Rust-Analyse und -Optimierung mittels LLVM

Seminar effiziente Programmierung

Steffen Schubert

Universität Hamburg
Fachbereich Informatik

1. Februar 2021

Motivation

1. Wie hängen Rust und LLVM überhaupt zusammen?
2. Was kann man mit LLVM so machen? Und wie?
3. Wozu wollen wir mit LLVM analysieren?
4. Wie sieht so ein LLVM Tool eigentlich aus? - Ein Beispiel

Gliederung

1. Rustc
   - Rustc
   - Rustc Backend - LLVM
2. LLVM Analyse
   - Tools
   - Analyse Pass - Demo
3. Polly
   - Einführung
   - SCoP's
   - Polytopmodell
   - Transformieren - parallelisieren
   - Polly heute

Rustc

Rustc [1]

- Rusts eigener Compiler
- Direkt im Rust Projekt enthalten
- Wird eigentlich immer implizit über Cargo aufgerufen

Rustc Compiler Pipeline [1]

- Cargo ruft rustc auf
- Verschiedene IR in der Pipeline
- Eine davon LLVM-IR
- LLVM Übernimmt und optimiert mit seinen Passes

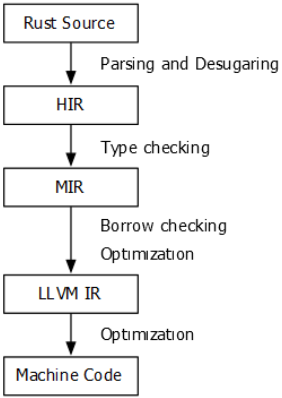


Abbildung: Rustc Pipeline, grobe Abbildung ([2])

LLVM Analyse

Opt [3]

- LLVM Optimizer
- Produziert neuen IR-Code aus altem IR-Code
- LLVM Passes für z.B.
  - Loops
  - Controllflowgraph
  - Callgraph
  - Und vor allem: Optimierung

Command Line:

$opt -passname file.bc (oder auch file.ll)
$opt -analyze -passname file.bc (für analyse Passes)

Rustc [1]

- Wir können LLVM Passes direkt an Rustc übergeben

Command Line:

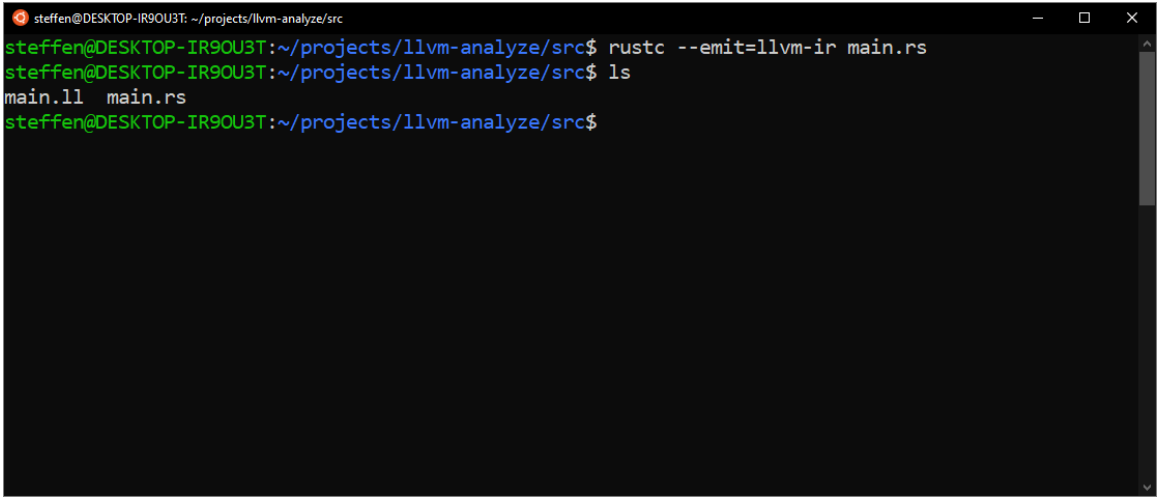
```
$rustc -C passes=val
```

- Nicht jeder Pass möglich
- Analysemodus nicht möglich

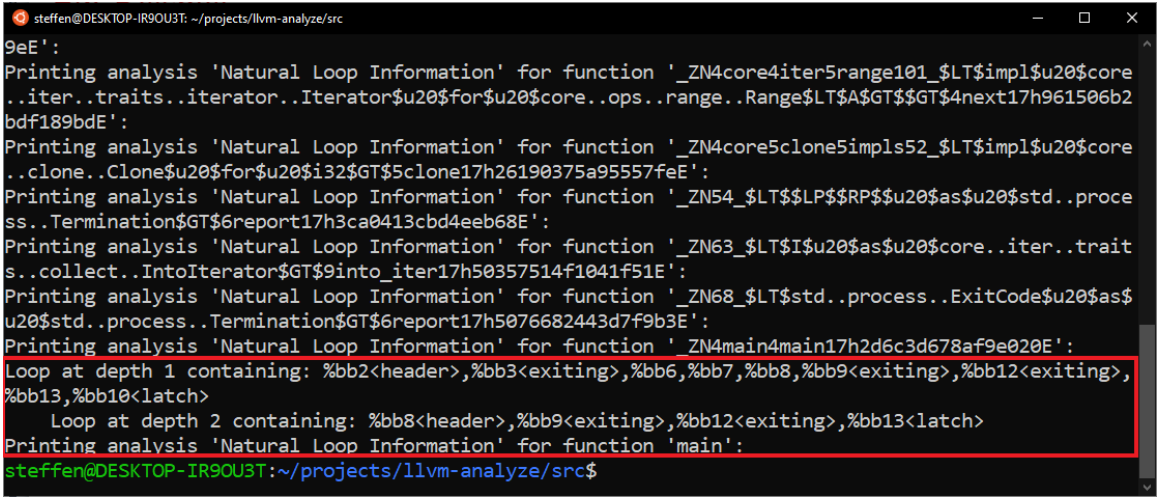
Ein Beispiel - main.rs

```
1   fn main() {
2       let mut n = 1;
3
4       for _i in 1..10 {
5           for _j in 1..20 {
6               n += 1;
7           }
8       }
9
10      println!("{}", n);
11  }
```

Ein Beispiel - Die IR ausgeben lassen



Ein Beispiel - $opt -analyze -loops main.ll



Wofür also Analyse mit LLVM? [4]

- In Rust eher weniger als debug Hilfe
- Mit Hilfe der Analysen werden Informationen gesammelt
- Die Informationen werden von Transformpasses genutzt
- Weiterer Optimierungsvorgang wird einfacher
- Durch die Modularität von LLVM können alle weiteren Passes von den Analysen profitieren

Von LLVM-IR weiter zu Machine Code

- Nicht der gleiche LLVM-IR Code wie in der Rustc pipeline
- Rust spezifische Funktionen noch in IR vorhanden
  - Z.B. Panic
- Lösung: Rust libraries manuell verlinken

Polly

Polly [5] [6]

- Projekt in LLVM
- Optimiert Schleifen
- Benutzt mathematische Repräsentation als Polyeder
- Initial gedacht für Parallelisierung

Funktionsweise [7]

- Satz von LLVM Passes
- 3 Verschiedene Aufgaben:
  - Übersetzen von IR zu Polytopmodell
  - Analyse und Optimierung
  - Übersetzen zurück zu IR

SCoP's - Static COntroll Parts [7]

- Codebereiche mit statischer Funktion
- Kann wie ein Funktionsaufruf behandelt werden
- Ersetzbar mit optimiertem Funktionsaufruf

