Fraunhofer SCAI

Institut für Informatik
Universität zu Köln

Diplomarbeit

im Diplomstudiengang Wirtschaftsmathematik
der Universität zu Köln

Effiziente Vektorisierung durch semi-automatisierte Code-Optimierung im Polyedermodell

von

Dustin Feld
dustin.feld@scai.fraunhofer.de

Gutachter:	Prof. Dr. Michael Jünger (Universität zu Köln)
Gutachter extern:	Dr. Thomas Soddemann (Fraunhofer SCAI Sankt Augustin)
Eingereicht am:	11. November 2011

Erklärung

Ich versichere an Eides statt, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit wurde bisher in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt und noch nicht veröffentlicht.

Dustin Feld

Eulerstr. 14, 51065 Köln

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation 1
1.2 Aufgabenstellung 1
1.3 Struktur der Arbeit 2

2 Grundlagen **3**

2.1 Mathematische Grundlagen 3
2.1.1 Grundlagen der Linearen Algebra 3
2.1.2 Grundlagen der linearen (ganzzahligen) Optimierung 14
2.2 Technische Grundlagen 22

3 Das Polyedermodell **29**

3.1 Abhängigkeiten 29
3.1.1 Skalare Abhängigkeiten 29
3.1.2 Dynamische Abhängigkeiten 30
3.2 Verschachtelte Schleifen 31
3.3 Indexraum (Domäne) 32
3.4 Gleichförmige und nicht-gleichförmige Abhängigkeiten 32
3.5 Zusammenfassung der Begriffe 34
3.6 Abhängigkeitstest 36
3.6.1 Heuristik-basierter Ansatz 37
3.6.2 Direkte Lösung 39
3.7 Schleifentransformationen 40
3.7.1 Polyeder-Transformation 41
3.7.2 Gültige (legale) Transformation 42
3.7.3 Unimodulare Transformationen 43
3.7.4 Kommutatives Band 45
3.7.5 Gültige (legale) Parallelisierung 46
3.7.6 Vektorisierbarkeit 50
3.7.7 Parallelisierung: Space-Time-Mapping vs. Partitionierungsansatz . 50
3.8 Tiling 56

3.8.1 Formalisierung . . . 56
3.8.2 Gültige Tile-Form . . . 59
3.8.3 Parallelisierbare Tiles . . . 63
3.8.4 Optimale Tile-Form . . . 63
3.8.5 Optimale Tile-Größe . . . 65
3.8.6 Kommunikationsminimales Tiling nach Xue . . . 66
3.9 PluTo . . . 70
3.9.1 Idee und Umsetzung . . . 70
3.9.2 Arbeitsweise . . . 71
3.9.3 Ergebnisse (und Schwächen) . . . 81

4 Konzept und Implementierung 83
4.1 Gesamtkonzept . . . 84
4.2 Neue Elemente . . . 86
4.3 Erweiterungen der Optimierungsphase . . . 87
4.3.1 Lexikografische Minimierung . . . 87
4.3.2 Die Optimierungsprobleme (ILPs) . . . 90
4.3.3 Die Volumenbeschränkung . . . 91
4.4 Der Kern von PluTo-SIMD . . . 93
4.4.1 Transformationssetup und Bestimmung der vektorisierbaren Schleife 93
4.4.2 Auslesen der Hardware-Cache-Parameter . . . 99
4.4.3 Cache-spezifisches Tiling . . . 101
4.5 Gesamtprozess . . . 116

5 Analyse der Ergebnisse 117
5.1 Erweiterungen in der Optimierungsphase . . . 118
5.2 SIMD- und Cache-spezifischer Tiling Algorithmus . . . 120
5.2.1 SIMD-spezifisches Tiling mit **manuellen** Quantitäten . . . 122
5.2.2 SIMD-spezifisches Tiling mit **Cache-spezifischen** Quantitäten . . . 128
5.2.3 Cache-Miss-Rate Benchmark der Matrix-Multiplikation . . . 136
5.2.4 Laufzeit der Kompilierung durch PluTo und PluTo-SIMD . . . 137
5.2.5 Schwächen der Methoden mit dem GCC . . . 138
5.2.6 Vergleich GCC vs. ICC . . . 140
5.2.7 Anwendung auf MPEG4 oder Motion-Tracking . . . 142

6 Resümee & Ausblick 145

A Appendix 149
A.1 Arbeitsweise für mehrere Statements . . . 149
A.2 Wahl der Level-2-Tile Schleife . . . 153

Abkürzungsverzeichnis

C Cache-Hit-Rate (CHR)
Cache-Miss-Rate (CMR)

D Directed-Acyclic-Graph (DAG)

F First-In-First-Out (FIFO)
Fourier-Motzkin-Elimination (FME)
Floating-Point Unit (FPU)

G größter gemeinsamer Teiler (ggT)

H High-Performance-Computing (HPC)

I Integer Linear Program (ILP oder IP)

K kleinstes gemeinsames Vielfache (kgV)

L Least-Frequently-Used (LFU)
Least-Recently-Used (LRU)
Left-Hand-Side (LHS)
Lower-Bound (LB)
Linear Program (LP)

M Multiple Instruction Multiple Data (MIMD)
Multiple Instruction Single Data (MISD)
Multi Media Extension (MMX)
Most-Recently-Used (MRU)

R Right-Hand-Side (RHS)

S Single Instruction Multiple Data (SIMD)
Single Instruction Single Data (SISD)
Streaming SIMD Extensions (SSE)

T Translation Lookaside Buffer (TLB)
Tile-Space-Graph (TSG)

U Upper-Bound (UB)

Symbolverzeichnis

$\mathbb{N}, \mathbb{N}^{>0}$	Halbring der natürlichen Zahlen, echt positive natürliche Zahlen
$\mathbb{Z}, \mathbb{Z}^{+}$	Ring der ganzen Zahlen, nichtnegative ganze Zahlen
$\mathbb{Q}, \mathbb{Q}^{+}$	Körper der rationalen Zahlen, nichtnegative rationale Zahlen
$\mathbb{R}, \mathbb{R}^{+}$	Körper der reellen Zahlen, nichtnegative reelle Zahlen
$\mathbb{K}, \mathbb{K}^{+}$	Allgemeiner Körper, nichtnegative Elemente des Körpers
$\mathcal{I}_N$	N-dimensionale Einheitsmatrix
M_Φ, Φ	Abbildungsmatrix der Abbildung Φ
z^*, x^*	Optimaler Zielfunktionswert z und zugehöriger x-Vektor
$\bar{a}, \bar{b}, \bar{c}$	Durch Simplexalgorithmus transformierte Elemente a, b und c
$\prec, \succ$	lexikographisch kleiner, größer
ρ	prozentualer Anteil (z. B. parallelisierbarer Anteil eines Codes)
$\wp$	Anzahl von Recheneinheiten (z. B. Prozessoren)
$\mathcal{I}$	Indexvektor einer verschachtelten Schleife, eines Statements, etc.
$\mathcal{Z}_\mathcal{I}$	Zugriffsfunktion mit Indexvektor $\mathcal{I}$
$\mathcal{D}^S$	Indexraum (Domäne) eines Statements S
$\mathbb{O}$	Nullmatrix
$e^{S_i \to S_j}$	Abhängigkeit zwischen Statement S_i und S_j
$\mathcal{T}_S$	allgemeine Transformation des Statements S
$\phi_S()$	Eindim. lin. Transformation eines Statements S (Hyperebenennormale)
$\psi_S()$	Eindim. lin. Transformation eines Statements S (Hyperebene)
$\Phi_S()$	Mehrdim. lin. Transformation eines Statements S
$\mathcal{P}_e$	Abhängigkeitspolyeder einer Abhängigkeit e
$\vec{s}, \vec{t}$	Quelle (source) und Ziel (target) einer Abhängigkeit
θ, σ	Schedule, Allokation eines Space-Time-Mappings
G_S, H_S	Clustering-, Tiling-Matrix
$\vec{p}$	Vektor der Programmparameter

1 Einleitung

1.1 Motivation

In den vergangenen zwei Jahrzehnten haben sich Vektoreinheiten als Beschleuniger in CPUs auch im Bereich der PCs etabliert. Allerdings sind selbst moderne Compiler nicht generell in der Lage, dieses Potenzial auszuschöpfen, wenn der Programmierer die entsprechenden Codes nicht explizit für den jeweiligen Beschleuniger schreibt. Für Vektoreinheiten heißt dies, dass die gewöhnlichen Befehle für mathematische Operationen durch entsprechende '*Intrinsics*' für den Vektorbefehlssatz der CPU ersetzt werden müssen. Viele Codes erlauben solche Ersetzungen nativ jedoch gar nicht oder nicht gewinnbringend, könnten aber oft durch Transformationen in eine entsprechende Form gebracht werden. Diese Transformationen können von vielen aktuellen Compilern nicht oder nicht immer realisiert werden, da die hierzu nötigen mathematischen Operationen und Analysen nicht implementiert oder noch gar nicht entwickelt sind.
Darüber hinaus ist in aktuellen Compilern i. a. R. keine Anpassung des resultierenden Codes an die gegebene Hardware vorhanden. So wird ein Code nicht an Parameter wie die vorhandenen Register oder den Cache angepasst, obwohl gerade die Speicherzugriffe in vielen Codes einen, oft sogar *den*, limitierenden Laufzeitfaktor darstellen.

1.2 Aufgabenstellung

Im Fokus dieser Arbeit steht die Entwicklung und Umsetzung semi-automatisierter Code-Optimierungen, die gezielt für eine Vektorisierung sowie angepasst an die gegebene Hardware vorgenommen werden um den End-Compiler in der Vektorisierung zu unterstützen, sodass eine effiziente solche auch ohne Implementierung von *Intrinsics* stattfindet. Diese Aspekte sollen in das Transformations-Framework *PluTo* [Bona] integriert werden. Das PluTo-Framework ist entwickelt worden, um Codes für parallele

Architekturen zu transformieren und für Speicherzugriffe zu optimieren. Die grundlegende Transformation für diese Ziele stellt die Blockung (*Tiling*) verschachtelter Schleifen dar, welche die Iterationen einer Verschachtelung mit dem Ziel einer verbesserten Datenwiederverwendung umsortiert. Allerdings ist in PluTo keinerlei automatischer **Bezug zur vorhandenen Hardware** (insbesondere dem Cache) integriert und die Transformationen zielen nicht auf eine **effektive Vektorisierung** ab. In PluTo-SIMD sollen Methoden und Transformationen für diese beiden Hauptaspekte entwickelt und implementiert werden. Hierzu ist eine Analyse der Zugriffsstruktur eines Codes nötig, um die Transformationen *angepasst* durchführen zu können. Da der User aufgrund der abstrakten Transformation eines solchen Codes und der dadurch veränderten Strukturen eine Anpassung an die Hardware nur mit viel Aufwand und entsprechender Kenntnis der Thematik durchführen kann, soll dies in PluTo-SIMD semi-automatisch vonstattengehen. Das heißt, dass der User über Parameter in einer Konfigurationsdatei die 'Rahmenbedingungen' für die Transformation bestimmen kann und das Framework diese ohne Eingreifen des Users durchführt. Die nötige mathematische Analyse des Codes (Transformationen, Normen, etc.) wird in dieser Arbeit vorgestellt und ist in PluTo-SIMD implementiert.
Die Transformationen werden im Rahmen einer Source-To-Source-Kompilierung stattfinden, sodass aus dem vom User gegebenen Quellcode ein optimierter solcher entsteht, der seinerseits durch einen *beliebigen End-Compiler* in Maschinencode übersetzt werden kann.

1.3 Struktur der Arbeit

In *Kapitel 2* werden die für die Methoden des Frameworks nötigen mathematischen Grundlagen der Linearen Algebra und ganzzahligen Optimierung sowie die technischen Grundlagen erläutert. *Kapitel 3* beleuchtet das der Transformation zugrunde liegende Polyedermodell und stellt so die Verbindung von Quellcodes zum mathematischen Modell und der darauf aufbauenden Transformation zur Speicherzugriffsoptimierung und Parallelisierung sowie Vektorisierung her. Am Ende des Kapitels wird das auf diesem Modell beruhende PluTo-Framework dargestellt. *Kapitel 4* stellt die entwickelten Modelle sowie deren Implementierung in PluTo-SIMD dar, um im Anschluss die hieraus resultierenden Performance-Vorteile in *Kapitel 5* vorzustellen. In *Kapitel 6* wird ein Resümee der entwickelten Modelle und deren Ergebnisse gezogen sowie ein kurzer Ausblick auf zukünftig geplante Erweiterungen gegeben.

2 Grundlagen

2.1 Mathematische Grundlagen

2.1.1 Grundlagen der Linearen Algebra

Ganzzahlige Vektoren und Matrizen Ein Vektor oder eine Matrix werden als **ganzzahlig** bezeichnet, sofern deren Einträge ausschließlich ganze Zahlen sind, d. h.

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,N-1} & a_{1,N} \\ a_{2,1} & & \dots & & a_{2,N} \\ \vdots & & & & \vdots \\ a_{M-1,1} & & \dots & & a_{M-1,N} \\ a_{M,1} & a_{M,2} & \dots & a_{M,N-1} & a_{M,N} \end{pmatrix} \text{ ist ganzzahlig} \Leftrightarrow \forall i,j : a_{i,j} \in \mathbb{Z} \quad (2.1)$$

Eine solche Matrix wird im Folgenden kurz als $A \in \mathbb{Z}^{M \times N}$ mit $M, N \in \mathbb{N}^{>0}$ angegeben. Die N-dimensionale **Einheitsmatrix** ($a_{i,i} = 1,\ a_{i \neq j} = 0$) wird im Folgenden durch $\mathcal{I}_N$ repräsentiert. Der **Rang** $rg(A)$ einer Matrix A entspricht der Dimension des Bildraumes der von dieser Matrix über einem Vektorraum V induzierten linearen Abbildung $v \mapsto A \cdot v$ (mit $v \in V$).

Die Determinante Die Determinante für quadratische Matrizen über einem Körper $\mathbb{K}$ ist die eindeutig definierte Abbildung $det : A^{N \times N}(\mathbb{K}) \to \mathbb{K}$ mit den Eigenschaften:

$$(i) \quad det \text{ ist linear in jeder Zeile} \quad (2.2)$$

$$(ii) \quad \text{Ist der (Zeilen-)Rang kleiner als } N, \text{ so ist } det(A) = 0 \quad (2.3)$$

$$(iii) \quad det(\mathcal{I}_N) = 1 \quad (2.4)$$

Die Zahl $\hat{a} = det(A) \in \mathbb{K}$ heißt Determinante von A. Für einen Beweis zu Existenz und Eindeutigkeit, siehe [Jä96, S. 135].

Eine Berechnungsvorschrift ist der Laplacesche Entwicklungssatz nach der j-ten Spalte: $det(A) = \sum_{i=1}^{N} (-1)^{i+j} * a_{ij} * det(A_{ij})$, wobei A_{ij} durch Weglassen der i-ten Zeile und j-ten Spalte aus A entsteht. Da der Fokus im Folgenden nicht auf der Berechnung solcher Determinanten liegt, wird nun nicht näher darauf eingegangen. Es sei erwähnt, dass dieser Entwicklungssatz hauptsächlich theoretischer Natur ist und es deutlich effizientere Methoden zur praktischen Berechnung der Determinante gibt.

Lemma 2.1. *Für Determinanten quadratischer Matrizen A, B, C mit $C = A \cdot B$ gilt:*

$$det(C) = det(A \cdot B) = det(A) \cdot det(B) \tag{2.5}$$

Diese Aussage lässt sich durch wiederholte Anwendung auf beliebig viele Matrizen erweitern.

Inverse (ganzzahlige) Matrizen Die Inverse zu einer invertierbaren quadratischen Matrix $A \in \mathbb{K}^{N \times N}$ ist die eindeutig definierte Matrix $A' \in \mathbb{K}^{N \times N}$, sodass $A * A' = A' * A = \mathcal{I}_N$ gilt.

Lemma 2.2. *Eine quadratische Matrix $A \in \mathbb{K}^{N \times N}$ über einem Körper $\mathbb{K}$ ist genau dann invertierbar, wenn gilt:*

$$det(A) \neq 0$$

Beweisskizze. Per Definition (2.3) ist $det(A) = 0$, falls $rg(A) < N$ ist. Sei also im Folgenden $rg(A) = N$. A kann in diesem Fall durch elementare Zeilentransformationen in $\mathcal{I}_N$ umgeformt werden. Diese beeinflussen die Determinante nur durch Multiplikation mit Elementen aus $\mathbb{K} \setminus 0$. Somit kann $det(A)$ nicht null sein, denn sonst müsste auch $det(\mathcal{I}_N) = 1$ null sein. *Details zum Beweis sind in [Jä96, Kap. 6] zu finden.* □

Die zu A inverse Matrix A' wird mit A^{-1} bezeichnet.

Eine allgemeinere Aussage als in Lemma 2.2 zur Invertierbarkeit einer quadratischen Matrix über einem Ring lautet:

Lemma 2.3. *Eine quadratische Matrix A über einem Ring ist genau dann invertierbar, wenn ihre Determinante invertierbar ist.*

Betrachten wir nun ganzzahlige Matrizen, also $A \in \mathbb{Z}^{N \times N}$. Da $\mathbb{Z}$ kein Körper ist, sondern ein Ring (es existieren Nicht-Nullelemente ohne multiplikativem Inversen in $\mathbb{Z}$), müssen weitere Eigenschaften für die Invertierbarkeit erfüllt sein.

Da in $\mathbb{Z} \setminus 0$ nur 1 und -1 invertierbar sind, gilt nach Lemma 2.3:

Satz 2.4. *Eine quadratische ganzzahlige Matrix $A \in \mathbb{Z}^{N\times N}$ ist genau dann ganzzahlig invertierbar (also $A^{-1} \in \mathbb{Z}^{N\times N}$), wenn gilt:*

$$|det(A)| = 1$$

Eine solche Matrix wird als *unimodular* bezeichnet und erhält besondere Bedeutung u. A. als Transformationsmatrix in der Code-Optimierung.

Lemma 2.5. *Ist A unimodular, so ist auch A^{-1} unimodular.*

Beweis.

$$A \text{ unimodular } \Rightarrow\ |det(A)| = 1\ \Rightarrow\ |det(A^{-1})| = \left|\frac{1}{det(A)}\right| = 1\ \Rightarrow\ A^{-1} \text{ unimodular}$$

□

Lineare und affine Abbildung Eine Abbildung $\Phi : \mathbb{K}^S \to \mathbb{K}^T$ $(S, T \in \mathbb{N}^{>0})$ ist genau dann ***linear***, wenn sie durch

$$\Phi(v) = M_\Phi \cdot v \ \text{ mit } v \in \mathbb{K}^S,\ M_\Phi \in \mathbb{K}^{T\times S} \tag{2.6}$$

dargestellt werden kann.

Eine Abbildung $\Phi : \mathbb{K}^S \to \mathbb{K}^T$ ist genau dann ***affin***, wenn sie durch

$$\Phi(v) = M_\Phi \cdot v + f_0 \ \text{ mit } v \in \mathbb{K}^S,\ f_0 \in \mathbb{K}^T,\ M_\Phi \in \mathbb{K}^{T\times S} \tag{2.7}$$

dargestellt werden kann (vgl. [Bon08]). Im Folgenden sind M_Φ und f_0, sowie v in der Regel ganzzahlig.
Da $\mathbb{Z}$ bzgl. Multiplikation und Addition abgeschlossen ist, gilt folgendes Lemma.

Lemma 2.6. *Für ganzzahlige M_Φ, f_0, v gilt:*

$$Bild(\Phi) \subset \mathbb{Z}\ .$$

Beweis.

$$\begin{aligned} \text{Es gilt:} \quad & M_\Phi \in \mathbb{Z}^{T\times S}, f_0 \in \mathbb{Z}^T, v \in \mathbb{Z}^S \\ \Phi(v) &= M_\Phi \cdot v + f_0 \\ &= \left(\underbrace{\sum_{j=1}^{|S|} \underbrace{m_{ij} * v_j}_{\in\mathbb{Z}} + f_{0_i}}_{\in\mathbb{Z}} \right)_i \quad \in \mathbb{Z} \end{aligned}$$

□

Der Rang der Abbildung Φ ist mit dem Rang der Abbildungsmatrix M_Φ identisch.

Kern einer Abbildung Unter dem Kern einer affinen Abbildung $\Phi : \mathbb{K}^S \rightarrow \mathbb{K}^T$ mit $\Phi(v) = M_\Phi \cdot v + f_0$ versteht man die Menge der Vektoren, die auf das Nullelement des Vektorraumes $\mathbb{K}^T$ abgebildet werden, also $Kern(\Phi) = \{v \in \mathbb{K}^S \mid \Phi(v) = \vec{0}\}$. Dieser bildet im Falle eines zugrundeliegenden Vektorraumes einen Untervektorraum von $\mathbb{K}^S$ und ist äquivalent zum Kern der Matrix M_Φ. Wenn M_Φ vollen Spaltenrang hat (*), ist der Kern einer linearen Abbildung der *Nullvektor*. Dies folgt sofort aus der Unterraumeigenschaft und der Dimensionsformel für lineare Abbildungen:

$$\begin{aligned} & dim(Kern(\Phi)) + rg(\Phi) && = S \\ \Rightarrow & dim(Kern(\Phi)) + rg(M_\Phi) && = S \\ \Rightarrow & dim(Kern(\Phi)) + S && = S \quad wegen(*) \\ \Rightarrow & dim(Kern(\Phi)) && = 0 \end{aligned}$$

Da der Untervektorraum den Nullvektor enthalten muss, gilt somit $Kern(\Phi) = \vec{0}$.

Der Orthogonalraum Zwei Elemente s und t eines Vektorraumes mit einem positiv definiten inneren Produkt $< ., . >$ heißen **orthogonal**, wenn:

$$< s, t > = \ 0 \tag{2.8}$$

In der geometrischen Interpretation stehen die beiden Vektoren *senkrecht aufeinander* und man schreibt $s \perp t$. Das Standardskalarprodukt ist das innere Produkt mit der Einheitsmatrix als Abbildungsmatrix. Kurz schreibt man $s.t$ oder $s^T \cdot t$ für das Standardskalarprodukt. Ausgeschrieben heißt das im N-dimensionalen Fall:

$$\begin{aligned} < s, t >_{\mathcal{I}} \ &= \ s^T \cdot \mathcal{I}_N \cdot t \\ &= \ s_1 * t_1 \ + \ \ldots \ + \ s_N * t_N \end{aligned} \tag{2.9}$$

Der Orthogonalraum (oder auch: das **orthogonale Komplement**) einer Teilmenge W eines Vektorraumes V ($W \subset V$) ist die Menge der auf W senkrecht stehenden Vektoren, d.h.

$$W^\perp \ = \ \{v \in V \mid v \perp w \ \ \forall w \in W\} \tag{2.10}$$

Statt $v \perp w \ \ \forall w \in W$ schreibt man auch kurz $v \perp W$.

Bemerkung 2.7. *$W^\perp$ ist ein Untervektorraum von V.*

(Endliche) Linear- und Konvexkombination Seien $v_1, ..., v_n$ Vektoren eines Vektorraumes V über einem Körper $\mathbb{K}$.

Dann ist $v \in V$ eine ***Linearkombination*** dieser Vektoren, falls:

$$\exists \lambda_1, ..., \lambda_n \in \mathbb{K} : \; v = \lambda_1 * v_1 + ... + \lambda_n * v_n \tag{2.11}$$

$v \in V$ ist eine ***Affinkombination*** dieser Vektoren, falls *zusätzlich* zu (2.11) gilt:

$$\sum_{i=1}^{n} \lambda_i = 1 \tag{2.12}$$

$v \in V$ ist eine ***Konvexkombination*** dieser Vektoren, falls *zusätzlich* zu (2.12) gilt:

$$0 \leq \lambda_i \leq 1 \quad \forall i \tag{2.13}$$

Eine geometrische Interpretation für zwei Vektoren ist in Abbildung 2.1 zu sehen.

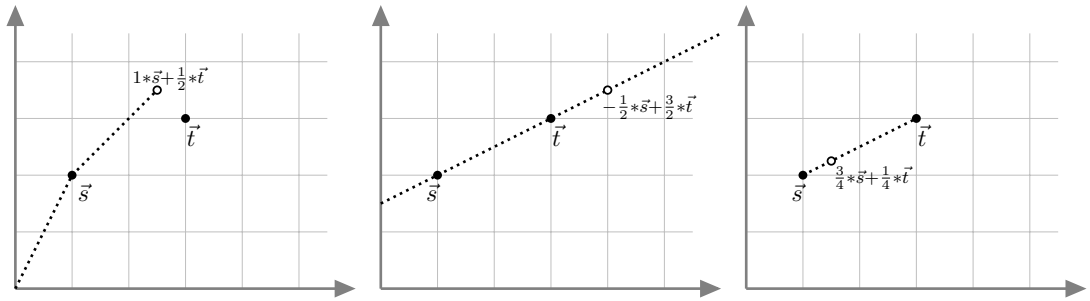


Abbildung 2.1: 2-D Linearkombination, Affinkombination und Konvexkombination

Konvexe Menge Eine Teilmenge $P \subseteq \mathbb{R}^N$ ist konvex, wenn für zwei beliebige Elemente $s, t \in P$ gilt, dass jede Konvexkombination dieser Elemente ebenfalls in P enthalten ist, also:

$$\forall s, t \in P \; \wedge \; \forall \lambda \in [0, 1] : \lambda * s + (1 - \lambda) * t \; \in \; P \tag{2.14}$$

Konvexe Hülle Die konvexe Hülle einer Menge von Vektoren $\{v_1, \ldots, v_k\} \in \mathbb{R}^N$ ist die Menge aller Konvexkombinationen dieser Vektoren $conv(v_1, \ldots, v_k)$. Es ist insbesondere die kleinste konvexe Menge, die all diese Vektoren enthält.

Konische Hülle Die konische Hülle einer Menge von Vektoren $\{v_1, \ldots, v_k\} \in \mathbb{R}^N$ ist die Menge aller ***konischen Linearkombinationen*** dieser Vektoren $cone(v_1, \ldots, v_k)$. Eine Linearkombination (2.11) heißt *konisch*, falls $\lambda_i \geq 0$.

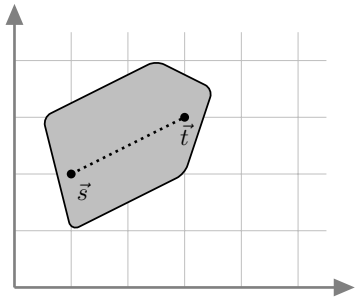


Abbildung 2.2: Konvexe Menge

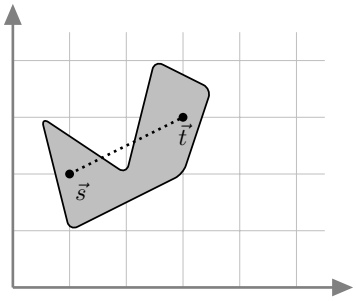


Abbildung 2.3: Nichtkonvexe Menge

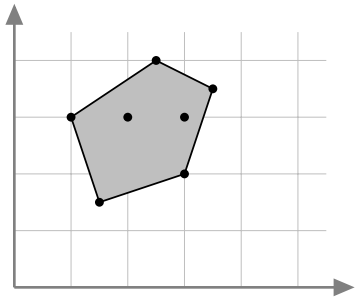


Abbildung 2.4: Konvexe Hülle

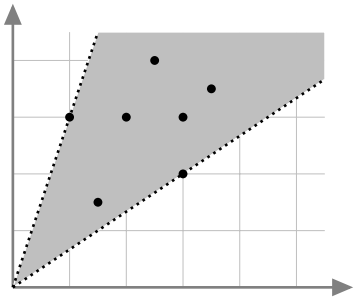


Abbildung 2.5: Konische Hülle

Vereinigung und Schnitt konvexer Mengen

Lemma 2.8. *Der Schnitt beliebig vieler konvexer Mengen ist eine konvexe Menge.*

Beweis. Seien P_i konvexe Mengen im $\mathbb{R}^N$. Seien $s, t \in \mathcal{P} = \bigcap_i P_i$ beliebig.
Zu zeigen: $\lambda * s + (1 - \lambda) * t \ \in \ \mathcal{P}$
Da alle P_i konvex sind und $s, t \in P_i \ \ \forall i$, gilt $\lambda * s + (1 - \lambda) * t \in P_i \ \ \forall i$.
Wenn jede dieser Konvexkombinationen von s und t aber in jedem P_i liegt, so auch im Schnitt der P_i. □

Lemma 2.9. *Die Vereinigung konvexer Mengen ist nicht zwingend konvex.*

Beweis. Sei $P_1 = \{x \in \mathbb{R} \mid x > 3\}$ und $P_2 = \{x \in \mathbb{R} \mid x < 1\}$.
Sei $\mathcal{P} = P_1 \cup P_2 = \{x \in \mathbb{R} \mid x < 1 \ \vee \ x > 3\}$.
Somit gilt $0 \in \mathcal{P}$ und $4 \in \mathcal{P}$, aber $\frac{1}{2} * 0 + \frac{1}{2} * 4 = 2 \notin \mathcal{P}$. □

Affiner Raum Eine unter Affinkombinationen abgeschlossene Menge von Vektoren ist ein affiner Raum [Bon08, S. 11].

Affine Hyperebenen Eine affine Hyperebene ist ein $(N-1)$-dimensionaler affiner Unterraum eines N-dimensionalen Vektorraums V über einem Körper $\mathbb{K}$ (ein klassisches Beispiel ist die Ebene im $\mathbb{R}^3$). Alle Elemente $v \in V$, welche die Gleichung

$$\begin{aligned} \mathbf{h}.v &= k \\ \text{also} \quad h_1 * v_1 + \ldots + h_N * v_N &= k \end{aligned} \tag{2.15}$$

für ein $k \in \mathbb{K}$ erfüllen, bilden somit eine solche Hyperebene (vgl. [Bon08]). Der Vektor $\mathbf{h}$ ist die Normale der Hyperebene und dessen Hyperebenen für verschiedene k sind parallel. Somit liegen zwei Vektoren v_1 und v_2 in derselben dieser Hyperebenen, wenn gilt:

$$\mathbf{h}.v_1 = \mathbf{h}.v_2 \quad \text{also} \quad \mathbf{h}.(v_1 - v_2) = 0 \tag{2.16}$$

Ist $k \in \mathbb{Z}$ und $v \in \mathbb{Z}^N$, so zerlegt (2.15) für verschiedene k den Raum durch parallele $(N-1)$-dimensionale Schnitte.

Eine affine Hyperebene kann auch als Funktion

$$\begin{aligned} \phi &: \mathbb{K}^N \to \mathbb{K} \\ \phi(v) &= \mathbf{h}.v + c \\ &\quad \text{mit } \mathbf{h} \in \mathbb{K}^N, c \in \mathbb{K} \end{aligned} \tag{2.17}$$

interpretiert werden, die Vektoren des Raumes auf seinen zugrundeliegenden Körper abbildet.

Affine Halbräume Affine Hyperebenen teilen ihren zugrundeliegenden Raum in zwei Halbräume. Man nennt den durch die Ungleichung $\mathbf{h}.v \geq k$ definierten Halbraum **positiven Halbraum**, den durch $\mathbf{h}.v \leq k$ definierten **negativen Halbraum**.

Polyeder und Polytope Ein (konvexes) **Polyeder** ist der Schnitt endlich vieler affiner Halbräume. Ein **Polytop** ist ein beschränktes Polyeder. Somit kann man jedes Polyeder und jedes Polytop im $\mathbb{R}^N$ durch endlich viele Ungleichungen beschreiben. Durch die zwei folgenden Eigenschaften kann man jedes Polyeder oder jedes Polytop in der allgemeinen

Form eines Ungleichungssystems

$$\{x \in \mathbb{R}^N \mid A \cdot x \leq b\} \quad \text{mit} \quad A \in \mathbb{R}^{N \times M},\ b \in \mathbb{R}^N \tag{2.18}$$

darstellen:

1. $\mathbf{h}.v \ \geq \ k \quad \Leftrightarrow \quad -\mathbf{h}.v \ \leq \ -k$
2. $\mathbf{h}.v \ = \ k \quad \Leftrightarrow \quad \mathbf{h}.v \ \geq \ k \wedge \mathbf{h}.v \ \leq \ k$

Im Folgenden sei (2.18) die allgemeine Form eines Polyeders.

Im Kontext dieser Arbeit sind hauptsächlich **ganzzahlige Polyeder**, im Folgenden $\mathbb{Z}$-Polyeder, von Interesse. Diese lassen sich wie folgt definieren:

$$\{x \in \mathbb{R}^N \mid A \cdot x \leq b\} \cap \mathbb{Z}^N \ \text{ mit } \ A \in \mathbb{R}^{N \times M},\ b \in \mathbb{R}^N$$

oder kurz:

$$\{x \in \mathbb{Z}^N \mid A \cdot x \leq b\} \ \text{ mit } \ A \in \mathbb{R}^{N \times M},\ b \in \mathbb{R}^N \tag{2.19}$$

Polyeder und Polytope sind konvexe Mengen. Eine alternative Repräsentation zum Ungleichungssystem eines Polytops ist die konvexe Hülle seiner Eckpunkte.

Seitenflächen und Facetten Jede Hyperebene $a.x = b$, für die $a.x \leq b$ von allen Punkten des Polyeders P respektiert wird, bildet durch den Schnitt mit dem Polyeder selbst eine **Seitenfläche**. Eine $(N-1)$-dimensionale Seitenfläche eines N-dimensionalen Polyeders heißt **Facette** des Polyeders.
Darüber hinaus ist das Polyeder selbst eine N-dimensionale Seitenfläche des Polyeders, formal ist auch die leere Menge $\emptyset$ eine Seitenfläche.
Eine geometrische Veranschaulichung dieser Begriffe ist in Abbildung 2.6 zu sehen.

Bemerkung 2.10. *Oft ist man interessiert an einer Beschreibung des Polyeders durch Facetten, da eine solche die Anzahl der notwendigen Ungleichungen minimiert. Dies ist bei der konkreten Lösung von Optimierungsproblemen auf Polyedern äußerst wichtig, da die Anzahl der Ungleichungen ein dominierender Laufzeitfaktor ist. Das Finden solcher Facetten für ein gegebenes Problem ist i. d. R. sehr aufwändig, in Anbetracht des Laufzeitgewinns trotzdem oft lohnenswert. Ein Beispiel für eine intensive Facettensuche ist das Traveling-Salesman-Polytop (Details dazu z. B. in [JRR97]).*

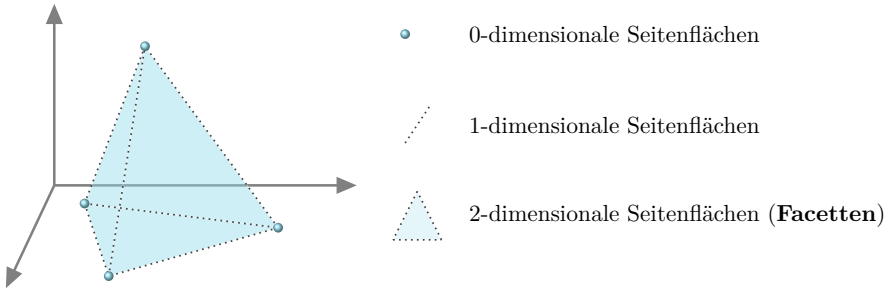


Abbildung 2.6: Pyramide als Polyeder im $\mathbb{R}^3$

Fourier-Motzkin-Elimination Das Eliminationsverfahren von Fourier und Motzkin (siehe z. B. [BT97]) ist eine Projektionsmethode zur Eliminierung von Ungleichungen in einem Ungleichungssystem und kann zur Beantwortung der Lösbarkeit eines solchen Systems genutzt werden. Das Verfahren beruht auf der sukzessiven Elimination von Variablen in einem Ungleichungssystem durch Projektionen. Details zur Methode sind beispielsweise in [Sch10, S.155 f.] zu finden.
Sei $A \cdot x \leq b$ mit $A \in \mathbb{R}^{M \times N},\ x \in \mathbb{R}^N,\ b \in \mathbb{R}^M$ ein Ungleichungssystem mit M Ungleichungen und N Variablen in Standardform.

$$\left[\begin{array}{rrcrrcr}
a_{1,1} * x_1 + & a_{1,2} * x_2 + & \dots & + & a_{1,N-1} * x_{N-1} + & a_{1,N} * x_N & \leq & b_1 \\
a_{2,1} * x_1 + & a_{2,2} * x_2 + & \dots & + & a_{2,N-1} * x_{N-1} + & a_{2,N} * x_N & \leq & b_2 \\
 & & \vdots & & & & & \\
a_{M-1,1} * x_1 + & a_{M-1,2} * x_2 + & \dots & + & a_{M-1,N-1} * x_{N-1} + & a_{M-1,N} * x_N & \leq & b_{M-1} \\
a_{M,1} * x_1 + & a_{M,2} * x_2 + & \dots & + & a_{M,N-1} * x_{N-1} + & a_{M,N} * x_N & \leq & b_M
\end{array}\right]$$

Um zu prüfen, ob dieses System eine Lösung besitzt, durchläuft man alle x_j und eliminiert diese wie folgt:

- Gleichung i ist untere Schranke (LB) α_i für x_j, falls $a_{i,j} < 0$
- Gleichung i ist obere Schranke (UB) β_i für x_j, falls $a_{i,j} > 0$
- Gleichung i ist in diesem Schritt irrelevant für x_j, falls $a_{i,j} = 0$

Normiere nun jede Ungleichung bzgl x_j.

Umgestellt erhält man z. B. für x_1:

$$\left[\begin{array}{lll}
x_1 \leq b_1 - \left(\frac{a_{1,2}}{a_{1,1}} * x_2 \right. & + \ldots + \frac{a_{1,N-1}}{a_{1,1}} * x_{N-1} + \frac{a_{1,N}}{a_{1,1}} * x_N & \left. \right) \mid a_{1,1} > 0 \\
x_1 \geq b_2 - \left(\frac{a_{2,2}}{a_{2,1}} * x_2 \right. & + \ldots + \frac{a_{2,N-1}}{a_{2,1}} * x_{N-1} + \frac{a_{2,N}}{a_{2,1}} * x_N & \left. \right) \mid a_{2,1} < 0 \\
\quad \vdots & & \\
x_1 \geq b_{M-1} - \left(\frac{a_{M-1,2}}{a_{M-1,1}} * x_2 \right. & + \ldots + \frac{a_{M-1,N-1}}{a_{M-1,1}} * x_{N-1} + \frac{a_{M-1,N}}{a_{M-1,1}} * x_N & \left. \right) \mid a_{M-1,1} < 0 \\
a_{M,2} * x_2 & + \ldots + \quad a_{M,N-1} * x_{N-1} + a_{M,N} * x_N \leq b_M & \mid a_{M,1} = 0
\end{array}\right]$$

Die rechten Seiten (RHS) der '$\leq$'-Ungleichungen sind nun obere Schranken β_i für x_1, die RHS der '$\geq$'-Ungleichungen sind untere Schranken α_i für x_1. Die Elimination von x_1 wird umgesetzt, indem man ein neues System von Ungleichungen mit allen Kombinationen (von α_i und $\beta_{i'}$) aus oberen und unteren Schranken $\alpha_i \leq \beta_{i'}$ für x_1 bildet. Die irrelevanten Ungleichungen werden übernommen, da sie x_j nicht enthalten. Auf diesem System (ggf. nach Umsortierung der Variablen in Standardform) wird nun die nächste Variable eliminiert.
Sind alle Variablen x_j auf diese Weise eliminiert worden, so bleibt eine Mengen von Ungleichungen *ohne* Unbekannte übrig.
Sind alle diese Ungleichungen *wahr*, so ist das System lösbar, ansonsten nicht.

Bemerkung 2.11. *Ein Problem dieses (dem Gaußschen-Eliminationsverfahren für Gleichungssysteme ähnlichen) Verfahrens ist die stark steigende Anzahl von Ungleichungen. Die Komplexität der Fourier-Motzkin-Elimination ist $\mathcal{O}(M^{2^N})$, hauptsächlich bedingt durch die Kombination aller LB und UB in jedem Projektionsschritt [Jim99].*

Das Farkas Lemma Das Farkas Lemma (vgl. [BT97, S. 165]) wird in der Literatur in diversen Formen und Spezialisierungen angegeben. Eine allgemeine Form lautet:

Lemma 2.12. ***(Farkas)*** *Sei $A \in \mathbb{R}^{M \times N}$ und $b \in \mathbb{R}^M$. Dann ist **genau eines** der folgenden Systeme (für x bzw. y) lösbar:*

- $A \cdot x = b, \quad x \geq 0$
- $y^T \cdot A \geq 0, \quad y^T \cdot b < 0$

Eine für den weiteren Kontext relevante Spezialisierung [Bon08, S. 13] folgt.

Lemma 2.13. ***(Farkas (affine Form))*** *Sei $\mathcal{P}$ ein nichtleeres Polyeder im $\mathbb{R}^N$ (beschrieben durch M Ungleichungen) folgender Form*

$$\begin{aligned} \mathcal{P} &= \{x \in \mathbb{R}^N \mid A \cdot x + b \geq 0\} \\ &mit \quad A \in \mathbb{R}^{M \times N}, b \in \mathbb{R}^M. \end{aligned} \tag{2.20}$$

Dann ist eine affine Abbildung nichtnegativ in ganz $\mathcal{P}$ genau dann, wenn sie eine ggf. konstant verschobene konische Linearkombination der Seitenflächen von $\mathcal{P}$ ist (vgl. [Sch86, S. 93]), also wenn

$$\phi(x) \equiv \lambda_0 \; + \; \sum_{k=1}^{M} \lambda_k \cdot (a_k \cdot x + b_k) \qquad mit \;\; \lambda_1, \ldots, \lambda_M \geq 0 \tag{2.21}$$

mit a_k der k-te Zeilenvektor von A.

Die nichtnegativen Konstanten λ_i werden in diesem Kontext ***Farkas Multiplikatoren*** *genannt.*

Beweisidee. (geometrisch, vgl. [Sch10, S. 35])]

Sei a_i der i-te Zeilenvektor von A.
Sei $\delta' = max\{c^T \cdot x \mid A \cdot x \leq b\}$ an Stelle x^*.
Seien o. B. d. A. die ersten k Zeilen der Matrix diejenigen, welche in x^* mit Gleichheit erfüllt sind. In der Abbildung wird deutlich, dass der Normalenvektor c eine konische Linearkombination der Vektoren a_1 und a_2 sein muss. Wäre c außerhalb dieses Kegels, so würde die Hyperebene kippen und einen anderen Optimalpunkt, eine andere Ecke, x^* bilden.

