral Features . . . . .	24
5.3	Change of Optimizations . . . . .	25
5.4	Feature Normalization . . . . .	26
5.5	Selection Time vs Run Time . . . . .	27

<b>6 Conclusion</b>	<b>30</b>
6.1 Conclusions . . . . .	30
6.2 Future Work . . . . .	30
<b>Appendices</b>	<b>32</b>
<b>A Example Correlation</b>	<b>33</b>
<b>B Example OpenScop</b>	<b>34</b>

# Chapter 1

## Introduction

Nowadays, computers have reached very complex architectures. Even Personal Computers have Multicore processors, and are able to perform vector instructions. However, in order to get optimal use out of these, efficient parallel code needs to be written. [Asanovic et al., 2006] Since writing efficient parallel code gets more complex as the hardware gets more complex, methods to automatically tune these codes need to be explored. Especially the optimization of loops tends to be important, since a high percentage of the time is spent within loops.

### 1.1 Motivation

A problem with tuning code optimizations, is that the optimization space is near infinite. For each additional optimization, the time it takes for reaching a solution will increase multiplicatively. For this reason, the optimization space needs to be limited.

Take the correlation.c code from the PolyBench benchmark as example, shown in Appendix A. After iteratively compiling the code over the limited optimization space shown in Table 3.2 for 10 minutes, it reached a speedup of 115 times.

From the speedup, it's clearly visible that the iterative loop optimization has improved the code a lot, and that it has promising results. However, a method to reach similar results, that does not take as much time as Iterative Compilation would, is necessary.

The Polyhedral Model is often used for automating optimization of loops, since it provides a numerical abstraction of the loops, which can then easily be processed by code. Examples of programs that use the Polyhedral Model are P<sub>Lu</sub>To [Bondhugula et al., 2008], LLVM/Polly [Grosser et al., 2011] and PoCC [Pouchet et al., ].

Currently, there are numerous methods being applied for the optimization of loops. The most efficient results would be reached by using Iterative compilation, while results can be collected faster when using a Static Cost Model approach such as P<sub>Lu</sub>To. Alternately, research has been done on the use of Machine Learning, in order to approximate the best sequence of loop optimizations that is found by Iterative compilation, by finding patterns in the given features.

In this report we will cover the design of a novel Polyhedral Optimization solution, using Machine Learning.

### 1.2 Background Information

This section will cover the background information that is necessary to properly understand the content of the thesis. First the used loop optimizations will be discussed in Section 1.2.1, then the concept of the Polyhedral Model in Section 1.2.2 and finally, Machine Learning and some models, that will be used, will be discussed in 1.2.3

#### 1.2.1 Loop Optimizations

Loop optimizations are often performed for improving memory locality and exploiting parallelism [Bacon et al., 1994]. A small explanation on the optimizations that are applied and why they are

effective will follow.

### **Thread-Level Parallelization**

While not actually an optimization, by allowing the compiler to parallelize the code on thread-level, a lot of other optimizations become effective. Thread-Level Parallelization is based on the concept of spreading tasks over different threads of a CPU or GPU. By spreading the tasks over, for example, 2 threads, it is possible to speed the code up by nearly 2 times. However, this speedup is gated by the overhead of splitting the tasks over the threads, and the amount of tasks that can simply not be parallelized. In order to make a code so that this speedup is maximized, other optimizations are done to exploit this parallelism.

### **Loop Tiling**

Loop Tiling is a powerful optimization that divides the iteration space in tiles, to improve cache reuse, and to exploit inner-loop parallelism. Tile Size is a very relevant factor for this optimization, and the prediction of optimal tile size is currently still heavily being researched.

### **Loop Fusion/Distribution**

Loop Fusion is the act of combining Loops, while Loop Distribution distributes them. By fusing loops, loop overhead can be reduced, instruction parallelism can be increased and data cache locality can be improved.

Loop Distribution can improve instruction cache locality by having shorter loop bodies and create smaller loops with fewer dependences.

PLuTo, which we will use for the optimizations, defines Loop Fusion as either max-fuse, smart-fuse or no-fuse. Max-fuse might prevent parallelization and vectorization by attempting to fuse all loops. Smart-fuse tries to work towards those optimizations using a heuristic. No-fuse ignores all fusions altogether.

### **Wavefronting**

Wavefronting is an operation that allows for tiles to be executed in a different order, such that the tiles are being executed in a pipeline parallel fashion. While this optimization can expose coarse-grained parallelism, it does so at the cost of potential cache reuse.

### **Pre-Vectorization**

One of the operations that is automatically optimized by compilers is Vectorization. By vectorizing, it allows the processor to use hardware elements such as an SIMD to simultaneously perform multiple independent operations. This is generally done for the inner-most loop.

By Pre-Vectorizing, the to be vectorized operations are pushed to the inner-most loop.

### **Loop Unrolling**

When unrolling loops, the loop body is replicated a number of times and reduces the loop iteration space by that same number. By doing so, loop overhead is avoided, instruction parallelism can be exploited, and register and data cache locality can be improved. However, the pressure on the registers and instruction memory do increase.

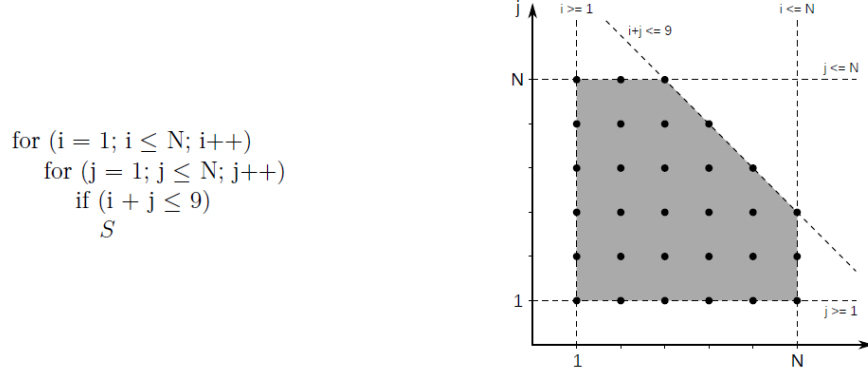
## **1.2.2 Polyhedral Model**

The Polyhedral model serves for an abstraction of affine loops, also called Static Control Parts (SCoPs), in the form of matrices. These matrices contain information on the Iteration Domain, the Read/Write Accesses, the Schedule and the Dependences.

This abstraction can be used for static loop analyses and can serve a useful tool for performing loop transformations.

## Iteration Domain

The Iteration Domain is composed of the vectors that bound the iteration space. An example is shown in 1.1. The dotted lines indicate the Iteration Vectors, and the Iteration Domain is a collection of these Iteration Vectors.



**Figure 1.1:** A SCoP (left) and its Iteration Space for  $N=6$  (right). Image from [Jimborean, 2012]

## Schedule

Schedules are used to determine the relative order of statements and in which loop body they are located. In case of the example code of Figure 1.1, there are 2 loops, the possible positions of the statement are: Before loop  $i$ , between loop  $i$  and loop  $j$ , and after loop  $j$ . If there are multiple statements, it would also differentiate which of the two statements occurs in which order. The schedule is also where most optimizations take place. By exchanging positions in the schedule it is possible to apply optimizations such as Loop Interchange.

## Read/Write Accesses

Read and Write accesses are also being displayed in the form of matrices. The amount of read and write accesses are tracked and for each access, the dimensions of the array can be tracked.

## Dependences

Dependences between statements are also part of the model. For each dependence, it keeps track which iteration of which statement depends on which iteration of another statement. Checking the Dependences is very important for making sure that the function of the code is not altered by applying optimizations.

For each Dependence, the type of Dependence is also tracked. These include Read-After-Write (RAW), Write-after-Read (WAR) and Write-after-Write (WAW). Note that Read-After-Read (RAR) is generally not tracked, due to being a false dependence, meaning that it would not affect the functionality of the code.

### 1.2.3 Machine Learning

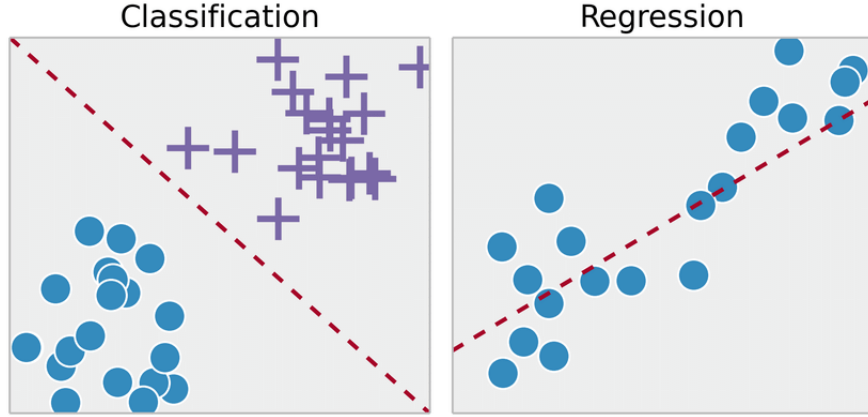
Machine Learning is used to create a model, that is found by recognizing patterns. [Nasrabadi, 2007] Generally, when solving problems with Machine Learning, the process will be as follows:

1. *Features* of the problem will be extracted and used as input for the model.
2. A collection of features of example problems will be used as *training set*, which will be used to create a model.
3. Then, the model will be tested in a *test set*, which contains example problems that are not used in the training set.



Problems are generally categorized as being *supervised* or *unsupervised*. In supervised problems, the training data already contains the answer of the problem, while in unsupervised it does not. In the case of supervised problems, which are mostly relevant for this thesis, the problems are further categorized as being *classification* problems and *regression* problems.

For classification problems, the goal is to find the category that the input belongs to. Regression problems want to predict a number of continuous variables as an output. A simple illustration is shown in Figure 1.2.



**Figure 1.2:** Classification (left) and Regression (right). Image from [Ngoma, 2017]

### Ordinal Regression

One of the Machine Learning models that will be introduced in this thesis is Ordinal Regression, as implemented by Joachims [Joachims, 2002], based on SVM. This method, rather than regressing towards a value such as Execution Time or Speedup, tries to predict a relative rank. such that the high difference in execution time does not weigh in too heavily.

### Multimodel

Besides using single Machine Learning algorithms without compiling and running any of the predicted results, it's also possible to use the results of compiling and running multiple Machine Learning algorithms, and select the best result out of that, as suggested by Park et al. [Park et al., 2013]. At the cost of some Selection Time, the Run Time will increase.

## 1.3 State of the Art

For transforming code to a Polyhedral Representation, LLVM/Polly [Grosser et al., 2011] and PoCC [Pouchet et al., ] are big names.

LLVM/Polly transforms LLVM Intermediate Representation (LLVM-IR) code to a Polyhedral Representation. LLVM-IR can be generated from multiple different programming languages, which makes this method very portable. LLVM/Polly also implements a number of Polyhedral Optimizers, like PLuTo [Bondhugula et al., 2008], and also allows for importing and exporting of Schedules, so that loops can manually be optimized.

PoCC (The Polyhedral Compiler Collection) is a collection of Polyhedral Representation and Optimization tools. It implements a toolchain containing CLooG, Clan and Candl for Polyhedral Representation, and Polyhedral Optimizers like PLuTo and LeTSeE.

Currently, Polyhedral Optimizations are generally done through a Static Cost model, like PLuTo [Bondhugula et al., 2008], or through Iterative Search, like LeTSeE [Pouchet et al., 2007a] [Pouchet et al., 2007b] [Pouchet et al., 2008]. Recently, people have started experimented with solutions to include Machine Learning as well, as done by [Ganser et al., 2017] and [Park et al., 2013].

We will compare Iterative Search, Park et al. and P<sub>Lu</sub>To in terms of Selection Time (the time it takes to select a solution), and Run Time (the time the selected solution takes to finish running once). To display this, a Pareto graph of the State of the Art solutions has been made and is displayed in Figure 1.3.

Notable is that Park et al.’s best model, even though it’s a Machine Learning method, takes hours to select the 30 results, while having results that are similar to simply running P<sub>Lu</sub>To –tile –parallel, which does the same thing within minutes.

Park et al.’s Multimodel does reach better times and does a better job approaching Iterative, at the cost of a bit of Selection Time.

## 1.4 Problem Statement

As was prefaced in Section 1.3, the main problem that we want to solve, is that there is a big trade-off between Selection Time and Run Time, and by using Machine Learning, we want to find a method that approaches the Run Time reached by Iterative methods, while also being fast in terms of Selection Time. The proposed solution would be to use the execution times of an Iterative method, then inserting it, together with other Polyhedral Features of the code, as input data of a Machine Learning application, which would in turn learn to predict which solution is optimal.