Ein Beispiel

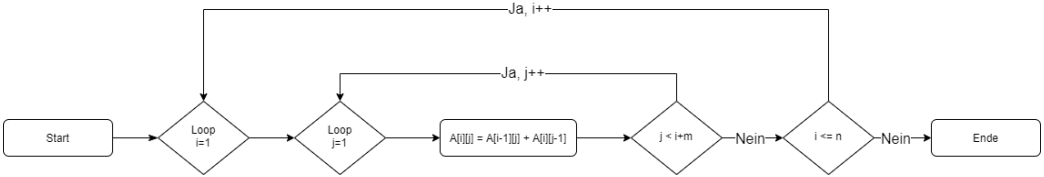


Abbildung: Kontrollflussgraph

Ein Beispiel

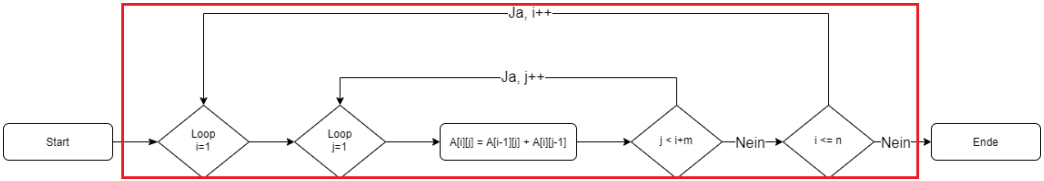


Abbildung: Kontrollflussgraph

Ist dies ein SCoP?

Ein Beispiel

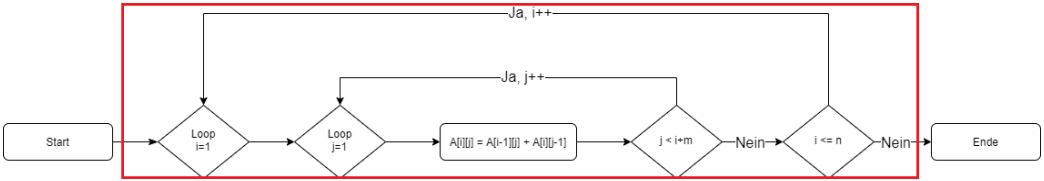


Abbildung: Kontrollflussgraph

Ist dies ein SCoP? Ja

Ein Beispiel

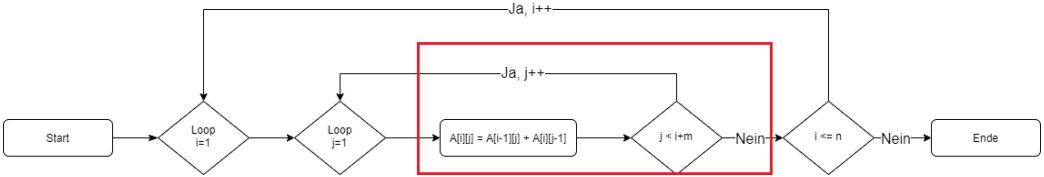


Abbildung: Kontrollflussgraph

Ist dies ein SCoP?

Ein Beispiel

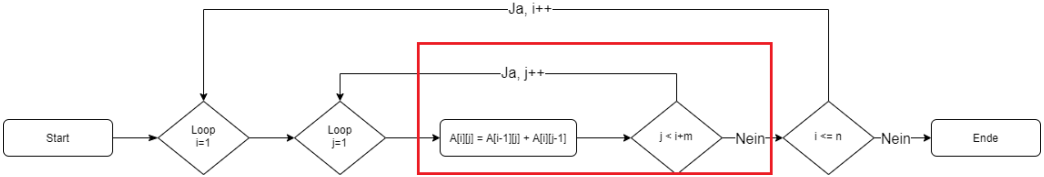


Abbildung: Kontrollflussgraph

Ist dies ein SCoP? Nein

Ein Beispiel

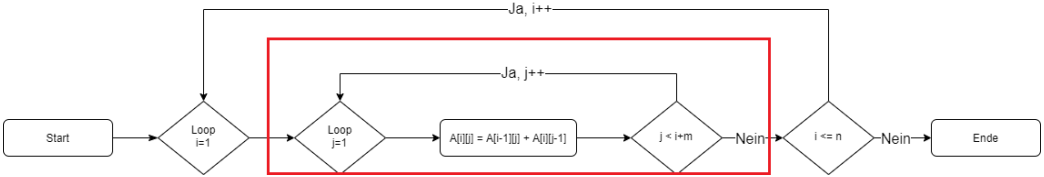


Abbildung: Kontrollflussgraph

Ist dies ein SCoP?