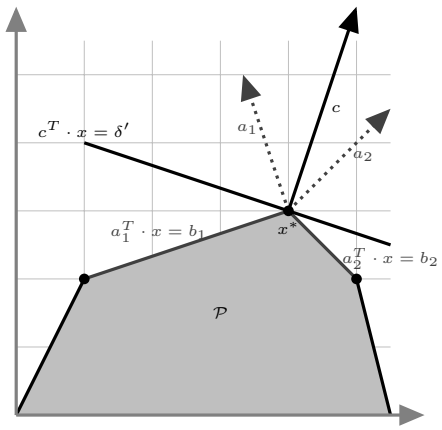


Abbildung 2.7: Farkas Lemma - geometrisch

Es existieren somit $\lambda_1, \ldots \lambda_k \geq 0$, sodass

$$\lambda_1 * a_1 + \ldots + \lambda_k * a_k = c^T \quad \text{und} \tag{2.22}$$

$$\lambda_1 * b_1 + \ldots + \lambda_k * b_k = \delta. \tag{2.23}$$

□

2.1.2 Grundlagen der linearen (ganzzahligen) Optimierung

Die ersten Methoden der linearen Optimierung wurden 1939 von Kantorovich [Kan39] erarbeitet und danach durch Dantzig u. A. im Projekt ***SCOOP*** *(**S**cientific **C**omputation **o**f **O**ptimum **P**rograms)* der U.S. Air Force mit K. Wood weiterentwickelt [GA05, Kap. 4]. Ursprünglich also zur Optimierung von Produktionsprozessen sowie zu Militärzwecken entwickelt, werden die Methoden dieses mathematischen Zweigs heute zur Lösung vielfältiger Probleme genutzt. Dazu haben sich weitere Spezialisierungen wie z. B. die *ganzzahlige lineare Optimierung* gebildet, dessen effiziente Lösung bis heute eine der wichtigsten offenen Fragen der theoretischen Informatik ist. Einige der bekanntesten Probleme dieses Bereichs sind das 'Traveling-Salesman-Problem' und das 'Rucksack-Problem'.

Optimierung über Polyedern Polyeder im $\mathbb{R}^N$, repräsentiert durch Ungleichungssysteme, sind wichtige Modelle in der mathematischen Optimierung. Grundsätzliches Ziel dieser Optimierung ist die Maximierung bzw. Minimierung einer gegebenen Zielfunktion, unter Einhaltung der *Restriktionen* des Polyeders. Allgemein kann man Optimierungsprobleme auf beliebigen geordneten Mengen definieren, hier sind aber als Mengen $\mathbb{R}, \mathbb{Q}$ oder $\mathbb{Z}$ und als Ordnung i. d. R. die natürliche Ordnung relevant.
Gewöhnlich werden an die Zielfunktion und die Restriktionen Bedingungen gesetzt, z. B. *Linearität*. Im Folgenden handelt es sich um **lineare Programme**, also der Maximierung bzw. Minimierung einer linearen Zielfunktion über einem konvexen Polyeder.
Auch an die Variablen können Bedingungen gebunden werden, beispielsweise können einige der Variablen als ganzzahlig gefordert werden. Das *lineare Programm* in einer Standardform lässt sich wie folgt mathematisch beschreiben:

$$\max \; c^T \cdot x \qquad \text{bzw.} \qquad \max \; \sum_{i=1}^{N} c_i \cdot x_i$$

$$s.t. \;\; A \cdot x \leq b \qquad\qquad s.t. \;\; \begin{aligned} a_{1,1} * x_1 + \ldots + a_{1,N} * x_N &\leq b_1 \\ &\vdots \\ a_{M,1} * x_1 + \ldots + a_{M,N} * x_N &\leq b_M \end{aligned}$$

$$x_i \in \mathbb{R}^+ \qquad\qquad x_i \in \mathbb{R}^+$$

s.t. steht für **subject to**.

Bemerkung 2.14. *Graphisch entspricht das Ziel eines solchen Maximierungsproblems der Verschiebung einer Hyperebene $c^T \cdot x$ im gegebenen Polyeder $\mathcal{P}$ (beschrieben durch*

$A \cdot x \leq b$) bis zum Maximalwert x^* (siehe Abbildung 2.8). Somit gilt $c^T \cdot x^* \geq c^T \cdot x \ \forall x \in \mathcal{P}$. Ein Minimierungsproblem verfolgt analog dazu die Verschiebung der Hyperebene zum Minimalwert.

Ein Optimalwert eines linearen Programms wird immer auch an einer Ecke des Polyeders angenommen [Aig06, Satz 13.15]. Ist die Zielfunktion parallel zu einer im Optimum mit Gleichheit erfüllten Ungleichung, so gibt es mehr als eine Optimallösung mit demselben Zielfunktionswert. Ist das Polyeder in Richtung der Verschiebung (also in Optimierungsrichtung) unbeschränkt, so gibt es keine Optimallösung, ebenso wenn das Polyeder leer ist.

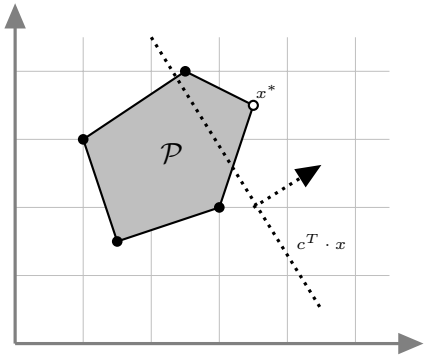


Abbildung 2.8: Maximierung einer linearen Zielfunktion

Bemerkung 2.15. *Zur Veranschaulichung der Probleme sind als Beispiele i. d. R. zweidimensionale Polyeder gewählt. Es sei erwähnt, dass die Prinzipien auf beliebig dimensionale Probleme übertragbar sind.*

Äquivalente Formen linearer Programme Verschiedene Formen von Ungleichungen können, wie bereits in (2.18) gesehen, ineinander überführt werden, sodass jedes gegebene lineare Programm (abgesehen von Ganzzahligkeitsrestriktionen) in die oben angegebene Standardform gebracht werden kann. Einige äquivalente Formen sind z. B. :

$$\begin{array}{lll} \max \ c^T \cdot x & & \min \ -c^T \cdot x \\ s.t. \ A \cdot x \leq b & \Leftrightarrow & s.t. \ A \cdot x \leq b \\ x_i \in \mathbb{R}^+ & & x_i \in \mathbb{R} \end{array} \quad \Leftrightarrow \quad \begin{array}{l} \min \ -c^T \cdot x \\ s.t. \ -A \cdot x \geq -b \\ x_i \in \mathbb{R}^+ \end{array}$$

Die verschiedenen Ungleichungen können auch innerhalb eines Programms vorkommen.

Dualität linearer Programme

Das zu einem linearen Programm in Standardform

$$\begin{array}{l} \max \ c^T \cdot x \\ s.t. \ A \cdot x \leq b \\ x_i \in \mathbb{R}^+ \end{array} \tag{2.24}$$

äquivalente duale Programm lautet:

$$\begin{aligned} \min \quad & y^T \cdot b \\ s.t. \quad & y^T \cdot A \geq c^T \\ & y_i \in \mathbb{R}^+ \end{aligned} \tag{2.25}$$

Satz 2.16. ***(schwacher Dualitätssatz)***

$$c^T \cdot x \leq y^T \cdot b$$

Beweis.

$$c^T \cdot x \leq (y^T \cdot A) \cdot x = y^T \cdot (A \cdot x) \leq y^T \cdot b$$

□

Der **starke Dualitätssatz** verschärft diese Aussage.

Satz 2.17. ***(starker Dualitätssatz)*** *Seien x^* die Optimallösung des primalen Programms (2.24) und y^* die Optimallösung des dualen Programms (2.25), dann gilt:*

$$c^T \cdot x^* = {y^*}^T \cdot b$$

Bemerkung 2.18. *Bei der Lösung des primalen Programms (z. B. durch den Simplex-Algorithmus) nähert sich der Zielfunktionswert von 'einer Seite' der Optimallösung an, beim dualen Programm nähert sich dieser von der 'anderen Seite' an. In der Optimallösung 'treffen' sich die Zielfunktionswerte der beiden Systeme.*

Lösung linearer Programme mit der Fourier-Motzkin-Elimination Die Fourier-Motzkin-Elimination (FME) beantwortet die Frage der Lösbarkeit eines Ungleichungssystems. Darüber hinaus lässt sich mit dieser Methode aber auch ein lineares Programm mit Zielfunktion lösen. Gegeben sei ein lineares Programm in Standardform.

$$\begin{aligned} \max \quad & c^T \cdot x \\ s.t. \quad & A \cdot x \leq b \\ & x_i \in \mathbb{R}^+ \end{aligned}$$

Ist das System lösbar, so kann man die Zielfunktion maximieren, indem man eine, die Zielfunktion repräsentierende, zusätzliche Variable x_{N+1} sowie eine zusätzliche Ungleichung $c^T \cdot x \leq x_{N+1}$ einführt, sodass x_{N+1} eine obere Schranke für die Zielfunktion

bildet. Das so entstandene System hat dann die Form:

$$\begin{pmatrix} A & 0 \\ c^T & -1 \end{pmatrix} \cdot x \leq \begin{pmatrix} b \\ 0 \end{pmatrix}$$

Wendet man nun die FME auf die ersten N Variablen an, so erhält man ein Ungleichungssystem $F \cdot x \leq f$ mit nur noch einer Variable x_{N+1} und diversen Ungleichungen der Form $\alpha_i \leq x_{N+1}$ und $x_{N+1} \leq \beta_i$. Da das Polyeder der **Schnitt** all dieser Halbräume ist, ist der maximale Wert für die Zielfunktion der größte Wert, der all jene Ungleichungen erfüllt, also

$$c^* = \max \ \ c^T \cdot x = \min_i \beta_i \ \ . \tag{2.26}$$

Um eine zu diesem Zielfunktionswert gehörige Lösung x zu erhalten, kann man $x_{N+1} = c^*$ wählen und die x_i sukzessive rückwärts in die zuvor projizierten Systeme einsetzen.

Auf gleiche Weise kann man durch das Hinzufügen der Ungleichung $c^T \cdot x \geq x_{N+1}$ in das System $A \cdot x \leq b$ das Minimum der Zielfunktion $c^T \cdot x$ finden, wobei man

$$c^* = \min \ \ c^T \cdot x = \max_i \alpha_i \tag{2.27}$$

setzt.

Sollen sowohl Maximum als auch Minimum gefunden werden, so kann man beide Ungleichungen hinzufügen und die ersten N Variablen eliminieren.

Damit ist ein prinzipiell einfacher Algorithmus zur Lösung linearer Programme vorhanden, welcher allerdings in der Praxis bereits bei relativ kleinen Systemen einen großen Nachteil hat, nämlich die *exorbitante* Laufzeit (Bemerkung 2.11).
Eine deutlich effizientere Methode zur Lösung eines solchen System stellt der *Simplex-Algorithmus* dar.

Der Simplex-Algorithmus Der Simplex-Algorithmus (im Folgenden zunächst immer der **primale** Simplex) ist ein Verfahren zur Lösung linearer Programme. Es handelt sich um ein iteratives Verfahren, welches, beginnend von einer *Startlösung*, die aktuelle Lösung sukzessive verbessert. Dazu wird die Zielfunktion solange in **zulässige Richtungen**, welche den Zielfunktionswert (im Fall eines **Maximierungsproblems**) erhöhen, 'verschoben', bis es keine Richtung mehr gibt, in welcher eine Lösung mit größerem Zielfunktionswert vorhanden ist.

Lemma 2.19. ***(Zulässige Richtung)*** *Sei $\mathcal{P}$ ein Polyeder und $x \in \mathcal{P}$. $\vec{r}$ ist eine zulässige Richtung, falls*

$$\exists \lambda > 0 : \; x + \lambda \cdot \vec{r} \in \mathcal{P}. \tag{2.28}$$

Aufgrund der Konvexität und Linearität ist ein lokales Optimum auch global optimal.

Der Simplex-Algorithmus iteriert nicht über beliebige Lösungsvektoren, sondern beschränkt sich auf die Eckpunkte des Polyeders (vgl. S. 15). Ein Wechsel eines solchen Eckpunktes zu einem benachbarten Eckpunkt im Polyeder während des Algorithmus ist ein Basiswechsel.

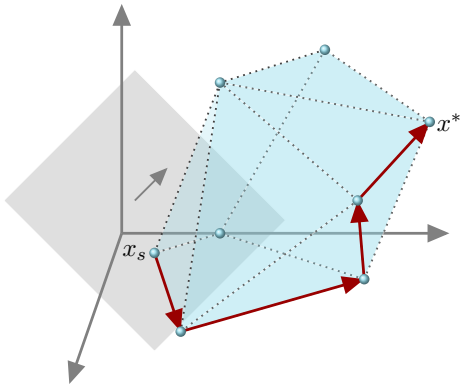


Abbildung 2.9: Der Simplex-Algorithmus

Sei also

$$\begin{aligned} \max \;\; & c^T \cdot x \\ s.t. \;\; & A \cdot x = b \\ & x_i \in \mathbb{R}^+ \end{aligned} \tag{2.29}$$

das zu lösende *primale* Programm. Liegt das Programm in der '$\leq$'-Form vor, so kann es durch Einführung von positiven *Schlupfvariablen* **s**, welche 'die Ungleichheit kompensieren', auf diese Form gebracht werden. Die modifizierten c_i werden im Laufe des Algorithmus als **reduzierte Kosten** bezeichnet. Um eine gültige Startlösung für den Algorithmus zu finden, bedarf es einer Vorlaufphase, der **PHASE I** des Simplex-Algorithmus. In dieser löst man, falls $b \geq 0$, das verwandte lineare Programm

$$\begin{aligned} \max \;\; & 1^T \cdot A \cdot x \\ s.t. \;\; & A \cdot x + z = b \\ & z \geq 0 \\ & x_i \in \mathbb{R}^+ \end{aligned} \tag{2.30}$$

mit dem Simplex-Algorithmus. z_i sind die sogenannten **künstlichen Variablen** und der Simplex-Algorithmus endet, wenn alle reduzierten Kosten *nichtpositiv* sind.

Dieses System ist nun im Gegensatz zu (2.29) direkt lösbar, da $x = \vec{0}$ und $z = b$ eine gültige Startlösung bilden. Ist $b \ngeq 0$, also $\exists k : \ b_k < 0$, so kann man diese Gleichungen mit -1 multiplizieren und erhält die gewünschte Form.

Das System (2.30) minimiert die künstlichen Variablen, da:

$$\begin{array}{llll} \max\ 1^T \cdot A \cdot x & \max\ 1^T \cdot (b - z) & 1^T \cdot b + \max\ -1^T \cdot z & \min\ z \\ s.t.\ A \cdot x + z = b \ \Rightarrow & s.t.\ A \cdot x + z = b \ \Rightarrow & s.t.\ A \cdot x + z = b \ \Rightarrow & s.t.\ A \cdot x + z = b \\ \quad z \geq 0 & \quad z \geq 0 & \quad z \geq 0 & \quad z \geq 0 \\ \quad x_i \in \mathbb{R}^+ & \quad x_i \in \mathbb{R}^+ & \quad x_i \in \mathbb{R}^+ & \quad x_i \in \mathbb{R}^+ \end{array}$$

Mit der nun gültigen Startlösung löst man dieses lineare Programm. Wird dabei eine Lösung mit $x = \tilde{x}$ und $s^* = \vec{0}$ gefunden, so hat man mit $\tilde{x}$ eine gültige Startlösung für (2.29) und kann das Programm in der **PHASE II**, ausgehend von dieser Lösung, lösen. Kann $\sum_i z_i$ hingegen nicht auf 0 minimiert werden, so ist (2.29) nicht lösbar.

Bemerkung 2.20. *Der Simplex-Algorithmus (bzw. dessen Durchführung) existiert in diversen Varianten. Für die praktische Anwendung der Methode sei hier auf [Dan66, Kap. 5] verwiesen.*

Bemerkung 2.21. *Der Simplex-Algorithmus hat im 'worst-case' exponentielle Laufzeit, beweisbar durch Anwendung auf einem Klee-Minty-Würfel [KM72]. Nichts desto trotz liefert er in der Praxis effiziente Ergebnisse. Das Problem 'Lösen eines LPs' ist allerdings nicht **NP-schwierig** [BT97, Kap. 8.4], da es in polynomieller Zeit mit der Ellipsoidmethode gelöst werden kann. Diese ist aber aus numerischen Gründen nicht praxistauglich für eine Umsetzung auf einem Rechner. Neben dem vorgestellten Verfahren auf dem primalen Problem gibt es auch eine duale Simplexmethode für das entsprechende duale Programm.*

Ganzzahlige Optimierungsprobleme Im Folgenden sind lineare Programme mit ganzzahligen Variablen, so genannte ILPs, von besonderer Bedeutung. Diese Einschränkung der Variablen macht das Problem 'Lösen eines ILPs' **NP-schwierig**. Zwei grundsätzliche Ansätze zur Lösung solcher Programme werden nun kurz vorgestellt, beide basieren auf der Lösung von LPs z. B. durch die Simplexmethode. Im Weiteren habe das gegebene Programm die Form

$$\begin{aligned} \max \;& c^T \cdot x \\ s.t. \;& A \cdot x \leq b \\ & x_i \in \mathbb{Z}^+ \end{aligned} \tag{2.31}$$

Die zugehörige **LP-Relaxierung** entsteht, wenn die Ganzzahligkeitsbedingung fallen gelassen wird, also

$$\begin{aligned} \max \;& c^T \cdot x \\ s.t. \;& A \cdot x \leq b \\ & x_i \in \mathbb{R}^+ \end{aligned} \tag{2.32}$$

Branch and Bound Bei diesem Verfahren wird zunächst die LP-Relaxierung des Programms gelöst. Ist nun mindestens eine, *als ganzzahlig geforderte*, Lösungskomponente nicht ganzzahlig, also $\exists k : x_k \notin \mathbb{Z}$, so wird das Problem an dieser Variable aufgeteilt. Man kann dieses Verfahren als binären Baum interpretieren, mit (2.32) als Wurzel. Sei $Z_1 < x_k < Z_2$ mit $Z_1, Z_2 \in \mathbb{Z}$, so wird (2.32) zu P_2 und P_3 geteilt, wobei P_2 (2.32) mit der Ungleichung $x_k \leq Z_1$ und P_3 (2.32) mit der Ungleichung $x_k \geq Z_2$ ergänzt wurde. In gleicher Weise fährt man nun in den so gewonnenen Unterproblemen fort. Hat man in diesem Verfahren an einer Stelle eine bisher höchste ganzzahlige Lösung durch die Relaxierung gefunden, so wird der zugehörige Zielfunktionswert und die Lösung selbst gespeichert und dieser Unterzweig kann als *abgearbeitet* gekennzeichnet werden. Dadurch kann man Unterprobleme, deren LP-Relaxierung eine schlechtere Lösung als diese besitzt, komplett streichen, da eine ganzzahlige Lösung in diesem Teilproblem nicht besser sein kann als eine reelle (da $\mathbb{Z} \subset \mathbb{R}$). Ist ein Unterprogramm durch die hinzugefügten Ungleichung *nicht lösbar*, so kann dieser Unterzweig auch missachtet werden. Sollte auf diese Weise keine ganzzahlige Lösung gefunden werden, so existiert keine. Ansonsten wurde der höchste Zielfunktionswert, sowie die zugehörige ganzzahlige Lösung, gespeichert.

Gomory-Cut-Verfahren Sei ein ganzzahliges Optimierungsproblem wie in (2.31) mit $A \in \mathbb{Z}^{M \times N}$, $b \in \mathbb{Z}^M$, $c \in \mathbb{Z}^N$ gegeben.
Zunächst wird wiederum die LP-Relaxierung des Programms gelöst. Ist hier eine berechnete Lösungskomponente nicht ganzzahlig (obwohl sie als solche gefordert ist), so kann für jedes $\bar{b}_k \notin \mathbb{Z}$ Ungleichung (2.33) dem System hinzugefügt werden:

$$\begin{array}{rccccc} [& \bar{b}_k & | & \bar{a}_{k,1} & \dots & \bar{a}_{k,N} \;\;] \\ \Rightarrow[& (\lfloor \bar{b}_k \rfloor - \bar{b}_k) & | & (\lfloor \bar{a}_{k,1} \rfloor - \bar{a}_{k,1}) & \dots & (\lfloor \bar{a}_{k,N} \rfloor - \bar{a}_{k,N})] \end{array} \tag{2.33}$$

Gilt $\bar{c}^* \notin \mathbb{Z}$ kann auch Ungleichung (2.34) dem System hinzugefügt werden:

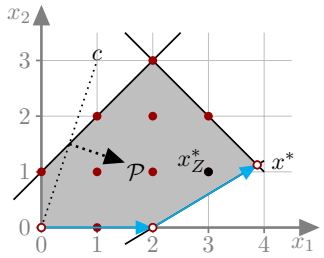
$$\begin{array}{rccccc} [& \bar{c}^* & | & \bar{c}_1 & \dots & \bar{c}_N \quad] \\ \Rightarrow[& (\lfloor \bar{c}^* \rfloor - \bar{c}^*) & | & (\lfloor \bar{c}_1 \rfloor - \bar{c}_1) & \dots & (\lfloor \bar{c}_N \rfloor - \bar{c}_N)] \end{array} \tag{2.34}$$

Diese Ungleichung wird von allen ganzzahligen Punkten im Polyeder erfüllt und ist somit eine gültige Einschränkung des Polyeders zur Lösung des ganzzahligen Programms [NW88, Prop. 3.1].

Ein **skizzierter Ablauf** einer Lösung mit dem **Gomory-Verfahren**:

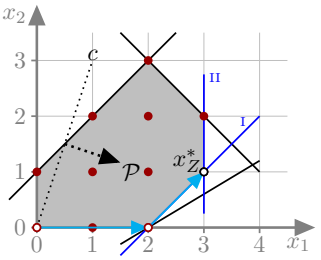
$$\begin{aligned} \max \quad & 3x_1 - x_2 \\ s.t. \quad & -x_1 + x_2 \leq 1 \\ & x_1 + x_2 \leq 5 \\ & 3x_1 - 5x_2 \leq 6 \\ & x_1, x_2 \in \mathbb{Z}^+ \end{aligned}$$

$\Rightarrow$



Lösen der LP-Relaxierung
→ **nicht-ganzzahlige** Optimallösung

$\Rightarrow$



Lösen mit zwei eingefügten Gomory-Cuts
→ **ganzzahlige** Optimallösung

Bemerkung 2.22. *Neben dem primalen Simplex-Algorithmus, welcher eine primal zulässige Lösung als Startlösung benötigt, gibt es auch den dualen Simplex-Algorithmus. Dieser benötigt eine zulässige Lösung des dualen Programms als Startkonfiguration. Diese Variante des Simplex-Algorithmus ist beispielsweise bei den Gomory-Cuts interessant. Fügt man nach der Lösung der LP-Relaxierung nämlich die durch (2.33) oder (2.34) generierten Ungleichungen dem bereits optimierten System hinzu, so ist die aktuelle Lösung nicht mehr zwingend primal zulässig, allerdings ist das System weiterhin dual zulässig. Man kann also mit der bis dato optimalen Lösung, als Startlösung für den dualen Simplex-Algorithmus, weitermachen und so schneller zu einer Optimallösung des modifizierten Systems kommen.*
Außerdem gibt es neben den genannten Verfahren noch das ***Branch-and-Cut*** *Verfahren [NW88, Kap. II.5.2], welches die beiden erwähnten kombiniert, indem nicht-ganzzahlige Lösungen zunächst durch einige Schnittebenen (Cuts) ausgeschlossen werden sollen und, falls dies nach einer festgelegten Anzahl von Schritten nicht gelingt, eine Branching-Variable gewählt werden muss. Durch die Schnittebenen können Schranken für das duale Programm gefunden werden, wodurch weniger Branch-Schritte notwendig sind. Diese Kombination ist beiden einzelnen Verfahren in der Praxis i. d. R. deutlich überlegen.*

Lexikographische Ordnung Die lexikographische Ordnung (endlicher Folgen) ist eine Relation $\prec$, die für zwei Folgen $s = s_1, s_2, \ldots$ und $t = t_1, t_2, \ldots$ wie folgt definiert ist:

Eine Folge s ist lexikographisch kleiner als eine Folge t, wenn

- $s_1 < t_1$ oder
- $s_1 = t_1 \quad \wedge \quad s_2 < t_2$ oder
- $s_i = t_i \; \forall i = 1, \ldots, k-1 \quad \wedge \quad s_k < t_k$.

Man schreibt $s \prec t$. Endliche Folgen können z. B. Vektoren im $\mathbb{R}^N$ sein, wodurch man so eine Ordnung auf Vektoren definieren kann.

Ein **lexikographisch minimaler** Vektor v^* einer Menge von Vektoren V ist ein Vektor, für den gilt:

$$v^* \prec v \quad \forall v \in V \setminus v^* \tag{2.35}$$

2.2 Technische Grundlagen

Parallelisierung und Vektorisierung In den letzten zwei Jahrzehnten hat sich die Taktfrequenz gängiger CPUs enorm gesteigert. Während der 1991 von Intel® eingeführte 'i486SX' mit gerade einmal 16 MHz verfügbar war, haben heutige Prozessoren Taktfrequenzen von bis zu 4 GHz. Derartige Taktfrequenzen erreichen allerdings eine physikalische Grenze, da mit diesen auch ein entsprechend hoher Stromverbrauch und eine hohe Abwärme einhergehen. Weil es immer aufwendiger wird, die durch höhere Leistung nahe der Fertigungsgrenze operierenden CPUs entsprechend zu kühlen und die Fertigungsgröße zu verringern bzw. die Transistorzahl zu erhöhen, ist der Einsatz von mehreren Prozessorkernen in einem PC in den letzten Jahren die gängige Methode zur Leistungssteigerung geworden.
Dieser Ansatz birgt allerdings Herausforderungen an die Softwareentwicklung, da auszuführende Programme für einen solchen Mehrkern-Prozessor ausgerichtet sein müssen. Diese Anpassung vorhandener Codes kann äußerst aufwendig und ggf. auch gar nicht möglich sein. Als Beispiel für einen nicht parallelisierbaren Code, siehe Abbildung 2.10. Da in jeder Iteration die Ergebnisse aus den beiden vorangegangenen Iterationen benötigt werden, ist es nicht möglich, zwei solche Iterationen parallel zueinander auszuführen. In vielen Codes muss die Struktur des Algorithmus verändert werden, um eine parallele Ausführung zu ermöglichen. Solch eine Anpassung des Codes darf die Ergebnisse des Programms selbstverständlich nicht verändern. Die Analyse und ggf. Anpassung von

```
fib[0]=0;
fib[1]=1;
  for(i=2; i<M; i++)
    fib[i]=fib[i-2]+fib[i-1];
```

Abbildung 2.10: Berechnung der ersten M Fibonacci-Zahlen

Codes, sowie die anschließende parallele Umsetzung ist eine Kernaufgabe aktueller Code-Beschleunigung.
Neben dem Einsatz mehrerer Prozessoren zur Nutzung von Parallelität, gibt es die Vektorrechner bzw. Vektoreinheiten. Im Gegensatz zu den grundsätzlich voneinander unabhängigen Prozessoren, welche verschiedene Operationen auf verschiedenen Daten durchführen können (**MIMD**, s. Tabelle 2.1), wird bei Vektoreinheiten dieselbe Operation auf mehreren Daten durchgeführt. Die Ausführung einer Operation auf den Daten erfolgt in einem Zeitschritt, was somit in einer Synchronität der parallelen Berechnungen resultiert. Auf Multicore-Prozessoren laufen die zugewiesenen Prozesse grundsätzlich asynchron ab, Synchronität muss, sofern nötig, explizit sichergestellt werden.

Amdahlsches Gesetz Amdahls Gesetz beschreibt den theoretisch maximal möglichen Speedup durch Parallelisierung eines Teils des zugrundeliegenden Programms. Grundlegende Annahme hierfür ist, dass nur ein Anteil ρ eines Programms parallelisierbar ist, somit wird ein Anteil $(1-\rho)$ des Programms definitiv seriell ausgeführt. Der aus dieser Annahme resultierende theoretische Speedup auf $\wp$ Prozessoren (Recheneinheiten) lässt sich durch

$$\text{Speedup}_N \;=\; \frac{1}{(1-\rho)+\frac{\rho}{\wp}+\sigma(\wp)} \tag{2.36}$$

berechnen.
Die Funktion $\sigma(\wp)$ beschreibt den, durch eine hohe Anzahl paralleler Prozessoren steigenden, Kommunikationsaufwand zwischen den Prozessoren. Allgemein ist diese Abschätzung des Speedups aber eher theoretischer Natur, da Komponenten wie der Cache außer acht gelassen werden, welche in der Praxis große Speedup-Einflüsse haben. Durch den Einfluss der Kommunikation in diesem Modell verliert die Funktion ihre *Monotonie*, womit bei einer endlichen Anzahl von $\bar{\wp}$ Prozessoren ein maximaler Speedup von $\text{Speedup}_{\bar{\wp}} = \max_{\wp}\{\text{Speedup}_{\wp}\}$ erreicht wird. Die Existenz eines solchen Maximums entspricht der realen Beobachtung.

Flynnsche Klassifikation 1966 hat Flynn eine Klassifizierung von Computersystemen bzw. Recheneinheiten basierend auf der Anzahl ihrer Daten- und Instruktionsströme [Fly72] vorgestellt, welche sich zur Einordnung von Architekturen bis heute eignet und genutzt wird. Die daraus resultierenden Abkürzungen sind in Tabelle 2.1 gelistet.

	Single Instruction	Multiple Instruction
Single Data	**SISD**	**MISD**
Multiple Data	**SIMD**	**MIMD**

Tabelle 2.1: Flynnsche Klassifikation

Jede dieser Klassen erfordert angepasste Konzepte bei der Programmierung und Anpassung von Algorithmen. Quellcodes die auf **SISD**-Basis programmiert wurden zu parallelisieren, also auf **MIMD** zu übertragen, oder zu vektorisieren, also auf **SIMD** zu übertragen, ist eine in dieser Arbeit fokussierte Konvertierung.

Beispiele für die einzelnen Klassen (vgl. [Dun90]) sind:

- **SISD**: PCs, von-Neumann-Architekturen, Harvard-Architekturen
- **MISD**: äußerst selten, Schachcomputer
- **SIMD**: Vektorrechner, SSE, MMX, 3DNow!
- **MIMD**: Supercomputer, Verteilte Systeme, Multicore-Prozessoren

Vektorrechner und SSE Während Vektorrechner bis in die 1990er Jahre hauptsächlich als HPC(*High Performance Computing*)-Computer (wie z. B. die Cray™ Supercomputer) vertreten waren und zur Lösung extrem rechenlastiger Algorithmen in Unternehmen wie der NASA oder der U.S. Air Force eingesetzt wurden, hielten sie 1997 mit dem Intel® Registersatz MMX™ und 1999 mit SSE (Streaming SIMD Extension) im Pentium®-III-(Katmai)-Prozessor als **SIMD**-Einheiten Einzug in den Bereich der *Personal Computer*. Beide Registersätze wurden vor allem zur Beschleunigung von Multimediaanwendungen entwickelt. **MMX** nutzt die acht vorhandenen 64-Bit großen Register des FPU (MM0 - MM7) der CPU, wodurch eine gleichzeitige Nutzung von MMX und *float-Operationen* nicht möglich ist. Die 64-Bit Register können in ein 64-Bit, zwei 32-Bit, vier 16-Bit oder acht 8-Bit Integer unterteilt werden, wobei die Operationen simultan auf diesen ausgeführt werden. Die unterstützten Operationen umfassen mathematische Operationen, Vergleiche, Datenbewegungen, Boolesche Logik und einige andere.

Für **SSE** wurden acht autarke 128-Bit Register (XMM0 - XMM7) in die CPU integriert. Diese führen (für SSE in der ersten Version) die Operationen auf vier 32-Bit *single-precision (float)* Werten aus, in SSE2 wurde dieses Prinzip bereits auf *double-precision* erweitert sowie die Registeraufteilungen von MMX und dessen Operationen übernommen. Der Instruktionssatz von SSE wurde auch im Folgenden in den Versionen SSE, SSE2, SSE3, SSE4.1, SSE4.2, SSE4a permanent erweitert und umfasst mittlerweile Instruktionen aus den Bereichen Arithmetik, Cache (Prefetch), Logik, Konvertierung, Datenbewegung, Shuffle, Vergleiche, bis hin zu komplexen Operationen wie einem vektorisierten Skalarprodukt zweier Vektoren.
Die Berechnung einer zweioperandigen Instruktion ist in Abbildung 2.11 dargestellt.

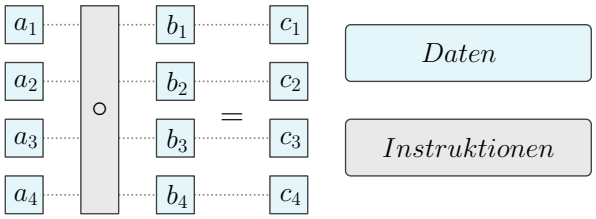


Abbildung 2.11: Zweioperandige SSE-Instruktion auf 4 Elementen (float)

Neben der SSE-Entwicklung von Intel hat AMD eine floating-point Variante von MMX in den FPU Registern entwickelt, **3DNow!™**. Allerdings integriert AMD seit 2001 im Athlon-XP™ SSE in seine CPUs und unterstützt seit 2011 kein 3DNow!™ mehr.
SSE ist eine Erweiterung der x86-Prozessor-Architektur.

Für Details zur Programmierung und Nutzung von MMX und SSE, siehe [Int99]. Eine ausführliche Erläuterung findet sich auch unter [1].

Cache-Hierarchie Damit der Prozessor mehrfach benötigte Daten nicht jedes Mal aus langsamen Speichern, wie dem RAM oder gar der Festplatte, laden muss, existieren CPU-nahe Speicherelemente, welche hierarchisch Daten zwischenladen und somit einen wiederholten schnellen Zugriff auf diese ermöglichen. Außerdem werden Daten, welche 'erwartungsgemäß' in naher Zukunft benötigt werden in diese Speicherbereiche vorgeladen (Prefetching [LCF+03]). Diese Mechanismen existieren als Hardware- und Softwaremechanismen. Da es nicht praktikabel ist, sehr schnelle (sowie prozessornahe) und gleichzeitig große Cache-Speicher zu realisieren, werden mehrere Caches hierarchisch angeordnet. Ein schneller L1-Cache, der relativ klein ist, wird bei der Suche nach Daten zuerst betrachtet. Sind diese hier nicht vorhanden, wird der langsamere, aber dafür größere

[1] http://softpixel.com/~cwright/programming/simd/ (21.06.2011)

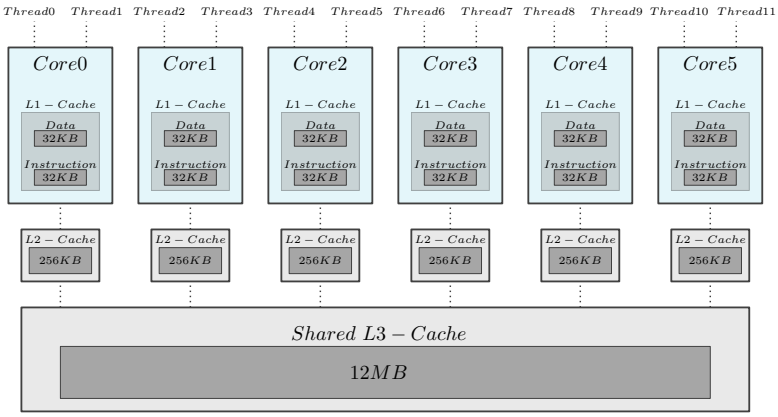


Abbildung 2.12: Cache-Hierarchie des Intel® Xeon® X5650

L2-Cache durchsucht, danach (falls vorhanden) der L3-Cache. Falls die Daten in keinem dieser Caches gefunden wurden, werden langsamere Speicher wie der Haupt- oder Hintergrundspeicher durchsucht. Es gibt verschiedene Verdrängungsstrategien, die bestimmen, welche Daten zuerst wieder aus dem Cache entfernt werden, um Platz für neue zu schaffen. Beispiele hierfür sind **FIFO** (First-In-First-Out), **LFU** (Least-Frequently-Used), **LRU** (Least-Recently-Used), **MRU** (Most-Recently-Used), allerdings gibt es auch andere Ansätze, wie eine zufällige Verdrängung der Daten (**Random**). Werden Daten in den Speicher geschrieben, kann dies entweder direkt in alle Cache-Level und den Hauptspeicher geschehen (*write-through*), oder sie werden zunächst lediglich auf einer Cache-Ebene abgelegt (*write-back*). Diese verschiedenen Strategien bieten für verschiedene Probleme spezifische Vor- und Nachteile, sodass eine verallgemeinerte Wertung nicht möglich ist.
Da viele Datenstrukturen linear im Speicher liegen und bei Zugriff auf (beispielsweise) Array-Elemente der Zugriff auf ein benachbartes Element in naher Zukunft wahrscheinlich ist (so die Annahme), werden nicht einzelne Datenelemente in den Cache geladen, sondern immer **Cache-Lines** einer CPU abhängigen Länge (z. B. L1/L2/L3-Cache-Line Größe des Xeon® X5650: 64Byte). Außerdem gibt es Hardware-spezifische Gründe für das Laden von Cache-Lines anstelle einzelner Werte.
Die Cache-Hierarchie eines Intel® Xeon® Prozessors ist in Abbildung 2.12 dargestellt.
Beispielhafte Attribute für die einzelnen Speicherebenen sind

- 1-3 Takte (das entspricht unter **1 ns** bei 2,67 GHz Taktung) und 8-64 KByte für *L1-Cache* (SRAM Speicher)
- 4-15 Takte (ca. **5 ns** bei 2,67 GHz Taktung) und 256 KByte-16 MByte für *L2-Cache* (SRAM Speicher),

- 2-64 Mbyte für *L3-Cache*,
- ca. **40 ns** und bis zu einigen GByte für den *Hauptspeicher* (DRAM) und
- ca. **5-10 ms** und bis zu einigen TByte für den *Hintergrundspeicher*

Prozessor-Cache Hits und Misses Wird bei der Suche eines bestimmten Datenelements dieses in einem beliebigen Level des Caches gefunden, wodurch kein Zugriff auf Haupt- oder Hintergrundspeicher nötig ist, so spricht man von einem 'Cache-Hit', ist dies nicht der Fall so von einem 'Cache-Miss'. Entsprechend des Verhältnisses zwischen Cache-Hits und Cache-Misses kann eine 'Cache-Hit-Rate' (CHR), sowie eine 'Cache-Miss-Rate' (CMR) ermittelt werden, wobei gilt $CHR = (1 - CMR)$. Um die Zugriffszeiten zu verringern, ist eine möglichst hohe Cache-Hit-Rate ein lohnender Ansatz. Dies gilt vor allem bei speichergebundenen Algorithmen (*memory-bound*).
Ein Hauptaspekt des in dieser Arbeit entwickelten Frameworks ist die Anpassung von Codes an die gegebene Hardware und somit eine Unterstützung des internen Prefetchings, um ein *flüssiges* Laden benötigter Daten in den Cache zu ermöglichen und die Cache-Hit-Rate zu erhöhen. Das Potenzial einer Erhöhung der Cache-Hit-Rate ist bereits anhand der Zugriffszeiten der einzelnen Speicherebenen erkennbar.
Zum Messen der Cache-Hit-Rate existieren diverse Tools wie *PAPI* [BDG+00] oder *Cachegrind* [NS03]. PAPI liest dazu die Hardware-Counter der CPU aus. Cachegrind simuliert die gegebene Hardwareumgebung und lässt das Programm darauf basierend laufen, wodurch sich die Laufzeit deutlich erhöht. Für eine Erklärung der Arbeitsweise von PAPI siehe [TJYD09], für Cachegrind siehe [Net04, Kap. 3].

Compiler und Compiler-Optimierungen Zur Übersetzung von Source-Codes in einen semantisch äquivalenten Maschinencode durchlaufen Compiler (für *Compilersprachen*) mehrere Schritte der Übersetzung. Ein typischer solcher Vorgang soll im Folgenden dargestellt werden (vgl. [ASLU08]).

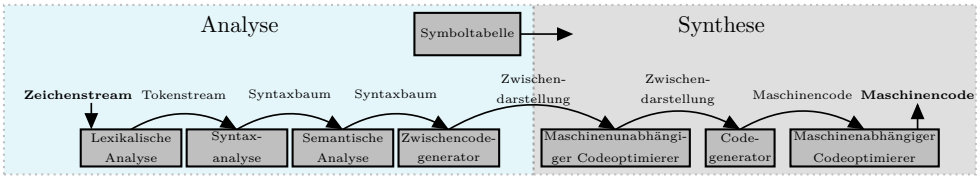


Abbildung 2.13: Phasen eines Compilers [2]

[2] vgl. [ASLU08, Abb. 1.6]

Zu Beginn der *Analyse* wird der Quellcode zur weiteren Verarbeitung vom Compiler eingelesen. Anschließend werden Syntax und Semantik auf Grundlage des aus dem Programmcode erstellten Syntaxbaums, einer abstrakten Datenstruktur zur Formalisierung des Quellcodes, gescannt und auf Korrektheit überprüft, wobei bei Fehlern oder Warnungen entsprechende Meldungen für den Nutzer ausgegeben werden können. Informationen zu jedem Quellcodesymbol werden in der Symboltabelle gesammelt. Sind somit alle Informationen zum vorliegenden Programm gesammelt und wurde dieses als 'korrekt' befunden, so werden eine daraus generierte Zwischendarstellung sowie die Symboltabelle vom **Front-End** an die *Synthese*, das **Back-End**, übergeben.
Das **Back-End** kann nun noch maschinenunabhängige Codeoptimierungen vor die endgültige Codeerzeugung schalten. Dies sind z.B. Code-Reduzierungen und Transformationen, die eine bessere Performance im Endprogramm hervorbringen. Der durch den Codegenerator erzeugte Maschinencode kann im Anschluss noch von einem maschinenabhängigen Codeoptimierer, durch maschinenspezifische Optimierungen, verbessert werden. Der resultierende Endcode kann direkt auf der Maschine ausgeführt werden.
Ein **Source-To-Source**-Compiler ist von der grundlegenden Funktionsweise identisch, übersetzt aber nicht in Maschinencode, sondern wiederum in Sourcecode. Dies kann zum einen sinnvoll sein, um von einer Sprachversion in eine andere zu übersetzen, wie beim 'Python 2to3' [hta], welcher einen Python2-Code in Python3-Code konvertiert, oder zur Übersetzung von Quellcode einer Sprache in eine andere, sogar z. B. von einer domänenspezifischen Sprache in eine allgemeine. Eines der verbreitetsten Tools für solch eine Transformation ist DMS® [BPM04]. Zum anderen sind Tools zur Optimierung des Quellcodes, z. B. zur Parallelisierung mittels OpenMP®, von dieser Art, indem sie den Quellcode an geeigneten und korrekten Stellen mit entsprechenden Pragmas versehen.
Solch ein optimierender Source-To-Source-Compiler kann als Vorstufe einer traditionellen Kompilierung verstanden werden. Dies ermöglicht eine Anpassung des Quellcodes an gegebene Architekturen ohne Spezifizierung des nachzuschaltenden Compilers.
Das in dieser Arbeit weiterentwickelte Framework **PluTo** [Bona; BHRS08] ist ein solcher Source-To-Source-Compiler, welcher hauptsächlich zur Optimierung von Datenlokalität sowie zur Parallelisierung von Quellcodes entwickelt wird.
Compiler für **C** und **C++** in Maschinensprache sind beispielsweise das kostenlose Open Source Projekt '*GCC, the GNU Compiler Collection*' [htb] oder der Intel® Compiler **ICC** [huct], der für nichtkommerzielle Produkte kostenlos ist und durch teilweise bessere Optimierungen oft noch schnelleren Code erzeugt als der *GCC*. Einige Compiler werden auch direkt mit entsprechenden Entwicklungsumgebungen zur komfortablen Programmierung ausgeliefert, wie z. B. beim **Visual C++** von Microsoft®.

3 Das Polyedermodell

3.1 Abhängigkeiten

Bei der Parallelisierung oder Vektorisierung, bzw. bei der allgemeinen Transformation von gegebenen Quellcodes, spielen Abhängigkeiten von Daten zwischen Instruktionen eine elementare Rolle. Die Analyse und Einhaltung dieser Abhängigkeiten, auch im transformierten Code, ist unerlässlich um die semantische Korrektheit des Codes zu garantieren. Die Abhängigkeiten bilden gewissermaßen die Restriktionen der Codeoptimierung, deren Einhaltung, kombiniert mit der Optimierung einer gegebenen Funktion, einen für diese Aufgabe substanziellen Bestandteil bildet.