Since this implementation uses a similar approach as [Park et al., 2013], we will set their results as a baseline for the possible improvements.

In terms of Figure 1.3, the goal is to achieve similar or better Run Times than "Best Park et al.", while reducing the Selection Time to something similar to "P<sub>Lu</sub>To –tile –parallel".

## 1.5 Contributions

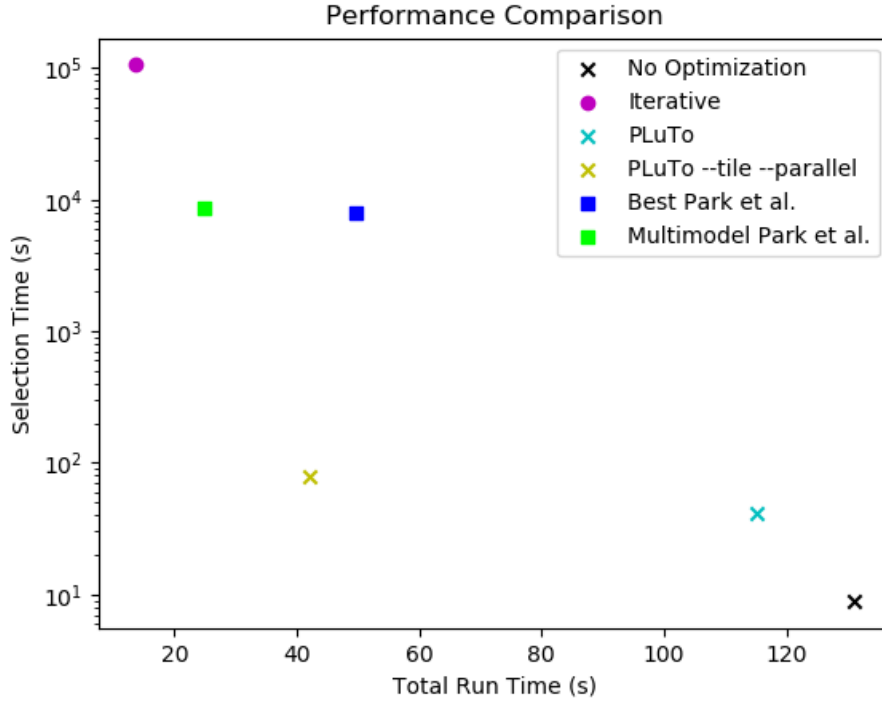
This thesis will have the following contributions on the area of Polyhedral Optimizations:

- An analysis of the State of the Art for Polyhedral Optimizers, in terms of Selection Time (time it takes to find a solution) and Run Time.
- A Machine Learning approach purely based on Polyhedral Information, rather than a pure flag tuning approach, that does not require any prior compiling and running of the test program has been created.
  - It is a direct improvement compared to Park et al.’s purely flag tuning based approach in terms of Run Time and is 72.8x faster in terms of Selection Time.
  - Unlike Park et al.’s approach, it has better results than simply performing P<sub>Lu</sub>To –tile –parallel, while also having a similar Selection Time as P<sub>Lu</sub>To
- A Multimodel approach, similar to the one by Park et al. has been created.
  - It is a direct improvement compared to Park et al.’s Multimodel approach in terms of both Run Time and is 10.90x faster in terms of Selection Time.
  - It reaches approximately 92.6% of the Run Time of Iterative Approaches in a factor 135x lower Selection Time, making it a good alternative for Iterative.
- The usage of Ordinal Regression, a Machine Learning model that has not been used in the context of Polyhedral Optimizations as of yet.

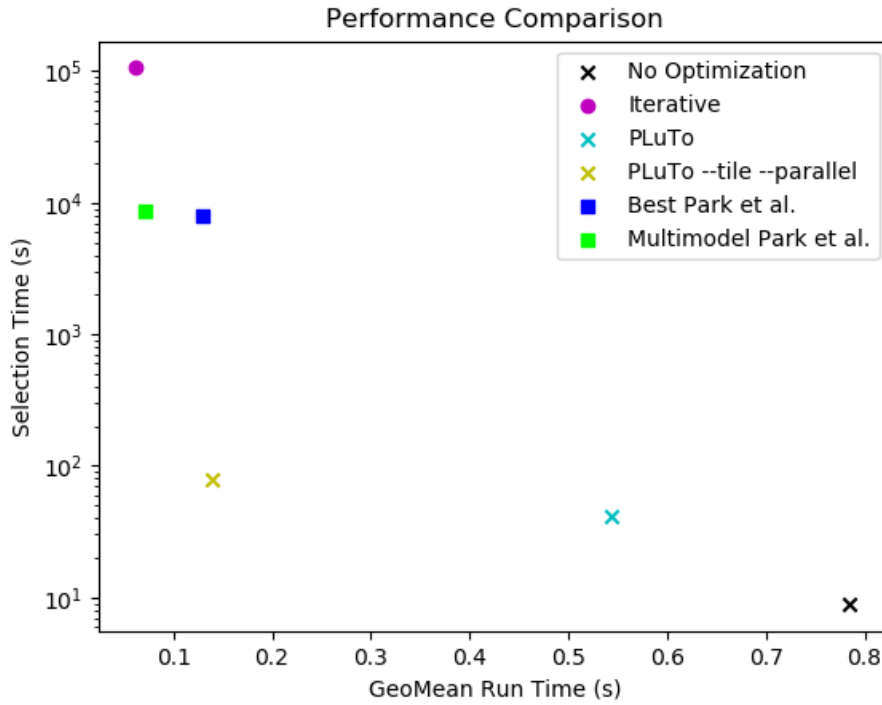
## 1.6 Outline

The thesis will follow the following structure:

- **Chapter 2** will go over the related works in the area of Loop Optimizations, Polyhedral Optimizers, application of Machine Learning for Iterative Compilations, Polyhedral Representation Tools .



(a) Total Run Time



(b) Geometric Mean of Run Time

**Figure 1.3:** Pareto graphs of the current state of the art. Selection Time is the time to select a result for all 30 PolyBench benchmarks. Total Run Time is the sum of the 30 Run Times, and GeoMean Run Time indicates the Geometric Mean of those 30 Run Times.

- **Chapter 3** will go in-depth on how Park et al. implemented their work, how we will alter it to a purely Polyhedral Model approach to improve its Selection Time, and how we will further improve on that.
- **Chapter 4** will discuss the setup of the experiments, the differences in implementation for the reproduction of Park et al.'s method and go over how the results of these experiments will need to be interpreted.
- **Chapter 5** will formulate research questions that need to be answered using experiments, and discuss the results of the experiments.
- **Chapter 6** will deliver the conclusions and discuss on potential improvements for future work.

## Chapter 2

# Related Work

For Loop Optimizations, several approaches have been made. Pochoir [Tang et al., 2011] is a compiler that optimizes stencil codes for parallelism. Halide [Ragan-Kelley et al., 2013] has designed a Domain Specific Language that can be used to make it easier to write efficient image processing code.

Polyhedral Optimizations have been researched in multiple methods. A purely iterative method would be LeTSeE [Pouchet et al., 2007a] [Pouchet et al., 2007b] [Pouchet et al., 2008]. LeTSeE limits the almost infinite search space, then filters it out so that only the legal and distinct schedules are left in order to be more time efficient. It also tries solving its time issues by generating a heuristic and solving it with a Genetic Algorithm.

Polyite [Ganser et al., 2017] finds a problem in LeTSeE, namely that it does not properly optimize for parallelization, so they add tiling and parallelization to the search space. Additionally, they also utilize a Genetic Algorithm to try and solve the search space more efficiently.

PLuTo [Bondhugula et al., 2008] is a purely heuristics-based method, that is also widely used for other optimizers. Both LeTSeE and Polyite implement it as well. The cost-model that is implemented can be used for, among other things, tiling, parallelization and loop fusion.

PoCC [Pouchet et al., ] implements a number of other optimizers, such as LeTSeE and PLuTo, so that they can be used in the same toolchain.

Park et al. [Park et al., 2013] uses a Machine Learning approach to solve the time issues. They try to predict the best combination of high-level optimizations that PLuTo (implemented within PoCC) can solve, by using either one of their 6 Machine Learning Models, or by using their multi-model that will run the best predicted combination of each of the models, and then choose the fastest one. The Machine Learning models in question are trained with the flags of the high-level optimizations and PAPI performance counters [Terpstra et al., 2010], such as L1 Cache Misses, amount of branch instructions and total amount of cycles, as features. They also experiment using two different hardware setups, and two different compilers, and play around with flags to optimize the Machine Learning Models with.

Joachims et al. [Joachims, 2002] alters the classical SVM to a Ranking SVM, also called Ordinal Regression, that regresses through relative ranking rather than using actual values. This Machine Learning algorithm will also be used in this thesis.

Multiple methods have used Iterative Compilation as a baseline for training Machine Learning Models. Milepost GCC [Fursin et al., 2011] uses the results of Iterative Compilation, alongside several flags and program features, to train a Decision Tree. Cosenza et al. [Cosenza et al., 2017] iteratively compiled stencil tuning configurations and used the execution time for training an Ordinal Regression Model for autotuning stencil computations.

A number of tools exist for extracting Polyhedral Representation; Clan [Bastoul, 2008] can be used to output an OpenSCoP file [Bastoul, 2011] that contains Polyhedral Information, Candl [Bastoul and Pouchet, 2012] can extract code dependences from code, CLooG [Bastoul, 2013] can generate normal code back from that OpenSCoP file. An example of such an OpenSCoP file is

shown in [Appendix B](#)

LLVM/Polly [[Grosser et al., 2011](#)] is an entire framework around that can transform code to LLVM/IR, an intermediate representation in which polyhedral optimizations can be done. It can generate a jSCoP file, which has a similar goal as the OpenSCoP file, and can be directly compiled from the LLVM/IR code.

## 2.1 Conclusions

Park et al. is one of the most direct references in the area of Polyhedral Optimizations using a Machine Learning approach. This is why we will take this as a baseline for our own approach.

In the following chapter, we will go more in depth on Park et al.’s method, why it is not quite what we want to reach, and how we plan to change it, such that it will solve our problems.

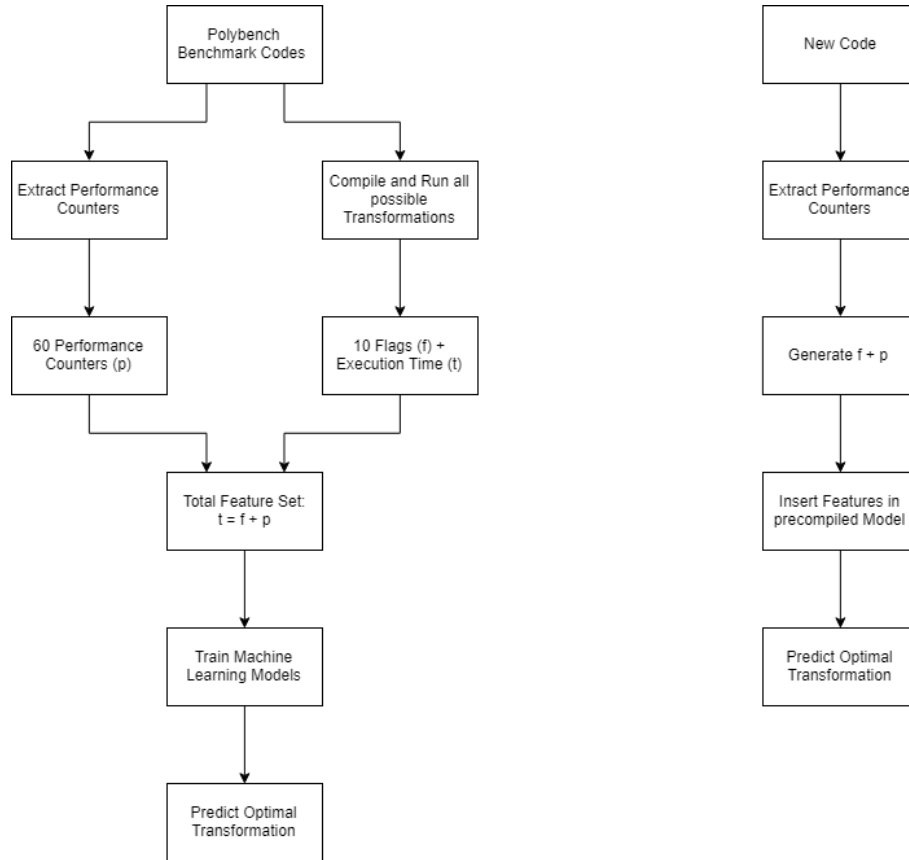
## Chapter 3

# Methodology

This Chapter will discuss the methods that are used, and will explain the thought process of why it would be an improvement, compared to the method that Park et al. used. To start off with, we will take Park et al.'s method as a baseline in Section 3.1, and discover improvements in Section 3.2, then implement a minimal approach that should be sufficient in Section 3.3 and a more complete approach in Section 3.4.