Ein Beispiel

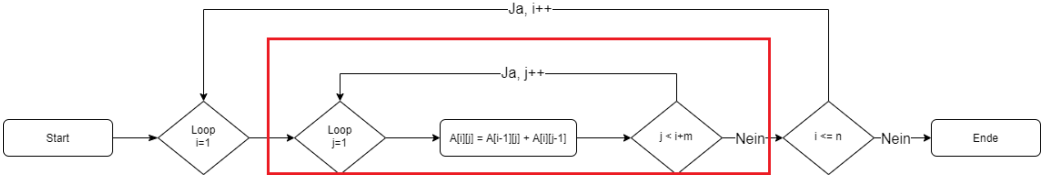


Abbildung: Kontrollflussgraph

Ist dies ein SCoP? Ja

The Polyhedral Model - Das Polytopmodell [7]

- SCoP's werden analysiert
- Iterationsraum von Schleifen beschrieben als ganzzahlige Polyeder
- Über Transformationen optimierbar

Der Iterationsraum

```
1  for(i = 1; i <= n; i++) {
2    for(j = 1; j < i+ m; j++) {
3      A[i][j] = A[i-1][j] + A[i][j-1]
4    }
5  }
```

Der Iterationsraum

```
1  for(i = 1; i <= n; i++) {
2    for(j = 1; j < i+ m; j++) {
3      A[i][j] = A[i-1][j] + A[i][j-1]
4    }
5  }
```

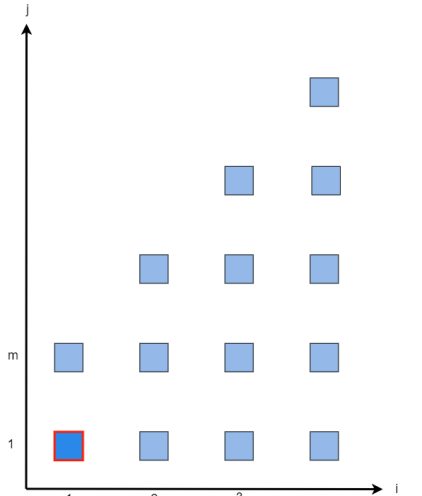


Abbildung: Iterationsraum (basierend auf [8], S.4)

Der Iterationsraum

```
1  for(i = 1; i <= n; i++) {
2    for(j = 1; j < i+ m; j++) {
3      A[i][j] = A[i-1][j] + A[i][j-1]
4    }
5  }
```

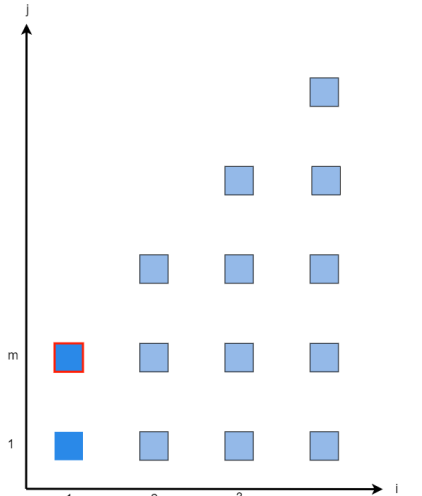


Abbildung: Iterationsraum (basierend auf [8], S.4)

Der Iterationsraum

```
1  for(i = 1; i <= n; i++) {
2    for(j = 1; j < i+ m; j++) {
3      A[i][j] = A[i-1][j] + A[i][j-1]
4    }
5  }
```