3.1.1 Skalare Abhängigkeiten

Datenabhängigkeiten sind vorhanden, wenn auf dieselben Daten in mehreren Statements eines Codeabschnittes zugegriffen wird. Ein **Statement** ist eine Instruktion, welche von beliebigen Variablen des zugrundeliegenden Codes abhängen kann. Statements werden im Folgenden mit S_i bezeichnet und die Menge aller Statements eines Codes mit $\mathcal{S}$.
Betrachte wie in [Muc97, Kap. 9] zwei Statements S_1 und S_2 (wobei in der Ausführungsreihenfolge S_1 vor S_2 steht), dann ist eine Datenabhängigkeit eine

- **Echte-Abhängigkeit** (*true dependence* oder *flow dependence*)[read-after-write], wenn auf Daten, die in S_2 gelesen werden, in S_1 geschrieben wird. Dies wird durch die binäre Relation $S_1 \, \delta^f \, S_2$ formalisiert.
- **Gegen-Abhängigkeit** (*antidependence*)[write-after-read], wenn auf Daten, die in S_1 gelesen werden, in S_2 geschrieben wird. Dies wird durch die binäre Relation $S_1 \, \delta^a \, S_2$ formalisiert.

- **Ausgabe-Abhängigkeit** (*output dependence*)[write-after-write], wenn auf dieselben Daten in S_1 und S_2 geschrieben wird. Dies wird durch die binäre Relation $S_1 \, \delta^o \, S_2$ formalisiert.
- **Eingabe-Abhängigkeit** (*input dependence*)[read-after-read], wenn dieselben Daten in S_1 und S_2 gelesen werden. Dies wird durch die binäre Relation $S_1 \, \delta^i \, S_2$ formalisiert.
  Eingabe-Abhängigkeiten sind ***für die Ausführungsreihenfolge irrelevant****, da zwei Lesezugriffe auf diesselben Daten jene nicht beeinflussen.*

Allgemein werden diese Abhängigkeiten mit $S_1 \, \delta \, S_2$ beschrieben. Für die Menge aller Abhängigkeiten zwischen Statements aus $\mathcal{S}$ kann der sogenannte **Abhängigkeitsgraph** (**DDG** - *data dependence graph*) erstellt werden. Dieser ist ein gerichteter, kreisfreier Graph (**DAG**), in welchem jedes Statement S_i durch einen Knoten vertreten ist und für jede Abhängigkeit $S_i \, \delta \, S_j$ eine Kante $e^{S_i \to S_j}$ existiert.

Beispiel 3.1.

```
1 A = B + C;
2 B = D * A;
3 A = D;
4 C = C + 1;
```

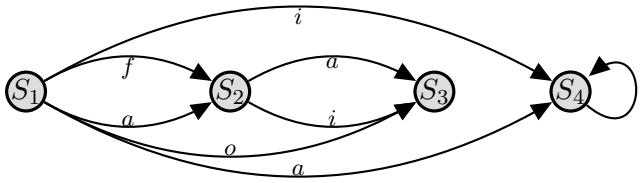


Abbildung 3.1: Quellcodeausschnitt mit zugehörigem Abhängigkeitsgraph (**DDG**)

3.1.2 Dynamische Abhängigkeiten

Das Prinzip der Abhängigkeiten lässt sich auch auf nicht-skalare Variablen, wie z. B. Arrays, übertragen. Abhängigkeiten zwischen zwei Zugriffen auf dasselbe Array können wie bei den *skalaren Abhängigkeiten* klassifiziert werden. Der grundsätzliche Unterschied ist dabei aber, dass das Zugriffsmuster auf das Array betrachtet werden muss, um festzustellen, ob und wie auf dieselben Array-Elemente zugegriffen wird. Hierzu wird jeder solche Zugriff durch eine Array-Zugriffsfunktion (*Array-Access-Function*) repräsentiert, welche die Zugriffsstrukturen der Dimensionen eines Arrays, abhängig von den umgebenden Schleifenindizes, formalisiert.
Die formale Repräsentation dieser Funktion ist in Abschnitt 3.5 zu finden.

3.2 Verschachtelte Schleifen

Der Fokus dieser Arbeit liegt auf der Betrachtung verschachtelter Schleifen, da ein Großteil der Rechenzeit wissenschaftlicher Quellcodes in solchen verbracht wird. Für die folgenden Betrachtungen sind ***for***-Schleifen die Grundlage, allerdings existieren auch Modelle für ***while***- oder ähnliche Schleifenkonstrukte.
Sind mehrere Schleifen ineinander **verschachtelt**, so unterscheidet man zwischen **perfekt verschachtelten** Schleifen (*perfectly nested loops*) und **nicht-perfekt verschachtelten** Schleifen (*not-perfectly nested loops*).

```
for(i_1 = ...; ...; i_1++)
{
  for(i_2 = ...; ...; i_2++)
  {
    ⋮
    for(i_D = ...; ...; i_D++)
    {
      S_1(i_2, i_3, i_8);
      ⋮
      S_T(i_1, i_D);
    }
  }
}
```

Abbildung 3.2: Perfekt verschachtelte Schleife

```
for(i_1 = ...; ...; i_1++)
{
  S_1(i_1);
  for(i_2 = ...; ...; i_2++)
  {
    S_2(i_1, i_2);
    S_3(i_2);
  }
  ⋮
  for(i_D̃ = ...; ...; i_D̃++)
  {
    S_9(i_2, i_3, i_8);
    ⋮
    S_T(i_1, i_D̃);
  }
}
```

Abbildung 3.3: Nicht-perfekt verschachtelte Schleife

Zu jedem Statement steht in Klammern, von welchen Schleifenindizes es abhängt.

Bei perfekt verschachtelten Schleifen sind *alle* Statements dieser Verschachtelung in der innersten Schleife. Für eine D-fach perfekt verschachtelte Schleife (somit eine **Schleife der Tiefe D**) mit T Statements liegt also eine Struktur wie in Abbildung 3.2 vor. Die **Tiefe eines Statements** gibt an, auf welcher Verschachtelungsstufe es steht bzw. in wie viele Schleifen es verschachtelt ist. Im perfekt verschachtelten Fall befinden sich alle Statements auf Tiefe D. Im nicht-perfekt verschachtelten Fall in Abbildung 3.3 ist S_1 auf Tiefe 1, alle anderen angegebenen Statements ($S_2 - S_T$) sind auf Tiefe 2.

Bemerkung 3.2. *Diverse Modelle betrachten ausschließlich den perfekt verschachtelten Fall zur Transformation, da dieser eine einfachere Analyse ermöglicht. Es ist möglich, eine nicht-perfekt verschachtelte Schleife in eine perfekt verschachtelte durch Einfügen von* ***if****-Bedingungen umzuwandeln [Xue97, Abu-Sufah Transformation]. Der umgewandelte Code ist allerdings i. d. R. deutlich weniger performant. In den Modellen, die in dieser Arbeit verwendet werden, wird auch* ***nativ*** *der nicht-perfekt verschachtelte Fall analysiert und transformiert.*

3.3 Indexraum (Domäne)

Der **Indexraum** (oder die **Domäne**) $\mathcal{D}_S$ eines Statements S ist die Menge aller Indexvektoren, die von den dieses Statement umgebenden Schleifen durchlaufen werden. Es gilt $\mathcal{D}^S \subset \mathbb{Z}^N$ für ein Statement, das von N Schleifenindizes abhängt. Hier wird dies i. d. R. ein $\mathbb{Z}$-Polyeder sein, sodass der Indexraum durch ein Ungleichungssystem $\{x \in \mathbb{Z}^N \mid D \cdot x \leq d\}$ mit $D \in \mathbb{Z}^{M \times N}$ und $d \in \mathbb{Z}^M$ beschrieben werden kann.
Dies ist nicht möglich, wenn 'Löcher' im Indexraum vorhanden sind, z. B. bei $i = i + 2$ statt $i{+}{+}$ im **for**-Schleifenkopf oder *nach* einer nicht-unimodularen Transformation eines $\mathbb{Z}$-Polyeders (Abschnitt 3.7.3, Abbildung 3.11).

3.4 Gleichförmige und nicht-gleichförmige Abhängigkeiten

Der **Abhängigkeitsvektor** (*dependence vector*) zweier voneinander abhängigen Array-Zugriffe ${\mathcal{Z}_\mathcal{I}}^k = Z_\mathcal{I}^k \cdot \mathcal{I} + z^k$ und ${\mathcal{Z}_\mathcal{I}}^l = Z_\mathcal{I}^l \cdot \mathcal{I} + z^l$ (o. B. d. A. seien die beiden Indexvektoren gleich, ansonsten kann die Vereinigung der beiden Mengen von Indizes als Indexvektor angenommen werden), ist wie folgt definiert.

$$d_{k,l} \;=\; \left(Z_\mathcal{I}^l \cdot \mathcal{I} + z^l\right) \;-\; \left(Z_\mathcal{I}^k \cdot \mathcal{I} + z^k\right) \;=\; \left(Z_\mathcal{I}^l \;-\; Z_\mathcal{I}^k\right) \cdot \mathcal{I} + \left(z^l \;-\; z^k\right) \tag{3.1}$$

Eine Abhängigkeit zwischen zwei Zugriffen auf ein Array heißt **gleichförmige Abhängigkeit** (*uniform dependence*), wenn die Abhängigkeitsstruktur in jedem Iterationspunkt identisch, also der Abhängigkeitsvektor konstant, ist. Man nennt den Abhängigkeitsvektor in diesem Fall auch **Distanzvektor** (*distance vector*). Dies heißt formell, dass $Z_\mathcal{I}^k \equiv Z_\mathcal{I}^l$ gilt. Gilt dies nicht, so heißt die Abhängigkeit **nicht-gleichförmige Abhängigkeit** (*non-uniform dependence*).

Bemerkung 3.3. *Im Gegensatz zu den skalaren Abhängigkeiten, muss bei dynamischen Abhängigkeiten der Kontext der umgebenden Schleifen betrachtet werden. Die Abhängigkeitsanalyse beschreibt in diesem Fall, welche* ***Iterationen*** *(aufgrund ihrer adressierten Daten) voneinander abhängig sind, um beispielsweise korrekte Umordnungen dieser zu generieren. In der Visualisierung der Abhängigkeiten wird für eine Abhängigkeit zwischen zwei Iterationen $I^k \delta I^l$ ein Pfeil von I^k nach I^l gezeichnet.*

Die Indexräume, zusammen mit den Abhängigkeitsvektoren für eine Kante $e \in DDG$, bilden das sogenannte **Abhängigkeitspolyeder** (*dependence polyhedron*) $\mathcal{P}_e$ [DV96, Kap. 3].

Abhängigkeitspolyeder für gleichförmige und nicht-gleichförmige Abhängigkeiten sind in Abbildung 3.4 und 3.5 dargestellt.

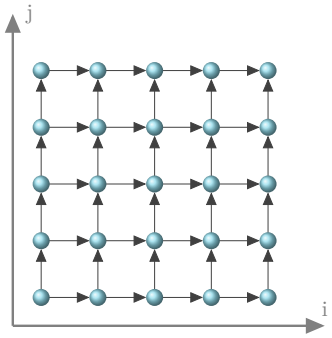


Abbildung 3.4: Gleichförmige Abhängigkeiten in einem 2-D Indexraum

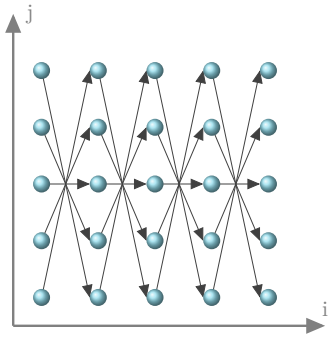


Abbildung 3.5: Nicht-gleichförmige Abhängigkeiten in einem 2-D Indexraum

Bemerkung 3.4. *Skalare Abhängigkeiten können als null-dimensionale dynamische Abhängigkeiten identifiziert werden (durch welche in jeder Iteration also auf dasselbe Element zugegriffen wird) und sind somit im Modell der dynamischen Abhängigkeiten enthalten.*

3.5 Zusammenfassung der Begriffe

Der **Indexvektor** $\mathcal{I}$ enthält die für den betrachteten Kontext (Statement, Array-Zugriff, Schleifenrumpf, etc.) relevanten Schleifen-Indizes.

Beispiel 3.5.

$A[i+k][k] \quad \Rightarrow \quad \mathcal{I} = \begin{pmatrix} i \\ k \end{pmatrix}$

```
for(i=0;i<M;i++)
  for(j=i;j<N;j++)
    for(k=0;k<K;k++)
```

$\Rightarrow \quad \mathcal{I} = \begin{pmatrix} i \\ j \\ k \end{pmatrix}$

Die **Array-Zugriffsfunktion** $\mathcal{Z}_\mathcal{I}$ eines M-dimensionalen Zugriffs $A[\bar{z}_1 \cdot \mathcal{I} + z_1] \dots [\bar{z}_M \cdot \mathcal{I} + z_M]$ mit N-dimensionalem Indexvektor wird repräsentiert durch seine Zugriffsmatrix $Z_\mathcal{I}$, seinen Indexvektor $\mathcal{I}$ sowie den konstanten Vektor des Zugriffs z, wobei $\bar{z}_i$ der i-ten Zeile der Zugriffsmatrix Z entspricht.

$$Z_\mathcal{I} \cdot \mathcal{I} + z = \begin{pmatrix} \bar{z}_1 \\ \vdots \\ \bar{z}_N \end{pmatrix} \cdot \mathcal{I} + \begin{pmatrix} z_1 \\ \vdots \\ z_M \end{pmatrix} = \begin{pmatrix} z_{1,1} & \dots & z_{1,N} \\ \vdots & \ddots & \vdots \\ z_{M,1} & \dots & z_{M,N} \end{pmatrix} \cdot \begin{pmatrix} i_1 \\ \vdots \\ i_N \end{pmatrix} + \begin{pmatrix} z_1 \\ \vdots \\ z_M \end{pmatrix} \tag{3.2}$$

Beispiel 3.6.

$$A[3*i+j+7][6*i+j-2*k] \Rightarrow \begin{pmatrix} 3 & 1 & 0 \\ 6 & 1 & -2 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \\ k \end{pmatrix} + \begin{pmatrix} 7 \\ 0 \end{pmatrix}$$

Ein Zugriff wird somit durch eine affine Abbildung beschrieben.

Der **Abhängigkeitsvektor** zweier voneinander abhängigen Array-Zugriffe ist wie folgt definiert.

$$d_{k,l} = \left(Z_\mathcal{I}^l - Z_\mathcal{I}^k\right) \cdot \mathcal{I} + \left(z^l - z^k\right) \tag{3.3}$$

Ist dieser konstant, so nennt man ihn auch **Distanzvektor**. Ein Abhängigkeitsvektor ist genau dann ein Distanzvektor, wenn die **Abhängigkeit gleichförmig** ist.

Der **Indexraum** $\mathcal{D}^S$ eines Statements S ist die Menge aller adressierten Indexvektoren der umgebenden Schleifen.

Das **Abhängigkeitspolyeder** $\mathcal{P}_e$ beinhaltet die Abhängigkeiten $\vec{i}_k \delta \vec{j}_l$ durch $e^{S_k \to S_l}$ und kann als gerichteter Graph (V, E) mit $V = \mathcal{D}^{S_l} \bigcup \mathcal{D}^{S_k}$ und $E = \{(\vec{i}_k, \vec{j}_l) \mid \vec{i}_k \in \mathcal{D}^{S_k}, \vec{j}_l \in \mathcal{D}^{S_l} \text{ mit } \vec{i}_k \delta \vec{j}_l\}$ dargestellt werden (bei $S_k = S_l$ entsprechend in *einem* Indexraum).

Beispiel 3.7.

```
for(i=1 ; i<N+1 ; i++)
  for(j=1 ; j<M+1 ; j++)
    A[i][j] = A[i-1][j] + A[i][j-1];
```

In diesem Code sind zwei relevante Abhängigkeiten vorhanden, nämlich

- e_1: $A[i][j] \leftrightarrow A[i-1][j]$ (blau)
- e_2: $A[i][j] \leftrightarrow A[i][j-1]$ (rot)

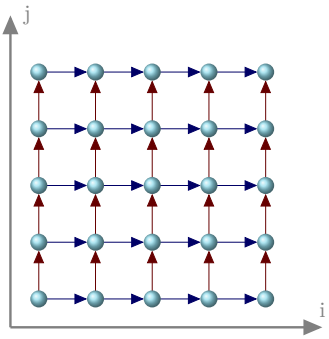


Abbildung 3.6: Abhängigkeitspolyeder für beide Abhängigkeiten

$\mathcal{P}_{e_1} = \{i' = i-1, j' = j, 1 \le i \le N, 0 \le j \le M\}$
$\mathcal{P}_{e_2} = \{i' = i, j' = j-1, 0 \le i \le N, 1 \le j \le M\}$

Der *Indexvektor* aus Beispiel 3.7: $\mathcal{I} = \begin{pmatrix} i \\ j \end{pmatrix}$

Die *Array-Zugriffsfunktionen* aus Beispiel 3.7:

$$1.)\ A[i][j] \to \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix}$$

$$2.)\ A[i-1][j] \to \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ 0 \end{pmatrix}$$

$$3.)\ A[i][j-1] \to \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ -1 \end{pmatrix}$$

Die *Distanzvektoren* aus Beispiel 3.7:

$$d_{2,1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix} - \left(\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ 0 \end{pmatrix} \right) = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \text{(blau)}$$

$$d_{3,1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix} - \left(\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ -1 \end{pmatrix} \right) = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \quad \text{(rot)}$$

Der *Indexraum* aus Beispiel 3.7: $\left\{ \begin{pmatrix} i \\ j \end{pmatrix} \in \mathbb{Z}^2 \mid 1 \le i \le N \wedge 1 \le j \le M \right\}$

Weiteres zu den beschriebenen Begriffen findet sich in [Ban97; Ban94; Xue00; Muc97].

3.6 Abhängigkeitstest

Um festzustellen, ob zwischen zwei Zugriffen $\mathcal{Z}^1$ mit (Z^1, z^1) und $\mathcal{Z}^2$ mit (Z^2, z^2) eine dynamische Abhängigkeit im Abhängigkeitspolyeder vorhanden ist, gilt es zu prüfen, ob es nicht-triviale Indexkombinationen $\vec{i}_1 \in \mathcal{D}^{S_1}$ und $\vec{j}_2 \in \mathcal{D}^{S_2}$ gibt, sodass

$$Z^1_{\vec{i}_1} \cdot \vec{i}_1 + z^1 \;=\; Z^2_{\vec{i}_2} \cdot \vec{j}_2 + z^2 \quad . \tag{3.4}$$

Existiert also ein Abhängigkeitsvektor, welcher *nicht der Nullvektor* ist, so ist eine tatsächliche Abhängigkeit vorhanden, sofern die entsprechenden Indexvektoren in den Indexräumen liegen.

Bemerkung 3.8. *Sind keine dynamischen Abhängigkeiten in einer verschachtelten Schleife vorhanden, so lassen sich die Schleifen beliebig parallelisieren und umordnen (transformieren). Auch wenn Abhängigkeiten vorliegen, so geben die Abhängigkeitsvektoren ggf. bereits Auskunft darüber, ob bestimmte Schleifen ohne Transformationen parallelisierbar sind (siehe Abschnitt 3.7.5).*

Sind die Indexräume $\mathbb{Z}$-Polyeder, so folgt:

Zwei Zugriffe (Z^1, z^1) und (Z^2, z^2) auf ein Array sind voneinander **abhängig**, falls Indexvektoren $\vec{i}_1$ und $\vec{j}_2$ (mit $\vec{i}_1 \neq \vec{j}_2$ bei $S_1 = S_2$) existieren, sodass

$$\vec{i}_1 \in \mathcal{D}^{S_1} \Rightarrow \qquad D^1_{\vec{i}_1} \cdot \vec{i}_1 \;\leq\; d^1 \tag{3.5}$$

$$\vec{j}_2 \in \mathcal{D}^{S_2} \Rightarrow \qquad D^2_{\vec{j}_2} \cdot \vec{j}_2 \;\leq\; d^2 \tag{3.6}$$

$$Z^1_{\vec{i}_1} \cdot \vec{i}_1 + z^1 \;=\; Z^2_{\vec{j}_2} \cdot \vec{j}_2 + z^2 \tag{3.7}$$

$$\vec{i}_1, \vec{j}_2 \text{ ganzzahlig}$$

mit $D^1_{\vec{i}_1} \cdot \vec{i}_1 \leq d^1$ und $D^2_{\vec{j}_2} \cdot \vec{j}_2 \leq d^2$ sind die $\mathbb{Z}$-Polyederbeschreibungen der Indexräume und $Z^1_{\vec{i}_1} \cdot \vec{i}_1 + z^1$ sowie $Z^2_{\vec{j}_2} \cdot \vec{j}_2 + z^2$ die Array-Zugriffsfunktionen.

Als Ungleichungssystem in Standardform mit $\mathcal{I} = \begin{pmatrix} \vec{i}_1 \\ \vec{i}_2 \end{pmatrix}$ ergibt sich:

$$\begin{pmatrix} D^1_{\vec{i}_1} & \mathbb{O} \\ \mathbb{O} & D^2_{\vec{j}_2} \\ Z^1_{i_1} & -Z^2_{i_2} \\ -Z^1_{i_1} & Z^2_{i_2} \end{pmatrix} \cdot \mathcal{I} \;\leq\; \begin{pmatrix} d^1 \\ d^2 \\ z^2 - z^1 \\ -z^2 + z^1 \end{pmatrix} \quad mit\ \mathcal{I} \in \mathbb{Z}^{|\vec{i}_1|+|\vec{i}_2|} \tag{3.8}$$

Somit kann die Identifikation dynamischer, affiner Abhängigkeiten in, durch $\mathbb{Z}$-Polyeder repräsentierten, verschachtelten Schleifen auf das Lösen eines linearen Ungleichungssystems zurückgeführt werden.

Eine direkte Lösung durch z. B. Fourier-Motzkin-Elimination oder das Lösen des IPs mit beliebiger Zielfunktion (z. B. der *Nullfunktion*) ist möglich, aber rechenintensiv. Im Folgenden werden einige Methoden aus [ASLU08, Kap. 11.6] beschrieben, welche die Lösung des gesamten Systems durch vorgeschaltete Tests und Heuristiken möglichst zu vermeiden versuchen.

3.6.1 Heuristik-basierter Ansatz

Ein Ansatz zum Ausschluss dynamischer Abhängigkeiten stellt der **ggT-Test** dar. Dieser stellt nicht die Frage nach der Lösbarkeit des gesamten Systems in (3.8), sondern beschränkt sich auf die Abhängigkeiten durch die Zugriffsfunktionen unter Vernachlässigung der Indexräume. Ist $Z^1_{\vec{i}_1} \cdot \vec{i}_1 + z^1 = Z^2_{\vec{j}_2} \cdot \vec{j}_2 + z^2$ *nicht* ganzzahlig lösbar, so kann das gesamte System (3.8), ungeachtet der anderen Ungleichungen, nicht ganzzahlig lösbar sein. Diese Bedingung (bzw. $Z^1_{\vec{j}_2} \cdot \vec{j}_2 - Z^1_{\vec{i}_1} \cdot \vec{i}_1 = z^1 - z^2$) ist eine Menge von M **linearen diophantischen Gleichungen**, deren Lösbarkeit durch Anwendung des **Euklidischen Algorithmus** [Bro71] geklärt werden kann, denn:

Lemma 3.9. *Eine **lineare diophantische Gleichung** $a_1 * x_1 + \ldots + a_N * x_N = c$ ist genau dann ganzzahlig lösbar (also mit $x_i \in \mathbb{Z}$), wenn c durch $ggT(a_1, \ldots, a_N)$ teilbar ist.*

Ist (3.7), durch diesen Test beantwortet, erfüllbar, so können durch das Lösen dieses Gleichungssystems mögliche Lösungen (und dadurch auch ganzzahlige solche) gefunden werden. Der Kontext der gegebenen Indexräume muss in diesem Falle mit in die Lösung einbezogen werden, um festzustellen, ob auch Lösungen innerhalb dieser existieren. Eine Möglichkeit dafür stellt der **Test auf unabhängige Variablen** dar. Bei diesem werden die diophantischen Gleichungen der Zugriffe mit den Schleifengrenzen (und somit den Indexräumen) kombiniert. Dazu werden beispielsweise für die Gleichung eines Zugriffes $i_k = c_1 * j_1 + \ldots + c_L * j_L + c_{L+1}$ und den *for*-Schleifenkopf $for(i_k = 0; i_k < K+1; i_k++)$ mit konstantem K die Ungleichungen $0 \leq c_1 * j_1 + \ldots + c_L * j_L + c_{L+1} \leq K$ generiert. Das aus all diesen Ungleichungen resultierende System kann nun beispielsweise mit der Fourier-Motzkin-Elimination gelöst werden, was im Resultat der direkten Lösung des Problems auf einem reduzierten System entspricht.

Bemerkung 3.10. *Das Vorschalten des 'Tests auf unabhängige Variablen' verringert ggf. die Anzahl der Variablen und Ungleichungen (durch die Kombination der Gleichungen mit den Ungleichungen), sodass die Lösung des resultierenden Systems oft effizienter ist als die des ursprünglichen in (3.8).*

Ein *graphentheoretisch basierter* Test, der die Frage nach der Lösbarkeit eines Ungleichungssystems für Abhängigkeiten beantworten kann, ist der **Schleifenresiduum-Test** nach Shostak [Sho81]. Liegen die Ungleichungen des Systems in der Form $i_k \leq i_l + c$ vor, so kann es mit einer vereinfachten Version des Schleifenresiduum-Tests gelöst werden. Ungleichungssysteme dieser Gestalt sind in der Abhängigkeitsanalyse üblich, da Schleifengrenzen häufig in dieser Form gegeben sind.
Aus allen so gegebenen Ungleichungen wird ein gewichteter und gerichteter Graph (V, E) konstruiert, mit

$$V = \{i_1, \ldots, i_N\} \quad \text{und} \quad E = \{(i_k, i_l) \text{ mit Gewicht } c \mid i_k \leq i_l + c\} \quad . \tag{3.9}$$

Aus diesem Graph lässt sich nun ableiten, dass für einen Pfad von i nach i' mit kumuliertem Gewicht $\tilde{c}$ gilt: $i \leq i' + \tilde{c}$. (Beispiel: $(i_1 \leq i_2 + c_1 \wedge i_2 \leq i_3 + c_2) \Rightarrow i_1 \leq i_3 + (c_2 + c_1)$)

Hat ein **Zyklus** in diesem Graph (somit $i \leq i + \tilde{c}$) ein *negatives* Gewicht $\tilde{c}$, so ist diese Ungleichung nicht erfüllbar. Somit ist das System nicht lösbar. Für die Suche nach negativen Zyklen in gerichteten Graphen existieren effiziente Algorithmen, so z. B. der von *Bellman, Ford und Moore* (z. B. in [Cor01]) zum Auffinden kürzester Wege zwischen zwei beliebigen Knoten eines gerichteten Graphen mit Komplexität $\mathcal{O}(|V|^2 * |E|)$ (für *alle* Knotenpaare insgesamt).

Beispiel 3.11.

$$\begin{aligned} i_1 &\leq i_2 + 4 \\ i_2 &\leq i_1 \\ i_2 &\leq i_3 - 3 \\ i_3 &\leq i_2 - 1 \\ i_3 &\leq i_1 - 2 \end{aligned}$$

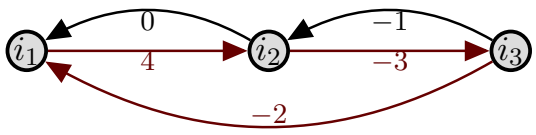


Abbildung 3.7: Schleifenresiduum mit negativem Zyklus

$i_1 \rightarrow i_2 \rightarrow i_3 \rightarrow i_1$ *ist ein negativer Zyklus. Dieses Ungleichungssystem ist somit nicht lösbar.*

Bemerkung 3.12. *Es kann sinnvoll sein, bereits gefundene Lösungen, sowie deren Systeme, zu speichern (**Memoisierung**), da bestimmte Zugriffsmuster regelmäßig wiederkehren und so ein einfaches Nachschlagen eines solchen Systems in einer Datenbank für dessen Lösung ausreicht. Allerdings kann solch ein Ansatz aufgrund des benötigten Speichers und des Aufwandes zum Finden der entsprechenden Systeme und Lösungen auch ineffizient sein. Außerdem müsste eine solche Datenbank angelegt werden und es müssten sinnvolle Kriterien bestehen, wann ein Zugriffsmuster 'häufig wiederkehrt'.*

3.6.2 Direkte Lösung

Eine direkte Lösung des allgemeinen gegebenen Systems aus (3.8) stellt, da es sich um ein allgemeines ILP handelt, eine *NP-schwierige* Aufgabe dar [ASLU08, Kap. 11.6.5]. Die Nutzung von Heuristiken, die die Lösbarkeit eines Teilproblems *verneinen*, ist somit grundsätzlich ein guter Ansatz. Eine approximative Lösung dieses Problems kommt i. d. R. nicht in Frage, da eine fälschliche Feststellung *keiner* Abhängigkeiten bei nachfolgenden Transformationen, wie z. B. einer Parallelisierung, fehlerhaften Code erzeugen könnte (*zuviele* festgestellte Abhängigkeiten wären hingegen legitim).
Aus praktischer Sicht ist eine heuristische oder approximative Herangehensweise i. d. R. auch nicht vonnöten, da die zu lösenden Ungleichungssysteme im Normalfall äußerst klein sind. Dies ist der Tatsache geschuldet, dass die Anzahl der Variablen im Ungleichungssystem der Anzahl der Schleifenindizes entspricht und die Anzahl der Ungleichungen proportional zur Anzahl der Array-Zugriffe und for-Schleifen ist. Somit sind die Anzahl sowohl der Zeilen, als auch der Spalten des Ungleichungssystems relativ klein.
Des Weiteren ist die Laufzeit des Kompilierungsvorgangs (in einem traditionellen Kompilierungsvorgang, an dessen Ende der gesamte zur Ausführung nötige, lauffähige Maschinencode steht) nicht allzu relevant, da es bei der Optimierung um das Beschleunigen dieses Endcodes geht. Darüber hinaus wird die Kompilierungszeit eines rechenintensiven Codes im Verhältnis zur Ausführungszeit (besonders bei mehrfacher Ausführung) marginal sein.

Somit ist eine direkte Lösung des Systems in (3.8) oftmals die Methode der Wahl.

Bemerkung 3.13. *Dieser Schritt der Abhängigkeitsanalyse (der Abhängigkeitstest) stellt im Normalfall lediglich deren erste Stufe dar. Sind nämlich Abhängigkeiten vorhanden, so folgen zur Optimierung meist problemspezifische Transformationen, welche die Datenstrukturen aus dieser zwar mitnutzen, allerdings eine deutlich intensivere Analyse der vorliegenden Abhängigkeiten fordern (siehe Abschnitt 3.7).*

3.7 Schleifentransformationen

In Anlehnung an das in dieser Arbeit erweiterte Framework werden Notationen aus [Bon08, Kap. 2.4] verwendet wie auch Konzepte hieraus dargestellt.

Um einen Quellcode mit dynamischen Abhängigkeiten z. B. **parallelisierbar**, **vektorisierbar** oder **blockbar** zu machen, muss die Struktur der Zugriffe auf die von den Abhängigkeiten betroffenen Datenstrukturen (i. d. R. Arrays) ggf. angepasst werden. Diese Transformation soll beispielsweise bestimmte Schleifen parallel ausführbar machen, ohne die Logik des Codes zu verändern. Somit ist ein **Mapping** der ursprünglichen Iterationsreihenfolge auf eine solche gesucht, welche die Voraussetzungen der gegebenen Transformationsidee, unter Einhaltung der Abhängigkeiten, erfüllt (z. B. Parallelität).

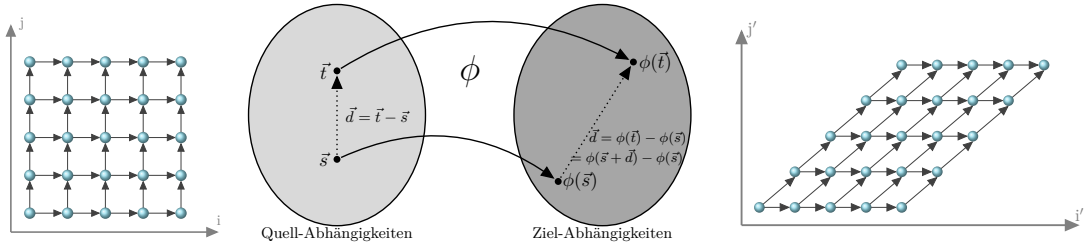


Abbildung 3.8: Abhängigkeitstransformation

Bemerkung 3.14. *Eine essentielle Beobachtung für das Modell ist, dass jeder Iterationspunkt in der Ausführung des Codes zu einem bestimmten Zeitpunkt ausgeführt wird. Diese Abbildung von einem Iterationspunkt auf seinen Ausführungszeitpunkt heißt* ***Scheduling****. Parallel auszuführende Iterationen haben beispielsweise denselben Zeitpunkt.*
Liegt eine verschachtelte for-Schleife vor, bei der in jedem for-Schleifenkopf der Laufindex inkrementiert wird, so entspricht die Reihenfolge der Ausführung der lexikographischen Ordnung, d. h. $\mathbf{i_1}$ ***wird vor*** $\mathbf{i_2}$ ***ausgeführt*** $\Leftrightarrow \mathbf{i_1} \prec \mathbf{i_2}$.

Die konkrete Transformation steht in der Mitte eines dreiteiligen Prozesses, nämlich

- Scan und Analyse des Codes sowie dessen Formalisierung in Datenstrukturen
- **Transformation** bzgl. gewünschter Eigenschaften auf diesen Datenstrukturen
- Erzeugung des transformierten Codes aus den transformierten Datenstrukturen

Die Bedingungen für **gültige (oder legale)** Transformationen werden im Folgenden erläutert. Zur Veranschaulichung (und wegen der hohen praktischen Relevanz) wird an einigen Stellen für die Abhängigkeiten $e^{S_i \to S_j}$ der Fall $S_i = S_j$ angenommen.

3.7.1 Polyeder-Transformation

Eine Polyeder-Transformation stellt eine Umordnung der adressierten Indexvektoren dar. Eine affine 1-dimensionale Transformation eines Statements S einer verschachtelten Schleife mit dynamischen Abhängigkeiten kann durch die affine Funktion

$$\begin{aligned} \phi_S(\vec{i}) &= \begin{pmatrix} t^S_{k,1} & \dots & t^S_{k,N_S} \end{pmatrix} \cdot \vec{i}_S + t^S_k \\ &= \begin{pmatrix} t^S_{k,1} & \dots & t^S_{k,N_S} & t^S_k \end{pmatrix} \cdot \begin{pmatrix} \vec{i}_S \\ 1 \end{pmatrix} \end{aligned} \tag{3.10}$$

beschrieben werden. Dadurch wird jedes Element des Quell-Indexraumes $\vec{i}$ durch $\phi_S(\vec{i})$ in eine Dimension im Ziel-Indexraum (hier: k) abgebildet und somit ggf. umgeordnet. Liegen mehrdimensionale Zugriffe vor, so erweitert dies die Transformation für M_S Dimensionen auf eine M_S-dimensionale Transformation (oder M_S eindimensionale)

$$\begin{aligned} \mathcal{T}_S(\vec{i}) &= T_S \cdot \vec{i} + t_S \\ &= \begin{pmatrix} \phi^1_S(\vec{i}) \\ \vdots \\ \phi^{M_S}_S(\vec{i}) \end{pmatrix} \\ &= \begin{pmatrix} t^S_{1,1} & \dots & t^S_{1,N_S} \\ \vdots & & \vdots \\ t^S_{M_S,1} & \dots & t^S_{M_S,N_S} \end{pmatrix} \cdot \begin{pmatrix} i^S_1 \\ \vdots \\ i_{N_S} \end{pmatrix} + \begin{pmatrix} t^S_1 \\ \vdots \\ t_{M_S} \end{pmatrix} \end{aligned} \tag{3.11}$$

mit $T_S \in \mathbb{Z}^{M_S \times N_S}$ und $t_S \in \mathbb{Z}^{M_S}$.

Satz 3.15. *Das Bild der affinen Transformation eines konvexen Indexraumes ist konvex.*

Beweis. Seien $\vec{i}$ und $\vec{j}$ beliebige Indexvektoren im Indexraum $\mathcal{D}^S$.
Da $\mathcal{D}^S$ konvex, gilt $\forall \lambda \in [0,1] : \lambda \cdot \vec{i} + (1-\lambda) \cdot \vec{j} \in \mathcal{D}^S$.
Im Bildraum gilt für eine beliebige Konvexkombination zweier abgebildeter Indexpunkte:

$$\begin{aligned} \lambda \cdot \mathcal{T}_S(\vec{i}) + (1-\lambda) \cdot \mathcal{T}_S(\vec{j}) &= \lambda \cdot \left(T_S \cdot \vec{i} + \vec{t_S} \right) + (1-\lambda) \cdot \left(T_S \cdot \vec{j} + \vec{t_S} \right) \\ &= T_S \cdot \left(\lambda \cdot \vec{i} + (1-\lambda) \cdot \vec{j} \right) + (\lambda + (1-\lambda)) \cdot \vec{t_S} \\ &= T_S \cdot \left(\lambda \cdot \vec{i} + (1-\lambda) \cdot \vec{j} \right) + \vec{t_S} && \in \mathcal{T}_S\left(\mathcal{D}_S\right) \\ & && \text{da } \mathcal{D}^S \text{ konvex} \end{aligned}$$

□

3.7.2 Gültige (legale) Transformation

Transformationsmatrizen T_S mit $d = M_S = N_S$ Zeilen und Spalten und vollem Spaltenrang sind oft vorausgesetzt, um eine Neuordnung zu erhalten, die den gesamten Indexraum vollständig durchläuft [Bon08, S. 22 *complete scanning order*]. Außerdem ist so ein **injektives** Mapping (da T_S invertierbar) induziert und kein Indexvektor wird mehrfach adressiert, wenn er es vor der Transformation nicht wurde. Singuläre und nichtquadratische Transformationen bergen weitere Probleme (z. B. kann sich die Dimension des Indexraumes ändern) und sind im hier entwickelten Framework nicht relevant.

Bemerkung 3.16. *Im Folgenden werden singuläre Transformationen nicht betrachtet, obwohl einige Konzepte übertragbar sind. Die zu generierenden Transformationsmatrizen sind nichtsingulär und quadratisch.*

Eine besondere Rolle in der Codeoptimierung spielen **unimodulare Transformationen**, also Transformationen mit unimodularer Transformationsmatrix (siehe Abschnitt 3.7.3). Für Transformationen, egal welchen Typs, muss die folgende Bedingung gelten.

Lemma 3.17. *Statementweise Transformationen $\mathcal{T}_S$ sind **gültig (oder legal)**, das heißt sie respektieren alle Abhängigkeiten, wenn*

$$\mathcal{T}_{S_j}(\vec{t}) - \mathcal{T}_{S_i}(\vec{s}) \succ \vec{0} \quad \forall (\vec{s}, \vec{t}) \in \mathcal{P}_e \;\; \forall e \in E$$

für voneinander abhängige Indexpunkte $\vec{s}$ (source), $\vec{t}$ (target) im Abhängigkeitspolyeder $\mathcal{P}_e$ gilt (mit E die Kantenmenge des DDG und $e^{S_i \to S_j} \in E$).

Bemerkung 3.18. *Lemma 3.17 besagt, dass ein Indexvektor $\vec{s}$, der aufgrund einer Abhängigkeit im Quell-Polyeder vor $\vec{t}$ ausgeführt werden muss, auch im Ziel-Polyeder vor $\vec{t}$ ausgeführt werden muss und somit **lexikographisch kleiner** sein muss ($\mathcal{T}_{S_i}(\vec{s}) \prec \mathcal{T}_{S_j}(\vec{t})$). Beispiel 3.31 verdeutlicht das Lemma.*

Eine *Abhängigkeitskante* e wird auf Level l (*Tiefe*, *Dimension*) **erfüllt**, falls es das erste Level (das kleinste l) ist, sodass für $(\vec{s}, \vec{t}) \in \mathcal{P}_e$ gilt:

$$\begin{aligned} & \phi^k_{S_j}(\vec{t}) - \phi^k_{S_i}(\vec{s}) \geq 0 \quad \forall k < l \\ \wedge \quad & \phi^l_{S_j}(\vec{t}) - \phi^l_{S_i}(\vec{s}) \geq 1 \end{aligned} \tag{3.12}$$

Bemerkung 3.19. *Die Hyperebenen ϕ können auch mit ihren korrespondierenden Schleifen(indizes) identifiziert werden, $\phi = \begin{pmatrix} 0 & 1 \end{pmatrix}$ bei Indexvektor $\mathcal{I} = \begin{pmatrix} i \\ j \end{pmatrix}$ z. B. mit $\phi \cdot \mathcal{I} = j$.*

3.7.3 Unimodulare Transformationen

Unimodulare Transformationsmatrizen haben eine besondere Bedeutung in der Codeoptimierung.

Beispiel 3.20.

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} \qquad \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad \begin{pmatrix} 1 & q \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ q & 1 \end{pmatrix}$$

Schleifenumkehr (**loop reversal**) — Schleifenvertauschung (**loop interchange**) — Schleifenverzerrung (**loop skewing**)

Abbildung 3.9: Beispiele für unimodulare Transformationsmatrizen

Neben der großen praktischen Bedeutung der in Beispiel 3.20 aufgeführten Transformationen z. B. für die Parallelisierung einer verschachtelten Schleife, sind unimodulare Matrizen aufgrund ihrer ganzzahligen Invertierbarkeit (Lemma 2.4) zur praktischen Transformation besonders geeignet, da einige Konzepte mit der Inversen der Transformationsmatrix arbeiten. Nicht-unimodulare Transformationsmatrizen mit $|Determinante| > 1$ können u. A. problematisch sein, da sie 'Löcher' im Ziel-Indexraum erzeugen (vgl. auch [Ram95]).