### 3.1 Park et al.

As earlier explained in Chapter 2, Park et al. also uses a Machine Learning approach for estimating optimal optimization sequences. A model of their approach is shown in Figure 3.1.



**Figure 3.1:** Model of Park et al.'s approach. Training method (left) and Prediction for new code (right)

### 3.1.1 Training Data

An important aspect of the design of a Machine Learning model is the Training Data. If the amount of training data is too limited, the model could become biased.

The idea of Park et al. is to use the 30 PolyBench benchmarks as training data. These are displayed in Table 3.1.

Benchmark	Description
2mm	2 Matrix Multiplications ( $\alpha * A * B * C + \beta * D$ )
3mm	3 Matrix Multiplications $((A*B)*(C*D))$
adi	Alternating Direction Implicit solver
atax	Matrix Transpose and Vector Multiplication
bicg	BiCG Sub Kernel of BiCGStab Linear Solver
cholesky	Cholesky Decomposition
correlation	Correlation Computation
covariance	Covariance Computation
deriche	Edge detection filter
doitgen	Multi-resolution analysis kernel (MADNESS)
durbin	Toeplitz system solver
fdtd-2d	2-D Finite Different Time Domain Kernel
gemm	Matrix-multiply $C=\alpha.A.B+\beta.C$
gemver	Vector Multiplication and Matrix Addition
gesummv	Scalar, Vector and Matrix Multiplication
gramschmidt	Gram-Schmidt decomposition
heat-3d	Heat equation over 3D data domain
jacobi-1d	1-D Jacobi stencil computation
jacobi-2d	2-D Jacobi stencil computation
lu	LU decomposition
ludcmp	LU decomposition followed by Forward Substitution
mvt	Matrix Vector Product and Transpose
nussinov	Dynamic programming algorithm for sequence alignment
seidel-2d	Seidel stencil computation
symm	Symmetric matrix-multiply
syr2k	Symmetric rank-2k update
syrk	Symmetric rank-k update
trisolv	Triangular solver
trmm	Triangular matrix-multiply

**Table 3.1:** PolyBench/C 4.2.1 benchmark codes

They apply Leave-One-Out Cross-Validation on these benchmarks, meaning that for each benchmark, they take the other 29 benchmarks as training set, and the chosen benchmark as test set.

They also limit the Optimization Space to the ones shown in Table 3.2. These are the optimizations that they considered as optimizations with an actual trade-off. Each benchmark will be compiled and ran with each of these optimizations.

Optimization	Options Park et al.	Reproduced?
Loop Fusion/Distribution	Max-fuse, smart-fuse, no-fuse	Yes
Loop Tiling	No tiling, tiling with size 32	Yes
Wavefronting	On, off	Yes
Thread-level Parallelizations	On, off	Yes
Pre-vectorization	On, off	Yes
SIMD-level Parallelizations	On, off	No
Register Tiling	Unroll factors: 1, 2, 4, 8	Yes

**Table 3.2:** Optimizations used for Park et al.



### 3.1.2 Features

The Features that Park et al. selected for training the Machine Learning Model, exist of several parts. First of all is the Execution Time of the optimized program. This is necessary for the prediction of future Execution Times.

Second are the optimization flags to encode the optimizations, and last are the PAPI hardware features, which are used as a static description of the code. This method was inspired by a previous work from Cavazos et al. [Cavazos et al., 2006]. The exact PAPI hardware counters that are extracted are shown in Table 3.3.

Category of PCs	List of PCs selected
Cache Line Access	CA-CLN, CA-ITV, CA-SHR
Level 1 Cache	L1-DCA, L1-DCH, L1-DCM, L1-ICA, L1-ICH, L1-ICM, L1-LDM, L1-STM, L1-TCA, L1-TCM,
Level 2 & 3 Cache	L2-DCA, L2-DCM, L2-DCR, L2-DCW, L2-ICA, L2-ICH, L2-ICM, L2-LDM, L2-STM, L2-TCA, L2-TCH, L2-TCM, L2-TCR, L2-TCW, L3-TCA, L3-TCM
Branch Related	BR-CN, BR-INS, BR-MSP, BR-NTK, BR-PRC, BR-TKN, BR-UCN
Floating Point	DP-OPS, FDV-INS, FML-INS, FP-INS, FP-OPS, SP-OPS
Interrupt/Stall	HW-INT, RES-STL
TLB	TLB-DM, TLB-IM, TLB-SD, TLB-TL
Total Cycle or Instruction	TOT-CYC, TOT-IIS, TOT-INS
Load/Store Instruction	LD-INS, SR-INS
SIMD Instruction	VEC-DP, VEC-INS, VEC-SP

**Table 3.3:** PAPI Counters used in Park et al. (Data from Park et al. [Park et al., 2013])

### 3.1.3 Modeling

For the Machine Learning Modeling, Park et al. experimented with multiple regression algorithms:

- Linear Regression (LR)
- Support Vector Machine (SVM)
- Instance-based Learning using K-Nearest Neighbor and Euclidean Distance (IBk)
- Instance-based Learning using Entropic Distance (K\*)
- M5 Model Tree Based Learning (M5P)
- Multi-Layer Perceptron (MLP)
- Ordinal Regression (OR) *Own Addition*

### 3.1.4 Step by Step

Park et al. essentially follows the following steps for training and testing the model:

1. Compile and run all unoptimized codes that are used for training with PAPI to extract PAPI hardware counters [Terpstra et al., 2010]. (See Table 3.3)
2. Compile and run every combination of optimizations to get execution time. (See Table 3.2)
3. Use the combination of optimizations, PAPI hardware counters and execution time as training data for each of the Machine Learning models.
4. Compile and run the test code with PAPI to extract the PAPI hardware counters.

5. Add the combinations of optimizations as features for the test code
6. Predict the speedup for each optimization using the created model.
7. (Optional) When calculating for multi-model: Compile and run the predicted best optimizations for each Machine Learning model, and select the best result.

### 3.1.5 Conclusions

The method proposed by Park et al. has one big issue that do not comply with the problem we want to solve. First of all, when compiling a new code, in order to extract features, a single compilation and run with PAPI is necessary. While this seems like it would add a rather low overhead, running with PAPI counters actually can end up taking up to half an hour.

Our goal is to create a method that does not need a single compilation for the prediction of new codes at all, and this does not comply with that prerequisite.

## 3.2 Improvements

In this section we will discuss the improvements over Park et al.'s method that will be implemented. These changes will be mainly about the Features and the Optimization Space.

### 3.2.1 Features

Since the biggest problem with Park et al.'s method is that they require a single compilation and run using PAPI to achieve a representation of the code, the solution would be to use alternate features to represent the code. This will be done in the form of Polyhedral Features. Another issue is the lack of Feature Normalization, which will be tested in multiple ways.

#### Polyhedral Features

From the OpenSCoP files, generated by Clan, Polyhedral Information can be generated. An example file is shown in Appendix B. Shortly summarized, there is a number of information that's about the entire code, and information such as Iteration Domain, Schedule (or Scattering Function), and Read/Write Accesses are done per statement. This means that depending on the number of statements in a SCoP, the number of features can vary.

The following Polyhedral Features are extracted:

- Number of Statements
- Number of Dependences
- Number of Iterators
- Number of different Iteration Domain edges
- Number of Read accesses
- Number of Read accesses per statement
- Number of Write accesses
- Number of RAW Dependences
- Number of WAR Dependences
- Number of WAW Dependences
- For each Statement:
  - Iteration Domain rows
  - Scattering rows
  - Amount of Reads
  - Amount of Write Access rows
  - Amount of Read Access rows

## Feature Normalization

Numerous Machine Learning models can get influenced by the Feature Vector Size. This is generally the case if the model uses Euclidean distance for calculations. To avoid this, it's important to normalize the Feature vector to values between 0 and 1. This was, as far as visible from Park et al.'s paper, not done for their models, and therefore has been reproduced as without normalization. The normalization can be done in multiple ways, and the following will be evaluated:

- Unit Vector Normalization
- Manual Normalization (If applicable, normalize using another feature, otherwise Min-Max normalization)

For Unit Vector Normalization, the method shown in Equation 3.1 is used for the features shown in Chapter 3.2.1. In the equation,  $\mathbf{x}'$  is the normalized Feature Vector and  $\mathbf{x}$  is the original Feature Vector.

$$\mathbf{x}' = \frac{\mathbf{x}}{\|\mathbf{x}\|} \quad (3.1)$$

In case of Manual Normalization, a number of features can be normalized using other features. The ones that are normalized this way are as follows:

- Number of RAW Dependences -> Number of RAW Dependences divided by Number of Dependences
- Number of WAR Dependences -> Number of WAR Dependences divided by Number of Dependences
- Number of WAW Dependences -> Number of WAW Dependences divided by Number of Dependences
- Iteration Domain rows -> Iteration Domain rows divided by Number of Iteration Domain edges
- Amount of Reads -> Amount of Reads divided by Number of Read Accesses

For the other features, a min-max normalization is applied, as shown in Equation 3.2. Here,  $x_n$  is a value in the feature vector,  $x'_n$  the normalized value, and  $\max(x_n)$  and  $\min(x_n)$  are defined as the maximum and minimum values of  $x_n$  over all feature vectors.

$$x'_n = \frac{x_n - \max(x_n)}{\max(x_n) - \min(x_n)} \quad (3.2)$$

The results of the different Normalizations will be tested in Chapter 5.4.

### 3.2.2 Optimization Space

The Optimizations that Park et al. used, are shown in Table 3.2. The first proposed change will be to omit the flag for Thread-level Parallelization. Since the goal of this thesis is to optimize towards parallel compilation, the end result should always include Thread-level Parallelization. Leaving the option to not parallelize in, will likely only improve the failure rate of the Machine Learning models.

Another optimization that was missing in Park et al.'s approach was more options for Tiling. Tile Size is often researched for parallel optimization [Coleman and McKinley, 1995] [Ryoo et al., 2008], and is yet unsolved, since it depends on a lot of hardware factors. By changing Tile Size, the amount of cache misses can heavily decrease, increasing the spatial locality of the code.

For the scope of this project, tile size shall only be checked for sizes of 16, 32, 64 and 128 in all dimensions (i.e. 16x16x16x16 if the code consists of 4 loops), since the amount of iterations that are added for each tiling configuration, would end up being too high. However, the implementation could be altered for a pure tile size estimation in future works.

The final Optimization set is shown in Table 3.4.

Optimization	Options
Loop Fusion/Distribution	Max-fuse, smart-fuse, no-fuse
Loop Tiling	No tiling, tiling with size 16, 32, 64 or 128 in all dimensions
Wavefronting	On, off
Thread-level Parallelizations	On
Pre-Vectorization	On, off
Register Tiling	Unroll factors: 1, 2, 4, 8

**Table 3.4:** *Our Optimization Space*

### 3.2.3 Conclusions

By switching the PAPI hardware features out for Polyhedral Features, the Selection Time should be reduced down to seconds, and if proper features have been selected, the Run Time should stay approximately the same.

Feature Normalization has been added so that a specific feature’s significance does not get blown up in case of models that use Euclidean distance or would get influenced by the feature size in some other way.

By removing the option to turn Thread-level Parallelization off, the miss-rate of the Machine Learning should decrease, and therefore it should improve the average results. In exchange, we also try the addition of Tile Sizes, which could lead to better results.

## 3.3 Minimal Approach

A minimal approach for achieving a faster Selection Time has been made and can be seen in Figure 3.2. This model changes only the PAPI hardware features to Polyhedral Features, and does not apply Feature Normalization and the alternate Optimization Space yet. This is done to compare the Features in vacuum.