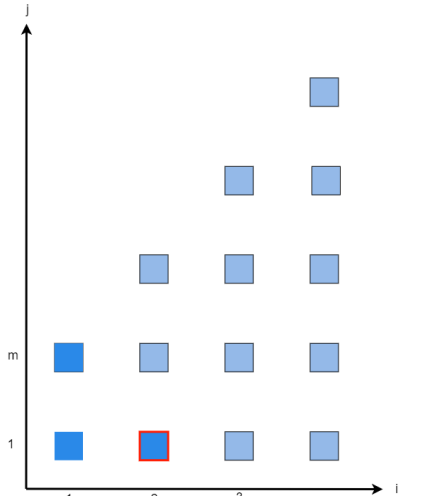


Abbildung: Iterationsraum (basierend auf [8], S.4)

Der Iterationsraum

```
1  for(i = 1; i <= n; i++) {
2    for(j = 1; j < i+ m; j++) {
3      A[i][j] = A[i-1][j] + A[i][j-1]
4    }
5  }
```

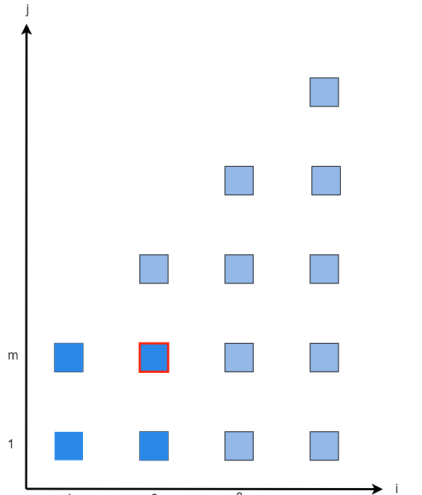


Abbildung: Iterationsraum (basierend auf [8], S.4)

Der Iterationsraum

```
1  for(i = 1; i <= n; i++) {
2    for(j = 1; j < i+ m; j++) {
3      A[i][j] = A[i-1][j] + A[i][j-1]
4    }
5  }
```

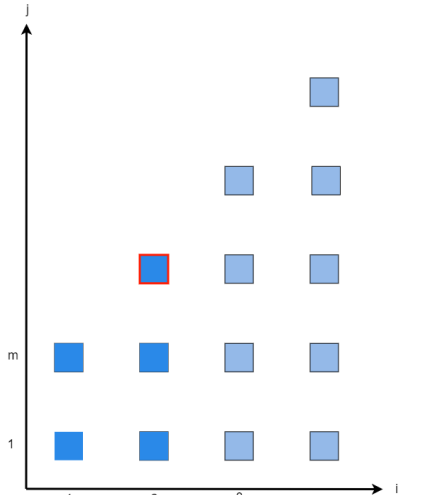


Abbildung: Iterationsraum (basierend auf [8], S.4)

Der Iterationsraum

```
1  for(i = 1; i <= n; i++) {
2    for(j = 1; j < i+ m; j++) {
3      A[i][j] = A[i-1][j] + A[i][j-1]
4    }
5  }
```

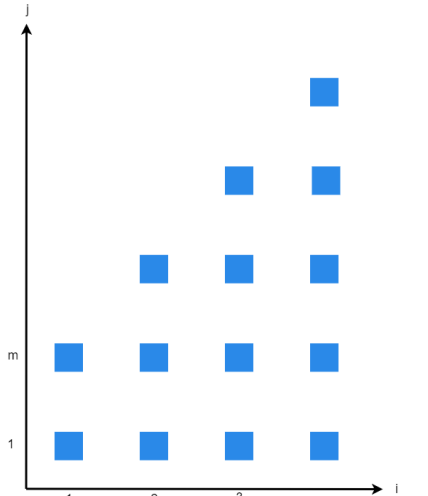


Abbildung: Iterationsraum (basierend auf [8], S.4)

Der Iterationsraum

```
1  for(i = 1; i <= n; i++) {
2    for(j = 1; j < i+ m; j++) {
3      A[i][j] = A[i-1][j] + A[i][j-1]
4    }
5  }
```