Die Abbildungen 3.10 und 3.11 zeigen, wie das Abhängigkeitspolyeder (bzw. die Zugriffsstruktur des Statements) aus Beispiel 3.7 durch zwei Transformationsmatrizen modifiziert wird.

$$T_S \cdot \mathcal{I} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix} \qquad (3.13)$$

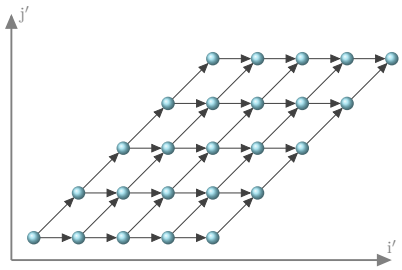


Abbildung 3.10: Unimodulare Transformation

$$T_S \cdot \mathcal{I} = \begin{pmatrix} 2 & 1 \\ 0 & 2 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix} \qquad (3.14)$$

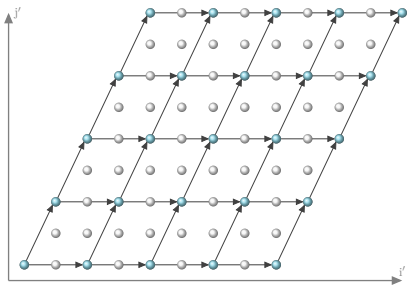


Abbildung 3.11: Nicht-unimodulare Transformation

Bemerkung 3.21. *Der **Zusammenhang** zwischen der **Determinante** der Transformationsmatrix und den **'Löchern'** im Indexraum lässt sich anschaulich durch die geometrische Bedeutung der Determinante erklären, denn die Determinante des Spanns von aufspannenden Vektoren eines **Parallelepipeds** P entspricht dessen Volumen $Vol(P) = |det(P)|$ (Lemma 3.49). Wird also ein Parallelepiped P im $\mathbb{R}^N$ durch T in $T \cdot P$ transformiert, so heißt das für das Volumen: $Vol(T \cdot P) = |det(T \cdot P)| = |det(T)| \cdot |det(P)|$. Ist nun die Transformationsmatrix unimodular, so ist $|det(T)| = 1$ und das heißt, dass $Vol(T \cdot P) = |det(P)|$. Somit ist die Transformation des Parallelepipeds **volumenerhaltend**. Gilt $|det(T)| > 1$, so erhöht sich das Volumen des Parallelepipeds bei gleichbleibender Anzahl an Indexpunkten, wodurch die 'Löcher' entstehen.*

Die 'Löcher' sorgen somit für größere Iteratoren, was in wissenschaftlichen Anwendungen (bei großen Problemen) den Bereich des Datentyps (z. B. *int*) überschreiten könnte und allein deshalb möglichst vermieden oder zumindest beschränkt werden sollte.

Beispiel 3.22. *Die **unimodulare** Transformation (3.13) hat $det(T) = 1$, wodurch das Volumen erhalten bleibt und keine 'Löcher' im Indexraum entstehen. Die **nichtunimodulare** Transformation (3.14) hat $det(T) = 4$, wodurch sich das Volumen des Indexraumes vervierfacht und 'Löcher' in diesem entstehen.*

Unimodulare Transformationen können, ohne Verlust der Unimodularität, auch beliebig miteinander kombiniert werden, denn:

Satz 3.23. *Das Produkt beliebig vieler unimodularer Matrizen ist unimodular.*

Beweis. Seien $U_1, \ldots, U_M \in \mathbb{Z}^{N \times N}$ unimodulare Matrizen. Dann gilt:

$$\left| det\left(\prod_{i=1}^{M} U_i \right) \right| \overset{\text{Lemma 2.1}}{=} \left| \prod_{i=1}^{M} det\left(U_i\right) \right| = \prod_{i=1}^{M} |det\left(U_i\right)| \underset{\text{unimodular}}{\overset{\text{da } U_i}{=}} \prod_{i=1}^{M} 1 = 1$$

□

Daraus lässt sich direkt folgern:

Korollar 3.24. *Die Verknüpfung beliebig vieler unimodularer Transformationen ist eine unimodulare Transformation.*

Beispiel 3.25. *Eine Schleifenvertauschung mit* ***anschließender*** *Schleifenumkehr des zweiten Indexes eines zweidimensionalen Indexvektors bzgl. eines Statements S hat die folgende Form:*

$$T_S^* = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \cdot \left(\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix} \right) = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix}$$
$$= \begin{pmatrix} j \\ -i \end{pmatrix}$$

Eine Schleifenvertauschung mit ***vorangegangener*** *Schleifenumkehr des zweiten Indexes eines zweidimensionalen Indexvektors bzgl. eines Statements S hat die folgende Form:*

$$T_S^* = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \cdot \left(\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix} \right) = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix}$$
$$= \begin{pmatrix} -j \\ i \end{pmatrix}$$

3.7.4 Kommutatives Band

Definition 3.26 (vgl. [Bon08, Definition 12])**.** *Die Transformationen $\phi^k_{S_i}$ der Level $k = p, \ldots, p+s-1$ bilden ein* ***Kommutatives Band*** *(Permutable Band) der transformierten verschachtelten Schleife genau dann, wenn*

$$\forall k \in \{p, \ldots, p+s-1\} : \phi^k_{S_j}(\vec{t}) - \phi^k_{S_i}(\vec{s}) \geq 0, \quad (\vec{s}, \vec{t}) \in \mathcal{P}_e, e \in E_p$$

mit E_p die Menge der Abhängigkeiten des DDG, die bis zu Level $p-1$ noch nicht erfüllt sind.

Die ϕ^i bilden die Dimensionen der Transformation $\mathcal{T}$ und ein solches Band somit eine zusammenhängende Menge solcher Dimensionen bzw. eine Menge aufeinanderfolgender Schleifen in einer transformierten verschachtelten Schleife.
Die Schleifen innerhalb eines Kommutativen Bandes können beliebig untereinander vertauscht werden, ohne dass dies Abhängigkeiten verletzt, denn Lemma 3.17 bleibt durch solche Vertauschungen offensichtlich gültig, da die Differenz $\mathcal{T}_{S_j}(\vec{t}) - \mathcal{T}_{S_i}(\vec{s})$ durch Vertauschung positiver Einträge lexikographisch positiv bleibt (wenn sie es vor dieser war). Dies ist zur Parallelisierung und zur Optimierung von Speicherzugriffen eine wichtige Eigenschaft.

3.7.5 Gültige (legale) Parallelisierung

Die Parallelisierung einer Schleife (und somit die parallele Ausführung ihrer betreffenden Indexiterationen) ist nur dann legal, wenn zwischen Iterationen dieser Schleife keine Abhängigkeiten bestehen, die Iterationen müssen im Moment der Parallelität unabhängig voneinander sein. Verschiedene Arten von Parallelität können zur Klassifikation in entsprechende Gruppen unterteilt werden, welche sich u. A. in der Kommunikation zwischen den Prozessoren (bzw. allgemein Recheneinheiten) unterscheiden.
In einigen Fällen kann bei einer nativ nicht parallelisierbaren Schleife durch Transformationen Parallelisierbarkeit erreicht werden. Solche Transformationen zu finden ist eines der Hauptforschungsgebiete innerhalb des Polyedermodells.

Äußere Parallelität (vgl. [Bon08, Definition 13]) Eine Transformationshyperebene $\{\phi_{S_1}, \ldots, \phi_{S_M}\}$ heißt **Außen-Parallel** dann und nur dann, wenn

$$\phi_{S_j}(\vec{t}) - \phi_{S_i}(\vec{s}) = 0 \tag{3.15}$$

für $(\vec{s}, \vec{t}) \in \mathcal{P}_e$ für alle Abhängigkeiten $e \in E$ zwischen Statements im DDG.

Die äußere Parallelität wird auch als *kommunikationsfreie Parallelität* bezeichnet, da keinerlei Abhängigkeiten in der Hyperebene ϕ 'vorhanden sind' und die so entstehenden Stränge also vollkommen unabhängig und kommunikationslos parallel ausgeführt werden können. Somit ist in diesem Fall auch keine Synchronisierung der parallelen Stränge erforderlich, wie in Abbildung 3.12 durch die Hyperebene $\phi = (0, 1)$ bzw. j.

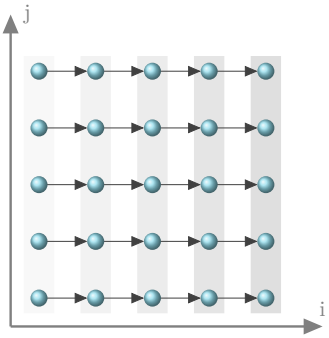


Abbildung 3.12: Außen-Parallele Schleife

Innere Parallelität (vgl. [Bon08, Definition 14]) Eine Transformationshyperebene $\{\phi^k_{S_1}, \ldots, \phi^k_{S_M}\}$ heißt **Innen-Parallel** dann und nur dann, wenn

$$\phi^k_{S_j}(\vec{t}) - \phi^k_{S_i}(\vec{s}) = 0 \tag{3.16}$$

für $(\vec{s}, \vec{t}) \in \mathcal{P}_e$ für alle Abhängigkeiten $e \in E_k$ zwischen Statements im DDG, die bis zum $(k-1)$-ten Level noch nicht erfüllt wurden.

Bemerkung 3.27. *Sind bis zu einem Level $k-1$ bereits **alle** Abhängigkeiten erfüllt, so sind die darauf folgenden Level somit automatisch alle Innen-Parallel.*

Eine Innen-Parallele Schleife darf im Kontext einer verschachtelten Schleife immer weiter nach innen verlagert werden, ohne dass dies Abhängigkeiten oder die Parallelität verletzt, **nicht** aber nach außen. Synchronität und Kommunikation der Recheneinheiten ist ggf. erforderlich, wie in Abbildung 3.13 durch die Hyperebene $\phi = (0, 1)$ bzw. j.

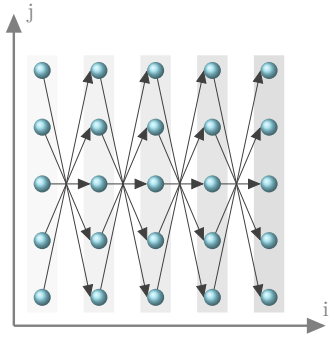


Abbildung 3.13: Innen-Parallele Schleife

Bemerkung 3.28. *Äußere und innere Parallelität werden auch als **doall**-Parallelität bezeichnet.*

Verzögerte Parallelität (vgl. [Bon08, S.30]) Verzögerte Parallelität (*Pipelined Parallelism*) tritt auf, wenn Schleifen durch einen Versatz (*delay*) parallelisierbar werden.

Im abgebildeten Fall kann Parallelität durch einen Versatz von 1 realisiert werden (nativ ist keine der Schleifen parallelisierbar), wodurch die Recheneinheiten verzögert beginnen zu rechnen. Auch hier müssen die parallelen Berechnungen synchronisiert ausgeführt werden, wie in Abbildung 3.14 durch die Hyperebene $\phi = (1, 1)$ bzw. $i + j$. In diesem Fall könnten beide Schleifen, i oder j, verzögert parallel zur jeweils anderen ausgeführt werden.

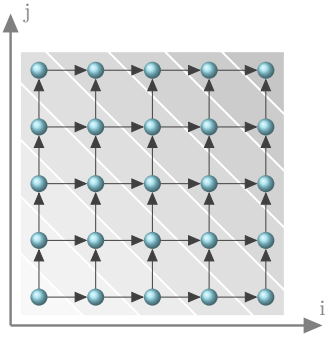


Abbildung 3.14: Verzögert Parallele Schleife

Bemerkung 3.29. *Verzögerte Parallelität wird auch **doacross**-Parallelität genannt.*

Eine legale Transformation, welche eine parallele transformierte j-Schleife ermöglicht, wäre z. B. $\mathcal{T} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$. Die daraus entstehende Verzögerung ist in Abbildung 3.15 verdeutlicht.

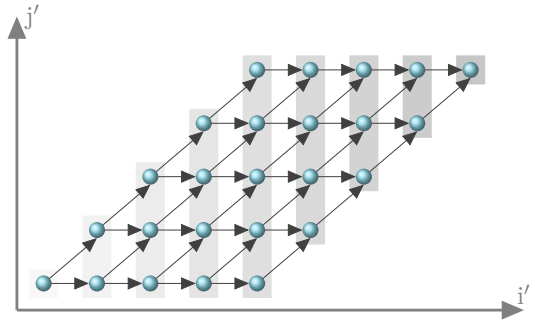


Abbildung 3.15: Transformation bei verzögerter Parallelität

Bemerkung 3.30. *Indexpunkte i_1, i_2, für die $\phi(i_1) = \phi(i_2)$ gilt, befinden sich in Richtung der Hyperebene ϕ (oder im transformierten Indexraum bzgl. der ϕ-Achse) auf derselben Stufe. Wenn also z. B. für alle Abhängigkeiten $\vec{s} \rightarrow \vec{t}$ und eine Hyperebene ϕ die Bedingung (3.15) gilt, so 'kreuzen' die Abhängigkeiten diese Hyperebene nicht, da alle voneinander abhängigen Indexpunkte jeweils in derselben Hyperebene liegen, sodass alle Indexpunkte auf dieser Hyperebene mit $\phi(\vec{i}) = k$ unabhängig von anderen mit $\phi(\vec{i}) \neq k$ ausgeführt werden können. Somit lassen sich die verschiedenen ϕ-Hyperebenen kommunikationslos parallel ausführen.*

Analysen zur Parallelität wie in diesem Fall sind komplexer als Extreme wie z. B. innere oder äußere Parallelität und können oft nicht durch einfache Regeln wie in (3.15) oder (3.16) durchgeführt werden. Zwei Konzepte zur *allgemeinen* Lokalisierung von Parallelisierungsmöglichkeiten, sowie deren konkrete Umsetzung, sollen im Folgenden kurz erwähnt und gegenübergestellt werden. Auf den Partitionierungsansatz wird im Folgenden noch sehr ausführlich eingegangen, da dieser in der Umsetzung gewählt wurde.

Beispiel 3.31. *Lemma 3.17 sowie die Parallelitätskonzepte sollen am folgenden Beispiel durch verschiedene Transformationen veranschaulicht werden.*

```
1 for(i=1; i<M; i++)
2   for(j=1; j<M; j++)
3     A[i][j]=...A[i-1][j+1]...;
```

Es existiert ein Statement und eine gleichförmige Abhängigkeit $d = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$, somit gilt: $\mathcal{T}(\vec{t}) - \mathcal{T}(\vec{s}) = T \cdot d$.

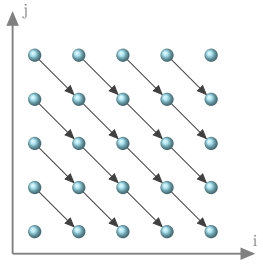


Abbildung 3.16: Ausgangspolyeder

Gültigkeit:

$$T = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \Rightarrow T \cdot d = \begin{pmatrix} 1 \\ -1 \end{pmatrix} \succ \vec{0}$$

entspricht der gegebenen Codestruktur und ist somit gültig. In der gegebenen Reihenfolge sind alle benötigten Daten zu jedem Zeitpunkt vorhanden.

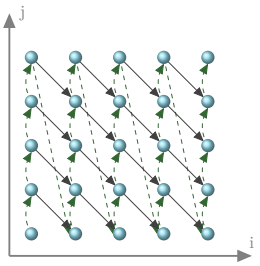


Abbildung 3.17: Gültige Transformation

$$T = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \Rightarrow T \cdot d = \begin{pmatrix} -1 \\ 1 \end{pmatrix} \not\succ \vec{0}$$

entspricht der Schleifenvertauschung und ist nicht gültig, da Daten erst berechnet werden würden, **nachdem** sie benötigt werden.

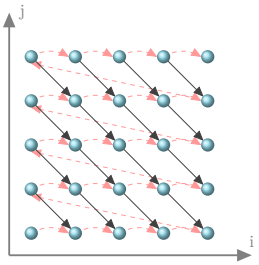


Abbildung 3.18: Nicht gültige Transformation

Parallelität:

$$T = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \Rightarrow T \cdot d = \begin{pmatrix} 1 \\ -1 \end{pmatrix} \succ \vec{0}$$

ist gültig und erzeugt innere Parallelität durch $\begin{pmatrix} 0 & 1 \end{pmatrix}$, da bereits auf Level $k = 1$ alle Abhängigkeiten erfüllt sind. Deshalb sind sie auf Level 2 nicht mehr relevant, somit ist ϕ^2 (inner-) parallel und darf nicht nach außen verlagert werden.

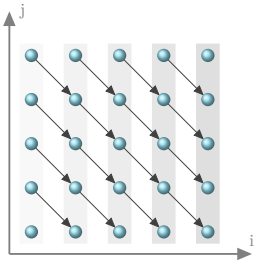


Abbildung 3.19: Native innere Parallelität

$$T = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \Rightarrow T \cdot d = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \succ \vec{0}$$

ist gültig und erzeugt äußere Parallelität, da $\phi^2(\vec{t}) - \phi^2(\vec{s}) = 0$ für die einzige vorhandene Abhängigkeit gilt.

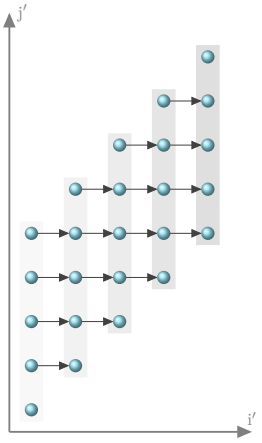


Abbildung 3.20: Transformation für äußere Parallelität

3.7.6 Vektorisierbarkeit

Eine Schleife ist vektorisierbar, wenn diese *zusätzlich* zu der 'gewöhnlichen' Parallelisierbarkeit die Eigenschaft besitzt, die innerste Schleife der Verschachtelung zu sein oder gültigerweise ins Innerste der Verschachtelung (durch Schleifenvertauschungen) verschoben zu werden. Die Iterationen der innersten Schleife, die parallel durchlaufbar sind, können dann auch parallel in den Registern einer Vektoreinheit berechnet werden. Für eine *praktische und effiziente* Vektorisierung durch die SSE-Register aktueller CPUs müssen weitere Eigenschaften erfüllt sein, welche ausführlich in Kapitel 4 erläutert werden. Eine parallele Berechnung durch Vektoreinheiten ist vollständig synchronisiert, da die parallelen Berechnungen in den Registern dieser 'in einem Schritt' durchgeführt werden.

3.7.7 Parallelisierung: Space-Time-Mapping vs. Partitionierungsansatz

Die zwei wesentlichen Ziele für Schleifentransformationen in dem im Rahmen dieser Arbeit entwickelten Framework sind

- **Vektorisierung** und Parallelisierung von Schleifen und
- Optimierung der **Datenlokalität**.

Bisher wurde betrachtet, **wie** bei gegebenen Transformationen **erkannt** werden kann, ob diese gültig sind oder welche Schleifen durch diese parallel ausführbar werden.

Das Ziel eines Source-To-Source Compilers ist hingegen, Transformationen zu **konstruieren** und durchzuführen, welche die gewünschten Eigenschaften im transformierten Code erzeugen. Hierzu werden nun zwei Ansätze, das **Space-Time-Mapping** und der **Partitionierungsansatz**, vorgestellt, welche beide den Aspekt der Parallelisierung (bzw. Vektorisierung) verfolgen. Für die Optimierung der Datenlokalität ist der Partitionierungsansatz prädestiniert, in den traditionellen Methoden durch Space-Time-Mappings ist eine solche Optimierung nicht vorgesehen.
Beim *Partitionierungsansatz* wird nach **Hyperebenen** ϕ^i gesucht, welche den Indexraum für eine gute Datenlokalität in Blöcke (*Tiles*) zerlegen. Diese Blöcke können dann ggf. parallel zueinander ausgeführt werden.
Beim *Space-Time-Mapping* werden **Funktionen** θ **und** σ gesucht, welche jedem Iterationspunkt konkret einen Zeitpunkt und einen Ort der Ausführung zuweisen.

In beiden Fällen wird auf Grundlage der gefundenen Hyperebenen und Funktionen eine gültige Transformation erstellt.

Im Folgenden werden die Unterschiede zwischen den beiden Methoden dargestellt.

Da letztendlich der Partitionierungsansatz gewählt wurde und das Space-Time-Mapping den Aspekt der Blockung i. d. R. nicht ins Modell aufnimmt, wird der Vergleich der beiden Ansätze bereits an dieser Stelle, *vor* der methodischen Erläuterung der Blockung in Abschnitt 3.8, vorgenommen.
Im Anschluss daran wird gezeigt, wie die Generierung solcher Hyperebenen theoretisch, sowie im Source-To-Source Compiler PluTo, durch ganzzahlige Optimierung realisiert wird.

Wenn im Folgenden von 'Recheneinheiten' gesprochen wird, sind Prozessoren oder Register einer Vektoreinheit gemeint, die Konzepte sind aber auf andere parallele Architekturen übertragbar.

Space-Time-Mapping Im Ansatz des Space-Time-Mappings werden zwei Funktionen, θ und σ, gesucht, welche jedem Iterationspunkt explizit einen Zeitpunkt sowie einen Ort der Berechnung zuweisen. Man nennt die ein Space-Time-Mapping charakterisierenden Funktionen **Schedule** (θ) für die Zuweisung des Zeitpunktes und **Allokation** (σ) für die Zuweisung der Recheneinheit. An diese Funktionen können nun Bedingungen gestellt werden, in Ansätzen für eine Parallelisierung sind dies typischerweise die Folgenden:

Das *Schedule* θ muss die Abhängigkeiten respektieren, es muss also

$$\theta(\vec{t}) - \theta(\vec{s}) \geq 1 \tag{3.17}$$

für alle Abhängigkeiten $\vec{s} \rightarrow \vec{t}$ erfüllen, d. h. dass eine Iteration frühestens einen Zeitschritt nachdem alle erforderlichen Daten berechnet wurden, ausgeführt werden darf. Iterationen i_1, i_2, für die $\theta(\vec{i_1}) = \theta(\vec{i_2})$ gilt, werden auf denselben Zeitpunkt abgebildet und somit parallel zueinander ausgeführt.
Die *Allokation* σ darf zwei Iterationen $\vec{i_1}, \vec{i_2}$ desselben Zeitschritts nicht auf dieselbe Recheneinheit abbilden, also $\sigma(\vec{i_1}) \neq \sigma(\vec{i_2})$ falls $\theta(\vec{i_1}) = \theta(\vec{i_2})$.
Basierend auf diesen beiden Funktionen wird eine d-dimensionale Transformation $\mathcal{T} = (\phi^1, \ldots, \phi^d)^T$ (oft nicht unbedingt affin, siehe 'Vergleich der beiden Ansätze') generiert.

Ist eine d-dimensionale Transformation $\mathcal{T} = (\phi^1, \ldots, \phi^d)^T$ bestimmt, so kann man die resultierenden Schleifen entweder als Zeit-Schleife (*time*) oder als Ort-Schleife (*space*)

kennzeichnen. Eine Zeit-Schleife wird sequentiell von einer Recheneinheit verarbeitet, eine Ort-Schleife hingegen wird parallel auf verschiedenen Recheneinheiten ausgeführt. In Beispiel 3.33 ist i eine Zeit-Schleife und j eine Ort-Schleife. Somit entspricht i dem Zeitpunkt der Ausführung und j der dazu vorgesehenen Recheneinheit.

Bemerkung 3.32. *Ein Space-Time-Mapping kann somit als Abbildung des Indexraumes beliebiger Dimensionalität auf eine 2-dimensionale Zuordnung verstanden werden, in welcher die Iterationen nach Ausführungszeitpunkt und Recheneinheit angeordnet sind. Dies impliziert natürlicher Weise eine Synchronität der Rechenschritte, da deren Zeitpunkt vollständig und eindeutig vor Beginn der Ausführung determiniert ist.*

Ein solches Mapping nur auf Grundlage der Bedingung einer 'legalen' Zuweisung von Recheneinheit und Zeitpunkt vorzunehmen, führt zwar zu semantisch korrektem und u. U. parallelem Code, allerdings ist beispielsweise die Kommunikation zwischen Recheneinheiten (z. B. Prozessoren) in diesem Modell *nicht* beachtet. Das folgende Beispiel soll die Problematik weiter motivieren.

Beispiel 3.33.

```
1 for(i=1; i<M; i++)
2   for(j=1; j<M; j++)
3     A[i][j]=...A[i-1][j-1]...;
```

$$\text{Schedule}: \quad \theta(i,j) = i$$
$$\text{Allokation}: \quad \sigma(i,j) = j$$

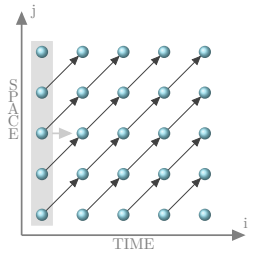


Abbildung 3.21: Schleife für Space-Time-Mapping

In Beispiel 3.33 ist der Code zwar ohne Transformation parallelisierbar, allerdings müssen bei dem gegebenen Schedule $\theta(i,j) = i$ in jeder i-Iteration Daten von einer Recheneinheit (hier beispielsweise Prozessoren) zu einer benachbarten übertragen werden. Selbst wenn sich die Prozessoren einen gemeinsamen Speicher teilen, der diese Daten enthält, so können die Prozessoren die Daten nicht aus ihrem L1-Cache wiederverwenden, was der Fall wäre, wenn die vorhandenen 'Stränge' (diagonal) jeweils auf einem Prozessor gerechnet werden würden.

Die üblichen Schedule-Allokation-basierten Ansätze zur Parallelisierung sind z. B. in [Fea92a; Fea92b; DR95; Gri04] zu finden.

Partitionierungsansatz Partitionen-basierte Ansätze bilden nicht jeden Indexpunkt explizit auf einen Zeitpunkt und einen Ort ab, sondern partitionieren den Indexraum in Blöcke (*Tiles*), welche dann z. T. parallel zueinander ausgeführt werden, ohne die innere Ausführung verschiedener solcher Blöcke untereinander zu synchronisieren. Damit dies möglich ist, müssen die gebildeten Blöcke wiederum Voraussetzungen erfüllen, die von den Daten-Abhängigkeiten im vorliegenden Code abhängen.
Da dieser Ansatz hier gewählt wurde und somit im Folgenden ausführlich erläutert wird, soll an dieser Stelle nicht detaillierter darauf eingegangen werden. Einen üblichen Partitionierungsansatz findet man neben den Arbeiten von Bondhugula [Bon08; BHRS08] z. B. in [LL97].

Vergleich der beiden Ansätze Ein Problem bei vielen Schedule-Allokations-Ansätzen ist, dass das theoretisch bzgl. eines Kriteriums optimale Ergebnis in der Praxis zwar parallel ausführbar ist, allerdings nicht performant rechnet, da sehr viele **Hardware-spezifische** Kriterien außer Acht gelassen werden. Dazu gehören unter anderem *Anzahl von Recheneinheiten*, *vorhandener Cache*, *weitere vorhandene Beschleuniger* etc.
Beispielsweise kann ein theoretisch zeitoptimales Space-Time-Mapping, welches von der Annahme beliebig vieler Recheneinheiten ausgeht, in der Praxis äußerst schlecht performen, wie in **Abbildung 3.23** durch ein **Feautrier** Space-Time-Mapping (sowohl Schedule als auch Allokation). Dies liegt daran, dass **in der Praxis** bei einer Anzahl $\wp \ll M$ Recheneinheiten (z. B. Prozessoren, vgl. Beispiel 3.33), die theoretische Wiederverwendbarkeit der Daten auf einem Prozessor aus dem Cache dadurch verhindert wird, dass zunächst die gesamte j-Schleife durchlaufen wird und die Prozessoren währenddessen viele Male neue Daten benötigen. Die so berechneten Daten können also bereits wieder aus dem Cache verdrängt worden sein, wenn die nächste i-Iteration erreicht wird. Außerdem ist die hier berechnete Schedule-Funktion nach Feautrier keine lineare Funktion, sondern muss durch 'teure' **if**-Abfragen verschiedene Fälle selektieren.

Bemerkung 3.34. *Der Feautrier-Scheduler führt jeden Iterationspunkt zum frühest möglichen Zeitpunkt aus und hat somit als theoretische Laufzeit (unter der Annahme beliebig vieler Prozessoren) die Laufzeit des* ***längsten Pfades*** *der Abhängigkeitspolyeder, ist also unter diesen Annahmen theoretisch zeitoptimal.*

Darüber hinaus kann notwendige Kommunikation zwischen den Recheneinheiten Laufzeitgewinne wieder egalisieren, wie in **Abbildung 3.22** durch die Nutzung der gegebenen inneren Parallelität, wobei hier die Auslastung der Prozessoren (wenn es in der Praxis

ausreichend viele für die Space-Dimension gäbe) gleichmäßig wäre. Hier müssten die Prozessoren allerdings, in **jedem** Iterationsschritt von i, alle benötigten Daten von anderen Recheneinheiten verwenden. Das bedeutet, selbst wenn alle erforderlichen Daten in den Arbeitsspeicher passen, eine theoretische *L1-Cache-Miss-Rate* von **100 %**.

Ein Space-Time-Mapping, welches die *theoretischen Vorteile* beider anderer Versionen vereint, wäre das in **Abbildung 3.24**. Allerdings würden auch hier (in der Praxis) bei $\wp \ll M$ benötigte Daten bereits wieder aus dem Cache verdrängt worden sein. Außerdem wären wiederum Fallunterscheidungen, ggf. in der innersten Schleife, erforderlich. *Dieses ist kein Ergebnis der üblichen Schedule-Allokator-Kombinationen, sondern ein (zum Vergleich) manuell erstelltes.*

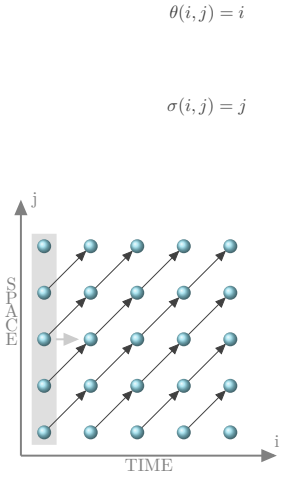


Abbildung 3.22: Gegebene Parallelität: theoretisch zeitoptimal aber hohe Kommunikation

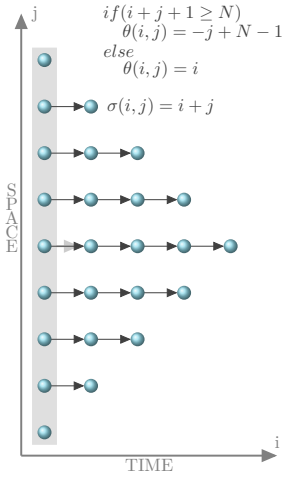


Abbildung 3.23: Feautrier: theoretisch zeitoptimal aber schlechte Datenwiederverwendung

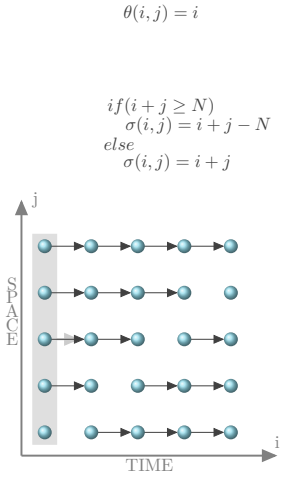


Abbildung 3.24: Mögliche Kombination der Vorteile beider anderer Versionen (manuell erstellt)

Da die Speicheranbindung auf aktuellen Prozessoren oft ein limitierender Laufzeit-Faktor eines Algorithmus ist (besonders bei speichergebundenen [*memory-bound*] Algorithmen), ist eine Beschleunigung durch bessere Speichernutzung in Form von Daten-Wiederverwendung auf einem Prozessor und geringer Kommunikation zwischen verschiedenen Prozessoren ein Hauptaspekt der praktischen Beschleunigung von Codes.

Im Ansatz von [Bon08] wird *Kommunikation* und *Daten-Wiederverwendung* als Optimierungsziel ins Modell mit aufgenommen, um somit u. A. die angesprochenen Probleme in den dargestellten Space-Time-Mapping-Ansätzen zu lösen.
Durch die Partitionierung des Indexraumes und die Parallelität solcher Blöcke entsteht

eine **grobe Parallelität**, die durch eine Parallelität von Schleifen im Inneren der Blöcke noch erweitert werden kann, z. B. indem diese parallelen Schleifen im Inneren (wenn möglich) vektorisiert werden. Die Cache-Vorteile bei einem so partitionierten Code spielen eine entscheidende Rolle in den hier umgesetzten Transformationen.
Außerdem ist eine Blockung zur Optimierung dieser beiden Kennzahlen oft bereits auf seriellen Prozessoren ohne parallele Recheneinheiten beschleunigend, ohne die mögliche Parallelität auszunutzen, da Speicherzugriffszeiten verringert werden.

Zur Umsetzung einer **effizienten Vektorisierung** wurde hier der Partitionierungsansatz gewählt, da

- die Blockung einer parallelisierbaren Schleife in Register-konforme Blöcke für die Vektorisierung i. d. R. nötig ist, damit die Schleifenblöcke auf die **SIMD-Register** passend verteilt werden können. Dies gilt sowohl bei der Vektorisierung durch explizite SIMD-Instruktionen im Code, als auch bei der Vektorisierung (also die Generierung dieser Instruktionen) durch einiger Compiler,
- die **Speicheranbindung** bei der Vektorisierung eine entscheidende Rolle spielt und eine effektive Wiederverwendung sowie ein guter Datenfluss durch die Cache-Hierarchie den Vektorisierungsspeedup unterstützen,
- eine **Kombination** von Parallelisierung (Multicore) und Vektorisierung (SSE) in diesem Modell gut realisierbar ist und
- die Phasen der **ganzzahligen Optimierung** zur Transformation im Partitionierungsansatz eine an die Architektur angepasste Optimierung ermöglichen.

3.8 Tiling

Tiling (oder *Blockung*) ist eine Kerntransformation zur Optimierung von Speicherzugriffen und Datenwiederverwendung. Die **Idee** dahinter ist eine Umsortierung der Datenzugriffe in der Art vorzunehmen, dass

- berechnete Daten möglichst oft und zeitnah *wiederverwendet* werden und
- benötigte Daten möglichst in einem Level der *Cache*-Hierarchie vorliegen.

Neben diesen **Zielaspekten** sind

- eine *Parallelität* von verschiedenen Tiles sowie
- geringe *Kommunikation* zwischen solchen Tiles

Anforderungen, welche die entsprechenden Tilings optimalerweise erfüllen sollten.

Notwendige Bedingung für ein Tiling, welches die Semantik des vorliegenden Codes nicht verändert, ist, dass

- die Umordnung die *Abhängigkeiten* respektiert.

Ein solches Tiling nennt man **gültig**.

3.8.1 Formalisierung

Zur Erfüllung dieser Ziele wird der **Indexraum** in (i. d. R. - bis auf randnah - gleichgroße) Blöcke partitioniert, welche jeweils zusammenhängend ausgeführt werden.

Ein rechteckiges Tiling eines Statements S in einer zweifach verschachtelten Schleife mit Tile-Quantitäten (q_i, q_j) ist in Abbildung 3.25 dargestellt.

```
1 for(i=0; i<M; i++)
2  for(j=0; j<N; j++)
3   S(i,j);
```

$\Rightarrow$

```
1 for(ii=0; ii≤floor(M-1,qi); ii++)
2  for(jj=0; jj≤floor(N-1,qj); jj++)
3   for(i=qi*ii; i≤min(M-1,qi*ii+(qi-1)); i++)
4    for(j=qj*jj; j≤min(N-1,qj*jj+(qj-1)); j++)
5     S(i,j);
```

Abbildung 3.25: Codeveränderung eines rechteckigen Tilings

Die Umordnung der Iterationsreihenfolge durch ein solches rechteckiges Tiling ist in Abbildung 3.26 visualisiert.

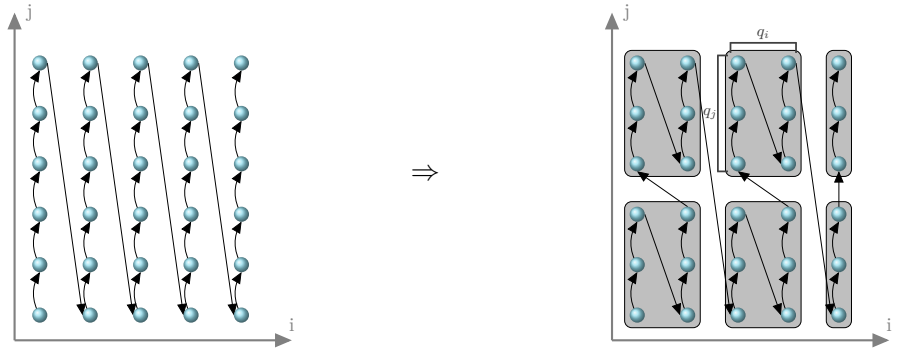


Abbildung 3.26: Iterationsänderung eines rechteckigen Tilings

Die Indexpunkte werden nach dem Tiling in einer anderen Reihenfolge als zuvor adressiert. Daraus resultiert auch ein veränderter Zugriff auf den Speicher. Dies kann sich bereits Performance-technisch auswirken, da die Daten in einer festen Reihenfolge im Speicher liegen. Vorteilhaft sind z. B. Zugriffe auf im Speicher benachbarte oder nahe beieinanderliegende Elemente, da durch Systemeinrichtungen wie Prefetching diese bereits in den Cache vorgeladen sein können, ohne dass sie zuvor adressiert wurden, was somit in einem Zugriff auf ein Cache-Element statt auf den Haupt- oder Hintergrundspeicher bewirkt (siehe Kapitel 2.2). Außerdem bewirkt ein linearer Zugriff auf den Speicher einen sehr guten Ladefluss der Daten.

Neben der natürlichen Blockung in rechteckige Tiles, sind aber auch andere Formen möglich und oft sogar nötig, um ein gültiges Tiling zu finden.

Ein Tiling eines für ein Statement S relevanten Indexraumes kann durch die begrenzenden Hyperebenen der Tiles beschrieben werden. Die Hyperebenen wiederum lassen sich durch ihre Normalen ϕ_S^i darstellen.

Lemma 3.35. *Ein vollständiges Tiling eines Statements S mit d-dimensionalem Indexraum lässt sich durch $\{\phi_S^1, \ldots, \phi_S^d\}$ Hyperebenen mit $\phi_S^i \in \mathbb{Q}^d \ \forall i = 1, \ldots, d$ beschreiben.*

Bemerkung 3.36. *$\phi_S^i \in \mathbb{Q}^d$ kann durch $\phi_S^i \in \mathbb{Z}^d$ ersetzt werden, indem der Normalenvektor mit dem kgV der Nenner multipliziert wird,* ***allerdings nur****, wenn der Normalenvektor* ***lediglich als Richtungsvektor*** *interpretiert wird. Ansonsten ist diese quantitative Veränderung* ***nicht*** *zulässig. Eine Erweiterung auf $\mathbb{R}^d$ ist hier nicht notwendig oder sinnvoll.*

Die ***Clustering*-Matrix** $G_S \in \mathbb{Z}^{d\times d}$ des Tilings eines Statements S enthält *spaltenweise* die **Hyperebenen**, welche das Tile begrenzen, also

$$G_S = \begin{pmatrix} \psi_S^1 & \dots & \psi_S^d \end{pmatrix} = \begin{pmatrix} g_{1,1}^S & \dots & g_{1,d}^S \\ \vdots & \ddots & \vdots \\ g_{d,1}^S & \dots & g_{d,d}^S \end{pmatrix} \quad . \tag{3.18}$$

Diese Hyperebenen entsprechen auch **quantitativ** den begrenzenden Hyperebenen.
Um ein vollständiges Tiling eines d-dimensionalen Indexraumes zu erhalten, muss $G_S \in \mathbb{Z}^{d\times d}$ invertierbar sein. Dann und nur dann entspricht die Vereinigung aller Tiles dem ursprünglichen Indexraum. Dies wird im Folgenden angenommen.

Die **Tiling-Matrix** $H_S = G_S^{-1} \in \mathbb{Q}^{d\times d}$ enthält *zeilenweise* die **Hyperebenennormalen**, also

$$H_S = G_S^- 1 = \begin{pmatrix} \phi_S^1 \\ \vdots \\ \phi_S^d \end{pmatrix} = \begin{pmatrix} h_{1,1}^S & \dots & h_{1,d}^S \\ \vdots & \ddots & \vdots \\ h_{d,1}^S & \dots & h_{d,d}^S \end{pmatrix} \tag{3.19}$$

Beweis. Da $H_S = G_S^{-1}$ gilt: $H_S \cdot G_S = \begin{pmatrix} \phi_S^1 \\ \vdots \\ \phi_S^d \end{pmatrix} \cdot \begin{pmatrix} \psi_S^1 & \dots & \psi_S^d \end{pmatrix} = \mathcal{I}_d$

d. h. :
$$\phi_S^i \cdot \psi_S^i = 0 \quad \forall i = 1, \dots, d$$
$$\phi_S^i \cdot \psi_S^j = 1 \quad \forall i \neq j$$

Somit ist jedes $\phi_S^{i'}$ orthogonal zu allen ψ_S^i mit $i \neq i'$ und somit die Normale zu der von diesen definierten Hyperebene. □

Im Folgenden wird mit Φ_S die Menge aller Tiling-Hyperebenen für Statement S bezeichnet, also $\Phi_S = (\phi_S^1, \dots, \phi_S^d)^T$, äquivalent zu H_S.

Bemerkung 3.37. *Sollen nur einige Schleifen $d' < d$ einer verschachtelten Schleife geblockt werden, so können die entsprechend anderen $(d - d')$ Schleifen einfach in der gesamten Schleifenbreite (quasi in einen Block) geblockt werden.*

Es werden also bei der Suche eines Tilings einer d-fach verschachtelten Schleife mit T Statements für jedes der T Statements d Hyperebenen $\{\phi^1, \dots, \phi^d\}$ gesucht, bzw. für jede der d Dimensionen T Hyperebenen $\{\phi_{S_1}, \dots, \phi_{S_T}\}$.

Bemerkung 3.38. *Gibt es im Folgenden in einem betrachteten Kontext nur **ein** Statement, so wird der Exponent bzw. der Index S weggelassen.*

3.8.2 Gültige Tile-Form

Für Abbildung 3.25 ist $G = \begin{pmatrix} q_i & 0 \\ 0 & q_j \end{pmatrix}$ (*Clustering*) und $H = \begin{pmatrix} \frac{1}{q_i} & 0 \\ 0 & \frac{1}{q_j} \end{pmatrix}$ (Tiling). Dies entspricht einem rechteckigen Tiling, denn:

Lemma 3.39. *Ein Tiling ist **rechteckig**, dann und nur dann, wenn G (und somit H) Diagonalmatrizen sind. In diesem Fall gilt*

$$\begin{aligned} g_{i,i} &= \tfrac{1}{h_{i,i}} \qquad \forall i = 1, \ldots, d \quad \textit{und} \\ g_{i,j} &= h_{i,j} = 0 \quad \forall i \neq j \quad . \end{aligned}$$

Rechteckige Tilings sind allerdings nicht immer möglich, da sie durch die Umordnung der Indexreihenfolge (Abbildung 3.26) auf diese Weise die Abhängigkeiten verletzen können. Dies kann entweder bedeuten, dass gar kein gültiges Tiling beliebiger Form existiert (Abbildung 3.27) oder dass lediglich kein gültiges rechteckiges Tiling existiert, aber eine andere Form ein gültiges Tiling ermöglichen würde (Abbildung 3.28, grüne Tiles). Dementsprechend muss eine Bedingung gefasst werden, wann ein Tiling (bzw. seine charakterisierenden Hyperebenen) gültig ist.