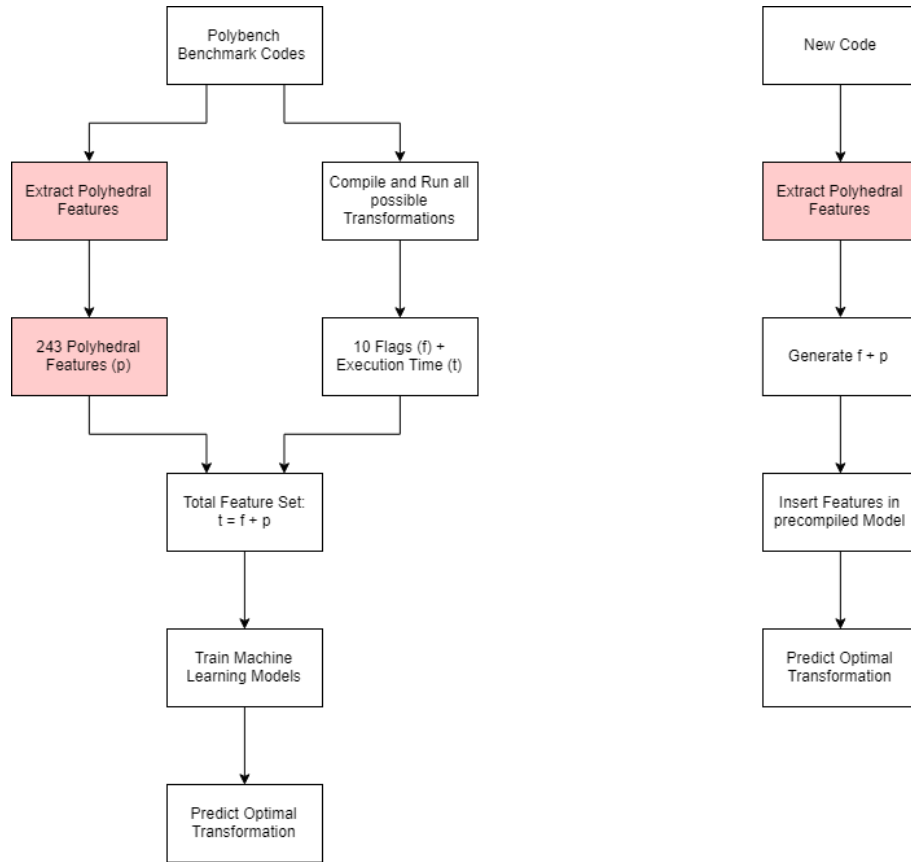
## 3.4 Complete Approach

Once the minimal approach has been completed and tested, a second, more in-depth approach will be performed. This will include the change of the Iterative version that is being used, and the addition of Feature Normalization. This is shown in Figure 3.3. This approach should be purely a Run Time improvement compared to the Minimal Approach

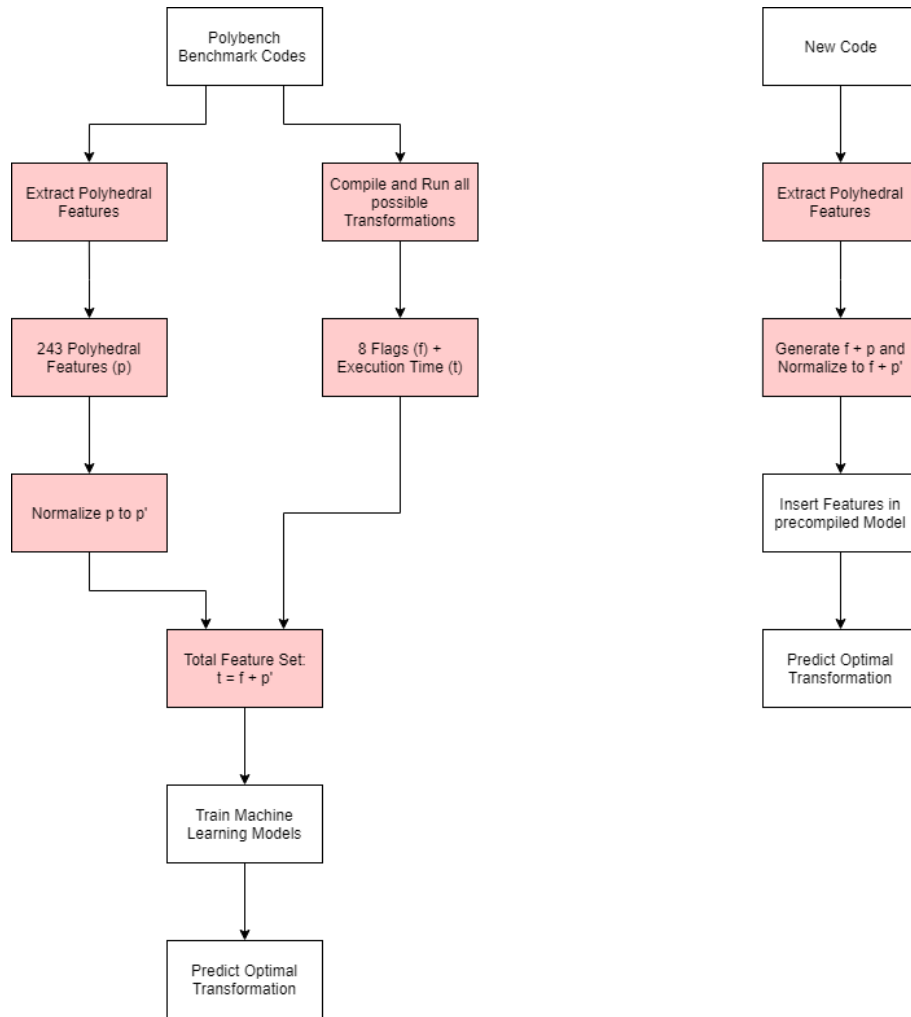
## 3.5 Conclusions

Park et al.’s method has been observed, and from the observations came a number of possible improvements. These improvements include the usage of Polyhedral Features rather than PAPI hardware counters as static code describing features, so that the high overhead of extracting those PAPI hardware counters can be removed. Additionally, Feature Normalization methods are explored and a different Optimization Space is explored.

These improvements are then implemented in a minimal approach, that only changes the features to Polyhedral Features, and a complete approach, that include all of the aforementioned changes.



**Figure 3.2:** Model of the minimal approach. Training method (left) and Application for new code (right). Red indicates the change compared to Park et al.'s model.



**Figure 3.3:** Model of the complete approach. Training method (left) and Application for new code (right). Red indicates the change compared to Park et al.'s model.

# Chapter 4

## Experiment Setup

This Chapter will explain the Hardware setup and the tools used for performing the experiments, the differences between Park et al.’s approach in their own paper and in our reproduction of it. It will also explain how to interpret the results.

### 4.1 Setup

#### 4.1.1 Hardware

The benchmarks are being performed on a system with

- Intel Xeon E5-2697v4 18x 2.30GHz Processor

#### 4.1.2 Software

The following versions of software are being used:

- PLuTo 0.11.4
- Clan 0.8.0
- Candl 0.6.2
- CLooG 0.18.4
- PolyBench/C 4.2.1

### 4.2 Reproduction Park et al.

There’s a number of differences with the original method of Park et al. and our reproduction of it. This could lead to a difference in the results of their paper and this thesis. The differences between the reproduction and the original method are as follows:

- PolyBench 4.2.1 has been used for benchmarking instead of PolyBench 2.1
  - Since PolyBench 2.1, a number of benchmarks has been removed and added.
- PLuTo 0.11.4 has been used instead of the PLuTo (0.5.5) implementation within PoCC (1.4.2) for optimizations
  - PoCC could not handle newer codes of PolyBench 4.2.1 and ended up slower for the other codes. This is probably due to the older version of PLuTo installed in PoCC.
  - SIMD-level Parallelizations were not available on PLuTo, so these have been skipped.
- Not all of the PAPI hardware counters used in Park et al. could be extracted. Instead, all available PAPI hardware counters have simply been used.

- Ordinal Regression has also been tested as Machine Learning model in addition to the other models.

As a result of the changes in the setup, the results of Park et al.'s paper are not directly comparable, and only the results of the reproduction will be used for comparison.

### 4.3 Selection Time

The term Selection Time has been used a number of times so far, defined as the time it takes to select a solution. This is not to be confused with Compilation Time, which is the time to compile a solution.

In the case of Iterative compilation, the time to select a solution is equal to the sum of the compilation time and run time of all optimizations. This is mathematically visualized in Equation 4.1.

$$t_{select,iterative} = t_{compile,unoptimized} + t_{run,unoptimized} + \sum_k t_{compile,k} + t_{run,k} \quad (4.1)$$

For P<sub>Lu</sub>To and P<sub>Lu</sub>To -tile -parallel, a selection is already made, so it only needs to be precompiled with P<sub>Lu</sub>To for that optimization sequence, as shown in Equation 4.2.

$$t_{select,P\text{Lu}To} = t_{precompile} \quad (4.2)$$

Park et al. requires the time to compile and run the unoptimized code, then the time for a Machine Learning Model to find a solution, and finally the time for precompiling. This is shown in Equation 4.3.

$$t_{select,park} = t_{compile,PAPI\_unoptimized} + t_{run,PAPI\_unoptimized} + t_{ML} + t_{precompile} \quad (4.3)$$

Our methods using Polyhedral Features, both the minimal and the complete approaches, replace the compilation time and run time of the unoptimized code with PAPI by a feature extraction time, as shown in Equation 4.4.

$$t_{select,polyfeat} = t_{feature\ extract} + t_{ML} + t_{precompile} \quad (4.4)$$

Estimates for what these times would be are shown in Table 4.1. These times will later be confirmed in Chapter 5.

Type	Selection Time Estimate
$t_{select,iterative}$	hours
$t_{select,P\text{Lu}To}$	seconds
$t_{select,park}$	hours
$t_{select,polyfeat}$	seconds
$t_{compile,unoptimized}$	less than a second
$t_{run,unoptimized}$	seconds to minutes
$\sum_k t_{compile,k} + t_{run,k}$	hours
$t_{compile,PAPI\_unoptimized}$	less than a second
$t_{run,PAPI\_unoptimized}$	minutes
$t_{precompile}$	seconds
$t_{feature\ extract}$	seconds
$t_{ML}$	seconds

**Table 4.1:** Estimates for the Selection Times

### 4.4 Performance Metrics

There's a lot of room of interpretation when talking about performance, so this section will explain a little bit on how the results will be evaluated and the reasoning behind it.



An example of results is displayed in Figure 4.1. The bolded results are the best result per benchmark per criterium. When looking at the results, there is no clear 'better' model; for some benchmarks, OR Park et al. performs better, for others MLP Park et al. performs better. For some performance metrics, such as Total Time and Average Time, MLP Park et al. performs better, for GeoMean of Time, OR Park et al. performs better.

Benchmark	Time		Speedup		Speedup Percentage	
	OR Park et al.	MLP Park et al.	OR Park et al.	MLP Park et al.	OR Park	MLP Park et al.
correlation	<b>0.180</b>	0.661	<b>60.03</b>	16.35	<b>51.85%</b>	14.12%
covariance	<b>0.178</b>	0.960	<b>61.84</b>	11.47	<b>49.03%</b>	9.09%
2mm	<b>0.171</b>	0.321	<b>14.35</b>	7.66	<b>43.16%</b>	23.03%
3mm	0.269	<b>0.169</b>	13.47	<b>21.52</b>	40.85%	<b>65.25%</b>
atax	<b>0.002</b>	0.031	<b>6.48</b>	<b>0.47</b>	<b>85.55%</b>	6.17%
bicg	<b>0.002</b>	0.017	<b>9.95</b>	1.26	<b>88.44%</b>	11.19%
doitgen	0.562	<b>0.412</b>	1.35	<b>1.84</b>	51.20%	<b>69.83%</b>
mvt	<b>0.002</b>	0.004	<b>24.05</b>	13.88	<b>81.11%</b>	46.82%
gemm	0.120	<b>0.060</b>	4.97	<b>9.89</b>	50.28%	<b>100.00%</b>
gemver	<b>0.004</b>	0.016	<b>14.12</b>	3.43	<b>91.32%</b>	22.20%
gesummv	<b>0.001</b>	<b>0.001</b>	8.07	<b>11.85</b>	63.80%	<b>93.72%</b>
symm	<b>0.994</b>	3.815	<b>3.75</b>	<b>0.98</b>	<b>95.83%</b>	24.97%
syr2k	0.176	<b>0.148</b>	27.94	<b>33.34</b>	81.79%	<b>97.60%</b>
syrk	0.146	<b>0.137</b>	6.22	<b>6.61</b>	76.54%	<b>81.41%</b>
trmm	0.069	<b>0.353</b>	<b>36.09</b>	7.09	<b>66.02%</b>	12.98%
cholesky	<b>0.170</b>	0.368	<b>8.31</b>	3.84	<b>98.73%</b>	45.65%
durbin	0.030	<b>0.007</b>	<b>0.19</b>	<b>0.80</b>	18.12%	<b>77.09%</b>
gramschmidt	<b>0.385</b>	<b>0.385</b>	<b>43.09</b>	<b>43.09</b>	<b>74.88%</b>	<b>74.88%</b>
lu	<b>0.242</b>	1.153	<b>18.94</b>	3.97	<b>90.98%</b>	19.08%
ludcmp	<b>2.135</b>	4.254	<b>1.98</b>	<b>0.99</b>	<b>0.35%</b>	0.18%
trisolv	<b>0.003</b>	0.004	<b>1.42</b>	<b>0.96</b>	<b>43.99%</b>	29.83%
deriche	<b>0.523</b>	0.694	<b>1.30</b>	<b>0.98</b>	<b>22.28%</b>	16.81%
floyd-warshall	31.379	<b>13.148</b>	<b>0.51</b>	<b>1.23</b>	8.23%	<b>19.64%</b>
nussinov	11.481	<b>4.747</b>	<b>0.54</b>	<b>1.31</b>	40.90%	<b>98.91%</b>
adi	<b>1.225</b>	9.945	<b>10.17</b>	1.25	<b>94.38%</b>	11.63%
fdtd-2d	0.272	<b>0.193</b>	5.74	<b>8.08</b>	65.72%	<b>92.50%</b>
heat-3d	<b>2.033</b>	5.797	<b>1.58</b>	<b>0.55</b>	<b>24.55%</b>	8.61%
jacobi-1d	<b>0.001</b>	<b>0.001</b>	2.02	<b>2.07</b>	92.21%	<b>94.63%</b>
jacobi-2d	0.311	<b>0.231</b>	5.91	<b>7.96</b>	68.64%	<b>92.47%</b>
seidel-2d	2.239	<b>1.537</b>	9.29	<b>13.53</b>	64.04%	<b>93.28%</b>
<b>Total</b>	55.309	<b>49.568</b>				
<b>Average</b>	1.844	<b>1.652</b>	<b>13.45</b>	7.94	<b>60.83%</b>	48.45%
<b>GeoMean</b>	<b>0.130</b>	0.205	<b>6.04</b>	3.82	<b>46.96%</b>	29.70%
<b>Total Speedup</b>	2.371	<b>2.646</b>				