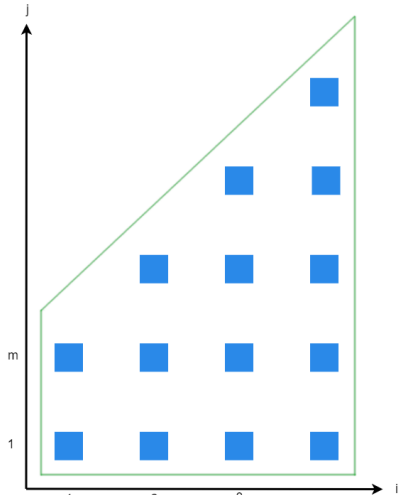


Abbildung: Iterationsraum (basierend auf [8], S.4)

Abhängigkeiten

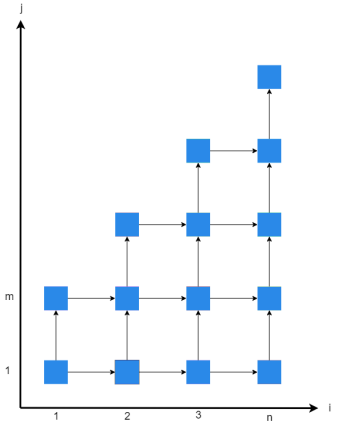


Abbildung: Abhängigkeiten (basierend auf [8], S.4)

Transformationen [7]

- Iterationsraum wird als Domain bezeichnet
- Jede Domain hat eine Schedule der sie folgt
- Durch transformation der Schedule wird die Domain parallelisierbar

Abhängigkeiten

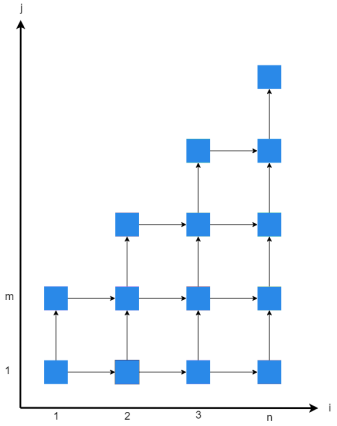


Abbildung: Abhängigkeiten (basierend auf [8], S.4)

Transformation

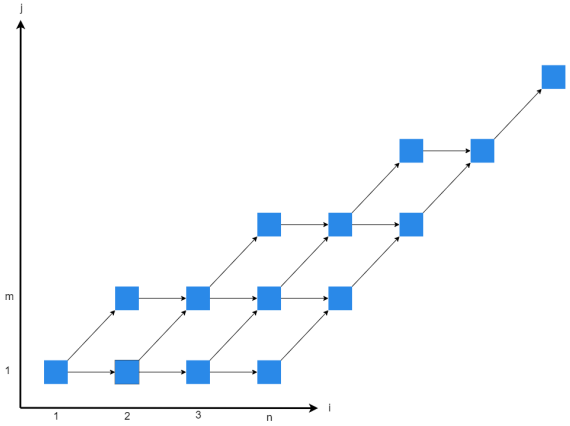


Abbildung: Transformation (basierend auf [8], S.4)

Parallelisiert

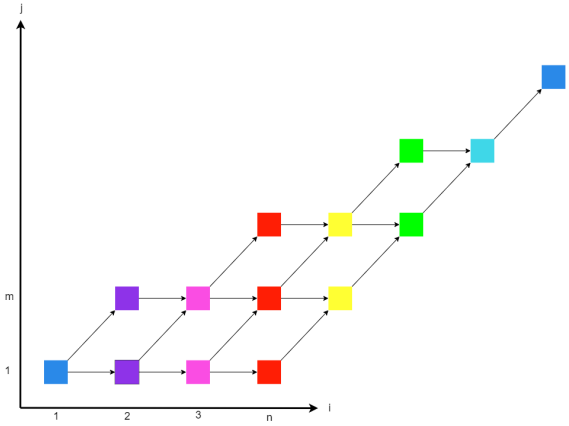


Abbildung: Transformation (basierend auf [8], S.4)