Lemma 3.40. *Mengen von Tiling-Hyperebenen $\{\Phi_{S_1}, \ldots, \Phi_{S_T}\}$ für eine verschachtelte Schleife mit Statements $S_1, \ldots, S_T$ sind **gültig**, dann und nur dann, wenn für jede Kante $e^{S_i \to S_j}$ im DDG, also für jede Abhängigkeit zwischen Statements, gilt*

$$\Phi_{S_j}(\vec{t}) - \Phi_{S_i}(\vec{s}) \geq \vec{0} \quad \forall (\vec{s}, \vec{t}) \in \mathcal{P}_{e^{S_i \to S_j}} \quad (\textit{komponentenweise})$$

Beweiserläuterung. Für einen vollständigen Beweis, siehe [Bon08, Theorem 1] der auf dem Beweis für eindimensionale Tiling-Hyperebenen aus [IT88] beruht.
Die Bedingung beinhaltet, dass alle Abhängigkeiten zwischen zwei Statements nur in *einer* Richtung, und zwar (in allen Dimensionen) in 'positive' Richtung über jede der Hyperebenen, von einem Tile in ein anderes übergehen. Dadurch sind die Tiles untereinander in lexikographischer Reihenfolge ausführbar. Gilt beispielsweise für zwei Tiles B_1 und B_2 mit $\vec{s} \in B_1$ und $\vec{t} \in B_2$ und $\vec{s}\delta\vec{t}$, so muss B_1 **vor** B_2 ausgeführt werden, da Daten aus B_1 in B_2 benötigt werden. Existieren nun auch $\vec{s'} \in B_2$ und $\vec{t'} \in B_1$ und $\vec{s'}\delta\vec{t'}$, so müsste außerdem B_2 **vor** B_1 ausgeführt werden, was sich widerspricht. □

Für ein gültiges Tiling muss Lemma 3.40 gelten, da Grundvoraussetzung ist, dass die Tiles **autark** sind, d. h. dass sie nur zu Beginn und am Ende ihrer Ausführung Daten

mit anderen Tiles kommunizieren müssen. Findet Kommunikation zwischen zwei Tiles in beide Richtungen statt, so sind diese **wechselseitig abhängig** (Abbildung 3.28, rote Tiles). Findet die Kommunikation über eine Hyperebene zwar nur in eine Richtung statt, aber in 'negative', so ist die lexikographische Ausführungsreihenfolge der Tiles nicht gültig. Dieses Problem kann z. B. durch Schleifenumkehr gelöst werden (Abbildung 3.29, hier würde eine Schleifenumkehr der j-Schleife das Tiling gültig machen).

Sind Tiling-Hyperebenen nach Lemma 3.40 für alle d Dimensionen und alle T Statements gefunden (und somit Tiling-Matrizen für alle Statements), so sind die Statements entlang diesen gültig blockbar, bzw. in den durch diese Hyperebenen transformierten Abhängigkeitspolyedern rechteckig blockbar (Bemerkung 3.44).

Korollar 3.41. *Die Transformation von Statements $\{S_1, \ldots, S_T\}$ durch Tiling-Hyperebenen ist eine **gültige** Transformation.*

Beweis. Es gilt nach Lemma 3.40 komponentenweise $\Phi_{S_j}(\vec{t}) - \Phi_{S_i}(\vec{s}) \geq \vec{0}\ \forall e^{S_i \to S_j}$.
Fall 1: $S_i = S_j$
Somit ist $\Phi_{S_i} = \Phi_{S_j}$, im Folgenden einfach Φ_S. Φ_S ist (äquivalent zu H_S) nichtsingulär. Betrachte den Grenzfall $\Phi_S(\vec{t}) - \Phi_S(\vec{s}) = \vec{0}$. Es gilt also $\Phi_S(\vec{t}) = \Phi_S(\vec{s}) \Leftrightarrow \vec{t} = \vec{s}$. Allerdings können für eine 'echte' Abhängigkeit $\vec{s} \to \vec{t}$ *source* und *target* nicht identisch sein. Also folgt $\Phi_S(\vec{t}) - \Phi_S(\vec{s}) \geq \vec{0}$ und $\Phi_S(\vec{t}) - \Phi_S(\vec{s}) \neq \vec{0}\ \forall e^{S \to S}$, womit $\Phi_S(\vec{t}) - \Phi_S(\vec{s}) \succ \vec{0}$ gilt und Lemma 3.17 erfüllt ist.
Fall 2: $S_i \neq S_j$
Hier *können* im Grenzfall Abhängigkeiten $e^{S_i \to S_j}$ über verschiedene Statements durch unterschiedliche Transformationen $\Phi_{S_i} \neq \Phi_{S_i}$ auf denselben Iterationspunkt abgebildet werden. Ist dies der Fall, so können die Statements im Inneren der Schliefe untereinander in der entsprechenden Reihenfolge abgelegt werden, sodass die Abhängigkeiten respektiert werden. Dies ist immer möglich, da dies auch im ursprünglichen Quellcode entsprechend geordnet war (es können nicht $e^{S_i \to S_j}$ **und** $e^{S_j \to S_i}$ existieren, der so entstehende DDG wäre insbesondere nicht kreisfrei, es müsste S_i vor S_j und gleichzeitig S_j vor S_i ausgeführt werden → Wsp.) □

Bemerkung 3.42. *Für eine gültige Tile-Form ist jede Tile-Größe gültig, da die Gültigkeit entlang der unbeschränkten Hyperebenen vorausgesetzt wurde.*

Beispiel 3.43. *Für Abbildung 3.27 mit Abhängigkeitsvektor $\vec{v} = \begin{pmatrix} 1 \\ N-1-2*j \end{pmatrix}$ kann es kein zweidimensionales Tiling geben, da*

$$\Phi(\vec{t}) - \Phi(\vec{s}) = \Phi(\vec{v}) = \begin{pmatrix} h_{1,1} & h_{1,2} \\ h_{2,1} & h_{2,2} \end{pmatrix} \cdot \begin{pmatrix} 1 \\ N-1-2*j \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \end{pmatrix} \ \forall 0 \leq j \leq N \quad (3.20)$$

aufgrund der enthaltenen nicht-gleichförmigen Abhängigkeit nicht allgemein (also durch eine konstante Matrix) erfüllbar ist.

Für Abbildung 3.28 mit Distanzvektoren $\vec{d_1} = \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \vec{d_2} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \vec{d_1} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ *ist kein rechteckiges zweidimensionales Tiling gültig, da aus*

$$\Phi(\vec{t}) - \Phi(\vec{s}) = \Phi(\vec{d_1}) = \begin{pmatrix} h_{1,1} & 0 \\ 0 & h_{2,2} \end{pmatrix} \cdot \begin{pmatrix} 1 \\ -1 \end{pmatrix} \overset{!}{\geq} \begin{pmatrix} 0 \\ 0 \end{pmatrix} \quad (3.21)$$

$$\wedge \quad \Phi(\vec{t}) - \Phi(\vec{s}) = \Phi(\vec{d_2}) = \begin{pmatrix} h_{1,1} & 0 \\ 0 & h_{2,2} \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} \overset{!}{\geq} \begin{pmatrix} 0 \\ 0 \end{pmatrix} \quad (3.22)$$

$$\wedge \quad \Phi(\vec{t}) - \Phi(\vec{s}) = \Phi(\vec{d_3}) = \begin{pmatrix} h_{1,1} & 0 \\ 0 & h_{2,2} \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix} \overset{!}{\geq} \begin{pmatrix} 0 \\ 0 \end{pmatrix} \quad (3.23)$$

folgt, dass $h_{1,1} \geq 0$ *und* $h_{2,2} \leq 0 \wedge h_{2,2} \geq 0 \Rightarrow h_{2,2} = 0$*, was also ein zweidimensionales Tiling verhindert. Dies ist in der Abbildung durch die wechselseitigen Abhängigkeiten durch* d_1 *(3.21) und* d_3 *(3.23) zwischen den roten Tiles (Abbildung 3.28) sichtbar.*

Für Abbildung 3.29 mit Distanzvektor $\vec{d} = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$ *ist ein direktes rechteckiges Tiling mit* ***positiven*** $h_{1,1}, h_{2,2}$ *nicht möglich, da aus*

$$\Phi(\vec{t}) - \Phi(\vec{s}) = \Phi(\vec{d}) = \begin{pmatrix} h_{1,1} & 0 \\ 0 & h_{2,2} \end{pmatrix} \cdot \begin{pmatrix} 1 \\ -1 \end{pmatrix} \overset{!}{\geq} \begin{pmatrix} 0 \\ 0 \end{pmatrix} \quad (3.24)$$

folgt, dass $h_{2,2}$ *für ein gültiges Tiling negativ sein muss. Dies bedeutet eine Kombination einer Tiling-Matrix* $\tilde{H} = \begin{pmatrix} \tilde{h}_{1,1} & 0 \\ 0 & \tilde{h}_{2,2} \end{pmatrix}$ *mit* $h_{1,1}, h_{2,2} > 0$ *mit einer Schleifenumkehr der* j*-Schleife mit Transformationsmatrix* $U = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$*, somit* $H = U \cdot \tilde{H}$*.*

Allgemein lassen sich durch Nicht-Diagonalmatrizen mit Rang d in einem d-dimensionalen Indexraum, neben den rechteckigen Tiles, beliebige d-dimensionale ***Parallelepipeds*** (Parallelflächner) zum Tiling verwenden.

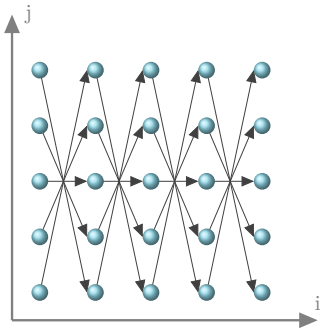


Abbildung 3.27: Kein 2-D Tiling möglich

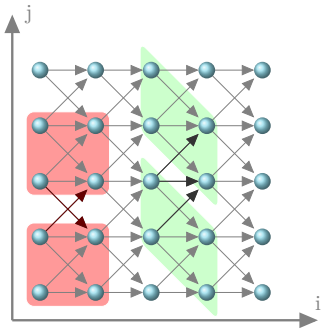


Abbildung 3.28: Tiling legal (grün) und illegal (rot)

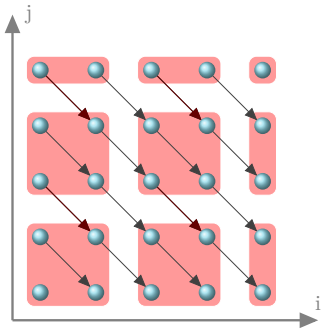
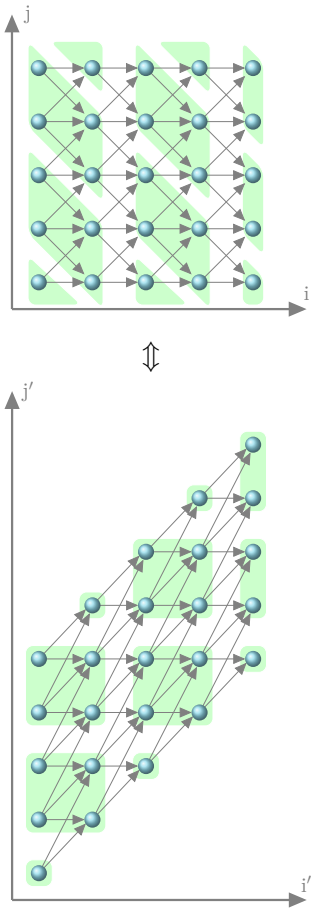


Abbildung 3.29: Tiling illegal **ohne** Schleifenumkehr

Bemerkung 3.44. *Ein nicht-rechteckiges Tiling eines Statements S durch Tiling-Hyperebenen $\{\phi_S^1, \ldots, \phi_S^d\}$ im Quell-Indexraum ist äquivalent zu einem rechteckigen Tiling im durch $\{\phi_S^1, \ldots, \phi_S^d\}$ transformierten Ziel-Indexraum. Somit vereint ein Tiling allgemeiner Form die Blockung des Indexraumes mit der dazu ggf. nötigen Transformation.*
Dies ist der Tatsache geschuldet, dass die d-dimensionale Clustering-Matrix die Matrix aufspannender Vektoren eines d-dimensionalen Parallelepipeds ist und folgender Satz gilt.



Satz 3.45. *Jedes n-dimensionale Parallelepiped P ist ein affines Bild des Einheitswürfels E.*

Beweisidee. Analog zum Gaußschen Eliminationsverfahren kann man jede nichtsinguläre Matrix durch elementare Zeilen- (oder Spalten-) Transformationen auf Diagonalgestalt, und somit auch auf die Einheitsmatrix, bringen. Dabei handelt es sich um affine Transformationen. □

Somit ist das Bild einer affinen Transformation eines d-dimensionalen Quaders ein d-dimensionales Parallelepiped $\mathcal{T}(Q) = P$ und jedes d-dimensionale Parallelepiped das Bild einer affinen Transformation eines d-dimensionalen Quaders (da jeder d-dimensionale Quader wiederum eine d-dimensionale Transformation des Einheitswürfel durch eine 'streckende' Diagonalmatrix ist).

3.8.3 Parallelisierbare Tiles

Zur Generierung von parallelem Code durch Tiling, müssen gesamte Tiles voneinander unabhängig ausgeführt werden. Welche Tiles parallel ausgeführt werden können, lässt sich auf Grundlage des Tile-Space-Graphen **TSG** feststellen, in welchem jedes Tile B_i als Knoten repräsentiert ist und eine Kante $B_i \rightarrow B_j$ vorhanden ist, falls B_j zur Ausführung Daten von B_i benötigt, also Kommunikation entlang dieser Kante stattfinden muss. Dieser Graph muss auf Grundlage eines gültigen Tilings kreisfrei sein, da keine wechselseitigen Abhängigkeiten bestehen. Die durch die parallele Ausführung verschiedener Tiles generierte Parallelität lässt sich als **grobe Parallelität** (*coarse-grained parallelism*) bezeichnen. Außerdem können nach dem Tiling (zusätzlich) Schleifen im Inneren der Tiles als *parallel ausführbare Schleifen* markiert werden (u. A. Abschnitt 3.7.5), was z. B. für die innere Vektorisierung nötig ist. Da die Methoden für *grobe* Parallelität in dieser Arbeit nicht weiterentwickelt wurden, wird hier nicht näher auf *diese* eingegangen. *Für Details und die mathematische Formalisierung, siehe [Xue00, Kap. 5].*

3.8.4 Optimale Tile-Form

Neben der Erfüllung der Abhängigkeiten können verschiedene Tile-Formen auch verschieden vorteilhaft in Hinblick auf die zwischen Tiles nötige Kommunikation (dem **Kommunikationsvolumen**) sein.
So ist in Abbildung 3.28 kein rechteckiges Tiling möglich, allerdings ist 'das' gültige Parallelepiped-Tiling **nicht eindeutig** bestimmt. Somit sollten über die Gültigkeit hinaus Kriterien gefunden werden, welches gültige Tiling bzw. welche gültige Hyperebenenkonfiguration die geringste Kommunikation verursacht.
Des Weiteren ist die Minimierung der **Wiederverwendungsdistanz** (*Reuse-Distance*) ein, für die *Laufzeit* entscheidendes, Kriterium. Dies entspricht der Minimierung der Länge der Abhängigkeits-'Pfeile' im Abhängigkeitspolyeder und kann *grundsätzlich* zusammen mit dem Kommunikationsvolumen berücksichtigt werden (s. Kapitel 3.9). Die Wiederverwendungsdistanz (bzgl. einer zu spezifizierenden *Norm*) gibt also an, in welchem 'Abstand' im Indexraum ein berechnetes Datenelement das nächste Mal benötigt wird. Je geringer diese Distanz (bei Wahl einer geeigneten Norm) ist, umso wahrscheinlicher ist es, das Element noch im Cache zu finden.

Bemerkung 3.46. *Da bei verschachtelten Schleifen die Schleifen von außen nach innen durchlaufen werden, also z. B. eine Abhängigkeit von Indexpunkt (i,j) auf (i,j+128) einen deutlich kleineren Sprung als eine von (i,j) auf (i+1,j) (bei ausreichender Quantität der*

Dimensionen) bedeutet, wäre die direkte lexikographische Minimierung der Distanzvektoren ein in dieser Hinsicht gutes Optimierungsziel.

Beispiel 3.47.

```
1 for(i=1; i<T; i++)
2   for(j=1; j<N-1; j++)
3     a[i][j]=0.333*(a[i-1][j]+a[i-1][j-1]+a[i-1][j+1]);
```

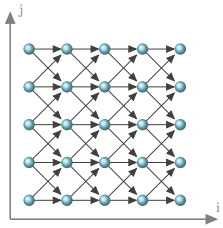


Abbildung 3.30: 1-D perfekt verschachtelter Jacobi

Für den Code aus Abbildung 3.30 sind unter anderem die drei verschiedenen Tilings mit $H_1 = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$ *(Abbildung 3.31),* $H_2 = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ *(Abbildung 3.32) oder* $H_3 = \begin{pmatrix} 2 & 1 \\ 1 & 0 \end{pmatrix}$ *(Abbildung 3.33) gültig (diese beschreiben das Tiling* ***ohne*** *konkrete Größe und sind lediglich als formbestimmend oder als* 1×1*-Blöcke zu verstehen).*

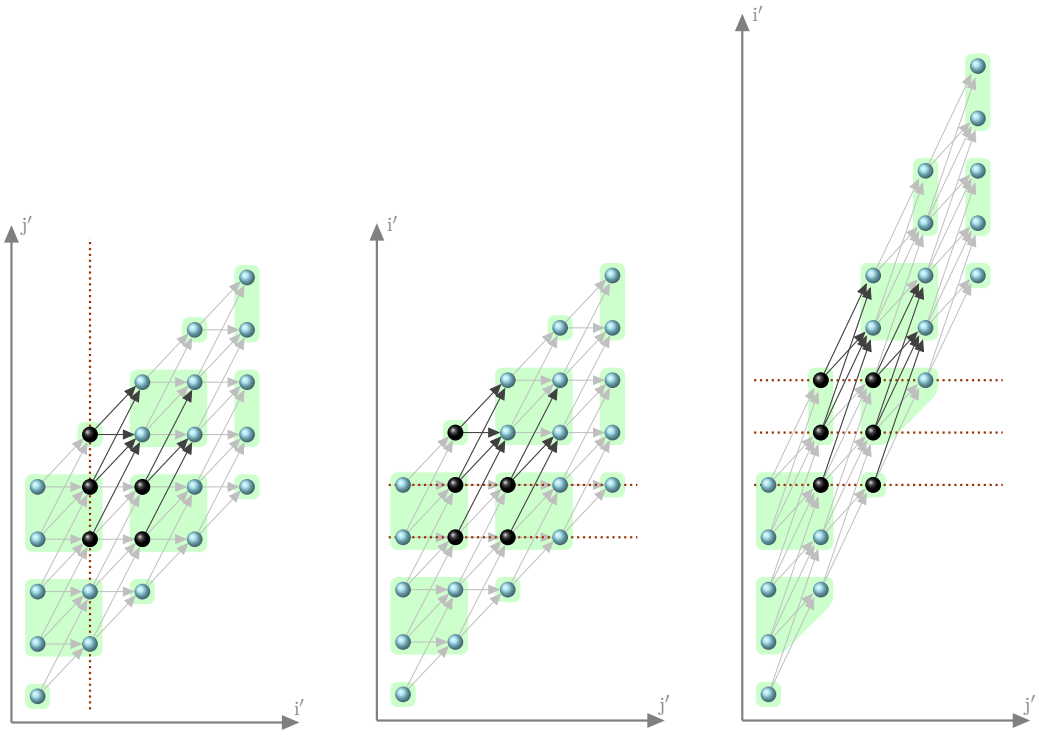


Abbildung 3.31: Hyperebenen $\phi^1 = (1,0)$, $\phi^2 = (1,1)$

Abbildung 3.32: Hyperebenen $\phi^1 = (1,1)$, $\phi^2 = (1,0)$

Abbildung 3.33: Hyperebenen $\phi^1 = (2,1)$, $\phi^2 = (1,0)$

Das Kommunikationsvolumen ist durch H_1 und H_2 geringer als durch H_3 (die für das ausgewählte Tile zu kommunizierenden Elemente sind durch die dunklen Kugeln und Pfeile gekennzeichnet). Die verschiedenen Wiederverwendungsdistanzen, welche aus den Distanzvektoren d_1, d_2, d_3 durch die Transformation entstehen und durch eine geeignete Norm durch $||\mathcal{T}(d_i)||$ gemessen werden können, sind durch die transformierten Abhängigkeiten (Pfeile) und deren Zugriff in verschiedene Dimensionen sichtbar.
*Außerdem sind für die verschiedenen Tilings verschieden viele Kommunikationslinien (lines of communication, vgl. [Bon08, Kapitel 3.6]) vorhanden, die beschreiben, auf wie viele vergangene Hyperebenen in einer **äußeren** Dimension für die Kommunikation zwischen zwei Tiles zurückgegriffen werden muss. Je weiter diese zurückliegen, desto unwahrscheinlicher ist es, die entsprechenden Daten im Cache zu finden (Cache-Hit).*

Bemerkung 3.48. *Die Tiles im transformierten Indexraum sind entlang beider Dimensionen i' und j' im Sinne der groben Parallelität nicht nativ parallelisierbar. Allerdings kann jedes Tile zu seinen linken-oberen und rechten-unteren Nachbarn parallel ausgeführt werden. Im Inneren der Tiles kann bei H_1 j' parallelisiert (oder vektorisiert) werden, bei H_2 und H_3 jeweils i' parallelisiert. Inwiefern verschiedene Tiles parallel ausführbar sind kann über den TSG geprüft werden, hier z. B. durch verzögerte Parallelität der Tiles untereinander.*

3.8.5 Optimale Tile-Größe

Die Anpassung der Tile-Größe an gegebenene Speicherbedingungen ist ein Hauptaspekt der in dieser Arbeit entwickelten Methode, weshalb in Kapitel 4 auf diesen Aspekt noch detailliert eingegangen wird.
An dieser Stelle seien noch die folgenden Beobachtungen festgehalten.

Lemma 3.49 (Satz 15 [Kön04])**.** *Das Volumen eines N-dimensionalen Parallelepipeds P, welches von den Vektoren $\{v_1, \ldots, v_N\}$ aufgespannt wird, ist gleich der Determinante der von diesen Spaltenvektoren gebildeten Matrix, also:*

$$Vol(P) = |det(v_1| \ldots |v_N)| \tag{3.25}$$

Im hier vorgestellten Kontext entspricht das Volumen der gebildeten Tiles also der Determinante der *Clustering*-Matrix $Vol(P) = |det(G)|$. Dieses Volumen entspricht damit der **durchschnittlich enthaltenen** ganzzahligen Punkte in einem durch G aufgespannten Parallelepiped (dem Berechnungsvolumen [*computation volume*] eines Tiles) und somit

der Anzahl der in einem solchen Tile adressierten Indexpunkte. Ist nun $det(G) \in \mathbb{Z}$ (bzw. $det(H^{-1}) \in \mathbb{Z}$), so ist diese Anzahl **sogar exakt**.

Bemerkung 3.50. *Somit ist das Berechnungsvolumen (die exakte Anzahl der in jedem Tile adressieren Indexpunkte) bei rechteckigem Tiling mit ganzzahligen Seitenlängen:*

$$Vol(R) = \left| det \begin{pmatrix} r_{1,1} & 0 & \dots & 0 \\ 0 & \ddots & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \dots & 0 & r_{d,d} \end{pmatrix} \right| = \left| \prod_{i=1}^{d} r_{i,i} \right| \quad . \tag{3.26}$$

Des Weiteren ist es auch exakt für unimodular transformierte, rechteckige Tilings mit ganzzahligen Seitenlängen, da $det(\mathcal{T}(R)) = det(T \cdot R) = det(T) \cdot det(R) = det(R) \in \mathbb{Z}$.

Ist $det(G) \notin \mathbb{Z}$, so variiert die Anzahl der ganzzahligen Punkte in den Tiles mit Durchschnittsanzahl $det(G)$.

3.8.6 Kommunikationsminimales Tiling nach Xue

In [Xue00, Kap. 6] wird ein Ansatz zur Generierung eines kommunikationsminimalen Tilings beschrieben, der dem hier verwendeten Ansatz nach [Bon08], der im folgenden Abschnitt ausführlich beleuchtet wird, ähnelt und kurz skizziert werden soll.

Sei D die Matrix, welche *spaltenweise* die Distanzvektoren d_i enthält.

Bei *Xue* werden für gegebene Distanzvektoren $d_1, \dots, d_M$ (also für **gleichförmige** Abhängigkeiten) einer **perfekt verschachtelten** Schleife zwei Größen miteinander in Verbindung gebracht, nämlich zum einen das **Berechnungsvolumen**

$$V_{comp}(H) = \frac{1}{|det(H)|} = |det(G)| \quad , \tag{3.27}$$

das der Menge an Indexpunkten in einem Tile mit Tiling-Matrix H entspricht, und zum anderen eine Approximation des **Kommunikationsvolumens**

$$\begin{aligned} V_{comm}(H) &\approx \frac{1}{|det(H)|} \sum_{i=1}^{N} \sum_{j=1}^{M} \vec{\phi}^i \cdot \vec{d}_j \\ &= \frac{1}{|det(H)|} \sum_{i=1}^{N} \sum_{j=1}^{M} (H \cdot D)_{i,j} \end{aligned} \tag{3.28}$$

die der Anzahl der Hyperebenen-kreuzenden Abhängigkeitsvektoren entspricht (Herleitung dazu in [Xue00, Kap. 6.1]).

Nun können kommunikationsminimale Hyperebenen ϕ_i bei vorgegebenem Inhalt eines Tiles $V_{comp}(H) = \upsilon$ (beispielsweise angepasst an die Cache-Größen) durch Lösung des folgenden Optimierungsproblems gefunden werden:

$$\begin{aligned} \min \quad V_{comm}(H) &= \frac{1}{|det(H)|} \sum_{i=1}^{N} \sum_{j=1}^{M} (H \cdot D)_{i,j} \\ s.t. \quad V_{comp}(H) &= \frac{1}{|det(H)|} = \upsilon \\ H \cdot D &\geq 0 \end{aligned} \tag{3.29}$$

$H \cdot D \geq 0$ entspricht der Gültigkeitsbedingung für Tiling-Hyperebenen aus Lemma 3.40 bei gleichförmigen Abhängigkeiten.
Da außerdem $\frac{1}{|det(H)|} = \upsilon$ vorgegeben und somit konstant ist, kann die Zielfunktion in $min\, V_{comm}(H) = \mathcal{F}(H) = \sum_{i=1}^{N} \sum_{j=1}^{M} (H \cdot D)_{i,j}$ in (3.29) mit resultierendem Optimalwert $\upsilon * \mathcal{F}(H)$ vereinfacht werden, wodurch die Zielfunktion linear wird. Lösungsmethoden zur Ermittlung von H für dieses Optimierungsproblem sind in [Xue00, Kap. 6] beschrieben.

Beispiel 3.51. *Um die approximative Eigenschaft von (3.28) zu erläutern, wird nun exemplarisch der Code aus Abbildung 3.30 mit* $G = \begin{pmatrix} 4 & 0 \\ -4 & 4 \end{pmatrix}$ *bzw.* $H = G^{-1} = \begin{pmatrix} \frac{1}{4} & 0 \\ \frac{1}{4} & \frac{1}{4} \end{pmatrix}$ *geblockt betrachtet. Nach (3.28) gilt*

$$\begin{aligned} V_{comm}(H) &\approx \frac{1}{|\frac{1}{16}|} \sum_{i=1}^{N} \sum_{j=1}^{M} \left(\begin{pmatrix} \frac{1}{4} & 0 \\ \frac{1}{4} & \frac{1}{4} \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & -1 \end{pmatrix} \right)_{i,j} \\ &= 16 * \sum_{i=1}^{N} \sum_{j=1}^{M} \left(\begin{pmatrix} \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{2} & \frac{1}{4} & 0 \end{pmatrix} \right)_{i,j} \\ &= 16 * \frac{3}{2} \\ &= 24 \end{aligned}$$

Die tatsächliche Anzahl zu kommunizierender Elemente (bzw. die Anzahl von Iterationspunkten in denen zu kommunizierende Daten generiert werden) ist 12*, wie in Abbildung 3.34 (schwarze Indexpunkte) zu sehen ist. Die Abweichung wird im Folgenden erklärt.*

Die relevanten Abhängigkeiten der drei Distanzvektoren $d_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ *(gepunktet),* $d_2 \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ *(durchgezogen) und* $d_3 = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$ *(gestrichelt) können auch einzeln betrachtet werden, entsprechend ihrer Spalten in D. Also ist die Anzahl der Kreuzungen zwischen* ϕ_i *und* d_j *durch* $\frac{1}{|det(H)|}\vec{\phi}_i \cdot \vec{d}_j$ *approximierbar. Sei* $\phi_1 = \begin{pmatrix} \frac{1}{4} & 0 \end{pmatrix}$ *und* $\phi_2 = \begin{pmatrix} \frac{1}{4} & \frac{1}{4} \end{pmatrix}$*, mit* $|det(H)| = \frac{1}{16}$ *ergibt sich somit:*

d_1 *verursacht* ***4*** *Kreuzungen mit* ϕ_1 *und* ***8*** *Kreuzungen mit* ϕ_2*. (tatsächlich* ***7****)* [1]

d_2 *verursacht* ***4*** *Kreuzungen mit* ϕ_1 *und* ***4*** *Kreuzungen mit* ϕ_2*.*

d_3 *verursacht* ***4*** *Kreuzungen mit* ϕ_1 *und* ***0*** *Kreuzungen mit* ϕ_2*.*

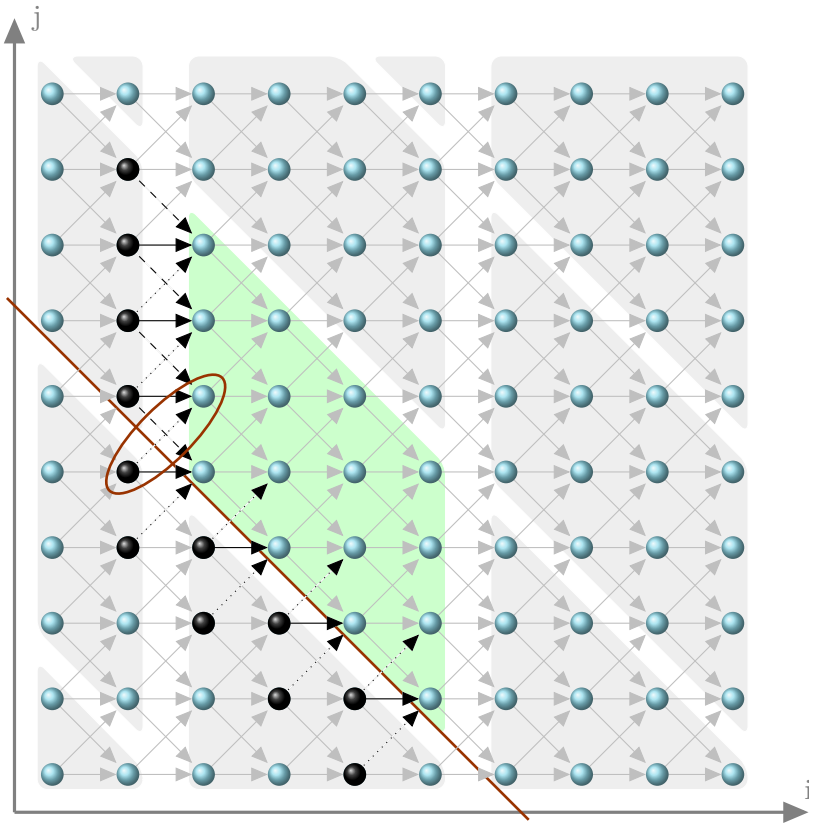
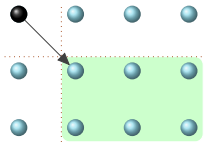


Abbildung 3.34: Kommunikationsvolumen für den 1-D Jacobi

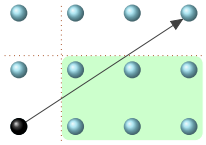
[1] Diese Ungenauigkeit ist darauf zurückzuführen, dass die rot eingekreiste Abhängigkeit auch als ϕ_2 kreuzend mitgezählt wird (da sie die Hyperebene (rote Linie) kreuzt), obwohl sie zwischen den zwei Tiles liegt. Deshalb wird hier durch die Approximation pro Tile eine Kreuzung zu viel durch V_{comm} gezählt. Die Abweichung aufgrund **dieses** Effektes ist aber allgemein gering.

*Insgesamt sind dies laut Abschätzung genau die insgesamt berechneten **24** Kreuzungen. Neben der bereits beobachteten geringen Abweichung, ist die Differenz zum echten Wert der zu kommunizierenden **Daten** der **Indexpunkte** (hier **12** [schwarze Kugeln]) allgemein durch die folgenden Beobachtungen zu erklären:*

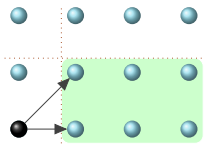
- *Abhängigkeiten, welche auf eine Ecke treffen, kreuzen alle sich dort treffenden Hyperebenen und werden auch entsprechend mehrfach gezählt, obwohl sie nur eine Abhängigkeit als Ursprung haben.*



- *Abhängigkeiten, die mehrere Hyperebenen überqueren und ggf. gar nicht mehr im Tile enden, werden auch entsprechend mehrfach gezählt, obwohl sie gar nicht relevant sind.*



- *Es werden die kreuzenden Abhängigkeiten gezählt, nicht aber die Indexpunkte denen sie entspringen und somit wird auch hier wieder Vieles mehrfach gezählt, falls aus Indexpunkten diverse kreuzende Abhängigkeiten entspringen.*



Somit entsteht häufig eine Überapproximation des echten Wertes, die aber proportional zu diesem ist, wodurch eine Minimierung dieses Wertes trotzdem ein grundsätzlich richtiger Ansatz ist. Exaktere Lösungen zur Kommunikationsminimierung sind in [Xue00, Kap. 6.6] referenziert.

Bemerkung 3.52. *Verschiedene Ansätze zu zeitoptimalen Tile-Größen und -Formen sind außerdem in [Xue00, Kap. 7] beschrieben, welche Optimierungsprobleme bzgl. verschiedener zeitrelevanter Größen lösen.*

3.9 PluTo

PluTo [Bonb] ist ein von *U. Bondhugula* in seiner Dissertation [Bon08] (bis 2008) entwickeltes und u. A. zusammen mit A. Hartono, J. Ramanujam und P. Sadayappan [BHRS08] erweitertes Framework zur semi-automatisierten Parallelisierung verschachtelter Schleifen in *Fortran-* und *C*-Codes, im Folgenden wird sich in der Beschreibung auf *C*-Codes beschränkt.
PluTo basiert auf *Cloog-ISL* [Bas] als Codegenerator und auf *Clan* [Bas08] und Teilen von *LooPo* [GL96] zum Scan des vorliegenden Codes. Zur Lösung der Optimierungsprobleme wird Piplib [Fea] verwendet.
Neben dem Tiling zur Lokalitätsoptimierung werden äußere, innere und verzögerte Parallelität von Tiles zur groben Parallelität mittels OpenMP[2]-Pragmas ermöglicht und für den **ICC** können Pragmas zur Vektorisierung gesetzt werden (wobei die vektorisierbare Schleife ins Innerste der Verschachtelung verschoben wird (vgl. Abschnitt 3.7.6)). Des Weiteren sind Schleifenmanipulationen wie z. B. *Unrolling* oder *Schleifenvereinigungen* möglich.

3.9.1 Idee und Umsetzung

PluTo transformiert durch '#pragma scop' und '#pragma endscop' umgebene verschachtelte Schleifen in *C*-Codes und erzeugt somit aus dem Kontext eines lauffähigen *C*-Codes eine optimierte, lauffähige Version dessen. Somit kann PluTo als Source-To-Source-Compiler verwendet werden, der gewisse Anforderungen an die zu transformierenden *Scopes* stellt, sodass diese vom Scanner und Parser verarbeitet werden können. Sie müssen im Wesentlichen folgende Bedingungen erfüllen (Details in der Pluto Dokumentation):

- Zugriffe sind skalare Elemente oder Array-Zugriffe
- Zugriffsfunktionen sind affin und hängen von den umgebenden Schleifenindizes ab
- *if*-Bedingungen enthalten als Operanden ausschließlich affine Funktionen, welche von den umgebenden Schleifenindizes abhängen

Der Code wird dann grundsätzlich zur Optimierung von Datenlokalität transformiert und kann durch Einflussnahme des Benutzers geblockt sowie durch OpenMP-Pragmas im Code parallelisiert werden, für eine anschließende Vektorisierung können Pragmas

[2] http://openmp.org/wp/ (18.07.2011)

für den *ICC* an eine entsprechend nach innen verlagerte und vektorisierbare Schleife gesetzt werden. Die Transformationen finden statementweise statt und die verschachtelten Schleifen verschiedener Statements können, wenn möglich und vom Benutzer so konfiguriert, vereint werden (*fuse*).
Neben dem einfachen Tiling ist auch ein zweifaches solches möglich, wodurch die geblockten Schleifen ein zweites Mal geblockt werden, um z. B. eine Anpassung an den *L1*- und *L2*-Cache vorzunehmen.

3.9.2 Arbeitsweise

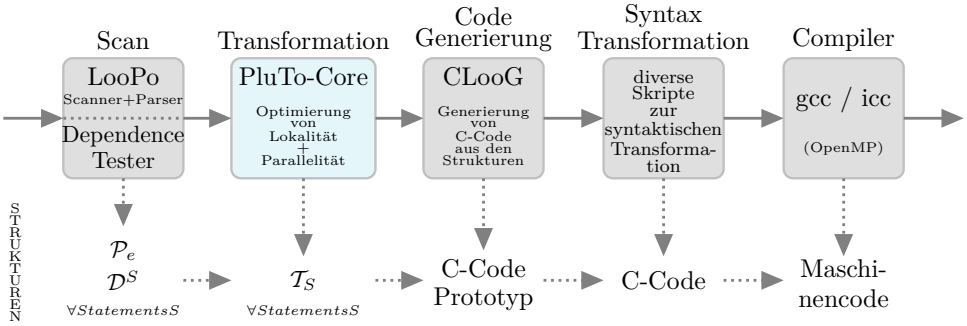


Abbildung 3.35: Ablauf eines Transformationsprozesses via PluTo

Der grundsätzliche Ablauf eines Transformationsprozesses in PluTo sowie die für diesen Kontext relevantesten Strukturen sind in Abbildung 3.35 dargestellt.

Bemerkung 3.53. *Die lexikographische Ordnung spielt in der Codeoptimierung (und somit auch in PluTo) eine besondere Rolle, u. A. da diese der Adressierungsreihenfolge in verschachtelten Schleifen entspricht. (s. a. Bem. 3.14 und Abschnitt 3.9.3)*

Der für das Tiling entscheidende Schritt ist die Generierung der Tiling-Hyperebenen [Bon08, Kap. 3], weshalb dieser Schritt nun explizit beschrieben wird.
In diesem Kapitel wird dieser Vorgang für Abhängigkeiten in *einem* Statement S (bzw. einen Block von Statements), also für *perfekt verschachtelte* Schleifen, beschrieben. Die Verallgemeinerung auf beliebig viele und beliebig verschachtelte Statements ist formell in Appendix A.1 (analog zu diesem Abschnitt) beschrieben, für Details siehe [Bon08, Kap. 3 und 4]. Sie ist für die hier erweiterten Elemente allerdings nicht essentiell.
Weil nun also **ein** Statement betrachtet wird, wird ϕ_S einfach als ϕ geschrieben.

Im Folgenden sei

$$\delta_e^k(\vec{s},\vec{t}) = \phi^k(\vec{t}) - \phi^k(\vec{s})$$

für $(\vec{s},\vec{t})$ Abhängigkeit aus $\mathcal{P}_e$ mit $e^{S\to S}$, also innerhalb des Statements S.

Gültigkeit In PluTo werden die Tiling-Hyperebenen ϕ^k (bzw. die entsprechenden Zeilen $\mathbf{h_k}$ der Tiling-Matrix H) eines Statements für jedes Level k iterativ gesucht, wobei für jede solche Hyperebene nach Lemma 3.40 die *Gültigkeitsbedingung* gelten muss:

$$\delta_e^k(\vec{s},\vec{t}) = \phi^k(\vec{t}) - \phi^k(\vec{s}) = \mathbf{h_k}\cdot\vec{t} - \mathbf{h_k}\cdot\vec{s} \geq 0 \quad \forall(\vec{s},\vec{t}) \in \mathcal{P}_e \tag{3.30}$$

Optimalität Um unter den gültigen Tilings ein bzgl. Kommunikationsvolumen und Wiederverwendung optimales Tiling zu finden, wird eine diese Größen beschränkende Ungleichung hinzugefügt, deren Schranke (Obergrenze) es in der folgenden Optimierung zu minimieren gilt. $\delta_e^k(\vec{s},\vec{t})$ eignet sich aus folgenden Gründen zum Messen beider Größen.
$\delta_e^k(\vec{s},\vec{t})$ entspricht zum einen der Wiederverwendungsdistanz in dieser Dimension k im Inneren der Tiles, zum anderen der Anzahl der von diesem Abhängigkeitsvektor 'überquerten' Hyperebenen (Kommunikationslinien), falls die Tiles, die an dieser Hyperebene aneinander treffen, hintereinander auf einer Recheneinheit gerechnet werden.
Werden diese Tiles *nacheinander* ausgeführt, so misst $\delta_e^k(\vec{s},\vec{t})$ grob die über die Hyperebene ϕ^k zu kommunizierende Datenmenge. Dies wird durch die Analogie zu (3.28) klar, da eine Beschränkung von $\delta_e^k(\vec{s},\vec{t})$ der Beschränkung der Zeilensummen von $H\cdot D$ (und somit der Anzahl von Kreuzungen zwischen dem Abhängigkeitsvektor zu $\mathbf{e}$ mit ϕ^k) unter Vernachlässigung des Faktors $\frac{1}{|det(H)|}$ entspricht.

Die Ungleichung zur *Volumenbeschränkung* (Kommunikationsvolumen und Wiederverwendungsdistanz) für Level k in PluTo lautet somit

$$\delta_e^k(\vec{s},\vec{t}) = \phi^k(\vec{t}) - \phi^k(\vec{s}) = \mathbf{h_k}\cdot\vec{t} - \mathbf{h_k}\cdot\vec{s} \leq \mathbf{u}\cdot\vec{p} + w = \nu(\vec{p}) \quad \forall(\vec{s},\vec{t}) \in \mathcal{P}_e \tag{3.31}$$

mit $u \in \mathbb{Z}^{N_p}$ und $w \in \mathbb{Z}$, wobei $\vec{p}$ der $N_{\vec{p}}$-dimensionale Vektor der Programmparameter bei **nicht-gleichförmigen Abhängigkeiten** ist, da der Abhängigkeitsvektor dann nicht konstant ist und ggf. nur über die Parameter abgeschätzt werden kann.

Beispiel 3.54.

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

$$\delta_e^k(\vec{s},\vec{t}) = \phi^k(\vec{t}) - \phi^k(\vec{s}) = \begin{pmatrix} h_{k,1} & h_{k,2} \end{pmatrix} \cdot \begin{pmatrix} j \\ i \end{pmatrix} - \begin{pmatrix} h_{k,1} & h_{k,2} \end{pmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix}$$
$$= (j-i) * h_{k,1} + (i-j) * h_{k,2}$$

Hier gilt $(i-j) \in [0, M]$ *und die Differenz kann somit nur durch den Programmparameter* M *abgeschätzt werden, nämlich durch* $u_1 * p_1 = u_1 * M$*. In dieser Abschätzung ist dann das Ziel, die Koeffizienten* u_i *der Programmparameter zu minimieren.*

Bei **gleichförmigen Abhängigkeiten** vereinfacht sich (3.31) zu:

$$\delta_e^k(\vec{s},\vec{t}) = \phi^k(\vec{t}) - \phi^k(\vec{s}) = \mathbf{h_k} \cdot \vec{t} - \mathbf{h_k} \cdot \vec{s} \leq w \tag{3.32}$$

Linearisierung bei nicht-gleichförmigen Abhängigkeiten Um eine Ungleichung für eine Kante $e \in E$ aus dem DDG bei nicht-gleichförmigen Abhängigkeiten zu linearisieren, wendet man das affine Farkas Lemma an, denn somit gilt nach (3.31)

$$\begin{aligned} \delta_e^k(\vec{s},\vec{t}) = \quad & \phi^k(\vec{t}) - \phi^k(\vec{s}) && \leq \mathbf{u} \cdot \vec{p} + w \quad \forall (\vec{s},\vec{t}) \in \mathcal{P}_e \\ \Rightarrow \quad & \mathbf{u} \cdot \vec{p} + w - (\phi^k(\vec{t}) - \phi^k(\vec{s})) && \geq 0 \qquad\quad \forall (\vec{s},\vec{t}) \in \mathcal{P}_e \quad . \end{aligned}$$

Daraus folgt mit Lemma 2.13:

$$\Rightarrow \quad \mathbf{u} \cdot \vec{p} + w - (\phi^k(\vec{t}) - \phi^k(\vec{s})) \equiv \lambda_0 + \sum_{l=1}^{M_e} \lambda_l {\mathcal{P}_e}^l \qquad \lambda_l \geq 0 \tag{3.33}$$

${\mathcal{P}_e}^l$ ist die LHS der l-ten Zeile des $\mathcal{P}_e$ beschreibenden Ungleichungssystems (also für das Ungleichungssystem in der Form $A \cdot x + b \geq 0$ die LHS von $a_i \cdot x + b_i \geq 0$, vgl. Lemma 2.13) und M_e die Anzahl der Ungleichungen dieses Systems.