**Figure 4.1:** Example results that will result from the experiment. For each of the PolyBench benchmarks, Time, Speedup and Speedup Percentage can be found. The best results are bolded. Speedup is the speedup compared to the Unoptimized code, Speedup Percentage is the percentage of speedup reached compared to the Optimal code found by Iterative Compilation

For this reason, it needs to be distinguished which metrics will be considered important in this thesis.

Total Time, Average (Arithmetic Mean) Time and Total Speedup are essentially the same, except that Average is normalized by the number of benchmarks, and Total Speedup is normalized by the total Run Time of the unoptimized code. A problem with the Total Time metric, is that high Run Time codes will have a bigger influence on the result than smaller codes. Therefore this metric will be used when optimization of high Run Time codes is actually more important than the ones of low Run Time codes.

GeoMean (Geometric Mean) of Time on the other hand, is less influenced by high Run Time codes, and will be used in a more average case.

Speedup is the speedup of the code compared to the unoptimized code. It's generally frowned upon to use the arithmetic mean for normalized numbers, due to the inaccuracies created when observing high variance results [Fleming and Wallace, 1986]. For this reason, only the GeoMean will be observed in regards of Speedups.

Speedup Percentage is the speedup of the code normalized to the optimal Speedup achieved with Iterative Compilation. Since this is also a normalized number, only the GeoMean shall be observed.

However, since the GeoMean of Time, GeoMean of Speedup and GeoMean of Speedup Percentage all have similar results, only the GeoMean of Time will be used.

## 4.5 Models

After running all results, a total of 7 columns of results per approach will be outputted. However, in the end, when using the models on a completely unknown code, only one of the 7 predictions will be used. For future reference, "Best Total Time *Approach (Model)*" will be the name of the model that had the best Total Time, "Best GeoMean Time *Approach (Model)*" will be the name of the model that had the highest GeoMean Time.

In addition to the "Best" models, the codes will also be tested using the Multimodel approach that Park et al. used. This means that, per benchmark, each predicted optimization will be compiled and run, and the prediction with the lowest Run Time will be chosen as final optimization. While this is against the goal of creating an optimizer that does not require any compiling and running, it is still an option to sacrifice a bit of Selection Time for a better Run Time. Additionally, it serves as a good evaluation method to see if the changes made it so that more optimal results are reached.

## 4.6 Conclusions

The Research Questions from [5.1](#) will be answered in the next Chapter. The results will be compared with the results of our reproduction of Park et al. and will mostly be evaluated using the Total Time and GeoMean of Time metrics.

# Chapter 5

## Results

This Chapter will go over the Research Questions, and the respective results of the experiments.

### 5.1 Research Questions

Experiments will be performed to answer the following Research Questions:

1. How does performance change when using Polyhedral Features instead of PAPI hardware counters as features? (Section 5.2)
2. How does changing the optimizations influence the average Run Time and the selected Run Time? (Section 5.3)
3. How do different types of Feature Normalization affect performance? (Section 5.4)
4. How do the results of every model look in terms of Selection Time and Run Time? (Section 5.5)

### 5.2 Polyhedral Features

The results of comparison of our minimal approach of Chapter 3.3 and Park et al.'s approach of Chapter 3.1, are seen in Figure 5.1.

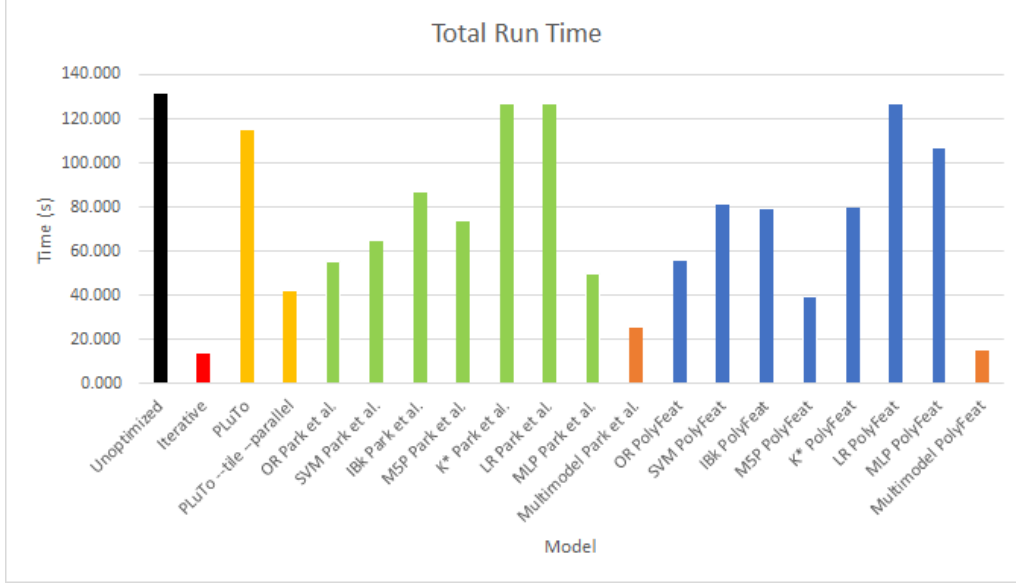
In these graphs, each of the Machine Learning Models, as well as the Multimodels for Park et al.'s approach and our Minimal Polyhedral Features approach are shown.

Regarding individual Machine Learning Model results, OR, IBk and LR performed similar for both approaches, SVM and MLP performed better for Park et al., and M5P and K\* performed better for our approach in terms of Total Time. In terms of Geometric Time, OR, SVM, M5P, K\* and LR had similar times, IBk performed better for Park et al. and MLP performed better for our minimal approach.

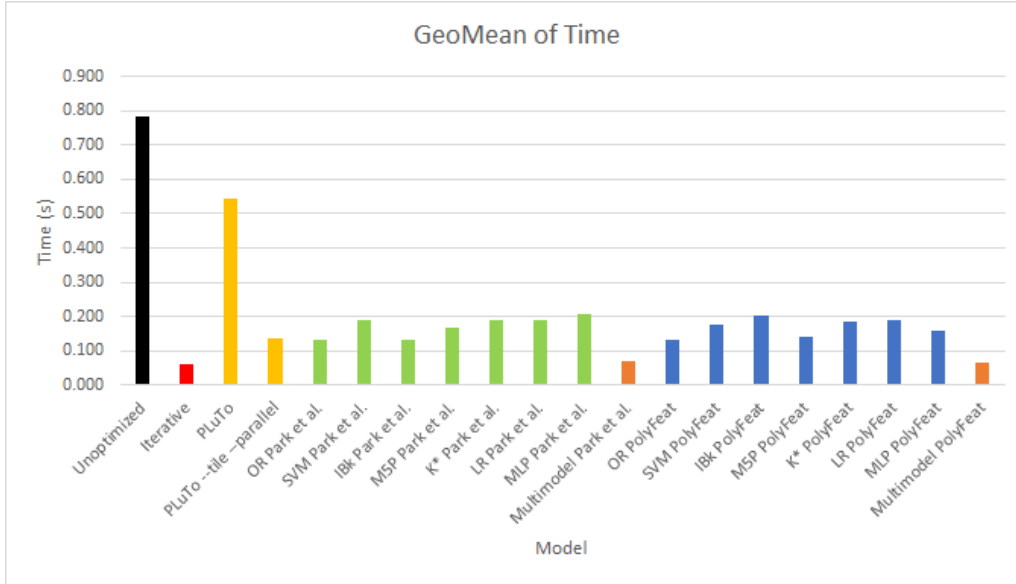
A more compact version of the results can be found in Table 5.1. PolyFeat has a better Best Total Time model than Park et al. in terms of Total Run Time and GeoMean Run Time. The results for Best GeoMean Time are the same. PolyFeat also improves on Park et al. for the Multimodel in terms of both Total Run Time and GeoMean Run Time.

Model	Total Run Time (s)	GeoMean Run Time (s)
Best Total Time Park et al. (MLP)	49.568	0.205
Best Total Time PolyFeat (M5P)	39.491	0.142
Best GeoMean Time Park et al. (OR)	55.309	0.130
Best GeoMean Time PolyFeat (OR)	55.309	0.130
Multimodel Park et al.	24.879	0.071
Multimodel PolyFeat	14.639	0.065

**Table 5.1:** Total Run Time and GeoMean Run Time of the different approaches



(a) Total Run Time



(b) Geometric Mean of Run Time

**Figure 5.1:** The resulting times and speedups of our Minimal Polyhedral Features approach (PolyFeat) and Park et al.’s approach (Park et al.)

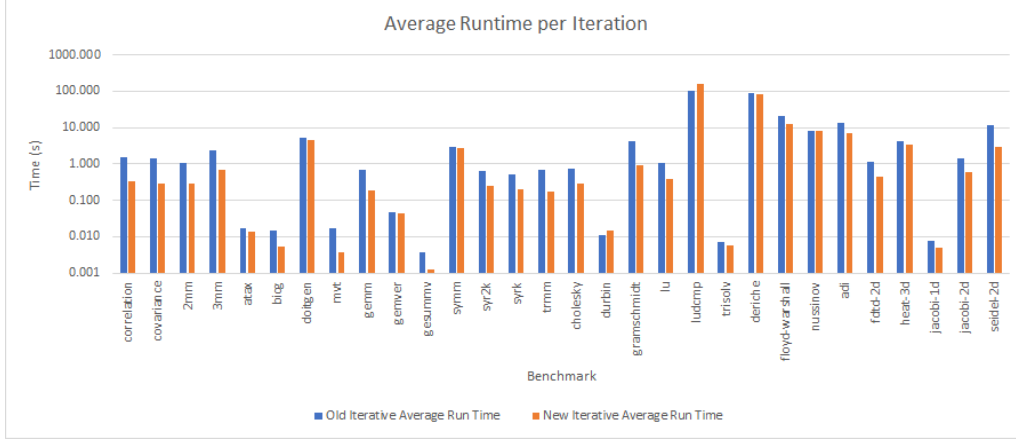
### 5.3 Change of Optimizations

The results of the changes explained in Chapter 3.2.2, have been tested and compared per benchmark in Figure 5.2. Note that the results are on a logarithmic scale.

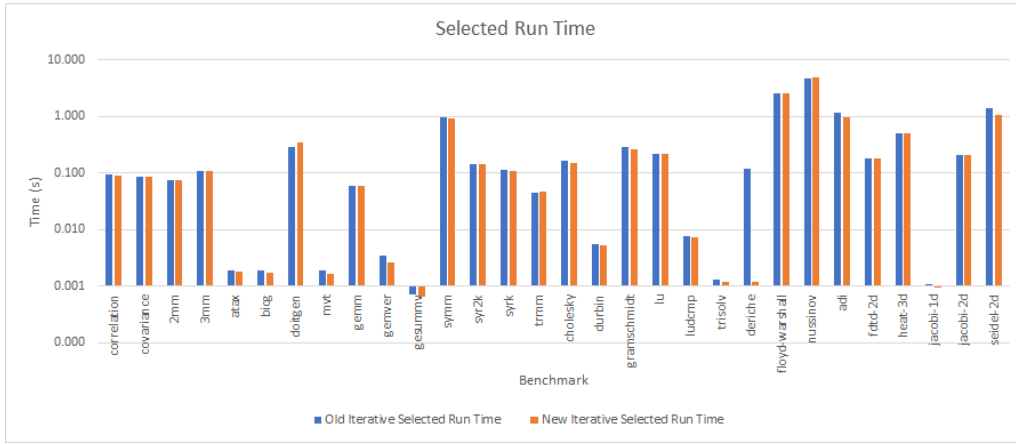
In the figures, Old Iterative indicates the Iterative Compilation of the optimizations that Park et al. used, and the New Iterative means the new optimizations that were proposed in Chapter 3.2.2.