Performance

Compiler and options	Large	Extra
gcc 7.2 -O3	4.250	60.130
Clang 10.0 -O3	3.817	58.848
Clang 10.0 -O3 with polly -std=c99 -fno-unroll-loops -O3 -mllvm -polly -mllvm -polly-process-unprofitable -mllvm -polly-use-llvm-names -ffast-math -march=native	0.709	6.268
Clang 10.0 -O3 with polly and tiling (96, 2048, 256), interchange and packing -std=c99 -fno-unroll-loops -O3 -mllvm -polly -mllvm -polly-process-unprofitable -mllvm -polly-use-llvm-names -ffast-math -march=native	0.265	2.041
Clang 10.0 -O3 with polly and autotuning Large: (50 128 256) Extra: (64 50 256) -std=c99 -fno-unroll-loops -O3 -mllvm -polly -mllvm -polly-process-unprofitable -mllvm -polly-use-llvm-names -ffast-math -march=native	0.229	1.907

Abbildung: Polybench syr2k benchmark [10]

Polly heute [6]

- Polly dient heutzutage als Infrastruktur für z.B.:
  - Speicherverwaltungsoptimierung
  - Programm verifikation
  - Kommunikationsoptimierung
  - Code Generation etc.

Polly in Rust [9]

- Polly nicht ganz standardmäßig in LLVM
- Man kann Rust selber bauen mit Polly aktiviert
- Polly erreichbar über rustc in den LLVM-args im codegen Flag

Command Line:

```
$rustc -C llvm-args=--polly
```

Zusammenfassung

- Rustc benutzt LLVM als Backend zum Optimieren
- Opt ist das Haupttool von LLVM, um Passes auszuführen
- Auch Rustc kann LLVM Passes anwenden
- LLVM kann mit den Analyseergebnissen die weitere Optimierung vereinfachen
- Polly ist ein Tool aus der LLVM Toolchain, das Schleifen optimiert und noch vieles mehr

Literaturverzeichnis I

[1] *Guide to Rustc Development.* https://rustc-dev-guide.rust-lang.org/ Letzter Zugriff: 27.01.2021

[2] Niko Matsakis: *Introducing MIR.* https://blog.rust-lang.org/2016/04/19/MIR.html Letzter Zugriff: 27.01.2021

[3] *opt - LLVM optimizer.* https://llvm.org/docs/CommandGuide/opt.html Letzter Zugriff: 27.01.2021

[4] *LLVM's Analysis and Transform Passes.* https://llvm.org/docs/Passes.html Letzter Zugriff: 27.01.2021

[5] *Polly.* https://polly.llvm.org/ Letzter Zugriff: 27.01.2021

[6] *Polyhedral Info.* https://polyhedral.info/ Letzter Zugriff: 27.01.2021

Literaturverzeichnis II

[7] Tobias Grosser, Hongbin Zheng, RagheshAloor, Andreas Simbürger, Armin Größlinger und Louis-Noel Pouchet: *Polly - Polyhedral optimization in LLVM.* https://polly.llvm.org/publications/grosser-impact-2011.pdf Letzter Zugriff: 27.01.2021

[8] Tobias Grosser, Hongbin Zheng: *Polyhedral Transformations for LLVM.* https://llvm.org/devmtg/2010-11/Grosser-Polly.pdf Letzter Zugriff: 27.01.2021

[9] *Polly in Rust.* https://github.com/rust-lang/rust/pull/78566 Letzter Zugriff: 27.01.2021

[10] Xingfu Wu, Michael Kruse, Prasanna Balaprakash, Hal Finkel, Paul Hovland, Valerie Taylor und Mary Hall: *Autotuning PolyBench Benchmarks with LLVM Clang/Polly Loop Optimization Pragmas Using Bayesian Optimization.* https://arxiv.org/pdf/2010.08040.pdf Letzter Zugriff: 28.01.2020