Beispiel 3.55. *Die* ***echte Abhängigkeit*** $a[i'][j'] \leftrightarrow a[j][i]$ *aus Beispiel 3.54 würde durch* $\{i = j', j = i', 1 \leq i \leq M, 1 \leq j \leq M, i - j \geq 1\}$ *beschrieben werden und die* ${\mathcal{P}_e}^i$ *wären*

$$i - 1 \geq 0 \ , \quad M - i \geq 0 \ , \quad j - 1 \geq 0 \ , \quad M - j \geq 0 \ , \quad i - j - 1 \geq 0 \quad . \tag{3.34}$$

Das System in (3.33) kann nun durch Koeffizientenvergleich und anschließende Anwendung der Gaußelimination und durch FME zur Elimination der λ_i (wegen der Ungleichungen für λ_i) gelöst werden, was in einer Menge von Ungleichungen resultiert, welche lediglich von den $\{h_{k,1}, \ldots, h_{k,d}\}$ und $\{u_1, \ldots, u_{N_p}, w\}$ abhängen und somit eine für das Ungleichungssystem kompatible Form haben.
Ein Beispiel hierzu findet sich auch in [Bon08, Beispiel 3.11.1].

Nulllösung Da der *Nullvektor* als Lösung für jede Hyperebene unzulässig ist (dieser bildet keine gültige Hyperebene), muss darüber hinaus eine Ungleichung zur Vermeidung dieser hinzugefügt werden. Allerdings kann $\mathbf{h_k} \neq \vec{0}$ nicht als einzelner konvexer Raum beschrieben werden. Ein Ausweg aus diesem Problem ist, lediglich positive Koeffizienten $h_{k,i}$ zuzulassen und

$$\sum_{i=1}^{d} h_{k,i} \geq 1 \tag{3.35}$$

zum System hinzuzufügen. Dies verhindert allerdings Schleifentransformationen wie *Schleifenumkehr* oder Kombinationen mit dieser. In PluTo ist es auf diese Weise realisiert worden.

Linear unabhängige Lösungen Da lediglich linear unabhängige Hyperebenen zu den bisher gefundenen gesucht werden sollen, kann der *Orthogonalraum* zu diesen gebildet werden, um mittels jenem linear abhängige Lösungen wie folgt auszuschließen.

Lemma 3.56. *Sei H die Matrix, welche zeilenweise Hyperebenen(-Normalen) enthält. Dann bilden die Zeilen von*

$$H^{\perp} = \mathcal{I} - H \left(H^T \cdot H\right)^{-1} H^T \tag{3.36}$$

den Orthogonalraum zu H (bzgl. des Standardskalarprodukts). [Pen55; LP92; BRS07]

Definition 3.57. *$H^{k^{\perp}}$ ist der Orthogonalraum zu den Hyperebenen $\{\phi^1, \ldots, \phi^{k-1}\}$.*

Ein linear unabhängiger Vektor ($\neq \vec{0}$) zu $\phi^1, \ldots, \phi^{k-1}$ darf nun nicht im Kern des Orthogonalraums dieser liegen, muss also mindestens eine Nicht-Nullkomponente im Orthogonalraum besitzen. [Bon08, Kap. 3.9]
Da auch die Bedingung $H^{\perp} \cdot \mathbf{h_k} \neq \vec{0}$ keinen konvexen Raum bildet, müsste $H^{i^{\perp}} \cdot \mathbf{h_k} > 0$ **oder** $H^{i^{\perp}} \cdot \mathbf{h_k} < 0$ für mindestens eine Zeile i von $H^{\perp}$ gelten. Da die Kombinationsmöglichkeiten für solch eine Überprüfung enorm schnell anwachsen, wird in PluTo auch in

diesem Fall eine vereinfachende Annahme getroffen. Diese besagt, dass alle Ungleichungen *positiv* sind und somit nur gefordert werden muss, dass **mindestens eine** mit '≥ 1' erfüllt wird, während alle '≥ 0' sind, wodurch lediglich

$$H^{k^\perp} \cdot \mathbf{h_k}^T \geq \vec{0} \;\; und \;\; (1 \ldots 1) \cdot (H^{k^\perp} \cdot \mathbf{h_k}^T) \geq 1 \tag{3.37}$$

hinzugefügt werden müssen.

Zielfunktion Unter den gegebenen **Restriktionen** soll nun der Vektor $(\mathbf{u}, w, \mathbf{h_k})$ lexikographisch minimiert werden, d. h. zuerst werden die Koeffizienten der volumenbeschränkenden Funktion $\mathbf{u} \cdot \vec{p} + w$ minimiert. Somit werden zunächst die Koeffizienten der (oftmals *sehr* großen) Programmparameter minimiert und im Anschluss daran der konstante Teil w der Funktion (bei gleichförmigen Abhängigkeiten ausschließlich w, siehe (3.32)). Dadurch sind Einschränkungen gesetzt, welche das Kommunikationsvolumen beschränken und unter welchen nun der Vektor $\mathbf{h_k}$ lexikographisch minimiert wird (d. h. es werden unter diesen Einschränkungen nun Hyperebenenvektoren gesucht, deren Koeffizienten bzgl. der lexikographischen Ordnung möglichst klein sind).

$$\min_{\prec}(u_1, \ldots, u_{N_p}, w, h_{k,1}, \ldots, h_{k,d}) \tag{3.38}$$

Bemerkung 3.58. *Eine Minimierung von* $\mathbf{u}$ *bzgl. der lexikographischen Ordnung impliziert indirekt eine Priorisierung der Parameter in gegebener Reihenfolge. Die Priorisierung der* u_i *durch die lexikographische Ordnung ist allerdings zufällig bzw. hat keinen Bezug zu der Größe der Parameter. Sind beispielsweise zwei Parameter in der Reihenfolge* M, N *gegeben mit* $M \ll N$*, so kann eine durch die lexikographische Ordnung generierte Lösung zu deutlich schlechterer Performance (durch höhere Kommunikation und Wiederverwendungsdistanz) führen.*
Dieses Problem wird in Kapitel 4 näher betrachtet und im ersten Abschnitt in 4.3.1 ein Lösungsansatz für die Problematik geliefert.

Bemerkung 3.59. *Die lexikographische Minimierung der Hyperebenkoeffizienten (Unterschied zu Bemerkung 3.46) ist u. A. sinnvoll, da so Einheitsvektoren als Hyperebenen präferiert werden, was die Zugriffsstruktur auf die Arrays und die Schleifenköpfe einfach hält. Grundsätzlich ist eine Minimierung der Hyperebenenkoeffizienten auch sinnvoll, um die Menge an 'Löchern' im Indexraum zu minimieren (Bemerkung 3.21), allerdings* ***kann*** *die lexikographische Minimierung in dieser Hinsicht unvorteilhaft sein, da diese nicht zwangsläufig die Determinante der Clustering-Matrix minimiert.*

Resultierendes System Aus diesen Elementen resultiert das zu lösende *Optimierungsproblem* für Level k und Statement S mit d-dimensionalem Indexraum:

$$
\begin{array}{lrll}
\min_{\prec} & (\mathbf{u}, w, \mathbf{h_k}) & & \\
\text{s.t.} & \phi(\vec{t}) - \phi(\vec{s}) & \geq 0 & \forall e^{S \to S} \\
& \phi(\vec{t}) - \phi(\vec{s}) & \leq \nu(\vec{p}) & \forall e^{S \to S} \\
& \mathbf{h_k} & \neq \vec{0} & \\
& {H^k}^{\perp} \cdot \mathbf{h_k}^T & \neq \vec{0} & \\
& \mathbf{h_k} & \in \mathbb{Z}^d & \\
\multicolumn{4}{c}{\downarrow \quad \text{bzw. in PluTo} \quad \downarrow} \\
\min_{\prec} & (\mathbf{u}, w, \mathbf{h_k}) & & \rightarrow (3.38) \\
\text{s.t.} & \phi(\vec{t}) - \phi(\vec{s}) & \geq 0 & \rightarrow (3.30) \\
& \phi(\vec{t}) - \phi(\vec{s}) & \leq \nu(\vec{p}) & \rightarrow (3.31) \\
& (1 \ldots 1) \cdot \mathbf{h_k}^T & \geq 1 & \rightarrow (3.35) \\
& {H^k}^{\perp} \cdot \mathbf{h_k}^T & > \vec{0} & \rightarrow (3.37) \\
& \mathbf{h_k} & \in \mathbb{Z}^{+d} & \\
\multicolumn{4}{c}{\downarrow \quad \text{bzw. ausgeschrieben} \quad \text{und bei} \quad \text{gleichförmigen Abhängigkeiten} \quad \downarrow} \\
\min_{\prec} & (w, h_{k,1}, \ldots, h_{k,d}) & & \\
\text{s.t.} & \sum_{j=1}^{d} h_{k,j} * d_j^l & \geq 0 & \forall \text{ Abhängigkeitsvektoren } d^l \\
& \sum_{j=1}^{d} h_{k,j} * d_j^l & \leq w & \forall \text{ Abhängigkeitsvektoren } d^l \\
& \sum_{j=1}^{d} h_{k,j} & \geq 1 & \\
& \sum_{j=1}^{d} ({H^k}^{\perp})_{i,j} h_{k,j} & \geq 0 & \forall i \\
& \sum_{i=1}^{d'} \sum_{j=1}^{d} ({H^k}^{\perp})_{i,j} h_{k,j} & \geq 1 & \\
& h_{k,j} & \in \mathbb{Z}^+ & \forall j \in 1, \ldots, d
\end{array}
$$

mit $d' = k - 1 \leq d$ die Dimension des Orthogonalraums für Level k.

Für die gesuchte Tiling-Matrix H (für Statement S) ist $\mathbf{h_k}$ die k-te Tiling-Hyperebene bzw. die k-te Zeile. (3.30) und (3.31) werden ggf. (bei nicht-gleichförmigen Abhängigkeiten) durch Farkas linearisiert.

Die Lösung dieses ILPs in PluTo wird durch die Piplib [Fea] realisiert, welche das lineare Programm durch einen Gomory-Cut Algorithmus löst.

Bemerkung 3.60. *Approximative Lösungsansätze der ganzzahligen Optimierungsprobleme sind nicht möglich, da nicht-exakte Lösungen im Transformationsprozess fehlerhaften Code erzeugen könnten.*

Bemerkung 3.61. *PluTo bestimmt die* ***formgebenden Hyperebenen*** *des Tilings ohne die Quantität der Tiles miteinzubeziehen. Es wird im Anschluss im transformierten Raum ein rechteckiges Tiling mit benutzergegebenen- oder default-Parametern generiert (siehe Bemerkung 3.44).*

Bemerkung 3.62. *'Global' gesehen, also bezogen auf $min_{\prec}(\mathbf{u}, w, \mathbf{h_k})$, ist die lexikographische Minimierung sinnvoll, denn somit werden zunächst die Koeffizienten u_i der (i. d. R. sehr großen) Programmparameter in der Volumenbeschränkung minimiert und im Anschluss daran der konstante Teil w dieser Funktion. Sind hierdurch die beschränkenden Nebenbedingungen für die Hyperebenen gesetzt, wird ein 'möglichst kleiner' Hyperebenenvektor gesucht. Trotz dieser korrekten Annahme ist die zusätzliche lexikographische Minimierung im Inneren von $\mathbf{h_k}$ und vor allem im Inneren von $\mathbf{u}$ nicht unbedingt sinnvoll.*
Dieses Problem wird auch in Kapitel 4 näher betrachtet und im zweiten Abschnitt in 4.3.1 wird ein Lösungsansatz für die Problematik geliefert.

Bemerkung 3.63. *Eine lexikographische Minimierung kann vorgenommen werden, indem man sukzessive jede Variable minimiert und den Optimalwert in das System einfügt. Ist beispielsweise u_1^{min} gefunden, so können die Ungleichungen $u_1 \leq u_1^{min}$ und $u_1 \geq u_1^{min}$ (also resultierend $u_1 = u_1^{min}$) hinzugefügt werden und es kann auf diesem System mit u_2 fortgefahren werden.*

Beispiel 3.64 (gleichförmige Abhängigkeiten)**.**
Gegeben sei der Code des 1-D perfekt verschachtelten Jacobi mit Statement S aus Abbildung 3.30 mit den folgenden Abhängigkeiten und zugehörigen Indexräumen:

$$(1)\ S\delta^f S: \quad a[i-1][j-1] \leftrightarrow a[i][j] \quad \textit{mit } d_1 = \begin{pmatrix} i \\ j \end{pmatrix} - \begin{pmatrix} i-1 \\ j-1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

$$(2)\ S\delta^f S: \quad a[i-1][j] \leftrightarrow a[i][j] \quad \textit{mit } d_2 = \begin{pmatrix} i \\ j \end{pmatrix} - \begin{pmatrix} i-1 \\ j \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$$(3)\ S\delta^f S: \quad a[i-1][j+1] \leftrightarrow a[i][j] \quad \textit{mit } d_3 = \begin{pmatrix} i \\ j \end{pmatrix} - \begin{pmatrix} i-1 \\ j+1 \end{pmatrix} = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

$$\textit{für alle: Indexraum} \quad \mathcal{D}^S = \{(i,j) \mid 1 \leq i \leq T-1, 1 \leq j \leq N-2\}$$

Seien die Abhängigkeiten im Folgenden von Iterationspunkt $\begin{pmatrix} i_1 \\ j_1 \end{pmatrix}$ *nach* $\begin{pmatrix} i_2 \\ j_2 \end{pmatrix}$.

Level: k=1 *Bestimmung der ersten Hyperebene* ϕ_S^1*:*

Abhängigkeit (1)
Gültigkeitsbedingung:

$$\begin{aligned} \phi_S^1(i_2,j_2) - \phi_S^1(i_1,j_1) &= \begin{pmatrix} h_{1,1} & h_{1,2} \end{pmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix} - \begin{pmatrix} h_{1,1} & h_{1,2} \end{pmatrix} \cdot \begin{pmatrix} i-1 \\ j-1 \end{pmatrix} \\ &= \begin{pmatrix} h_{1,1} & h_{1,2} \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix} \\ &= h_{1,1} + h_{1,2} \geq 0 \end{aligned}$$

Volumenbeschränkung:

$$\begin{aligned} \phi_S^1(i_2,j_2) - \phi_S^1(i_1,j_1) &= \begin{pmatrix} h_{1,1} & h_{1,2} \end{pmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix} - \begin{pmatrix} h_{1,1} & h_{1,2} \end{pmatrix} \cdot \begin{pmatrix} i-1 \\ j-1 \end{pmatrix} \\ &= \begin{pmatrix} h_{1,1} & h_{1,2} \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix} \\ &= h_{1,1} + h_{1,2} \leq w \end{aligned}$$

Analog für ***Abhängigkeit (2)***
Gültigkeitsbedingung: $\phi_S^1(i_2, j_2) - \phi_S^1(i_1, j_1) = h_{1,1} \geq 0$
Volumenbeschränkung: $\phi_S^1(i_2, j_2) - \phi_S^1(i_1, j_1) = h_{1,1} \leq w$

Ebenfalls analog für ***Abhängigkeit (3)***
Gültigkeitsbedingung: $\phi_S^1(i_2, j_2) - \phi_S^1(i_1, j_1) = h_{1,1} - h_{1,2} \geq 0$
Volumenbeschränkung: $\phi_S^1(i_2, j_2) - \phi_S^1(i_1, j_1) = h_{1,1} - h_{1,2} \leq w$

Daraus resultiert inkl. der Ungleichung zur Vermeidung der Nulllösung für die erste Hyperebene ϕ_S^1 *das folgende Optimierungsproblem:*

$$
\begin{array}{lll}
min_{\prec}(w, h_{1,1}, h_{1,2}) & & \\
s.t. \quad h_{1,1} + h_{1,2} & \geq 0 & \\
h_{1,1} + h_{1,2} & \leq w & \\
h_{1,1} & \geq 0 & \\
h_{1,1} & \leq w & \Rightarrow \textit{Lösung: } (1,1,0) \qquad (3.39) \\
h_{1,1} - h_{1,2} & \geq 0 & \textit{also } \phi_S^1 = h_1 = (1,0) \qquad (3.40) \\
h_{1,1} - h_{1,2} & \leq w & \\
- - - - - - - - & - - - & \\
h_{1,1} + h_{1,2} & \geq 1 & \\
h_{1,1}, h_{1,2} & \in \mathbb{Z} &
\end{array}
$$

Berechne den orthogonalen Unterraum zu dieser Lösung:

$$
\begin{aligned}
(\phi_S^1)^{\perp} &= \mathcal{I}_2 - \phi_S^1 \cdot \left({\phi_S^1}^T \cdot \phi_S^1 \right)^{-1} {\phi_S^1}^T \\
&= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} - \begin{pmatrix} 1 & 0 \end{pmatrix} \left(\begin{pmatrix} 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} \right)^{-1} \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} - \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \\
&= \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}
\end{aligned}
$$

Also ist $\begin{pmatrix} 0 & 1 \end{pmatrix}$ *Basis des Orthogonalraums. Da nur nicht-triviale Linearkombinationen dieser als weitere Lösungen zulässig sind, muss also für* $\phi_S^2 = \begin{pmatrix} h_{2,1} & h_{2,2} \end{pmatrix}$ *gelten:* $h_{2,2} \neq 0$*, oder in Ungleichungsform* $h_{2,2} \geq 1 \vee h_{2,2} \leq -1$*. Hier sollen o. B. d. A. positive*

Lösungen gefunden werden, also muss $h_{2,2} \geq 1$ *zum Ungleichungssystem hinzugefügt werden.*

Für die zweite Hyperebene ϕ_S^2 *resultiert das folgende Optimierungsproblem:*

$$
\begin{array}{lll}
min_{\prec}(w, h_{2,1}, h_{2,2}) & & \\
s.t. \quad h_{2,1} + h_{2,2} & \geq 0 & \\
h_{2,1} + h_{2,2} & \leq w & \\
h_{2,1} & \geq 0 & \\
h_{2,1} & \leq w & \Rightarrow \textit{Lösung: } (2,1,1) \qquad (3.41) \\
h_{2,1} - h_{2,2} & \geq 0 & \textit{also } \phi_S^2 = h_2 = (1,1) \qquad (3.42) \\
h_{2,1} - h_{2,2} & \leq w & \\
h_{2,1} + h_{2,2} & \geq 1 & \\
\rightarrow \qquad h_{2,2} & \geq 1 & \\
h_{1,1}, h_{1,2} & \in \mathbb{Z} &
\end{array}
$$

Bemerkung 3.65. *Die ermittelten Lösungen* ϕ_S^1 *und* ϕ_S^2 *sind auch die in Abbildung 3.31 oder 3.32 dargestellten Tiling-Hyperebenen. An den Grafiken sieht man auch die minimale Wiederverwendungsdistanz von* $w = 1$ *in* ϕ_S^1*- bzw.* i'*-Richtung und* $w = 2$ *in* ϕ_S^2*- bzw.* j'*-Richtung.*

Beispiel 3.66 (nicht-gleichförmige Abhängigkeiten)**.**
Ein ausführliches Beispiel zu nicht-gleichförmigen Abhängigkeiten ist in [Bon08, Beispiel 3.11.1] zu finden.

3.9.3 Ergebnisse (und Schwächen)

- Durch die erläuterten Methoden sowie diversen Erweiterungen wird aus einem lauffähigen Source-Code ein, innerhalb der angegebenen Scopes optimierter, wiederum lauffähiger Source-Code generiert, welcher neben den erläuterten Methoden des Tilings zur Datenzugriffsoptimierung und der Parallelisierung auch Aspekte wie *Unrolling* und Pragma-Einbettung für eine Vektorisierung mit dem *ICC* eingliedern kann.
- Eben die Optimierung der Datenlokalität ist ein Aspekt, der PluTo von den meisten anderen Projekten rund um das Polyedermodell unterscheidet. Somit ist die Nutzung der zu diesem Zweck speziell geeigneten Partitionierungsmethode, anstatt der 'traditionellen' Space-Time-Methode, in anderen Projekten selten zu finden.
- Durch die Transformationen in PluTo werden erhebliche Performance-Gewinne, sowohl für den *GCC* als auch für den *ICC* erreicht. Details zu den Speedups sind in [Bon08, Kap. 6], in den PluTo-Dokumentationen und in den Vergleichen in Kapitel 5 zu finden. Grundsätzlich schlagen mit PluTo optimierte Codes sowohl nicht optimierte, als auch Space-Time- (bzw. *Scheduling*-) basierte Ansätze für diverse Algorithmen (in [Bon08] kompiliert mit dem *GCC* und -O3 oder dem *ICC* und -parallel -fast) z. T. um ein Vielfaches (z. B. 1-D Jacobi: [Bon08, S. 131-132]).
- Die Wahl der lexikographischen Minimierung der Programmparameterkoeffizienten (Bemerkung 3.58) ist nur teilweise zu begründen.
- Die Bestimmung der Tile-Größe wird in PluTo dem User überlassen oder nach default-Parametern gesetzt. Beide Ansätze sind unvorteilhaft, da der User eine effiziente Tile-Größe (abhängig vom zugrundeliegenden Code) nicht einfach abschätzen kann und eine default-Setzung der Parameter weder verschiedenen Algorithmen noch verschiedenen Hardware-Gegebenheiten gerecht wird.
- Des Weiteren werden im default-Modus **alle** gegebenen Schleifen geblockt, was für eine Vektorisierung und auch andere Optimierungsziele z. T. unnötig ist und die Codestruktur (besonders bei vielfach verschachtelten Schleifen) sowie die Zugriffsstruktur auf den Speicher verkompliziert.
- Außerdem sind für eine Vektorisierung mit dem *GCC* weitere Kriterien an die Schleifen nötig, die bislang nicht oder nur gelegentlich (zufällig) erfüllt sind und im folgenden Kapitel weiter erläutert werden. Somit ist eine Vektorisierung mit dem *GCC*, selbst mit PluTo, bisher nur sporadisch möglich.

4 Konzept und Implementierung

In diesem Kapitel werden die Konzepte sowie grundlegende Implementierungsdetails des entwickelten Frameworks '**PluTo-SIMD**', welches auf PluTo aufbaut, beschrieben.

Anknüpfend an die in Abschnitt 3.9.3 aufgezeigten Schwächen von PluTo wurden Erweiterungen, die im Wesentlichen die folgenden Punkte betreffen, in PluTo-SIMD implementiert:

- Die Möglichkeit zur Modifikation der **Zielfunktion** in der Optimierungsphase ist in PluTo-SIMD gegeben. Hier kann neben der nativen lexikographischen Minimierung der Programmparameterkoeffizienten in gegebener Reihenfolge auch bzgl. der Parametersumme oder in modifizierter Parameterreihenfolge optimiert werden.
- Die automatische Bestimmung **optimaler Tile-Größen**, bzw. erste Methoden dazu, vor allem in Hinblick auf eine Vektorisierung, sind der Hauptaspekt der Weiterentwicklung des PluTo-Frameworks im Rahmen von PluTo-SIMD. Der automatisierte Bezug der Optimierung zu der gegebenen Hardware ist bislang *nicht* vorhanden und wird im Folgenden sowohl theoretisch als auch experimentell detaillierter betrachtet. Da ein guter Ladefluss direkt vom Cache abhängt, sollte ein Bezug zu diesem in der Optimierung ein lohnender Ansatz sein. Diese Verknüpfung wird in PluTo-SIMD realisiert.
- Außerdem werden in PluTo-SIMD **bestimmte Schleifen** (passend zur Vektorisierung) gezielt geblockt.
- Zusätzlich ermöglicht es PluTo-SIMD, **Block-Größen für jedes Statement einzeln**, automatisch vom Framework angepasst, setzen zu lassen, was ein durch den Hardwarebezug sinnvoller, neuer Aspekt des Frameworks ist. Eine solche Statementspezifische Setzung der Blockgröße ist in PluTo *grundsätzlich nicht* möglich.

Diese Konzepte sollen zu einer verbesserten Vektorisierung durch einen End-Compiler mit optimierten Speicherzugriffen führen.

4.1 Gesamtkonzept

Alle entwickelten Erweiterungen des PluTo-Frameworks wurden so integriert, dass sie das bestehende Framework in all seinen Optionen **nicht** beeinflussen. PluTo ist somit mit allen Funktionen unbeeinflusst in PluTo-SIMD vorhanden und kann, wenn keine der PluTo-SIMD-Optionen aktiviert ist, in seiner ursprünglichen Art und Weise verwendet werden. Die hinzugefügten Optionen in PluTo-SIMD können außerdem mit den bereits vorhandenen Optionen von PluTo kombiniert werden.

Die Erweiterungen von PluTo-SIMD lassen sich in zwei Gruppen einteilen.

Die Optimierungsphase

Es wurden neue Funktionen integriert, durch welche der User die Attribute der Optimierungsphase modifizieren kann (vgl. Kapitel 3.9).

SIMD- und Cache-spezifischer Tiling-Algorithmus

Der Kern von PluTo-SIMD besteht aus dem neuen *SIMD- und Cache-spezifischen Tiling-Algorithmus*. Durch diesen werden zwei Hauptziele verfolgt:

- Das Blocken lediglich der für die Vektorisierung **relevanten Schleifen** in **Register-konforme Blöcke** (zur Vorverarbeitung für den Compiler)
- sowie eine **Anpassung** der Tile-Größe **an die vorhandene Hardware** (Cache) durch die Analyse der **relevanten Zugriffe** und eine Abbildung von der Iterationsanzahl der vektorisierten Schleife auf die Menge der **benötigten/zu ladenden Daten**.

Alle Erweiterungen werden kurz durch die behandelnde Problematik motiviert. Im Anschluss daran wird das Konzept und dessen Implementierung sowie mögliche und teilweise auch bereits implementierte zusätzliche Möglichkeiten der Erweiterung dieser erläutert.

Im Folgenden wird ein **ein**faches Tiling einer verschachtelten Schleife auch als **Level-1-Tiling** bezeichnet, ein **zwei**faches Tiling entsprechend als **Level-1- und Level-2-Tiling**.
In **PluTo** werden im einfachen Tiling **alle** Schleifen geblockt und im zweifachen Tiling darauf aufsetzend **alle** Schleifen ein zweites Mal geblockt.
In **PluTo-SIMD** *ergänzen* sich die zwei Tile-Level (und setzen nicht unbedingt aufeinander auf), indem sie eine Schleife an den Level-1-Cache anpassen (Level-1-Tiling) und ggf. zusätzlich eine zweite Schleife an den Level-2-Cache (Level-1- und Level-2-Tiling).

'Angepassen' heißt hier im Wesentlichen, dass alle in einem Tile benötigten Daten auf einmal in das entsprechende Cache-Level geladen werden können und die Iterationsanzahl der (für das Tiling relevanten) Schleifen Register-konform ist. Im Detail wird hierauf in den folgenden Abschnitten eingegangen.

Folgende Funktionen für $x, y \in \mathbb{Q}$ werden zum Tiling in den for-Schleifenköpfen eingefügt:

$min(x, y)$:	liefert x, falls x<y, sonst y (die kleinere der beiden Zahlen)
$max(x, y)$:	liefert x, falls x>y, sonst y (die größere der beiden Zahlen)
$floor(x)$:	liefert die größte ganze Zahl, die nicht größer als x ist (rundet ab)
$ceil(x)$:	liefert die kleinste ganze Zahl, die nicht kleiner als x ist (rundet auf)

Sie werden verwendet, um die Einteilung der ursprünglichen Schleifen in Blöcke vorzunehmen (siehe Beispiel 4.1).

Beispiel 4.1.

```
1 for(i=0; i<M; i++)
2   for(j=0; j<N; j++)
```

```
1 for(ii=0; ii≤floor(M-1,32); ii++)
2   for(jj=0; jj≤floor(N-1,32); jj++)
3     for(i=32*ii; i≤min(M-1,32*ii+31); i++)
4       for(j=32*jj; j≤min(N-1,32*jj+31); j++)
```

Abbildung 4.1: Originale Schleifen (links) und in 32-er Blöcke eingeteilte Schleifen (rechts)

4.2 Neue Elemente

Optionen in PluTo-SIMD (die grau unterlegten Optionen sind die Erweiterungen durch PluTo-SIMD):

```
This is a modified Version of PluTo to improve SIMD-Usage


   ___      _____                        ___  __ _   __ __
  / o |/7  /_  _/_                     ,' _/ / // \,' //  \
 / _,'///7/7/ /,'o|  < < ___ > >  _\ `. / // \,' // o /
/_/  ///__//_/ /_,'               /___,'/_//_/ /_//__,'


Usage: polycc <input.c> [options]

Options:
      --tile                 Tile for locality
      --parallel             Automatically parallelize using OpenMP pragmas
      | --parallelize
      --l2tile               Tile a second time (typically for L2 cache) - disabled by default
      --multipipe            Extract two degrees of pipelined parallelism if possible;
                               by default one degree is extracted (if it exists)
      --rar                  Consider RAR dependences too (disabled by default)
      --[no]unroll           Unroll-jam (disabled by default)
      --ufactor=<factor>     Unroll-jam factor (default is 8)
      --[no]prevector        Make code amenable to compiler auto-vectorization (with ICC) - enabled by
                               default
      --context=<context>    Parameters are at least as much as <context>
      --bee                  Generate pragmas for Bee+Cl@k

      --indent  | -i         Indent generated code (disabled by default)
      --silent  | -q         Silent mode; no output as long as everything goes fine (disabled by default)
      --help    | -h         Print this help menu
      --version | -v         Display version number

   Fusion                 Options to control fusion heuristic
      --nofuse               Do not fuse across SCCs of data dependence graph
      --maxfuse              Maximal fusion
      --smartfuse [default]  Heuristic (in between nofuse and maxfuse)

   PluTo-SIMD             Options to control the PluTo-SIMD-Modifications (all disabled by default)
      --simd                 Generate vectorization modifications for SIMD
      --swapoptvariables     Swap two variables order in lexicographic optimization
      --minparamsum          Minimize the sum of the parameter-coefficients u1+u2+... instead of just
                               lex-min
      --ipdebug              Print the IPs used for optimization
      --nobound              Do not optimize communication-volume (Option was already given by PluTo)

   Debugging
      --debug        Verbose output

To report bugs, please send an email to the author <bondhugula.1@osu.edu>
The SIMD-Modifications were included by Dustin Feld <dustin.feld@scai.fraunhofer.de>
```

Abbildung 4.2: Zusammenfassung der Command-Line-Optionen in PluTo-SIMD

4.3 Erweiterungen der Optimierungsphase

4.3.1 Lexikografische Minimierung

Die Option '`--minparamsum`'

Problematik:
Vergleiche Bemerkung 3.62.

Konzept:
PluTo löst die Optimierungsprobleme durch die Piplib [Fea] von P. Feautrier. Dieser Bibliothek wird ein ILP ohne Zielfunktion übergeben, woraufhin es lexikographisch minimiert wird, d. h. die Reihenfolge, in der die Variablen stehen, bestimmt die Optimierungspriorität.
Es gibt an sich keinen Grund für die gegebene Priorisierung im Inneren von $\mathbf{u}$, also z.B. u_1 vor u_2 zu minimieren (und nicht andersherum), wenn die konkreten Werte der entsprechenden Parameter nicht bekannt sind (was sie zum Kompilierungszeitpunkt oft nicht sind). Mit dieser Option wird zuerst **die Summe** aller Parameter-Koeffizienten minimiert, danach werden alle Variablen unter dieser Restriktion wie bisher minimiert, was in $\min_{\prec}(u_1 + u_2 + ... + u_n, u_1, u_2, ..., u_n, w, c_1, c_2, ...)$ resultiert.

Implementierung:
Hierzu ist eine Modifikation des ILPs notwendig. Die Variable `npar` im PluTo-Framework beschreibt die Anzahl der für ein Statement (bzw. dessen Indexraum) relevanten Parameter. Es muss im ILP zunächst eine Variable $\tilde{y}$ durch die bereits in PluTo vorhandene Funktion `pluto_matrix_add_col` eingeführt werden sowie eine zu dieser Variable gehörige Zeile durch `pluto_matrix_add_row`, welche die Ungleichung $\tilde{y} \geq u_1 + u_2 + ... + u_n$ enthalten wird. Danach soll die Variable $\tilde{y}$ in der lexikographischen Minimierung als erste minimiert werden, somit muss sie vor die erste problemspezifische Spalte gesetzt werden ('problemspezifisch', da die allererste Spalte [B] der PluTo-Repräsentation des ILPs eine Variable enthält die bestimmt, ob es sich um eine Ungleichung oder Gleichung handelt). Die Platzierung der Zeile/Spalte ist über Parameter in den `add`-Funktionen steuerbar. Die eingefügte Zeile hat dann die Form $(1,\ 1,\ -1,\ -1,\ ...\ ,\ -1,\ 0,\ ...\ ,\ 0)$. Dies bedeutet: Es handelt sich um eine Ungleichung (die erste 1) und sie lautet: $\tilde{y} - u_1 - u_2 - ... - u_n \geq 0$.

Darüber hinaus muss das Ergebnis und die Existenz der Dummy-Variable $\tilde{y}$ im Lösungsvektor ignoriert werden, da dieser Wert ($\tilde{y}$) für die Weiterverarbeitung irrelevant ist und

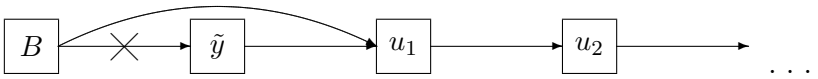


Abbildung 4.3: Überspringen der Dummy-Variable im Lösungsvektor

lediglich zur Beschränkung der Parametersumme dient. Der Lösungsvektor der PIP-Bibliothek (`solution`) ist als verkettete Liste gespeichert. Deshalb wurde das Ignorieren dieser Komponente über das Versetzen eines Pointers in dieser Liste realisiert (siehe Abbildung 4.3). Außerdem muss für das PluTo-SIMD-Framework die Anzahl der Elemente im Lösungsvektor `pipcst` $\rightarrow$ `ncols` im Fall einer so eingefügten Dummy-Variable (wegen des Überspringens einer Variable) um 1 verringert betrachtet werden, wenn die Lösung in das zu übergebende `sol`-Array übertragen wird.

Resultat:
Die Parameter**summe** ist minimal, dazu ist eine Zeile und eine Spalte im ILP zusätzlich nötig.

Bereich in PluTo für diese Erweiterung:

- ILP-Lösungsphase
- Manipulation der Restriktionen vor dieser

Die Option '`--swapoptvariables`'

Problematik:
Vergleiche Bemerkung 3.58.

Konzept:
Um die Priorisierung der Parameter im ILP zur Tiling-Hyperebenen-Berechnung beeinflussen zu können, wurde die Swap-Parameter-Funktion in PluTo-SIMD integriert. Hiermit ist es dem User möglich, die Reihenfolge der Variablen in der lexikographischen Minimierung $\min_{\prec}(u_1, u_2, ..., u_p, w, c_1, c_2, ...)$, und somit die Priorisierung der Parameter, zu verändern. Beispielsweise können die Prioritäten von u_1 und u_2 getauscht werden ($\Rightarrow \min_{\prec}(u_2, u_1, ..., u_p, w, c_1, c_2, ...)$), um das Kommunikationsvolumen, welches vom Parameter von u_2 abhängt, mit höchster Priorität (also als erstes) zu minimieren. Der User kann einen Tausch (*Swap*) zweier Variablen auf verschiedene Weisen steuern, nämlich

- durch die Datei `.swapoptcolumns`, indem er zwei Zeilen mit den zu tauschenden Variablenindizes (Stelle der Variable in der Minimierungsfunktion) angibt oder,
- falls **keine** solche Datei vorhanden ist, wird der User aufgefordert, die beiden Indizes via *stdin* zu bestimmen.

Die aktuelle Reihenfolge sowie die nach dem Tausch der Parameter, wird dem User (bei aktiver `--swapoptvariables` Option) bei der Ausführung ausgegeben.

Implementierung:
Der Algorithmus prüft zunächst, ob eine `.swapoptcolumns` Datei vorhanden ist. Ist dies nicht der Fall, wird eine solche erstellt und als temporär gekennzeichnet, damit sie am Ende des Vorgangs wieder gelöscht wird. In die Datei wird in diesem Fall die Eingabe der beiden Indizes vom User geschrieben. Dadurch ist ein einheitlicher '*Container*' für die zu tauschenden Variablen vorhanden, egal ob diese durch die Datei oder durch eine Eingabe spezifiziert wurden. Der Tauschvorgang wird im Falle einer nicht konformen `.swapoptcolumns` Datei deaktiviert, wenn z.B. nur ein Index angegeben wurde. Sind die beiden Indizes identisch, so wird der Tauschvorgang ebenfalls deaktiviert, da er keine Auswirkung hätte. Wenn die Indizes gültig sind, wird die Funktion `SIMD_swap_columns` aufgerufen. Diese vertauscht nun im Ungleichungssystem des ILP (in PluTo die Struktur `pipcst`) die beiden den Variablen entsprechenden Spalten. Daraufhin wird das Optimierungsproblem gelöst, dessen Lösung in der Liste `solution` verkettet gespeichert wird.
Damit dieser Tausch keinen Einfluss auf die Weiterverarbeitung von PluTo hat, müssen die korrespondierenden Lösungskomponenten im Lösungsvektor zurückgetauscht werden, denn der Tausch diente lediglich der *Reihenfolgenmanipulation im Optimierungsproblem*. Hierzu wird die Funktion `SIMD_swap_solution` aufgerufen, welche die verkettete Liste der Lösungskomponenten zunächst bis zur ersten angegebenen Spalte und im Anschluss weiter bis zur zweiten durchläuft und die beiden Einträge vertauscht.

Resultat:
Die Reihenfolge der Programmparameterkoeffizienten in der lexikographischen Minimierung kann manipuliert werden. Es ist außerdem möglich die Priorisierung beliebiger anderer Spalten in der Optimierungsmatrix zu vertauschen. Weiß der User vor dem Einsatz von PluTo-SIMD und bei vorhandenen nicht-gleichförmigen Abhängigkeiten im Code, dass ein Programmparameter besonders groß ist, so kann er diesen entsprechend nach vorne rücken.

Bereich in PluTo für diese Erweiterung:

- ILP-Lösungsphase
- Manipulation der Daten vor und nach dieser

Mögliche Erweiterungen:
Aktuell ist lediglich ein Tauschen zweier Variablen möglich, eine Erweiterung auf beliebig viele und somit eine beliebige Permutation der Variablen kann (auch darauf aufbauend) implementiert werden.

4.3.2 Die Optimierungsprobleme (ILPs)

Die Option '`--ipdebug`'

Problematik:
Bisher bekommt der User über keine Funktion direkten Einblick in die zu lösenden Optimierungsprobleme.

Konzept:
Das Generieren der Tiling-Hyperebenen ist ein iterativer Prozess, der in jedem Schritt ein ganzzahliges Optimierungsproblem löst. Um einen Einblick in diesen Prozess zu bekommen, werden mit dieser Option sowohl die ILPs, als auch deren Lösungen, via *stdout* ausgegeben. Somit kann der User nachvollziehen, wie PluTo (bzw. PluTo-SIMD) zu den generierten Lösungen, also Hyperebenen und somit Schleifentransformationen, kommt.

Implementierung:
PluTo hat Funktionen `pip_matrix_print` und `pip_quast_print` zur Ausgabe von Problem und Lösung implementiert, allerdings gab es keine Möglichkeit, diese in der Optimierungsphase zur Ausgabe der Probleme zu aktivieren. Über die neu implementierte Kommandozeilenoption werden die Informationen dem User direkt als Ausgabe dargestellt.

Resultat:
Der User erhält Einblick in die zu lösenden ILPs in PluTo-SIMD, allerdings resultiert dies unter Umständen in sehr vielen Ausgaben.

Bereich in PluTo für diese Erweiterung:

- ILP-Lösungsphase

4.3.3 Die Volumenbeschränkung

Die Option '--nobound'

Problematik:
Die *Volumenbeschränkung* ist keine notwendige Bedingung für ein gültiges Tiling. Da diese aber die Möglichkeiten für die Tiling-Hyperebenen einschränkt, kann es sinnvoll sein, ein beliebiges Kommunikationsvolumen zuzulassen, um u. U. bessere (lexikographisch kleinere) Hyperebenen zu generieren.

Konzept:
Im Optimierungsproblem zur Bestimmung der optimalen Tiling-Hyperebenen gibt es neben den Gültigkeitsbedingungen ('Legacy Constraints') noch diejenigen Hyperebenen, die das Kommunikationsvolumen bzw. die Reuse-Distanz minimieren ('Communication Boundary Constraints') (im Abschnitt 3.9.2 'Optimalität'). Diese sind keine Notwendigkeit, sondern ein Zusatz, der in einigen Fällen allerdings nicht gewünscht sein kann. Soll das Tiling beispielsweise auf einem Ein-Kern-Prozessor ausgeführt werden, ist das Kommunikationsvolumen zwischen den Tiles i. d. R. nicht entscheidend. Selbst auf Mehr-Kern-Prozessoren kann es sinnvoll sein, das Kommunikationsvolumen nicht in die Optimierung einzubeziehen, wenn die Kommunikation zwischen den Kernen z. B. sehr 'günstig' ist. Das Ignorieren des Kommunikationsvolumens kann nämlich Vorteile für die Bestimmung der Tiling-Hyperebenen bringen, da diese bei minimiertem Kommunikationsvolumen mehr Restriktionen unterliegen. Somit können die Hyperebenen unter Umständen für sich betrachtet weniger optimal generiert werden, da das Kommunikationsvolumen in der lexikographischen Minimierung zuallererst (durch $\mathbf{u}$ und w) behandelt wird. Es gibt also diverse Gründe, dem User die Möglichkeit zur Deaktivierung der Kommunikationsminimierung zu geben.

Implementierung:
PluTo hatte diese Funktion bereits inne, allerdings ist sie nicht in der Optionen-Beschreibung erwähnt, vielleicht war sie für die Entwickler angedacht und nicht für den User. Für die Optimierung auf Ein-Kern-Prozessoren z. B. für eine effiziente SSE-Nutzung ist sie in PluTo-SIMD wieder integriert worden.

Resultat:
Das Optimierungsproblem (ILP) hat weniger Ungleichungen und es können z. T. bessere

Tiling-Hyperebenen bestimmt werden. Allerdings hat man keine Kontrolle mehr über die Wiederverwendungsdistanz und das Kommunikationsvolumen.