In Figure 5.2a, the New Iterative approach tends to reach better results for all but one benchmark. This benchmark, *ludcmp*, tended to have some optimizations for which it wouldn’t complete running within 5 minutes. For these optimizations, a Run Time of 5 minutes is taken for training.

In Figure 5.2b, most results are fairly similar. The increase in speedup of *deriche* is likely an error, since this result was reached for optimizations that were also included in the old Iterative.



(a) Average Run Time



(b) Selected Run Time

**Figure 5.2:** The average Run Time of the Iterative Compilations per benchmark, and the Run Time that was selected as the best one from the Iterative Compilations.

Model	Total Run Time (s)	GeoMean Run Time (s)
Iterative Park et al.	13.545	0.061
Iterative Our Approach	13.110	0.050

**Table 5.2:** Total Run Time and GeoMean Run Time of the different versions of Iterative

When looking at the Total Run Time and GeoMean Run Time of the Iteratives, as shown in Table 5.2, our approach reaches better times for both of these.

## 5.4 Feature Normalization

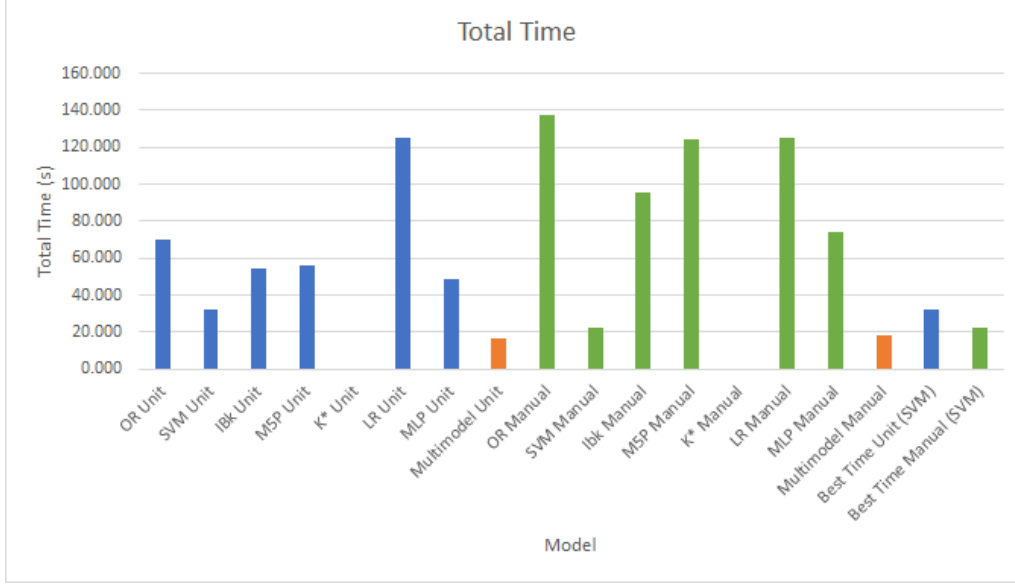
After changing the Optimizations, 2 different types of Feature Normalizations are applied on the Features, as explained in 3.2.1. The results are displayed in Figure 5.3.

One first thing to notice is the lack of  $K^*$  for both Normalizations. For some reason it couldn't find predictions for  $K^*$ , so this is omitted. Besides the lack of information for  $K^*$ , it also means that the Multimodel will be less accurate than the previous Multimodels.

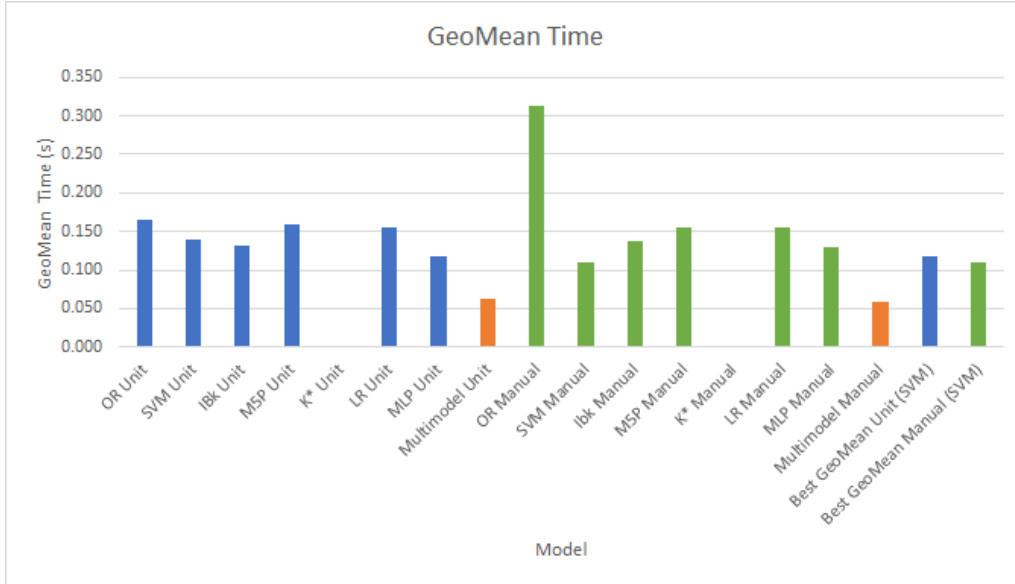
The summarized results are displayed in Table 5.3.

In terms of Total Run Time, the individual models seemed to take longer for the Manual Normalization. The MultiModel performed slightly better when using Unit Normalization, but the Best model performed better for Manual, due to SVM's performance.

In terms of GeoMean Time, the separate models seem similar. The Multimodels are comparable,



(a) Total Run Time



(b) GeoMean Run Time

**Figure 5.3:** The Total Run Time and GeoMean of Run Time of the Unit Vector Normalized Features (Unit) and Manually Normalized Features (Manual)

but the Manual normalization is slightly better. For the Best Model, Manual performed better again.

## 5.5 Selection Time vs Run Time

An updated Pareto graph is shown in Figure 5.4 and the data used for it in Table 5.4. A significant improvement compared to Park et al.'s method can be seen for all of the proposed methods; Selection Time is significantly reduced, and Run Time has improved. The proposed methods reached a Selection Time that is similar to P<sub>LuTo</sub> –tile –parallel, and improved on Run Times.

In regards of the Multimodel approach, all of the proposed methods reached an improved Run Time compared to Park et al., as well as a 10.9x improvement in Selection Time for Minimal

Model	Total Run Time (s)	GeoMean Run Time (s)
Best Total Time Unit (SVM)	32.278	0.140
Best Total Time Manual (SVM)	22.047	0.110
Best GeoMean Time Unit (MLP)	49.005	0.118
Best GeoMean Time Manual (SVM)	22.047	0.110
Multimodel Unit	16.901	0.062
Multimodel Manual	17.817	0.059

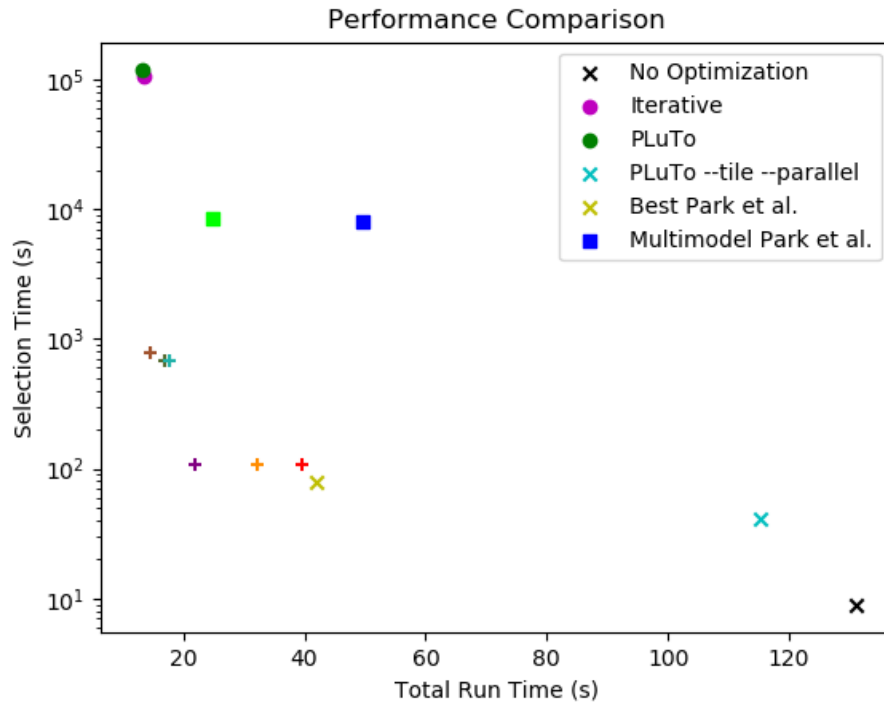
**Table 5.3:** Total Run Time and GeoMean Run Time of the different versions of Feature Normalization

Model	Total Selection Time (s)	Total Run Time (s)	GeoMean Run Time (s)
No Optimization	8.79	131.16	0.784
Iterative	105969.84	13.55	0.061
New Iterative	119519.27	13.11	0.050
PLuTo	41.14	115.12	0.544
PLuTo -tile -parallel	78.06	42.04	0.139
Best Total Time Park et al. (MLP)	7943.47	49.57	0.205
Best GeoMean Time Park et al. (OR)	7943.47	55.31	0.130
Best Total Time PolyFeat (M5P)	109.06	39.49	0.142
Best GeoMean Time PolyFeat (OR)	109.06	55.31	0.130
Best Total Time Unit (SVM)	109.10	32.28	0.140
Best GeoMean Time Unit (MLP)	109.10	49.00	0.118
Best Total Time Manual (SVM)	109.10	22.05	0.110
Best GeoMean Time Manual (SVM)	109.10	22.05	0.110
Multimodel Park et al.	8540.25	24.88	0.071
Multimodel PolyFeat	783.78	14.64	0.065
Multimodel Unit	676.60	16.90	0.062
Multimodel Manual	676.60	17.82	0.059

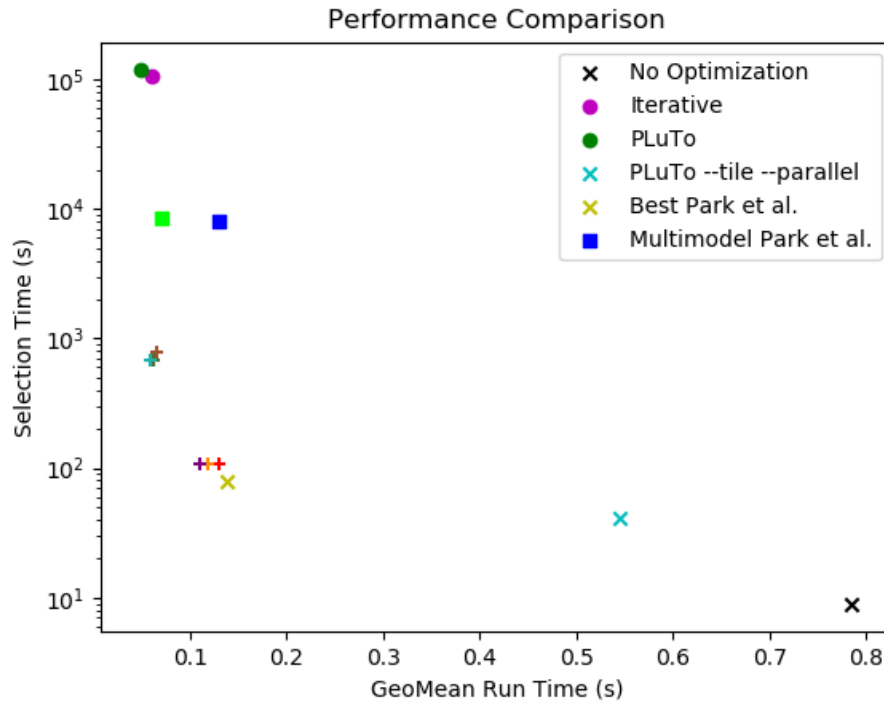
**Table 5.4:** Selection Times and Run Times of every model

Approach (PolyFeat) and 12.6x improvement for the complete approach (Unit and Manual). The Run Time of the Multimodel approaches are very similar to that of Iterative, and reaches 92.6% of Run Time Speedup of the old Iterative at a more than 135x lower Selection Time (PolyFeat). The Run Time for Unit Normalization reaches 77.5% of the Run Time Speedup of the new Iterative that it was based on, and the one for Manual Normalization reaches 73.6%, while being 176.6x faster in terms of Selection Time.

The Normalized models (Unit and Manual) tend to have better results in terms of Best single models. In terms of Multimodels, as they were missing one Machine Learning model (K\*), they performed worse than PolyFeat, but still reached a better GeoMean. Also, the Selection Time is lower than PolyFeat due to the lack of K



(a) Total Run Time



(b) Geometric Mean of Run Time

**Figure 5.4:** Pareto graphs of our Approaches, compared to the state of the art.



# Chapter 6

## Conclusion

This chapter concludes the results of the thesis. It will also discuss about the potential future works.

### 6.1 Conclusions

The initial problem to create a Polyhedral Optimizer, that does not require a prior compilation or run of the code, has been fulfilled, using the Polyhedral Features method of Chapter 3.3. This minimal method using Polyhedral Features already surpassed Park et al.'s method in terms of both Total Run Time and GeoMean Run Time and Selection Time, as shown in Chapter 5.2

In order to improve the results more, extra improvements have been added in Chapter 3.4. Changing the Optimizations led to slightly better average iterative results as shown in Chapter 5.3 and normalization of the Features led to a better Run Time for the "Best" Model, as seen in Chapter 5.5. However, the Run Time for the Multimodel seems to have worsened, due to the lack of the K\* Machine Learning model.

Chapter 5.5 shows the placement of all of the results and the final results of all models. The placement of all of our approaches in the Pareto graph of Figure 5.4 fill a relevant spot, and serve as a direct improvement on Park et al.'s approach.

To summarize: a Polyhedral Optimizer based on Machine Learning has been created. It does not require a single prior compile or run of the code, due to the use of Polyhedral Features instead of PAPI hardware counters. This makes our approach significantly faster in Selection Time than Park et al.'s approach. Additionally, it turned out to reach a faster Run Time as well, and reached a better Total Run Time than PLuTo -tile -parallel, which Park et al.'s approach could not beat. Additionally, although not in the original scope of the thesis, a Multimodel has been made, that would compile and run 7 optimizations; the selected optimization for each of the Machine Learning Models. At the cost of Selection Time, a higher Run Time can be reached using this method. This Multimodel is also faster in Selection Time than Park et al., due to not requiring any prior compilations and runs. The MultiModel of the minimal approach also reaches approximately 92.6% of the Iterative Run Time, while being more than 135 times faster with the selection process.

When using the Complete approach, only up to 77.5% of the Run Time Speedup can be reached while being 176.6x faster than the Iterative approach that was used for generating this. This is partially due to the lack of K\* in this multimodel.

### 6.2 Future Work

Originally, LeTSeE [Pouchet et al., 2007a] and Polyite [Gansser et al., 2017] were also supposed to be part of the performance comparisons, in order to have more Iterative Approaches to compare with. There was sadly no time to include this, so this has been omitted.

Additionally, Polyite seemed like an interesting Iterative Compilation base, to use for Machine Learning. This was also not reached due to time constraints, and difficulties getting Polyite to work.

Originally, there was a third approach planned. This approach would use only Polyhedral Features of the optimized code as Features, rather than Flags and Polyhedral Features of the unoptimized

code.

Currently, the approaches have only be tested on a single Hardware setup, with a single compiler. Park et al. tested this method on two Hardware setups, and two different compilers. Generating better results than Park et al. on multiple platforms rather than just one would prove more convincing than the single setup used in this thesis.

It's yet unclear why  $K^*$  did not predict values for the Normalized approaches. By getting  $K^*$  to work, the Run Time of the Multimodel should improve.

There is one big issue currently with the Polyhedral Features. Features with index  $i$  are only compared with other features with index  $i$ . In the case of the Polyhedral Features, this means that one code can have 42 Statements, another code can have 2. But only 2 Statements of each of the codes would be compared. And the 2 Statements don't necessarily are the most significant ones. To solve this, a special machine learning model can be made with a certain custom made kernel. This kernel would need to find either the most significant Statements, or would need to summarize all the Statement data within a smaller amount of Features that would be comparable.

It is also possible to change the scope of the project; Instead of trying to predict the best combination of optimizations, it's possible to use the same method to try and find the optimal Tile Sizes. The amount of different Tile Sizes that were currently tested are rather limited, due to the high amount of iterations caused by the other optimizations. But when it's not restrained by other optimizations, it would be possible to use this to approximate ideal Tile Sizes on a higher granularity.

# Appendices

## Appendix A

# Example Correlation

```
for (j = 0; j < _PB_M; j++)
{
    mean[j] = SCALAR_VAL(0.0);
    for (i = 0; i < _PB_N; i++)
        mean[j] += data[i][j];
    mean[j] /= float_n;
}

for (j = 0; j < _PB_M; j++)
{
    stddev[j] = SCALAR_VAL(0.0);
    for (i = 0; i < _PB_N; i++)
        stddev[j] += (data[i][j] - mean[j]) * (data[i][j] - mean[j]);
    stddev[j] /= float_n;
    stddev[j] = SQRT_FUN(stddev[j]);
    /* The following is an inelegant but usual way to handle
       near-zero std. dev. values, which below would cause a zero-
       divide. */
    stddev[j] = stddev[j] <= eps ? SCALAR_VAL(1.0) : stddev[j];
}

/* Center and reduce the column vectors. */
for (i = 0; i < _PB_N; i++)
    for (j = 0; j < _PB_M; j++)
    {
        data[i][j] -= mean[j];
        data[i][j] /= SQRT_FUN(float_n) * stddev[j];
    }

/* Calculate the m * m correlation matrix. */
for (i = 0; i < _PB_M-1; i++)
{
    corr[i][i] = SCALAR_VAL(1.0);
    for (j = i+1; j < _PB_M; j++)
    {
        corr[i][j] = SCALAR_VAL(0.0);
        for (k = 0; k < _PB_N; k++)
            corr[i][j] += (data[k][i] * data[k][j]);
        corr[j][i] = corr[i][j];
    }
}
corr[_PB_M-1][_PB_M-1] = SCALAR_VAL(1.0);
```

## Appendix B

# Example OpenScop

Example OpenScop of PolyBench's 2mm.c

```
#
#
#      <|
#      A
#      /\
#      <|  ["M#
#      A   | #
#      /\  ["M#
#      ["M# | # U"U#U
#      | # | # \.:/
#      | # | # ___| #
#      | " _ _ ' _ _ "
#      | " _ _ " _ _ " _ _ # _ _ #
#      |      # ## #####
#      \      .::: ' /
#      \      .::: ' /
#      :8a|      # # ##
#      ::88a      ###
#      ::::888a 8a ##::
#      :::::888a88a[]:::
#      :::::SUNDOGa8a::: ..
#      :::::8:::888:Y888:::.....
#      :::':::88:::888::Y88a-----
#      :::88a:::88a:Y88a
#      ' .: :Y88a:::8a:Y88a
#      : ' :8P:::88aa.
#      :: :Y88as88a...s88aa.
#
# [File generated by the OpenScop Library 0.9.0]

<OpenScop>

# ===== Global
# Language
C

# Context
CONTEXT
0 6 0 0 0 4

# Parameters are provided
1
<strings>
_PB_NI _PB_NJ _PB_NK _PB_NL
</strings>

# Number of statements
4

# ===== Statement 1
# Number of relations describing the statement:
3

# ----- 1.1 Domain
```

```

DOMAIN
6 8 2 0 0 4
# e/i| i j |_PB. _PB. _PB. _PB.| 1
1 1 0 0 0 0 0 0 0 ## i >= 0
1 -1 0 1 0 0 0 -1 ## -i+_PB_NI-1 >= 0
1 0 0 1 0 0 0 -1 ## _PB_NI-1 >= 0
1 0 1 0 0 0 0 0 ## j >= 0
1 0 -1 0 1 0 0 -1 ## -j+_PB_NJ-1 >= 0
1 0 0 0 1 0 0 -1 ## _PB_NJ-1 >= 0

# ----- 1.2 Scattering
SCATTERING
5 13 5 2 0 4
# e/i| c1 c2 c3 c4 c5 | i j |_PB. _PB. _PB. _PB.| 1
0 -1 0 0 0 0 0 0 0 0 0 0 0 ## c1 == 0
0 0 -1 0 0 0 1 0 0 0 0 0 0 ## c2 == i
0 0 0 -1 0 0 0 0 0 0 0 0 0 ## c3 == 0
0 0 0 0 -1 0 1 0 0 0 0 0 0 ## c4 == j
0 0 0 0 0 -1 0 0 0 0 0 0 0 ## c5 == 0

# ----- 1.3 Access
WRITE
3 11 3 2 0 4
# e/i| Arr [1] [2]| i j |_PB. _PB. _PB. _PB.| 1
0 -1 0 0 0 0 0 0 0 5 ## Arr == tmp
0 0 -1 0 1 0 0 0 0 0 ## [1] == i
0 0 0 -1 0 1 0 0 0 0 ## [2] == j

# ----- 1.4 Statement Extensions
# Number of Statement Extensions
1
<body>
# Number of original iterators
2
# List of original iterators
i j
# Statement body expression
tmp[i][j] = SCALAR_VAL(0.0);
</body>

# ===== Statement 2
# Number of relations describing the statement:
7

# ----- 2.1 Domain
DOMAIN
9 9 3 0 0 4
# e/i| i j k |_PB. _PB. _PB. _PB.| 1
1 1 0 0 0 0 0 0 0 ## i >= 0
1 -1 0 0 1 0 0 0 -1 ## -i+_PB_NI-1 >= 0
1 0 0 0 1 0 0 0 -1 ## _PB_NI-1 >= 0
1 0 1 0 0 0 0 0 0 ## j >= 0
1 0 -1 0 0 1 0 0 -1 ## -j+_PB_NJ-1 >= 0
1 0 0 0 0 1 0 0 -1 ## _PB_NJ-1 >= 0
1 0 0 1 0 0 0 0 0 ## k >= 0
1 0 0 -1 0 0 1 0 -1 ## -k+_PB_NK-1 >= 0
1 0 0 0 0 0 1 0 -1 ## _PB_NK-1 >= 0

# ----- 2.2 Scattering
SCATTERING
7 16 7 3 0 4
# e/i| c1 c2 c3 c4 c5 c6 c7 | i j k |_PB. _PB. _PB. _PB.| 1
0 -1 0 0 0 0 0 0 0 0 0 0 0 0
## c1 == 0
0 0 -1 0 0 0 0 1 0 0 0 0 0 0
## c2 == i
0 0 0 -1 0 0 0 0 0 0 0 0 0 0
## c3 == 0
0 0 0 0 -1 0 0 0 1 0 0 0 0 0
## c4 == j
0 0 0 0 0 -1 0 0 0 0 0 0 0 1
## c5 == 1