Bereich in PluTo für diese Erweiterung:

- ILP-Lösungsphase
- Manipulation der Daten vor der ILP-Lösungsphase

4.4 Der Kern von PluTo-SIMD

Die Option '`--simd`'

Diese Option aktiviert den Kern von PluTo-SIMD.

In diesem Abschnitt werden die in PluTo-SIMD integrierten Funktionen und Konfigurationsmöglichkeiten dargestellt sowie grundlegende Implementierungsdetails erläutert.

4.4.1 Transformationssetup und Bestimmung der vektorisierbaren Schleife

Problematik:

Die Bestimmung der Tile-Größen wird in PluTo dem User überlassen oder (im *default-Modus*) für ein einfaches Tiling (durch die Option '`--tile`') in jeder Dimension auf 32 gesetzt, die Multiplikatoren für das zweifache Tiling (wenn durch die Option '`--l2tile`' gesetzt) per *default* auf 8. Ein resultierendes rechteckiges Tiling ohne weitere Transformationen mit $d = 2$ hat z. B. eine Form wie in Abbildung 4.4.

```
1  for(iii=0;iii≤floor(M-1,256);iii++)
2    for(jjj=0;jjj≤floor(N-1,256);jjj++)
3      for(ii=8*iii;ii≤min(floor(M-1,32),8*iii+7);ii++)
4        for(jj=8*jjj;jj≤min(floor(N-1,32),8*jjj+7);jj++)
5          for(i=32*ii;i≤min(M-1,32*ii+31);i++)
6            for(j=32*jj;j≤min(K-1,32*jj+31);j++)
7              S(i,j);
```

Abbildung 4.4: Zweifaches Tiling nach PluTo (default)

Die Werte des einfachen Tilings $(q_1^{L1}, \ldots, q_d^{L1})$ geben für jede Dimension die **Quantität** an, mit welcher der Indexraum des Statements S im transformierten Zielraum rechteckig geblockt wird. Für das zweite (aufsetzende) Tiling geben die *Faktoren* $(q_1^{L2}, \ldots, q_d^{L2})$ an, in wie viele solche L1-Blöcke für das zweite Level geblockt werden soll, hier im Beispiel in jeweils acht 32-er Blöcke, also für das zweite Level insgesamt in $8 \cdot 32 = 256$-er Blöcke. Im Beispiel hieße das $(q_1^{L1}, q_2^{L1}) = (32, 32)$ und $(q_1^{L2}, q_2^{L2}) = (8, 8)$ mit resultierenden L2-Blöcken der Größe $(256, 256)$.

Bemerkung 4.2. *Damit eine Schleife mit Index i praktisch geblockt wird, muss für die Tile-Quantität $q_i^{L1} < M$ gelten (mit M der Parameter, der die i-Schleife begrenzt). Ansonsten passen alle Iterationen dieser Schleife in einen Tile-Block, sodass kein tatsächliches Tiling stattfindet (z. B. falls bereits in der ersten ii-Iteration min(M-1,$32 * ii + 31$)=M-1 gilt).*

Bemerkung 4.3. *Die zusätzlichen Schleifen, die durch das Tiling entstehen (hier: iii, ii, jjj, jj), werden im Folgenden als Steuerschleifen bezeichnet. Die Schleifen ii und iii werden als Mutter-Schleifen zu i bezeichnet, da die Gruppe iii, ii, i durch das Tiling aus i entsteht. Bei einfachem Tiling reduziert sich dies auf ii und i, womit zu jeder Schleife genau eine Mutter-Schleife existiert.*

Daraus resultieren die folgenden Probleme:

- Je mehr Dimensionen eine verschachtelte Schleife hat, desto mehr Indexpunkte befinden sich bei festen q_i^{L1} und q_i^{L2} innerhalb eines Tiles. Bei rechteckigem Tiling (hier im *Zielraum*) sind es $\prod_{i=1}^{d} q_i^{L1}$ (für das zweifache Tiling $\prod_{i=1}^{d} q_i^{L1} * q_i^{L2}$), allgemein $det(H^{-1}) = det(G)$. Somit ist ein solches **default**-Tiling (wenn überhaupt) nur für eine bestimmte Anzahl von Dimensionen und nur bei entsprechend passendem Algorithmus an die vorhandenen Cache-Größen angepasst.
- Eine Blockung *aller* Dimensionen ist in vielen Fällen nicht nötig und kann den Code (unnötig) **verkomplizieren** und die Anzahl an 'Sprüngen' im Arbeitsspeicher erhöhen. Für eine Vektorisierung ist das ausschließliche Blocken der vektorisierten Schleife in vielen Fällen ausreichend und sorgt für ein 'flüssiges' Laden der Daten in den Cache.
- Die Tile-Quantitäten müssen **vor Anlauf von PluTo** für alle Dimensionen angegeben werden, geblockt wird allerdings im Ziel-Indexraum. Eventuelle Schleifenvertauschungen oder andere Transformationen im Transformationsprozess machen eine gezielte Anpassung der einzelnen Tile-Quantitäten ohne Kenntnis der Transformation unmöglich.
- Ein für den Cache passendes Tiling sollte eine an die **Cache-Größe angepasste Menge** von **Daten** in einem Tile adressieren. Die Tile-Quantitäten alleine geben allerdings nur Auskunft über die Menge der **Indexpunkte** im Inneren eines Tiles. Werden in jeder Iteration der Schleife beispielsweise neue Daten aus n Arrays benötigt, so werden in einem Tile mit p Indexpunkten bereits $p{\cdot}n$ **Daten** adressiert. Für eine Anpassung **der adressierten Daten** an den gegebenen Cache muss die Array-Zugriffsstruktur analysiert werden, um so von der **Iterationsanzahl** auf die Anzahl der **neu zu ladenden Daten** zu schließen.

(Vergleiche auch Abschnitt 3.9.3.)

Hier realisierte Lösungen dieser Probleme sind:

Konzept:

- Es wird eine gezielte **Analyse der Zugriffsstruktur** vorgenommen und es werden lediglich die (im Anschluss) vektorisierten Schleifen geblockt. Hierzu wird die Tile-Größe, unter Einbeziehung der Zugriffsstruktur und der somit pro Block benötigten Daten, an die vorhandenen Cache-Größen (Level-1- und ggf. Level-2-Cache) angepasst.
- Durch die **gezielte Blockung** einzig der vektorisierten Schleife wird die Code-Struktur relativ einfach gehalten. Außerdem macht dies eine gezielte Anpassung an den Cache eindeutig möglich, ansonsten wäre bei vorgegebenem Inhalt eines Tiles u. A. das Verhältnis der q_i^{L1} (bzw. q_i^{L2}) zueinander relevant. Ein *experimenteller Ausblick* auf ein Cache-angepasstes Tiling aller Dimensionen wird in Kapitel 5 geliefert sowie die Erläuterung, in welchen Fällen eine solche Erweiterung sinnvoll ist.
- Durch die automatisierte Analyse wird in einem zweiphasigen Algorithmus zunächst in *Phase 1* bestimmt, welche Schleife vektorisierbar ist und in der *Analyse*, wie für die entsprechende Umsetzung die angepassten Tile-Größen aussehen. In *Phase 2* wird ein so **angepasstes Tiling** durchgeführt. Die (für die Analyse notwendigen) Transformationen aus dem Framework werden in die Analyse einbezogen, um eine exakte Anpassung zu generieren.
- Die Tile-Quantität für die vektorisierte Schleife muss die Schleife in **Register-konforme** Blöcke zerteilen, damit die Auto-Vektorisierung durch den Compiler (besonders durch den *GCC*) möglich ist.
- Durch die konkrete Anpassung der Tile-Größen an den Cache soll bei vielfach verschachtelten Schleifen **jedes Statement eine an dieses angepasste** Tile-Quantität erhalten. In PluTo ist lediglich eine globale (also für *alle* Statements identische) Setzung der Quantitäten möglich, in PluTo-SIMD kann ein spezifischer Quantitäten-Vektor $(q_1^{L1}, \ldots, q_d^{L1})^S$ (bzw. für L2) für jedes Statement berechnet und umgesetzt werden.
- Sind Level-1- und Level-2-Tiling aktiviert, so soll neben der innersten vektorisierten Schleife, welche an den Level-1-Cache angepasst geblockt wird, eine weitere Schleife **für den Level-2-Cache** geblockt werden, sodass entsprechend viele Level-1-Blöcke in einem großen Block gebündelt werden, welcher dann in den Level-2-Cache passt.

Implementierung:
Zur Umsetzung der Konzepte läuft PluTo-SIMD durch zwei Transformationsphasen mit einer zwischengeschalteten Analysephase:

- *Phase 1*: Transformationsphase mit default-Werten
- *Analyse*: Berechnung der angepassten Tile-Quantitäten
- *Phase 2*: Transformationsphase mit den generierten Werten

In *Phase 1* wird PluTo mit default-Werten ausgeführt. Dadurch wird jede Schleife, die *potenziell* blockbar ist, auch *praktisch* geblockt. Die Wahl der default-Werte hat den Hintergrund, dass manuell gesetzte Tile-Größen u. U. die Vektorisierung einer Schleife und somit die Vertauschung ins Innerste der Verschachtelung verhindern können. Eine angegebene Tile-Quantität von $q_i = 1$ blockt eine Schleife beispielsweise in 'Einer-Blöcke', lässt diese also im Resultat unberührt. Somit wird diese Schleife vom PluTo-Framework in einem solchen Fall als 'nicht zu blocken' gekennzeichnet und kann resultierend auch nicht als 'zu vektorisieren' behandelt werden, da das Blocken einer Schleife Voraussetzung für die Vektorisierungsfunktionalität (u. A. das Verschieben der Schleife ins Innere) in PluTo ist. Bei Anwendung eines PluTo-(default-)Tilings wird *jede* Schleife in 32-*er Blöcke* partitioniert, womit das Vektorisierungspotenzial durch diese Transformation nicht beeinträchtigt wird.

In der *Analyse*-Phase wird nun zunächst festgestellt, ob eine Schleife vektorisierbar ist, und wenn ja, welche. Daraufhin wird analysiert, an welcher Stelle sich die Mutter-Schleife der vektorisierten Schleife im Kontext der Verschachtelung befindet, im Folgenden an Stelle v. Hierzu müssen skalare Dimensionen (Bemerkung A.3) im Framework beachtet werden, da diese beim Durchlaufen aller Schleifen zur Ermittlung der Mutter-Schleife nicht mitgezählt werden dürfen. Somit bewirkt ein Tiling mit Tile-Quantitäten $(1, \ldots, 1, \underbrace{q_v}_{Stelle\ v}, 1, \ldots, 1)$ der transformierten verschachtelten Schleife eine Blockung der zu vektorisierenden Schleife mit Quantität q_v, während alle anderen Schleifen ungeblockt bleiben.

Während des Transformationsprozesses wird hierzu u. A. die Variable `SIMD_OuterBandSize` gesetzt, die den Umfang des Schleifenblocks (also die Anzahl aller Schleifen eines Levels) inkl. skalarer Dimensionen enthält. Dadurch kann man zu einer beliebigen geblockten Schleife die zugehörige Mutterschleife berechnen. Dies ist nötig, um zu der vektorisierten Schleife `SIMD_VectLoop` die entsprechende Mutterschleife `SIMD_MoveInLoop` (durch

Sprung in den nächsthöheren Tile-Block) zu berechnen. `SIMD_VectRowInTileSize` entspricht der gesuchten Stelle v im Level-1-Tiling, welche für die Blockung der vektorisierten Schleife zuständig ist. Diese kann ermittelt werden, indem man die **echten** Schleifen (keine skalaren Dimensionen) bis zu der `SIMD_MoveInLoop` zählt.

Für ein zusätzliches Level-2-Tiling wird hier der Ansatz verfolgt, die **äußerste** Level-2-Steuerschleife zusätzlich zur vektorisierten Schleife, zu blocken.
Dadurch ist die Mutterschleife des Level-2-Tilings in PluTo-SIMD immer die äußerste Schleife der Verschachtelung, wodurch jeder Index dieser Mutterschleife maximal lange verwendet wird. Da dieser Index wiederum (als Steuerschleife) die Indizes der Schleife des Level-2-Tilings beeinflusst, kann dies das Prefetching unterstützen. Eine Evaluation des Einflusses, welche Schleife zusätzlich für das Level-2-Tiling geblockt wird, wird in Kapitel 5.2.2 geliefert.
Dieser Ansatz ist neben des erwähnten Vorteils für das Prefetching hier auch experimentell begründet und kann für zukünftige Erweiterungen von PluTo-SIMD genauer beleuchtet werden, wodurch eine problemspezifisch optimale Schleife für das Level-2-Tiling ausgewählt werden könnte, die auch an die vorliegende Zugriffsstruktur angepasst ist.

Somit werden resultierend die zwei Tile-Quantitätenvektoren

$$\mathbf{q}^{L1} = (1, \ldots, 1, \underbrace{q_v^{L1}}_{Stelle\ v}, 1, \ldots, 1)^S \quad (4.1)$$

$$\mathbf{q}^{L2} = (\underbrace{q_1^{L2}}_{Stelle\ 1}, \ldots, 1, \ldots, \ldots, 1, 1)^S \quad (4.2)$$

(bei Statement-spezifischem Tiling (siehe Abschnitt 4.4.3) für jedes Statement S individuell) generiert. Die erste Stelle entspricht der ersten bzw. äußersten Schleife der Verschachtelung, die zweite Stelle der zweitäußersten usw.

Solch ein Tiling soll nun in *Phase 2* mit an den Cache angepassten Quantitäten q_v^{L1} und ggf. q_1^{L2} durchgeführt werden, indem zunächst alle generierten Transformationsinformationen (welche in `PlutoProg *prog` gespeichert werden) gelöscht werden und im Anschluss alle Operationen, die Einfluss auf `*prog` hatten, mit den neuen Tiling-Vorgaben von $\mathbf{q}^{L1}$ und ggf. $\mathbf{q}^{L2}$, erneut durchgeführt werden.

Die Bestimmung von q_v^{L1} und ggf. q_1^{L2}, unter den in 'Problematik' und 'Konzept' gegebenen Zielsetzungen, ist ein Kernaspekt in der effizienten Umsetzung der Vektorisierung

sowie zur generellen Cache-Optimierung und wird im Abschnitt 4.4.3 ausführlich in seiner Arbeitsweise und Konfigurationsmöglichkeit erläutert.

Resultieren würden im Konzept von PluTo-SIMD z. B. (für ein rechteckiges Tiling ohne weitere Transformation mit Tiefe $d = 2$) die folgenden beiden Beispiele. In Abbildung 4.5 ist die innere j-Schleife vektorisierbar, wodurch jj, also die Mutterschleife von j, die Schleife für das Level-2-Tiling ist. In Abbildung 4.6 musste die vektorisierbare i-Schleife nach innen verschoben/vertauscht werden, wodurch die Schleife für das Level-2-Tiling nicht der Mutterschleife der vektorisierten Schleife entspricht. Die erstellten Tiles für den Level-1-Cache sind in den Abbildungen dunkelgrau dargestellt, die für den Level-2-Cache hellgrau.

```
1 for(ii=0; ii≤floor(M-1,q_1^L2); ii++)
2   for(jj=0; jj≤floor(N-1,q_v^L1); jj++)
3     for(i=q_1^L2*ii; i≤min(M-1,q_1^L2*ii+(q_1^L2 − 1)); i++)
4       for(j=q_v^L1*jj; j≤min(N-1,q_v^L1*jj+(q_v^L1 − 1)); j++)
5         S(i,j);
```

Abbildung 4.5: Tiling nach PluTo-SIMD für zwei Cache Level mit j vektorisierbar

```
1 for(iii=0; iii≤floor(N-1,q_1^L2 * q_v^L1); iii++)
2   for(ii=q_1^L2*iii; ii≤min(floor(N-1,q_v^L1),q_1^L2*iii+(q_1^L2 − 1)); ii++)
3     for(j=0; j≤M-1; j++)
4       for(i=q_v^L1*ii; i≤min(N-1,q_v^L1*ii+(q_v^L1 − 1)); i++)
5         S(i,j);
```

Abbildung 4.6: Tiling nach PluTo-SIMD für zwei Cache Level mit i vektorisierbar

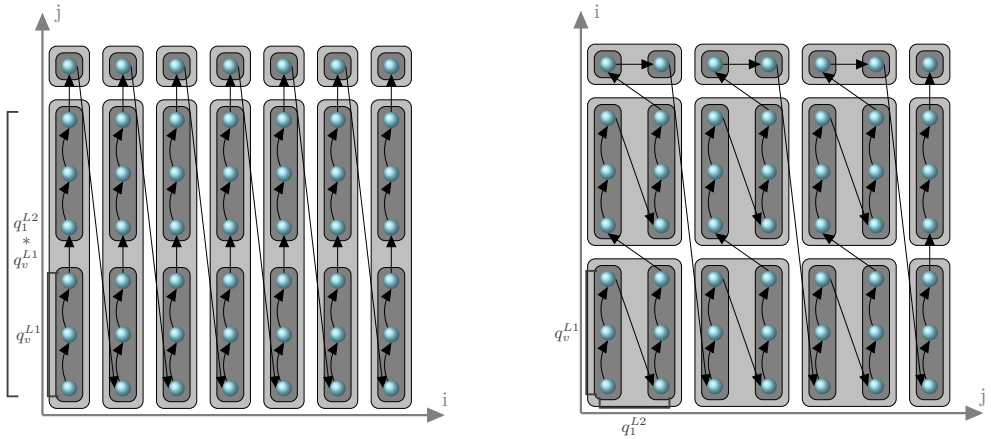


Abbildung 4.7: Visualisierung zu Abbildung 4.5 (links) und Abbildung 4.6 (rechts)

Resultat:
Ein Cache- und SIMD-spezifisches-Tiling wird (wenn möglich) generiert.

In **PluTo-SIMD** wird im *Level-1-Tiling* immer nur genau *eine* Schleife geblockt, kombiniert mit einem *Level-2-Tiling* immer genau *zwei*. Dies ist unabhängig von der Tiefe der Verschachtelung, womit nach der Transformation aus einer **d**-fach verschachtelten Schleife $\mathbf{d+1}$ bzw. $d+1+1 = \mathbf{d+2}$ Schleifen entstehen.

Im Ansatz von **PluTo** entstehen (per default), durch das Blocken aller Schleifen, $d+d = \mathbf{2*d}$ Schleifen bei einfachem Tiling bzw. $d+d+d = \mathbf{3*d}$ Schleifen bei zweifachem!

In den generierten Tiles sind (bis auf randnahe Blöcke) in PluTo-SIMD:

- in den Level-1-Tiles q_v^{L1} **Indexpunkte** und
- in den Level-2-Tiles $q_1^{L2} * q_v^{L1}$ **Indexpunkte** .

Bereich in PluTo für diese Erweiterung:

- *Phase 1* vor der Transformationsphase
- *Analyse* und *Phase 2* nach der (originalen) Transformationsphase

Erweiterungen:

- q_v^{L1} und ggf. q_1^{L2} können Cache-spezifisch und Register-konform angepasst werden,
- dazu muss die Menge der von den Indexpunkten **adressierten Daten** in den auf diese Weise zu generierenden Tiles gemessen und analysiert werden und in die Bestimmung der beiden Quantitäten (unter Beachtung der Cache-Größe) einbezogen werden.

Diese Erweiterungen sind in PluTo-SIMD umgesetzt und im folgenden Abschnitt 4.4.3 erläutert.

4.4.2 Auslesen der Hardware-Cache-Parameter

Die Funktion `get_hardware_cache_infos` wird im Kern von PluTo-SIMD zur Bestimmung verschiedener Hardware-Parameter aufgerufen.
Die Variablen der Form `_VAR` können in der Konfigurationsdatei (Abbildung 4.8) gesetzt werden.

Problematik:
Um das Tiling an den Cache anpassen zu können, muss vorliegen, wie groß die einzelnen Cache-Level der gegebenen CPU sind. Außerdem sollte geprüft werden, ob SSE-Unterstützung für eine Vektorisierung bei der vorliegenden CPU vorhanden ist.

Konzept:
Soll das Tiling an den Cache des vorhandenen Prozessors angepasst werden (`_TCS: Tile Cache-Specific: true`), so können die Cache-Informationen *entweder* automatisch aus der CPU ausgelesen werden (`_SCM: Set the Cache-Sizes manually: false`), *oder*, falls der User den Code z. B. für eine andere CPU optimieren möchte oder diese CPU von den vorhandenen Auslesefunktionen nicht unterstützt wird, kann der User die Cache-Parameter auch in der Konfigurationsdatei (Abbildung 4.8) im Abschnitt `MANUAL CACHE INFORMATION` manuell setzen (`_SCM: Set the Cache-Sizes manually: true`).

Implementierung:
Mittels des CPUID-Befehls, einem Maschinenstatusbefehl, der Zugriff auf Hardware-Parameter von x86-Prozessoren im laufenden Betrieb ermöglicht, können die Größen ausgelesen werden. Hierzu wird das EAX-Register (sowie unter Umständen weitere Register) mit Codenummern (in hexadezimaler Darstellung) gefüllt. Im Anschluss wird der CPUID-Befehl ausgeführt. Nun können die zu der entsprechenden Codenummer gehörenden Informationen aus den Registern EAX, EBX, ECX und EDX ausgelesen werden. Zu diesen Informationen gehören u. A. der Prozessorhersteller (Intel, AMD, etc.), Cache-Informationen, Informationen zu den Prozessorkernen, evtl. vorhandene Vektoreinheiten und viele weitere (Details z. B. in [Int11]).

Zur Ermittlung der Cache-Parametern wird bei Intel-Prozessoren ein HEX-Code zurückgegeben, mittels welchem über Tabellen die entsprechenden Parameter ermittelt werden können. Bei AMD-Prozessoren wird hingegen die Größe, Assoziativität oder Cache-Line-Size des Caches *numerisch* angegeben. Die Tabellen bzw. die daraus resultierenden Rückgabewerte zu den HEX-Codes für Intel-Prozessoren sind in PluTo-SIMD in der Datei '`src/cache_analysis/math_help.h`' gegeben und können dort (für künftige Prozessoren) erweitert werden. Alle Funktionen zur Ermittlung der Hardware-spezifischen Parameter sind im Ordner '`src/cache_analysis/`' zu finden und werden von der zentralen Routine '`get_hardware_cache_infos`' aufgerufen. Bei der Analyse aktueller Prozessoren sind besondere und oft mehrstufige Abfragen nötig (und implementiert), welche in den jeweils aktuellen Datenblättern von Intel zu CPUID veröffentlicht werden.

Resultat:
Es werden Informationen zur Größe und Eigenschaft (Assoziativität und Cache-Line-Size) aller Cache-Level sowie die Unterstützung von SSE (inkl. Versionsnummer) oder MMX, ausgelesen.

Erweiterungen:
Unter Verwendung der hierfür implementierten Funktionen ist es leicht möglich, weitere Informationen zur CPU auszulesen, was auch für weitere Hardware-Anpassungen der Code-Optimierung hilfreich sein kann.

Bereich in PluTo für diese Erweiterung:

- innerhalb des *Kerns von PluTo-SIMD*
- **vor** der Berechnung der q_v^{L1} und ggf. q_1^{L2}

4.4.3 Cache-spezifisches Tiling

Diese Option wird im Kern von PluTo-SIMD zur Bestimmung der Tile-Quantitäten aufgerufen.

Problematik:
Bei vorliegenden Cache-Größen und einer als 'zu vektorisieren' gekennzeichneten Schleife sollen die Tile-Quantitäten q_v^{L1} und ggf. q_1^{L2} nun so bestimmt werden, dass sie die folgenden Eigenschaften erfüllen:

- sie sollen durch Cache-passende Blöcke einen möglichst guten *Fluss* der Daten durch die Cache-Hierarchie gewährleisten
- sie sollen die Schleifen in *Register-konforme* Datenpakete blocken
- es soll ausgegeben werden, wie viel Prozent der zu ladenden Daten aus *innersten* Array-Dimensionen kommen und somit linear aus dem Speicher geladen werden können

Eine Konfigurationsdatei, durch die der User die Parameter dieser Option beeinflussen kann, ist als Beispiel in Abbildung 4.8 dargestellt.

Bemerkung 4.4. *Das Cache-spezifische Tiling vereint einige der bisher in Kapitel 4.4 vorgestellten Aspekte. Diese sind modular als Funktionen in PluTo-SIMD implementiert worden und werden in dieser Hauptfunktion miteinander verknüpft (siehe auch Abbildung 4.15).*

Konzept:
Ist die durch das in Abschnitt 4.4.1 vorgestellte Verfahren als 'zu vektorisieren' gekennzeichnete Schleife (im Folgenden charakterisiert durch ihren Schleifenindex i_v) ermittelt, so soll die Quantität dieser Tiling-Dimension an den Cache angepasst werden. Hierzu wird nun analysiert, wie viele **neue Daten pro Iteration** von i_v geladen werden müssen. Es wird also eine Abbildung von den **Iterationen** auf die **adressierten bzw. zu ladenden Daten** gesucht.

Mithilfe der Bibliothek Clan [Bas08] wird der ursprüngliche Quellcode analysiert und dessen Zugriffe mathematisch als Zugriffsfunktionen $Z \cdot \vec{i} + z$ formalisiert. Da aber der transformierte Code, also der Ziel-Code, optimiert werden soll, müssen auch die Zugriffe des **transformierten Codes** analysiert werden. Für jedes Statement S wurde eine Transformation $\mathcal{T}_S$ berechnet, durch welche die Statements im Framework transformiert werden (Abschnitt 3.7). Im Folgenden ist $\mathcal{T}_S(Z \cdot \vec{i} + z) = \tilde{Z} \cdot \vec{i} + \tilde{z}$ der transformierte Zugriff zu $Z \cdot \vec{i} + z$.

Bemerkung 4.5 (Transformation)**.** *Im PluTo-Framework wird die* ***Tiling-Matrix*** H_S *als Transformationsstruktur gespeichert. Um die Zugriffe zu transformieren wird allerdings die* ***Clustering-Matrix*** G_S *benötigt. Hierfür wurde ein Gauß-Jordan-Algorithmus zur Invertierung der Tiling-Matrix in PluTo-SIMD implementiert, da* $G_S = H_S^{-1}$ *gilt (vgl. Abschnitt 3.8.1).*

Sei $\mathcal{I} = \{i_1, \ldots, i_v, \ldots, i_d\}$ die Menge aller Schleifenindizes einer verschachtelten Schleife. Eine Dimension d' eines Zugriffs auf ein Array $\tilde{z}_{d',1} * i_1 + \ldots + \tilde{z}_{d',v} * i_v + \ldots + \tilde{z}_{d',d} * i_d + \tilde{z}_{d'}$ (vgl. (3.2)) ist **unabhängig** von der vektorisierten Schleife, wenn $\tilde{z}_{d',v} = 0$. Somit ist ein gesamter Zugriff $\tilde{Z} \cdot \vec{i} + \tilde{z}$ unabhängig von der vektorisierten Schleife, wenn dies für jede Dimension gilt, was resultierend heißt, dass die **v-te Spalte von $\tilde{\mathbf{Z}}$ (im Folgenden $\tilde{\mathbf{Z}}_{\mathbf{i_v}}$)** eine *Nullspalte* ist. Für solch einen Zugriff auf ein Array müssen innerhalb dieser i_v-Schleife keine neuen Daten geladen werden.

Beispiel 4.6 (Einfluss der Transformation)**.**
$a[i_1][i_v] \rightarrow Z_{i_v} = (0, 1)$*, also ein Zugriff der vektorisierten Schleife* i_v *mit Koeffizient* 1 *in der* ***zweiten*** *Dimension des Arrays.*
Durch Transformation $\mathcal{T} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ *ergibt sich*
$a[i_v][i_1] \rightarrow Z_{i_v} = (1, 0)$*, also ein Zugriff der vektorisierten Schleife* i_v *mit Koeffizient* 1 *in der* ***ersten*** *Dimension des Arrays.*

Es folgen Beispiele zur *Zählung von Zugriffen* der vektorisierten Schleife.

Beispiel 4.7.

```
1 for(i1=0; ...)
2   for(i2=0; ...)
3     for(iv=0; iv<qv^L1; iv++)
4       ... a[i1][i2] ...
```

Hier müssen in der innersten Schleife pro Iteration ***keine*** *neue Datenelemente für diesen Zugriff geladen werden.*

Hat $\tilde{Z}_{i_v}$ mindestens ein Nicht-Nullelement, so muss pro Iteration ein neues Datenelement für diesen Zugriff geladen werden.

Liegen nun in einer Schleife verschiedene Zugriffe (mit unterschiedlichen $\tilde{Z}_{i_v}$) auf dasselbe Array vor, so müssen diese verschiedenen Zugriffe gezählt werden.

Beispiel 4.8.

```
1 for(i1=0; ...)
2   for(iv=0; iv<qv^L1; iv++)
3     ... a[i1][iv] + a[iv][i1] ...
```

Hier müssen in der innersten Schleife pro Iteration ***zwei*** *neue Datenelemente geladen werden.*

Unterscheiden sich zwei Zugriffe $\tilde{Z}^1 \cdot \vec{i} + \tilde{z}^1$ und $\tilde{Z}^2 \cdot \vec{i} + \tilde{z}^2$ bezüglich i_v lediglich im konstanten Term ($\tilde{z}_v^1$ und $\tilde{z}_v^2$), so werden die Zugriffe als 'gleich' angesehen, da i. d. R. sehr große q_v^{L1} (in der Größenordnung von $\sim 10^3$) generiert werden. Somit wird eine Menge von adressierten Daten durch $i_v \in \{s, s+1, \ldots, s+q_v^{L1}-1\}$ in dieser Schleife geladen und die Konstanten werden vernachlässigbar, da deren Zugriffe (bis auf den Rand) automatisch mitgeladen werden.

Beispiel 4.9.

```
1 for(i1=0; ...)
2   for(iv=0; iv<qv^L1; iv++)
3     ... a[i1][iv+1] + a[i1][iv+2] ...
```

Hier müssen in der innersten Schleife pro Iteration ***nicht zwei*** *neue Datenelemente geladen werden, sondern nur* ***eins****, da eines in der vorherigen Iteration bereits benutzt wurde. Insgesamt sind es hier* $q_v^{L1}+\mathbf{1}$ *zu ladende Elemente,* ***nicht*** $2 * q_v^{L1}$.

Sind **mehrere** Statements (im Folgenden ein **Block Bl**) auf einer *Tiefe* der Schleife vorhanden, so werden die Zugriffe analog behandelt.

Beispiel 4.10.

```
1 for(i1=0; ...)
2   for(iv=0; iv < qv^L1;iv++)
3     {
4     ... a[i1][iv+1] ...;
5     ... a[i1][iv+2] ...;
6     }
```

Analog zu Beispiel 4.9 müssen hier in der innersten Schleife pro Iteration ***nicht zwei*** *neue Datenelemente geladen werden, sondern nur* ***eins****.*

Definition 4.11. *Zusammenfassend gilt, dass zwei Zugriffe $Z^1_{i_v}$ und $Z^2_{i_v}$ auf dasselbe Array für i_v* ***verschieden*** *sind, wenn $Z^1_{i_v} \neq Z^2_{i_v}$. Bei einer Menge von k Zugriffen $Z^1_{i_v}, \ldots, Z^k_{i_v}$ auf dasselbe Array müssen genau* `t` *neue Daten pro Iteration geladen werden, wenn genau* `t` *paarweise verschiedene $Z^x_{i_v} \neq \vec{0}$ vorhanden sind. Diese Menge der paarweise verschiedenen Zugriffe auf ein Array Arr_j in einem Statement S_i wird in PluTo-SIMD in einer verketteten Liste in $AAC_{S_i}[Arr_j]$ gespeichert.*

Bemerkung 4.12. *In der Implementierung von PluTo-SIMD entspricht die Struktur* `array_access_combination`*(AAC) einer solchen Liste von Zugriffen und ein Array mit einer Statement-Dimension und einer Array-Dimension der Sammlung all dieser.*

Die so definierte Menge an Elementen, die pro Iteration von i_v durch alle Zugriffe auf die verschiedenen Arrays in einem Block Bl von Statements geladen werden müssen, kann nun durch

$$\text{Elements per Iteration} : \text{ElPeIt} = ||\mathcal{T}(\mathcal{Z}(i_v))||^{Bl}_{\text{diff}} \tag{4.3}$$

(für alle Zugriffe eines Blocks zusammen) formalisiert werden.

Bemerkung 4.13. *In PluTo-SIMD gibt es für die Generierung des Cache-spezifischen Tilings die drei Optionen MIN, MAX und TPS (konfigurierbar durch die Option* `_TAA`*, vgl. Abbildung 4.8). Bei MIN und MAX wird für* ***alle*** *Statements global die Tile-Größe gesetzt, die aus der minimalen bzw. maximalen Anzahl von Zugriffen in einem Block resultiert. Dies hat den Hintergrund, dass es in PluTo nur möglich ist, Tile-Quantitäten für* ***alle*** *Statements global zu definieren. In PluTo-SIMD wurde das Framework erweitert, sodass eine exklusive Definition der Tile-Quantitäten für jedes Statement bzw. jeden Statementblock möglich ist. Dadurch kann mit der Option TPS die für jeden Block spezifisch durch ElPeIt ermittelte Quantität zur Generierung von q_v^{L1} genutzt werden, wodurch ein für jeden Statementblock individuelles Tiling entsteht. Die Auswirkungen der drei Optionen werden in Abschnitt 5.2.2 experimentell beleuchtet.*

Darauf aufbauend sollen nun so viele Iterationen in einem Tile abgearbeitet werden, dass die adressierten Daten insgesamt 'gut' in den Cache passen. Somit können die Daten für die gesamte $\mathbf{i_v}$-Schleife in den Cache geladen werden. 'Gut' heißt in diesem Fall, dass nicht unbedingt 100 % des Caches angenommen werden sollten, da zum einen die Abschätzung der Elemente nicht absolut exakt ist und zum anderen weitere Daten im Cache vorhanden sein können. Ein geeigneter Anteil des Caches, der für den Algorithmus zur Verfügung steht, kann durch den User über den Parameter `_ROC` (hier ρ) in der Konfigurationsdatei angegeben werden.

Dieser Prozentsatz

$$\text{Ratio of Cache to use} : \rho \tag{4.4}$$

ist per default auf 90 % gesetzt, die Wahl wird in Kapitel 5 durch die Evaluation begründet, da sie eine gute Wahl für die Matrix-Multiplikation darstellt. Allgemein sollte der Wert für einen theoretisch optimalen Datenfluss kleiner als 100 % sein, aber möglichst nahe an diesem liegen. Wäre der Cache exklusiv für den Algorithmus vorhanden und die Berechnung der adressierten Daten exakt, so wäre die theoretisch optimale Wahl $\rho = 1 = 100\,\%$.

Die Größe der Caches ($\mathcal{C}_{L1}$ und $\mathcal{C}_{L2}$) kann entweder aus der vorhandenen Hardware ausgelesen werden (`_SCM`=false) oder vom User manuell vorgegeben werden (`_SCM`=true). Ist die Cache-Größe in KB vorhanden, so kann durch

$$\text{Cache-Size in Elements} : \text{CaSiEl} = \frac{\mathcal{C}_{L1} * 1024}{\mathcal{D}} \tag{4.5}$$

berechnet werden, wie viele Elemente von einem Datentyp der Größe D *Bytes* in den Level-1-Cache passen.

Damit alle nötigen **Daten** für die i_v-Schleife in den Level-1-Cache passen, müsste also (ohne weitere beachtete Aspekte)

$$\text{CaSiEl} = \text{ElPeIt} * \text{Iterations} \quad bzw. \quad \text{Iterations} = \frac{\text{CaSiEl}}{\text{ElPeIt}} \tag{4.6}$$

oder durch ρ angepasst

$$\rho * \text{CaSiEl} = \text{ElPeIt} * \text{Iterations} \quad bzw. \quad \text{Iterations} = \frac{\rho * \text{CaSiEl}}{\text{ElPeIt}} \tag{4.7}$$

gelten.

Über die Cache-Anpassung hinaus soll die i_v-Schleife aber auch Register-konform für die Vektorisierung sein. Somit soll die Anzahl der Iterationen dieser Schleife ein Vielfaches von der Anzahl der Elemente pro Register in der Vektoreinheit sein, damit alle Berechnungen (und somit **Iterationen**) in vollständigen i_v-Schleifen, also bis auf ggf. die Rand-Blöcke, jeweils komplett auf die Vektorregister aufgeteilt werden können (die Größe eines Registers *in Bit* wird mit $\mathcal{R}$ bezeichnet). Hierzu wird die Anzahl

$$\text{Elements per Register} : \text{ElPeRe} = \frac{\mathcal{R}}{8 * \mathcal{D}} \tag{4.8}$$

berechnet und zu dem in (4.7) berechneten Wert der nächstkleinere, der ein Vielfaches von ElPeRe ist, gewählt.

Wenn ein Cache-spezifisches Tiling generiert werden soll, so berechnet sich die Quantität für den Level-1-Cache also insgesamt durch

$$\text{L1-Tile-Size} \quad : \quad q_v^{L1} \quad = \quad \left\lfloor \frac{\rho * \text{CaSiEl} * \frac{1}{\text{ElPeIt}}}{\text{ElPeRe}} \right\rfloor * \text{ElPeRe} \quad . \tag{4.9}$$

Der ideale Faktor für das Level-2-Tiling ist ***hier*** das Verhältnis der Cache-Größen zueinander, sodass der Block aus Level-1- und Level-2-Tiling in den Level-2-Cache passt. Für $\rho = 1$ und einem für den Algorithmus exklusiven Cache passt dies theoretisch perfekt.

Explizit ergibt sich somit für die beiden gesuchten Werte

$$\text{L1-Tile-Size} \quad : \quad q_v^{L1} \quad = \quad \left\lfloor \frac{\rho * \frac{\mathcal{C}_{L1} * 1024}{\mathcal{D}} * \frac{1}{||\mathcal{T}(\mathcal{Z}(i_v))||_{\text{diff}}}}{\frac{\mathcal{R}}{8 * \mathcal{D}}} \right\rfloor * \frac{\mathcal{R}}{8 * \mathcal{D}} \tag{4.10}$$

$$\text{L2-Tile-Size} \quad : \quad q_1^{L2} \quad = \quad \frac{\mathcal{C}_{L2}}{\mathcal{C}_{L1}} \tag{4.11}$$

mit

ρ	: Prozentualer Anteil der Cache-Größe die genutzt werden soll (`_ROC`)
$\mathcal{C}_{L1}$	: Level-1-Cache-Größe in KByte (`_ML1` $\vee$ Hardware)
$\mathcal{C}_{L2}$	: Level-2-Cache-Größe in KByte (`_ML2` $\vee$ Hardware)
$\mathcal{D}$	: Größe des Datentypen in Byte (`_SDT`)
$\|\|\mathcal{T}(\mathcal{Z}(i_v))\|\|_{\text{diff}}^{Bl}$	: Anzahl der neu zu ladenden Daten pro Iteration in Block Bl
$\mathcal{R}$	: Größe der Register der Vektoreinheit in Bit (`_RSZ`).

Implementierung:
Die Parameter für das Cache-spezifische Tiling können durch die Konfigurationsdatei '**cache_configuration.conf**' im Ordner 'src/cache_analysis' gesetzt werden, z. B. :

```
1  ##############################################################################
2  #-*-*-*-*-*-*-*-*-*-*OPTIONS FOR CACHE-AND-SIMD-USABILITY-*-*-*-*-*-*-*-*-*-*-#
3  ##############################################################################
4
5  _TCS: Tile Cache-Specific                                    : true
6
7  ########################(if Tile Cache-Specific is FALSE)#########################
8
9  _ST1: STATIC L1-tile-size                                    : 64
10 _ST2: STATIC L2-tile-factor                                  : 4
11
12 ########################(if Tile Cache-Specific is TRUE)##########################
13
14 _TAA: Tile in (MAX/MIN) most inner access amount             : max
15
16 _TL1: Tile for L1-Cache                                      : true
17 _TL2: Tile for L2-Cache                                      : false
18          (both Options may be true -> two level tiling)
19
20 _TPS: Tile per-statement-cache-specific                      : true
21          (deactivates max/min-ability)
22
23 ------------------------------MANUAL CACHE INFORMATION------------------------------
24
25 _SCM: Set the Cache-Sizes manually                           : false
26
27 _ML1: L1-Cache-Size in KB                                    : 64
28          (if it should be set by the user)
29
30 _ML2: L2-Cache-Size in KB                                    : 1024
31          (if it should be set by the user)
32
33 ------------------------------------PARAMETERS------------------------------------
34
35 _SDT: Size of Data-Type in Bytes (e.g. 4 for float)          : 4
36
37 _RSZ: Registersize in Bit (e.g. 128 for SSE)                 : 128
38
39 _ROC: Ratio of cache to use (e.g. 0.9 for 90%)               : 0.9
40
41 ##################################MODE########################################
42
43 _DBG: Debug-Mode for Cache-Tiling                            : true
44
45 ##############################################################################
46 #-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-#
47 ##############################################################################
```

Abbildung 4.8: Konfigurationsdatei für das Cache- und SIMD-angepasste Tiling

Ein Pseudocode zur Implementierung des gesamten Cache-spezifischen Tiling Algorithmus ist in Algorithmus 1 dargestellt.

Algorithmus 1: Cache-spezifischer Tiling Algorithmus

Data : Transformationen $\mathcal{T}_{S_i}$ sowie die Array-Zugriffsinformationen von Clan
Result : Cache-spezifisches q_v^{L1} und (bei entsprechender Konfiguration) q_1^{L2}

```
1  begin
2      ρ = _ROC ; D = _SDT ; R = _RSZ ;
3      if _TCS then                                   /* Cache-spezifisches Tiling wird generiert */
4          foreach Statement S_i do
5              foreach Array Arr_j do
6                  AAC_{S_i}[Arr_j] = ∅;
7                  foreach Zugriff Z * i⃗ + z in S_i auf Array Arr_j do
8                      Extrahiere Zugriff der zu vektorisierenden Schleife → Z_{i_v} ∈ ℤ^d;
9                      Transformiere diesen Zugriffsvektor durch T → T(Z_{i_v});
10                     if T(Z_{i_v}) ∉ AAC_{S_i}[Arr_j] then
11                         AAC_{S_i}[Arr_j] ← AAC_{S_i}[Arr_j] ∪ T(Z_{i_v});
12                         AAC_{S_i}[Arr_j] ↦ T(Z_{i_v}).counter = 1;
13                     else AAC_{S_i}[Arr_j] ↦ T(Z_{i_v}).counter++;
14         Finde Statemtents S_i in jeweils gleichem Block Bl_k;
15         foreach Block Bl_k von Statements S_i im transformierten Raum do
16             foreach Array Arr_j do AAC_{∪_i S_i}[Arr_j] = ⋃_i AAC_{S_i}[Arr_j];/* vgl.10-13 */
17             ||T(Z_{i_v})||^{Bl_k}_{diff} = #Zugriffe(Bl_k) = ElPeIt(Bl_k) = Σ_{allArr_j} |AAC_{∪_i S_i}[Arr_j]|;
18         if _SCM then                               /* wenn die Cache-Größe manuell angegeben wird */
19             C_{L1} = _ML1;                                          /* gegeben in KB */
20             C_{L2} = _ML2;                                          /* gegeben in KB */
21         else lese C_{L1} und C_{L2} aus der Hardware ;              /* gegeben in KB */
22         if !_TL1 && _TL2 then C_{L1} = C_{L2} ;                     /* EXKLUSIVES L2-Tiling */
23         CaSiEl = (C_{L1}*1024)/D;                                   /* berechnet in Bytes */
24         ElPeRe = R/(8*D);                                           /* berechnet in Bit */
25         if _TPS then                                /* individuelle Größe für jedes Statement/Block */
26             foreach Statement S_i do q(S_i)^{L1}_v = ⌊(ρ*CaSiEl)/(ElPeIt(Block(S_i))*ElPeRe)⌋ * ElPeRe;
27         else                                        /* gleiche Größe für jedes Statement */
28             if _TAA==max then ElPeIt = max{Zugriffe in einem Block};
29             else ElPeIt = min{Zugriffe in einem Block};          /* ⇒ _TAA==min */
30             q_v^{L1} = ⌊(ρ*CaSiEl)/(ElPeIt*ElPeRe)⌋ * ElPeRe;
31         if _TL1 && _TL2 then q_1^{L2} = C_{L2}/C_{L1} ;          /* ZUSÄTZLICHES L2-Tiling */
32     else                                            /* vom User definiertes statisches Tiling */
33         q_v^{L1} = _ST1;
34         q_1^{L2} = _ST2;
```

Resultat:
q_v^{L1} und ggf. q_1^{L2} werden, an die in jedem Tile zu adressierende Datenmenge und den vorhandenen Cache angepasst, generiert. Somit sind die beiden Quantitäten so gewählt, dass

- alle für die i_v-Schleife benötigten **Daten** in den Cache passen **und**
- die **Iterationen** der i_v-Schleife vollständig auf die Register der Vektoreinheit aufgeteilt werden können,

was in vielen Fällen den Cache-Fluss optimiert.