```

```

0 0 0 0 0 0 -1 0 0 0 1 0 0 0 0 0
## c6 == k
0 0 0 0 0 0 0 -1 0 0 0 0 0 0 0 0
## c7 == 0

# ----- 2.3 Access
READ
3 12 3 3 0 4
# e/i| Arr [1] [2]| i j k |_PB. _PB. _PB. _PB.| 1
0 -1 0 0 0 0 0 0 0 0 0 0 5 ## Arr == tmp
0 0 -1 0 1 0 0 0 0 0 0 0 0 ## [1] == i
0 0 0 -1 0 1 0 0 0 0 0 0 0 ## [2] == j

WRITE
3 12 3 3 0 4
# e/i| Arr [1] [2]| i j k |_PB. _PB. _PB. _PB.| 1
0 -1 0 0 0 0 0 0 0 0 0 0 5 ## Arr == tmp
0 0 -1 0 1 0 0 0 0 0 0 0 0 ## [1] == i
0 0 0 -1 0 1 0 0 0 0 0 0 0 ## [2] == j

READ
1 10 1 3 0 4
# e/i| Arr| i j k |_PB. _PB. _PB. _PB.| 1
0 -1 0 0 0 0 0 0 0 0 9 ## Arr == alpha

READ
3 12 3 3 0 4
# e/i| Arr [1] [2]| i j k |_PB. _PB. _PB. _PB.| 1
0 -1 0 0 0 0 0 0 0 0 0 10 ## Arr == A
0 0 -1 0 1 0 0 0 0 0 0 0 ## [1] == i
0 0 0 -1 0 0 1 0 0 0 0 0 ## [2] == k

READ
3 12 3 3 0 4
# e/i| Arr [1] [2]| i j k |_PB. _PB. _PB. _PB.| 1
0 -1 0 0 0 0 0 0 0 0 0 11 ## Arr == B
0 0 -1 0 0 0 1 0 0 0 0 0 ## [1] == k
0 0 0 -1 0 1 0 0 0 0 0 0 ## [2] == j

# ----- 2.4 Statement Extensions
# Number of Statement Extensions
1
<body>
# Number of original iterators
3
# List of original iterators
i j k
# Statement body expression
tmp[i][j] += alpha * A[i][k] * B[k][j];
</body>

# ===== Statement 3
# Number of relations describing the statement:
5

# ----- 3.1 Domain
DOMAIN
6 8 2 0 0 4
# e/i| i j |_PB. _PB. _PB. _PB.| 1
1 1 0 0 0 0 0 0 ## i >= 0
1 -1 0 1 0 0 0 -1 ## -i+_PB_NI-1 >= 0
1 0 0 1 0 0 0 -1 ## _PB_NI-1 >= 0
1 0 1 0 0 0 0 0 ## j >= 0
1 0 -1 0 0 0 1 -1 ## -j+_PB_NL-1 >= 0
1 0 0 0 0 0 1 -1 ## _PB_NL-1 >= 0

# ----- 3.2 Scattering
SCATTERING
5 13 5 2 0 4
# e/i| c1 c2 c3 c4 c5 | i j |_PB. _PB. _PB. _PB.| 1
0 -1 0 0 0 0 0 0 0 0 0 1 ## c1 == 1
0 0 -1 0 0 0 1 0 0 0 0 0 ## c2 == i
0 0 0 -1 0 0 0 0 0 0 0 0 ## c3 == 0

```

```

0 0 0 0 -1 0 0 1 0 0 0 0 0 0 ## c4 == j
0 0 0 0 0 -1 0 0 0 0 0 0 0 0 ## c5 == 0

# ----- 3.3 Access
READ
3 11 3 2 0 4
# e/i| Arr [1] [2]| i j |_PB. _PB. _PB. _PB.| 1
0 -1 0 0 0 0 0 0 0 0 0 13 ## Arr == D
0 0 -1 0 1 0 0 0 0 0 0 0 ## [1] == i
0 0 0 -1 0 1 0 0 0 0 0 0 ## [2] == j

WRITE
3 11 3 2 0 4
# e/i| Arr [1] [2]| i j |_PB. _PB. _PB. _PB.| 1
0 -1 0 0 0 0 0 0 0 0 0 13 ## Arr == D
0 0 -1 0 1 0 0 0 0 0 0 0 ## [1] == i
0 0 0 -1 0 1 0 0 0 0 0 0 ## [2] == j

READ
1 9 1 2 0 4
# e/i| Arr| i j |_PB. _PB. _PB. _PB.| 1
0 -1 0 0 0 0 0 0 0 14 ## Arr == beta

# ----- 3.4 Statement Extensions
# Number of Statement Extensions
1
<body>
# Number of original iterators
2
# List of original iterators
i j
# Statement body expression
D[i][j] *= beta;
</body>

# ===== Statement 4
# Number of relations describing the statement:
6

# ----- 4.1 Domain
DOMAIN
9 9 3 0 0 4
# e/i| i j k |_PB. _PB. _PB. _PB.| 1
1 1 0 0 0 0 0 0 0 ## i >= 0
1 -1 0 0 1 0 0 0 -1 ## -i+_PB_NI-1 >= 0
1 0 0 0 1 0 0 0 -1 ## _PB_NI-1 >= 0
1 0 1 0 0 0 0 0 0 ## j >= 0
1 0 -1 0 0 0 0 1 -1 ## -j+_PB_NL-1 >= 0
1 0 0 0 0 0 0 1 -1 ## _PB_NL-1 >= 0
1 0 0 1 0 0 0 0 0 ## k >= 0
1 0 0 -1 0 1 0 0 -1 ## -k+_PB_NJ-1 >= 0
1 0 0 0 0 1 0 0 -1 ## _PB_NJ-1 >= 0

# ----- 4.2 Scattering
SCATTERING
7 16 7 3 0 4
# e/i| c1 c2 c3 c4 c5 c6 c7 | i j k |_PB. _PB. _PB. _PB.| 1
0 -1 0 0 0 0 0 0 0 0 0 0 0 0 1
## c1 == 1
0 0 -1 0 0 0 0 0 1 0 0 0 0 0 0
## c2 == i
0 0 0 -1 0 0 0 0 0 0 0 0 0 0 0
## c3 == 0
0 0 0 0 -1 0 0 0 0 1 0 0 0 0 0
## c4 == j
0 0 0 0 0 -1 0 0 0 0 0 0 0 0 1
## c5 == 1
0 0 0 0 0 0 -1 0 0 0 1 0 0 0 0
## c6 == k
0 0 0 0 0 0 0 -1 0 0 0 0 0 0 0
## c7 == 0

# ----- 4.3 Access

```



```

READ
3 12 3 3 0 4
# e/i| Arr [1] [2]| i j k |_PB. _PB. _PB. _PB.| 1
0 -1 0 0 0 0 0 0 0 0 0 13 ## Arr == D
0 0 -1 0 1 0 0 0 0 0 0 0 ## [1] == i
0 0 0 -1 0 1 0 0 0 0 0 0 ## [2] == j

WRITE
3 12 3 3 0 4
# e/i| Arr [1] [2]| i j k |_PB. _PB. _PB. _PB.| 1
0 -1 0 0 0 0 0 0 0 0 0 13 ## Arr == D
0 0 -1 0 1 0 0 0 0 0 0 0 ## [1] == i
0 0 0 -1 0 1 0 0 0 0 0 0 ## [2] == j

READ
3 12 3 3 0 4
# e/i| Arr [1] [2]| i j k |_PB. _PB. _PB. _PB.| 1
0 -1 0 0 0 0 0 0 0 0 0 5 ## Arr == tmp
0 0 -1 0 1 0 0 0 0 0 0 0 ## [1] == i
0 0 0 -1 0 0 1 0 0 0 0 0 ## [2] == k

READ
3 12 3 3 0 4
# e/i| Arr [1] [2]| i j k |_PB. _PB. _PB. _PB.| 1
0 -1 0 0 0 0 0 0 0 0 0 15 ## Arr == C
0 0 -1 0 0 0 1 0 0 0 0 0 ## [1] == k
0 0 0 -1 0 1 0 0 0 0 0 0 ## [2] == j

# ----- 4.4 Statement Extensions
# Number of Statement Extensions
1
<body>
# Number of original iterators
3
# List of original iterators
i j k
# Statement body expression
D[i][j] += tmp[i][k] * C[k][j];
</body>

# ===== Extensions
<scatnames>
b0 i b1 j b2 k b3
</scatnames>

<arrays>
# Number of arrays
15
# Mapping array-identifiers/array-names
1 i
2 _PB_NI
3 j
4 _PB_NJ
5 tmp
6 SCALAR_VAL
7 k
8 _PB_NK
9 alpha
10 A
11 B
12 _PB_NL
13 D
14 beta
15 C
</arrays>

<coordinates>
# File name
polybench-c-4.2.1-beta/linear-algebra/kernels/2mm/2mm.c
# Starting line and column
88 0
# Ending line and column
103 0

```

```
| # Indentation  
| 2  
| </coordinates>  
| </OpenScop>
```

# Bibliography

- [Asanovic et al., 2006] Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., et al. (2006). The landscape of parallel computing research: A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.
- [Bacon et al., 1994] Bacon, D. F., Graham, S. L., and Sharp, O. J. (1994). Compiler transformations for high-performance computing. *ACM Computing Surveys (CSUR)*, 26(4):345–420.
- [Bastoul, 2008] Bastoul, C. (2008). Clan-a polyhedral representation extractor for high level programs.
- [Bastoul, 2011] Bastoul, C. (2011). Openscop: A specification and a library for data exchange in polyhedral compilation tools. Technical report, tech. rep., Paris-Sud University, France.
- [Bastoul, 2013] Bastoul, C. (2013). Cloog: The chunky loop generator.(2013).
- [Bastoul and Pouchet, 2012] Bastoul, C. and Pouchet, L. (2012). Candl: the chunky analyzer for dependences in loops. Technical report, tech. rep., LRI, Paris-Sud University, France.
- [Bondhugula et al., 2008] Bondhugula, U., Hartono, A., Ramanujam, J., and Sadayappan, P. (2008). A practical automatic polyhedral parallelizer and locality optimizer. In *Acm Sigplan Notices*, volume 43, pages 101–113. ACM.
- [Cavazos et al., 2006] Cavazos, J., Dubach, C., Agakov, F., Bonilla, E., O’Boyle, M. F., Fursin, G., and Temam, O. (2006). Automatic performance model construction for the fast software exploration of new hardware designs. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 24–34. ACM.
- [Coleman and McKinley, 1995] Coleman, S. and McKinley, K. S. (1995). Tile size selection using cache organization and data layout. In *ACM SIGPLAN Notices*, volume 30, pages 279–290. ACM.
- [Cosenza et al., 2017] Cosenza, B., Durillo, J. J., Ermon, S., and Juurlink, B. (2017). Autotuning stencil computations with structural ordinal regression learning. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*, pages 287–296. IEEE.
- [Fleming and Wallace, 1986] Fleming, P. J. and Wallace, J. J. (1986). How not to lie with statistics: the correct way to summarize benchmark results. *Communications of the ACM*, 29(3):218–221.
- [Fursin et al., 2011] Fursin, G., Kashnikov, Y., Memon, A. W., Chamski, Z., Temam, O., Namolalu, M., Yom-Tov, E., Mendelson, B., Zaks, A., Courtois, E., et al. (2011). Milepost gcc: Machine learning enabled self-tuning compiler. *International journal of parallel programming*, 39(3):296–327.
- [Ganser et al., 2017] Ganser, S., Grösslinger, A., Siegmund, N., Apel, S., and Lengauer, C. (2017). Iterative schedule optimization for parallelization in the polyhedron model. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(3):23.
- [Grosser et al., 2011] Grosser, T., Zheng, H., Aloor, R., Simbürger, A., Größlinger, A., and Pouchet, L.-N. (2011). Polly-polyhedral optimization in llvm. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, volume 2011.

- [Jimboean, 2012] Jimboean, A. (2012). *Adapting the polytope model for dynamic and speculative parallelization*. PhD thesis, Strasbourg.
- [Joachims, 2002] Joachims, T. (2002). Optimizing search engines using clickthrough data. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 133–142. ACM.
- [Nasrabadi, 2007] Nasrabadi, N. M. (2007). Pattern recognition and machine learning. *Journal of electronic imaging*, 16(4):049901.
- [Ngoma, 2017] Ngoma, Y. M. (2017). *Analysis of Control Attainment in Endogenous Electroencephalogram Based Brain Computer Interfaces*. PhD thesis, Tshwane University of Technology.
- [Park et al., 2013] Park, E., Cavazos, J., Pouchet, L.-N., Bastoul, C., Cohen, A., and Sadayappan, P. (2013). Predictive modeling in a polyhedral optimization space. *International journal of parallel programming*, 41(5):704–750.
- [Pouchet et al., ] Pouchet, L.-N., Bastoul, C., and Bondhugula, U. Pocc: the polyhedral compiler collection, 2010. URL <http://www.cse.ohio-state.edu/~pouchet/software/pocc>.
- [Pouchet et al., 2007a] Pouchet, L.-N., Bastoul, C., and Cohen, A. (2007a). Letsee: The legal transformation space explorer. *Third International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems (ACACES’07), L’Aquila, Italia*, pages 247–251.
- [Pouchet et al., 2008] Pouchet, L.-N., Bastoul, C., Cohen, A., and Cavazos, J. (2008). Iterative optimization in the polyhedral model: Part ii, multidimensional time. In *ACM SIGPLAN Notices*, volume 43, pages 90–100. ACM.
- [Pouchet et al., 2007b] Pouchet, L.-N., Bastoul, C., Cohen, A., and Vasilache, N. (2007b). Iterative optimization in the polyhedral model: Part i, one-dimensional time. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 144–156. IEEE Computer Society.
- [Ragan-Kelley et al., 2013] Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. (2013). Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530.
- [Ryoo et al., 2008] Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., and Hwu, W.-m. W. (2008). Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82. ACM.
- [Tang et al., 2011] Tang, Y., Chowdhury, R. A., Kuszmaul, B. C., Luk, C.-K., and Leiserson, C. E. (2011). The pochoir stencil compiler. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 117–128. ACM.
- [Terpstra et al., 2010] Terpstra, D., Jagode, H., You, H., and Dongarra, J. (2010). Collecting performance data with papi-c. In *Tools for High Performance Computing 2009*, pages 157–173. Springer.