Außerdem wird ausgegeben, wie viel Prozent der vektorisierten Zugriffe auf die innersten Dimensionen der Arrays stattfinden. Solche Zugriffe sind für Cache-Zugriffe optimal, da die Daten linear aus dem Speicher geladen werden können. Diese Kennzahl wird dem User im PluTo-SIMD Framework durch die Ausgabe '`[CACHE] p % of vectorized accesses are INNERMOST!`' mitgeteilt. Ein hohes `p` (optimaler Weise 100 %) ermöglicht außerdem ein gutes **Prefetching** der Daten.

Ausführliche Informationen zum Cache-spezifischen Tiling Algorithmus können in PluTo-SIMD in der Konfigurationsdatei durch den `DEBUG`-Modus ausgegeben werden.

Erweiterungen:

- Zugriffe, die für eine Vektorisierung ungünstig sind, könnten wiederum angepasst geblockt werden (siehe Kapitel 5 und 6), um in solchen Fällen das Prinzip des Hardware-spezifischen Tilings auf mehrdimensionale, Cache-angepasste Tilings zu erweitern.
- Die Optimierungsprobleme könnten Transformationen suchen, welche ein möglichst hohes `p` erzeugen bzw. sie könnten die zu vektorisierende Schleife (falls mehrere solche möglich sind) nach diesem Kriterium auswählen.

Bereich in PluTo für diese Erweiterung:

- innerhalb des *Kerns von PluTo-SIMD*

Einige Beispiele sollen im Folgenden den Prozess des Cache-spezifischen Tilings veranschaulichen und verdeutlichen. Hierzu wird die Konfigurationsdatei aus Abbildung 4.8 angenommen. Die Beispiele sind **konkrete Codes aus dem PluTo-Framework** [**Bona**], die z. T. in Kapitel 5 als Test-Beispiele dienen.

Beispiel 4.14.

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

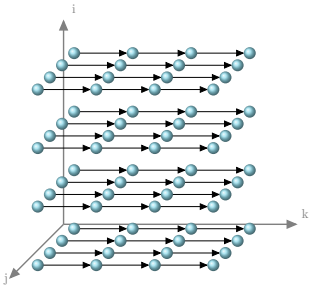


Abbildung 4.9: Standard Matrix-Multiplikation für $\alpha = 1$

***Phase 1**: Es gibt **ein** Statement S, für welches die Transformations-/Tiling-Matrix* $\mathcal{T}_S = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ *durch PluTo generiert wurde, also ein direktes rechteckiges Tiling.*

***Phase 2**: Die vektorisierte Schleife ist* $\mathbf{j}$ *(*$\mathbf{i}$ *wäre ebenso möglich,* $\mathbf{k}$ *ist durch die Abhängigkeiten nicht vektorisierbar), dies entspricht also allgemein den Tile-Quantitätenvektoren*

$$\mathbf{q}^{L1} = (1, \quad q_v^{L1}, \quad 1)^S \tag{4.12}$$

$$\mathbf{q}^{L2} = (q_1^{L2}, \quad 1, \quad 1)^S \ . \tag{4.13}$$

Da die Transformation in diesem Fall die Identität ist, gilt $\tilde{Z} \cdot \vec{i} + \tilde{z} = Z \cdot \vec{i} + z$*, sodass die Transformation keinen Einfluss auf die Zugriffe hat.*
Es gibt insgesamt drei Zugriffe auf Arrays durch die vektorisierte Schleife $\mathbf{j}$ *(*$C[i][\mathbf{j}] = C[i][\mathbf{j}] + alpha * A[i][k] * B[k][\mathbf{j}]$*), allerdings sind nur zwei davon verschieden, somit ist* $\boldsymbol{ElPeIt} = ||\mathbf{T}(\mathbf{Z_{i_v}})||^{\mathbf{S}}_{\boldsymbol{diff}} = \mathbf{2}$*.*
Es gilt durch die Zugriffe außerdem $\boldsymbol{p=100\ \%}$*.*
Die aus der Hardware gelesenen Cache-Größen sind $\mathcal{C}_{\mathbf{L1}} = \mathbf{32}$ *KB und* $\mathcal{C}_{\mathbf{L2}} = \mathbf{256}$ *KB.*
Die SSE-Register haben $\mathcal{R} = \mathbf{128}$*-Bit Größe und der Datentyp ist float, somit* $\mathcal{D} = \mathbf{4}$ *Byte.*
Als prozentuale Cache Auslastung wurde hier $\boldsymbol{\rho} = \mathbf{0.9}$ *gewählt.*
Somit passen $\boldsymbol{CaSiEl} = \frac{\mathbf{32*1024}}{\mathbf{4}} = \mathbf{8192}$ *Elemente vom Typ float in den Cache und* $\boldsymbol{ElPeRe} = \frac{\mathcal{R}}{\mathbf{8}*\mathcal{D}} = \frac{\mathbf{128}}{\mathbf{8*4}} = \mathbf{4}$ *Elemente in ein SSE-Register.*

Resultierend gilt also $\mathbf{q_v^{L1}} = \left\lfloor \frac{\boldsymbol{\rho}*\boldsymbol{CaSiEl}}{\boldsymbol{ElPeIt}*\boldsymbol{ElPeRe}} \right\rfloor * \boldsymbol{ElPeRe} = \left\lfloor \frac{\mathbf{0.9*8192}}{\mathbf{2*4}} \right\rfloor * \mathbf{4} = \mathbf{3684}$

und $\mathbf{q_1^{L2}} = \frac{\mathbf{C_{L2}}}{\mathbf{C_{L1}}} = \frac{\mathbf{256}}{\mathbf{32}} = \mathbf{8}$*, womit sich die Tile-Quantitätenvektoren*

$$\mathbf{q}^{L1} = (1, \quad 3684, \quad 1)^S \tag{4.14}$$

$$\mathbf{q}^{L2} = (8, \quad 1, \quad 1)^S \tag{4.15}$$

ergeben.

Mit dieser Konfiguration wird nun das Tiling in PluTo-SIMD generiert, woraus folgender Code resultiert.

```
1 for(t1=0;t1<=floor(M-1,8);t1++) {
2   for(t2=0;t2<=floor(N-1,3684);t2++) {
3     for(t3=0;t3<=K-1;t3++) {
4       for(t4=8*t1;t4<=min(M-1,8*t1+7);t4++) {
5 {
6  lbv=3684*t2; ubv=min(N-1,3684*t2+3683);
7 #pragma ivdep
8 #pragma vector always
9  for(t9=lbv; t9<=ubv; t9++) {
10    C[t4][t9]=beta*C[t4][t9]+alpha*A[t4][t3]*B[t3][t9];;
11    }
12 }
13         }
14       }
15     }
16   }
```

Abbildung 4.10: Durch PluTo-SIMD Cache-spezifisch transformierte Matrix-Multiplikation

Beispiel 4.15.

```
1 for(i=1; i<T; i++)
2   for(j=1; j<N-1; j++)
3     a[i][j] = 0.333*(a[i-1][j] + a[i-1][j-1]
                       + a[i-1][j+1]);
```

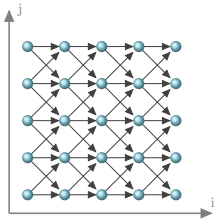


Abbildung 4.11: 1-D Jacobi, vgl. Abbildung 3.30

Phase 1*: Es gibt* ***ein*** *Statement S, für das PluTo die Transformations-/Tiling-Matrix* $\mathcal{T}_\mathcal{S} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$ *generiert hat, also ein Parallelepiped-Tiling (vgl. Beispiel 3.64).*

***Phase 2**: Die vektorisierte Schleife ist die transformierte j-Schleife.*
***Vor** der Transformation sind die Zugriffe durch die j-Schleife bei $\mathcal{I} = (i, j)$ jeweils in der zweiten Dimension des Arrays $Z^1_{i_v} = (0, 1)$, $Z^2_{i_v} = (0, 1)$ und $Z^3_{i_v} = (0, 1)$. Transformiert durch $\mathcal{T}$ ergibt sich $\tilde{Z}^1_{i_v} = (0, 1)$, $\tilde{Z}^2_{i_v} = (0, 1)$ und $\tilde{Z}^3_{i_v} = (0, 1)$, somit drei (bis auf die Konstanten) identische Zugriffe, also $\boldsymbol{ElPeIt} = ||\mathbf{T}(\mathbf{Z_{i_v}})||^{\mathbf{S}}_{\boldsymbol{diff}} = \mathbf{1}$.*
Es gilt durch die Zugriffe $\boldsymbol{p=100\,\%}$.
Die Daten des Caches und der SSE-Register sind identisch mit denen aus Beispiel 4.14. Resultierend gilt also $\mathbf{q^{L1}_v} = \left\lfloor \frac{\rho * \boldsymbol{CaSiEl}}{\boldsymbol{ElPeIt} * \boldsymbol{ElPeRe}} \right\rfloor * \boldsymbol{ElPeRe} = \left\lfloor \frac{\mathbf{0.9 * 8192}}{\mathbf{1 * 4}} \right\rfloor * \mathbf{4} = \mathbf{7372}$ und $\mathbf{q^{L2}_1} = \frac{\mathbf{C_{L2}}}{\mathbf{C_{L1}}} = \frac{\mathbf{256}}{\mathbf{32}} = \mathbf{8}$, womit sich die Tile-Quantitätenvektoren

$$\mathbf{q}^{L1} = (1, 7372, 1)^S \quad (4.16)$$

$$\mathbf{q}^{L2} = (8, 1, 1)^S \quad (4.17)$$

ergeben.

Mit dieser Konfiguration wird nun das Tiling in PluTo-SIMD generiert, woraus folgender Code resultiert.

```
1 for(t1=0;t1<=floor(T-1,8);t1++) {
2  for(t2=ceil(2*t1-1842,1843);t2<=min(floor(T+N-3,7372),
                                        floor(8*t1+N+5,7372));t2++) {
3   for(t3=max(max(1,8*t1),7372*t2-N+2);t3<=min(min(T-1,8*t1+7),
                                                 7372*t2+7370);t3++) {
4 {
5    lbv=max(7372*t2,t3+1); ubv=min(7372*t2+7371,t3+N-2);
6    #pragma ivdep
7    #pragma vector always
8    for(t6=lbv; t6<=ubv; t6++) {
9     a[t3][-t3+t6]=0.333*(a[t3-1][-t3+t6]+a[t3-1][-t3+t6-1]
                           +a[t3-1][-t3+t6+1]);;
10    }
11 }
12   }
13  }
14 }
```

Abbildung 4.12: Durch PluTo-SIMD Cache-spezifisch transformierter Jacobi

Beispiel 4.16.

```
1 /* Center and reduce the column vectors. */
2 for(i = 1; i <= N; i++)
3   for(j = 1; j <= M; j++)
4   {
5     data[i][j] -= mean[j];
6     data[i][j] /= sqrt(N) * stddev[j];
7   }
8
9 /* Calculate the M * M correlation matrix. */
10 for(j1 = 1; j1 <= M-1; j1++)
11 {
12   symmat[j1][j1] = 1.0;
13   for(j2 = j1+1; j2 <= M; j2++)
14   {
15     symmat[j1][j2] = 0.0;
16     for (i = 1; i <= N; i++)
17       symmat[j1][j2] += ( data[i][j1] * data[i][j2]);
18     symmat[j2][j1] = symmat[j1][j2];
19   }
20 }
```

Abbildung 4.13: Berechnung der Korrelationsmatrix

Da dieses Beispiel durch die Verschachtelungen deutlich komplizierter ist, soll es an dieser Stelle nicht in jedem Schritt vorgerechnet werden, die Ergebnisse sind:

- *es gibt 6 Statements*
- *alle werden durch die Identität 'transformiert'*
- *jeweils die zweite Schleife einer Verschachtelung wird (wenn möglich) vektorisiert*
- *es gibt verschiedene Tile-Größen für die Statements (durch* `_TPS`*)*
  $\mathbf{q_v^{L1}}(S1, S2) = 2456$, $\mathbf{q_v^{L1}}(S3) = 7368$, $\mathbf{q_v^{L1}}(S4, S5) = 3684$ *und* $\mathbf{q_v^{L1}}(S3) = 1$ *durch verschieden viele Zugriffe der jeweils vektorisierten Schleife [Statement 3 wird somit* ***nicht*** *vektorisiert (Blockgröße:* 1*)]*
- *außerdem gibt es durch verschiedene Kombinationen mit weiteren PluTo-Optionen verschieden viele Zugriffe auf die innersten Dimensionen, z. B.* $\mathbf{p}$*=88.89 % normal,* $\mathbf{p}$*=85.71 % ohne Schleifenvereinigungen oder* $\mathbf{p}$*=70.00 % mit maximaler Schleifenvereinigung*

Konfigurationsmöglichkeiten für die Transformation in PluTo wurden nicht verwendet, sodass (wenn nötig) Standardwerte verwendet wurden. Mit dieser Konfiguration wird das Tiling in PluTo-SIMD generiert, woraus folgender Code resultiert.

```
1 if(M >= 1) {
2  if(M >= 2) {
3   for(t2=0;t2<=floor(M-1,8);t2++) {
4    for(t3=ceil(t2-920,921);t3<=floor(M,7368);t3++) {
5     for(t5=max(1,8*t2);t5<=min(min(M-1,8*t2+7),7368*t3+7366);t5++) {
6 {
7 lbv=max(7368*t3,t5+1); ubv=min(M,7368*t3+7367);
8 #pragma ivdep
9 #pragma vector always
10 for(t10=lbv; t10<=ubv; t10++) {
11      symmat[t5][t10]=0.0;;
12     }
13 }
14    }
15   }
16  }
17 }
18 for(t2=1;t2<=M-1;t2++) {
19   symmat[t2][t2]=1.0;;
20  }
21 if(N >= 1) {
22  for(t2=0;t2<=floor(N,8);t2++) {
23   for(t3=0;t3<=floor(M,2456);t3++) {
24    for(t5=max(1,8*t2);t5<=min(N,8*t2+7);t5++) {
25 {
26 lbv=max(1,2456*t3); ubv=min(M,2456*t3+2455);
27 #pragma ivdep
28 #pragma vector always
29 for(t10=lbv; t10<=ubv; t10++) {
30     data[t5][t10]-=mean[t10];;
31     data[t5][t10]/=sqrt(N)*stddev[t10];;
32      }
33 }
34    }
35   }
36  }
37 }
```

```
38 if((M >= 2) && (N >= 1)) {
39  for(t2=0;t2<=floor(M-1,8);t2++) {
40   for(t3=ceil(2*t2-920,921);t3<=floor(M,3684);t3++) {
41    for(t4=1;t4<=N;t4++) {
42     for(t5=max(1,8*t2);t5<=min(min(M-1,8*t2+7),3684*t3+3682);t5++) {
43 {
44 lbv=max(3684*t3,t5+1); ubv=min(M,3684*t3+3683);
45 #pragma ivdep
46 #pragma vector always
47 for(t10=lbv; t10<=ubv; t10++) {
48         symmat[t5][t10]+=(data[t4][t5]*data[t4][t10]);;
49       }
50 }
51     }
52    }
53   }
54  }
55 }
56 if(M >= 2) {
57  for(t2=0;t2<=floor(M-1,8);t2++) {
58   for(t3=ceil(2*t2-920,921);t3<=floor(M,3684);t3++) {
59    for(t5=max(1,8*t2);t5<=min(min(M-1,8*t2+7),3684*t3+3682);t5++) {
60 {
61 lbv=max(3684*t3,t5+1); ubv=min(M,3684*t3+3683);
62 #pragma ivdep
63 #pragma vector always
64 for(t10=lbv; t10<=ubv; t10++) {
65       symmat[t10][t5]=symmat[t5][t10];;
66      }
67 }
68     }
69    }
70   }
71  }
72 }
```

Abbildung 4.14: Durch PluTo-SIMD Cache-spezifisch transformierte Berechnung der Korrelationsmatrix

4.5 Gesamtprozess

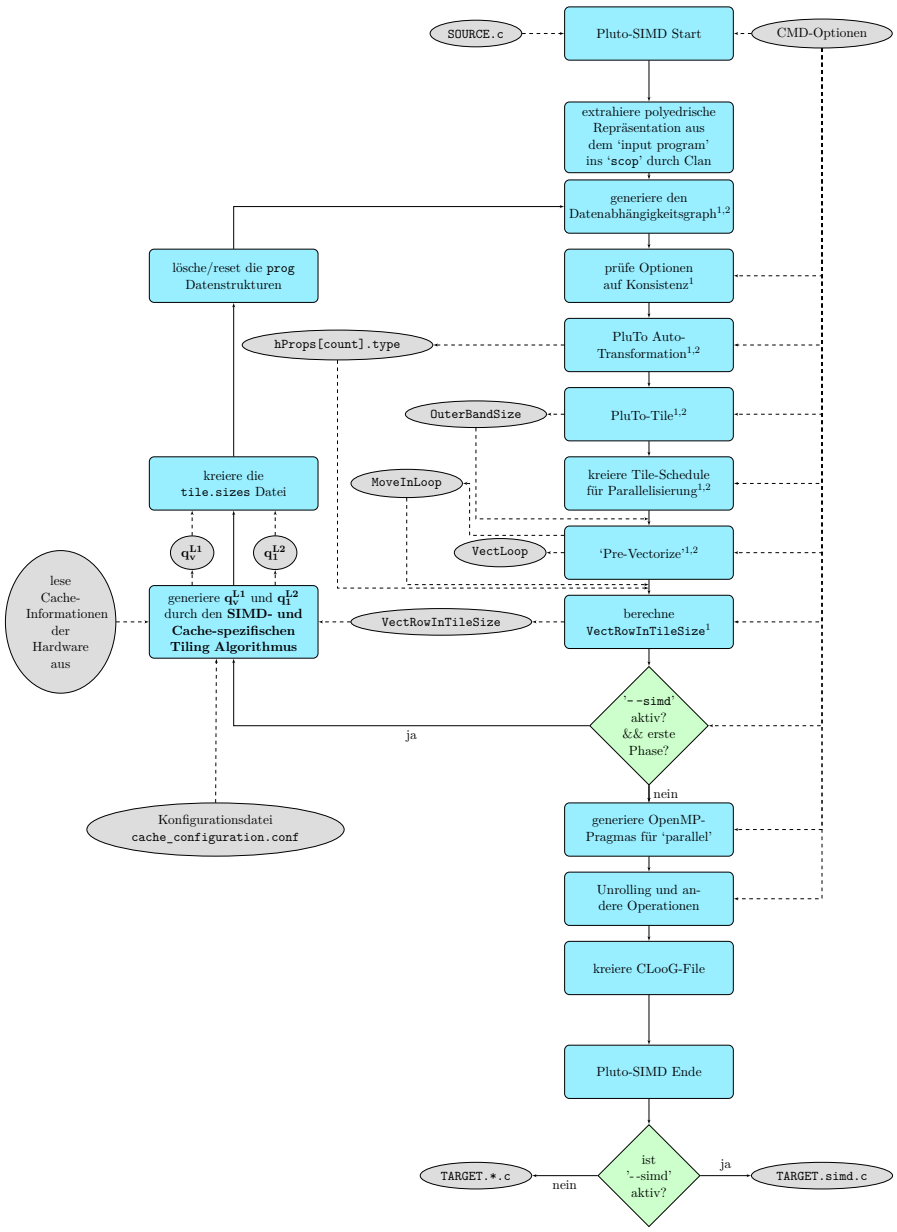


Abbildung 4.15: Flowchart zum Ablauf von PluTo-SIMD; Exponent 1 und 2 $\hat{=}$ Phase 1 und 2

5 Analyse der Ergebnisse

In diesem Kapitel werden Resultate der erarbeiteten und in PluTo-SIMD implementierten Konzepte dargestellt. Die Messungen sollen die theoretisch erläuterten Ideen praktisch untermauern sowie die resultierenden Performance-Auswirkungen zeigen.

Das Testsystem ist ein Knoten im Fraunhofer-Bull-Cluster (Sankt Augustin) mit folgenden (für diesen Kontext relevanten) Eckdaten:

$$\begin{aligned} CPU: &\quad Intel^{®}\ Xeon^{®}\ CPU\ X5650\ @\ 2.67GHz \\ L1-Cache: &\quad 32\ KByte\ (Daten) \\ L2-Cache: &\quad 256\ KByte \\ SSE-Version: &\quad 4.2 \end{aligned}$$

Als SIMD-Einheit wurde die SSE-Einheit (Streaming SIMD Extension) der CPU mit 128 Bit großen Registern verwendet und somit der Code für diese Parameter optimiert. Außerdem sind die dargestellten Ergebnisse in *single-precision* (float) gerechnet. Optimiert wurde entweder für den Level-1-Cache oder hierarchisch für Level-1- und Level-2-Cache. Alle Ergebnisse sind das arithmetische Mittel von 5 Werten, die aus 7 errechneten Werten durch Streichung des minimalen und maximalen Wertes ermittelt wurden.
Für das Cache-spezifische-Tiling gilt den obigen Werten zufolge:

$$\mathcal{C}_{L1} = 32\ [KByte], \quad \mathcal{C}_{L2} = 256\ [KByte], \quad \mathcal{D} = 4\ [Byte], \quad \mathcal{R} = 128\ [Bit]$$

Bemerkung 5.1. *Da es in dieser Arbeit in erster Linie um die Optimierung für eine effiziente* ***Vektorisierung*** *von Quellcodes geht, behandelt dieses Kapitel hauptsächlich die Optimierungen des SIMD- und Cache-spezifischen Tiling-Algorithmus von PluTo-SIMD (Abschnitt 5.2). Nichtsdestotrotz soll auch anhand eines entwickelten Aspekts durch Benchmarks auf die Erweiterungen in der Optimierungsphase (Abschnitt 5.1) eingegangen werden.*

5.1 Erweiterungen in der Optimierungsphase

Testcode:

```
1 #pragma scop
2   for (i=0; i<M; i++)
3     for (j=0; j<N; j++)
4       a[i][j] = 0.5*(a[M-i-1][j]+a[i][N-j-1]);
5 #pragma endscop
```

Abbildung 5.1: Code mit zwei nicht-gleichförmigen Abhängigkeiten

Dies ist ein 'künstlicher Code', bei dem die Auswirkungen eines Parametertauschs durch die gegebenen Abhängigkeiten deutlich werden. Die Ergebnisse in diesem Abschnitt sind mit **gcc -O0** kompiliert, um den Einfluss des Parametertauschs darzustellen. Durch den Parametertausch in der Optimierung wird im resultierenden Code die Reihenfolge der Schleifen vertauscht. Da der GCC auf höheren Optimierungsstufen ebenfalls Schleifen vertauscht und weitere Optimierungen durchführt, hier aber der reine Effekt durch die Schleifenvertauschung gezeigt werden soll, wurde diese Optimierungsoption gewählt.

Parametertausch

Der Code hat zwei nicht-gleichförmige Abhängigkeiten, deren Wiederverwendungsdistanz **M** und **N** sind. In der ursprünglichen Version durch PluTo wird in Iterationsschritt k bzgl. $\min_{\prec}(u_M, u_N, w, h_{k,1}, h_{k,2})$ optimiert, d. h. dass zunächst der Koeffizient u_M des Parameters M optimiert wird, danach erst der Koeffizient u_N des Parameters N. Dies erzeugt somit eine tendenziell gute Lösung (resultierend in einer **geringen Laufzeit**) wenn $M \gg N$, da u_M mit höchster Priorität minimiert wird. Diese Tendenz ist in Abbildung 5.2 zu sehen.
Durch den Tausch der beiden Koeffizienten mit der PluTo-SIMD Option `--swapopt variables` wird in Iterationsschritt k bzgl. $\min_{\prec}(u_N, u_M, w, h_{k,1}, h_{k,2})$ optimiert. Dies erzeugt also eine tendenziell gute Lösung wenn $N \gg M$, da nun u_N mit höchster Priorität minimiert wird. Diese Tendenz ist in Abbildung 5.3 zu sehen.
In den Benchmarks wurden M und N so gewählt, dass für jede Konfiguration $M*N = 2^{21}$ gilt, wodurch die Laufzeiten der verschiedenen Konfigurationen in der gleichen Größenordnung liegen, da immer 2^{21} Schleifeniterationen durchlaufen wurden. Es wurden alle Kombinationen $(M = 2^{20}, N = 2^1)$, $(M = 2^{19}, N = 2^2)$, ... ,$(M = 2^1, N = 2^{20})$ kompiliert und die resultierenden Laufzeiten ermittelt.

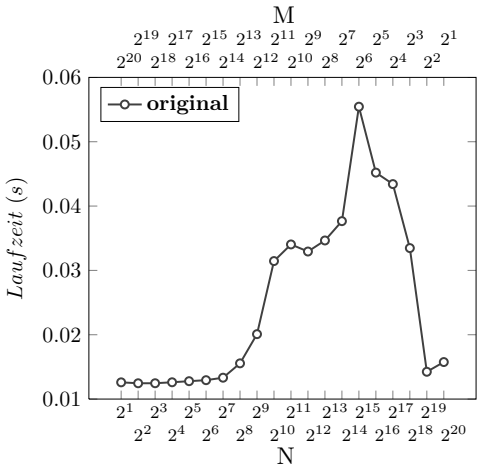


Abbildung 5.2: Ohne Tausch der Parameter

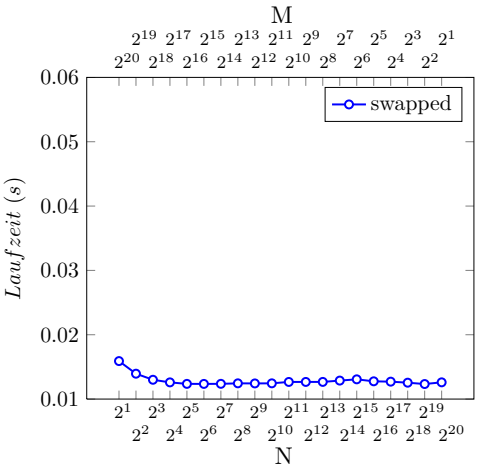


Abbildung 5.3: Mit Tausch der Parameter

Grundsätzlich ist die Laufzeit über die gesamte Parameter-Bandbreite durch den Tausch der Parameter deutlich ausgeglichener als im Original, da nach der 'gewöhnlichen' Optimierung durch PluTo beispielsweise die innere Schleife durch die äußere (erste) Array-Dimension läuft und nach der Optimierung mit Tausch der Parameter durch die innere (zweite) Array-Dimension. Hierdurch sind geringere Sprünge im Speicher nötig.

Die beiden 'extremsten' Konfigurationen (größte Differenz zwischen den beiden Parametern) der Messreihe sind in Abbildung 5.4 gegenübergestellt. Der Einfluss der Minimierungsreihenfolge wird hier sehr deutlich, da sich bei umgekehrter Reihenfolge der Parameter in der Optimierung die Laufzeitverhältnisse sehr exakt vertauschen. In diesen Extremkonfigurationen sind die Laufzeiten von der Größenordnung her auch gleich, da die Sprünge im Speicher hier durch den kleinen Programmparameter im gewöhnlich optimierten Code sehr klein sind und außerdem alle 'übersprungenen' Werte in eine Cache-Line passen, wodurch Elemente der nächsten Iteration automatisch mitgecached werden.

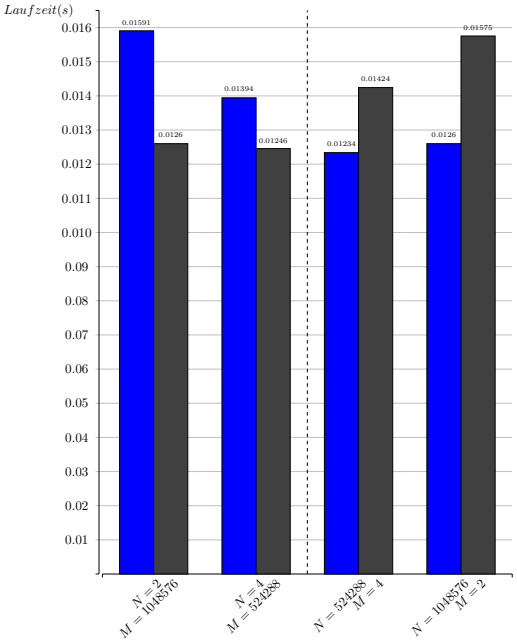


Abbildung 5.4: Laufzeit der *extremsten* Konfigurationen im Vergleich

5.2 SIMD- und Cache-spezifischer Tiling Algorithmus

Testcode 1: ***Matrix-Multiplikation***

```
1 #pragma scop
2  for(i=0; i<M; i++)
3    for(j=0; j<N; j++)
4      for(k=0; k<K; k++)
5        C[i][j] = C[i][j] + alpha*A[i][k]*B[k][j];
6 #pragma endscop
```

Abbildung 5.5: Original-Code der Matrix-Multiplikation

Testcode 2: ***Korrelationsmatrix***

```
1 /* Determine mean of column vectors of input data matrix */
2 for(j = 1; j <= m; j++)
3 {
4   mean[j] = 0.0;
5   for(i = 1; i <= N; i++)
6     mean[j] += data2[i][j];
7   mean[j] /= N;
8 }
9
10 /* Determine standard deviations of column vectors of data matrix. */
11 for(j = 1; j <= M; j++)
12 {
13   stddev[j] = 0.0;
14   for(i = 1; i <= N; i++)
15     stddev[j] += (data2[i][j] - mean[j]) * (data2[i][j] - mean[j]);
16   stddev[j] /= N;
17   stddev[j] = sqrt(stddev[j]);
18
19   /* The following is an inelegant but usual way to handle near-zero
20      std. dev. values, which below would cause a zero-divide. */
21   stddev[j] = stddev[j] <= eps ? 1.0 : stddev[j];
22 }
23
24 #pragma scop
25  /* Center and reduce the column vectors. */
26  for(i = 1; i <= N; i++)
27    for(j = 1; j <= M; j++)
28    {
29      data2[i][j] -= mean[j];
30      data2[i][j] /= sqrt(N) * stddev[j];
31    }
```

```
32 /* Calculate the M * M correlation matrix. */
33 for(j1 = 1; j1 <= M-1; j1++)
34 {
35   symmat[j1][j1] = 1.0;
36   for (j2 = j1+1; j2 <= M; j2++)
37   {
38     symmat[j1][j2] = 0.0;
39     for (i = 1; i <= N; i++)
40       symmat[j1][j2] += ( data2[i][j1] * data2[i][j2]);
41     symmat[j2][j1] = symmat[j1][j2];
42   }
43 }
44 #pragma endscop
45 symmat[M][M] = 1.0;
```

Abbildung 5.6: Original-Code der Berechnung der Korrelationsmatrix

Der Kern des Codes für die *Matrix-Multiplikation* ist vollständig 'im Scope', also als 'zu optimieren' gekennzeichnet. Aufgrund der relativ einfachen Struktur des Codes lassen sich Effekte gezielt an diesem zeigen.
Zugleich gehört dieser Algorithmus zu den wohl relevantesten im wissenschaftlichen Rechnen. Durch entsprechende Wahl der Parameter umfasst er die allgemeine Matrix-Matrix-Multiplikation, Matrix-Vektor-Multiplikation, das innere Produkt zweier Vektoren, die Multiplikation eines Vektors mit einem Skalar etc.
Die *Berechnung der Korrelationsmatrix* ist in statistischen Analysen häufig zu finden. Sie berechnet die Korrelationskoeffizienten einer mehrdimensionalen Zufallsvariable (Spalten der Matrix) in einem zweidimensionalen Schema `symmat`, wobei `symmat[i][j]` der Korrelation zwischen Spalte `i` und Spalte `j` entspricht. Dadurch ist die resultierende Matrix symmetrisch und hat Einsen auf der Diagonale.
Bei der *Berechnung der Korrelationsmatrix* wurde nur ein Teil des Codes als 'im Scope' (also 'zu optimieren') gekennzeichnet, nämlich derjenige, welcher tatsächlich die Korrelationsmatrix berechnet. Die Berechnung von Standardabweichungen und Mittelwerten der Spaltenvektoren der Matrix werden zwar mitgemessen, allerdings nicht optimiert. Dementsprechend ist nach Amdahls Gesetz (Abschnitt 2.2) auch nur ein gewisser Anteil des theoretischen Speedups erreichbar. Der Code ist aus den Beispielen des PluTo-Frameworks entnommen und soll gezielt die Beschleunigung dieses Scopes zeigen.
Dieser Code ist deutlich komplexer als jener der Matrix-Multiplikation und umfasst mehrere Statements inklusive Abhängigkeiten zwischen diesen. Dadurch können auch Effekte des Statement-spezifischen Tilings gezeigt werden.

Die Ergebnisse in diesem Abschnitt sind mit **gcc -O3** kompiliert, u. A. da die Autovektorisierung des GCC ab -O3 (wenn möglich) aktiviert wird. Außerdem soll in diesem Abschnitt gezeigt werden, welche Performance-Gewinne bei **aktuellster Compiler-Version** und auf **höchster Optimierungsstufe** des GCC durch PluTo-SIMD erzielt werden.

Bemerkung 5.2 (**zur Wahl der Benchmarks in diesem Kapitel:**). *Zu Beginn des Abschnitts 5.2.1 soll durch die 'feinskaligen Benchmarks' der Einfluss der SSE-Registerbreite auf die umgesetzte Vektorisierung des GCC gezeigt werden sowie die hieraus resultierende allgemein ermöglichte Vektorisierung durch ein Register-konformes Tiling. Aufbauend hierauf werden diese Ergebnisse noch für eine große Parameterbandbreite in den 'grobskaligen Benchmarks' bestätigt, sodass deren Gültigkeit auf eben diesem gesamten Parameterspektrum gezeigt wird.*
Die manuellen Quantitäten zeigen, dass im hier gemessen Bereich (bis $q_v^{L1} = 256$) für PluTo-SIMD gilt: je größer die Quantität, desto kürzer die Laufzeit. Dass diese Quantität aber einen Optimalwert erreicht und dieser dem theoretisch hergeleiteten aus Kapitel 4 entspricht, wird in Abschnitt 5.2.2 anhand fester Parameterkonstellationen (M,N,K) durch Variation der Quantitäten q_v^{L1} und q_1^{L2} in einem breiten Spektrum gezeigt. Die Legitimation zur Wahl einer festen Parameterkonstellation liefern die Ergebnisse aus Abschnitt 5.2.1, da die Verhältnisse der Laufzeiten über das gesamte Parameterspektrum äußerst konstant sind.
All diese Ergebnisse sollen anhand der beiden vorgestellten Codes aufgezeigt werden.
Im Anschluss werden noch kurz einige andere Aspekte beleuchtet, wie eine Messung der Cache-Misses, die Laufzeit der Kompilierung durch die Frameworks, Codes bei denen PluTo und/oder PluTo-SIMD keine gute Performance erzielen sowie ein Vergleich mit dem ICC als Back-End der Kompilierung.
Zum Schluss wird ein Code optimiert, der aufgrund kleiner Programmparameter ein SIMD-spezifisches Tiling mit manuellen Quantitäten motiviert und wegen seiner großen Bedeutung in aktuellen Algorithmen zur Videobearbeitung einen Anwendungsfall außerhalb wissenschaftlicher und statistischer Codes darstellt.

5.2.1 SIMD-spezifisches Tiling mit manuellen Quantitäten

Feinskaliger Benchmark
Diese Benchmarks sollen die Grundidee von PluTo-SIMD, also das Tiling lediglich der vektorisierten Schleife und ggf. einer weiteren für das Level-2-Tiling, und zwar in Register-konforme Größen, beleuchten. Hierzu wurde PluTo-SIMD mit verschiedenen manuellen

Tile-Quantitäten (`_TCS=false`) für ein Level (Abbildungen 5.7 und 5.9) sowie für zwei Level (Abbildungen 5.8 und 5.10) ausgeführt.
Die Codes wurden für verschiedene Matrixgrößen (jeweils $N \times N$) im Bereich $N = 995, 996, ..., 1005$ ausgeführt und deren Laufzeit jeweils mit dem ursprünglichen Code (source) sowie mit der durch PluTo optimierten Version verglichen.

Bemerkung 5.3. *In den Diagrammen mit PluTo-SIMD* ***Level-1****-Tiling werden die Werte mit einem* ***einfachen*** *PluTo-Tiling verglichen, in denen mit* ***Level-1- und Level-2****-Tiling durch PluTo-SIMD entsprechend mit einem* ***zweifachen*** *PluTo-Tiling.*

Die Tile-Quantitäten aus der Konfiguration für PluTo-SIMD sind in den Legenden eingetragen (PluTo generiert mit seiner default-Konfiguration).

Matrix-Multiplikation

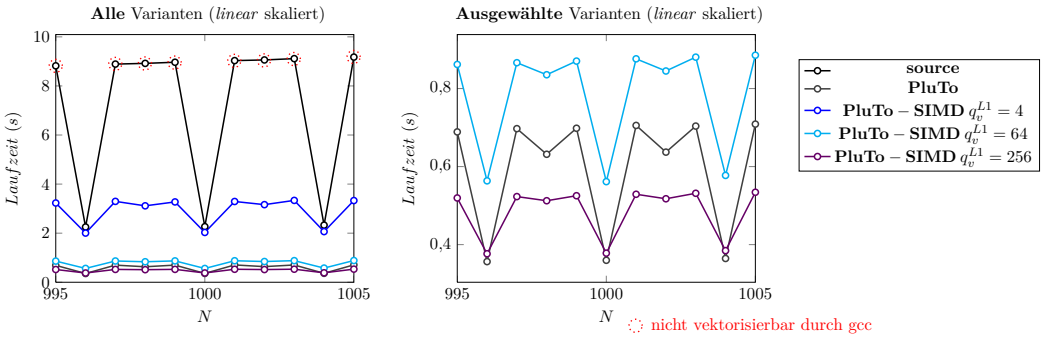


Abbildung 5.7: Matrix-Multiplikation **Level-1**-Tiling

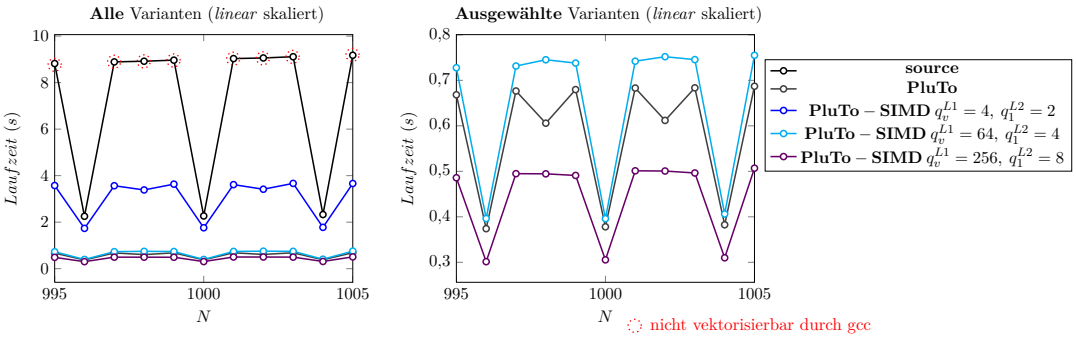


Abbildung 5.8: Matrix-Multiplikation **Level-1- und Level-2**-Tiling

Im originalen Code funktioniert die Autovektorisierung des GCC nur dann, wenn die Matrixbreite (und somit die Anzahl der Iterationen einer Schleife) ein Vielfaches von 4 ist (verifiziert durch die GCC `--ftree-vectorizer-verbose=2` Ausgabe). Dies korrespondiert zu ElPeRe $= \frac{\mathcal{R}}{8*\mathcal{D}} = \frac{128}{8*4} = 4$, also der Anzahl an Elementen, die in ein Register

der SSE-Einheit passen.
Durch die (von PluTo durch die 32-er Blockung aller Schleifen und von PluTo-SIMD durch Quantitäten in 4er-Vielfache) erzeugten Blöcke funktioniert in **allen** optimierten Codes bei **jeder** Matrixgröße die Autovektorisierung des GCC.
Die optimierten Codes sind darüber hinaus bei Register-konformen Parametern N (996, 1000, 1004) am schnellsten, da in der geblockten Schleife keine 'Reste' vorhanden sind, welche *seriell* abgearbeitet werden müssten.
Das zusätzliche Level-2-Tiling verbessert die Performance i. d. R., die Level-2-Quantität sollte allerdings nicht zu klein sein, da dies die Zugriffsstruktur sonst zu sehr 'zerstückelt'. Bereits durch die manuellen Quantitäten $q_v^{L1} = 256$ und $q_1^{L2} = 8$ sind die Ergebnisse von PluTo-SIMD schneller als die in allen Dimensionen geblockten Codes von PluTo. In den dargestellten Beispielen sind die Codes für größere q_v^{L1} und q_1^{L2} immer besser als für kleinere. Dies gilt allerdings bei größeren Quantitäten nicht mehr. Die optimale Größe der Quantitäten (durch den Cache-spezifischen Tiling Algorithmus) wird in Abschnitt 5.2.2 untersucht.

Korrelationsmatrix

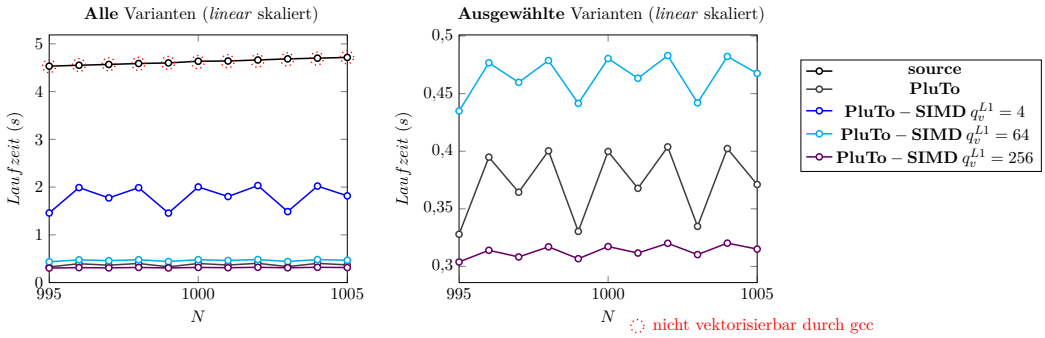


Abbildung 5.9: Korrelationsmatrix **Level-1**-Tiling

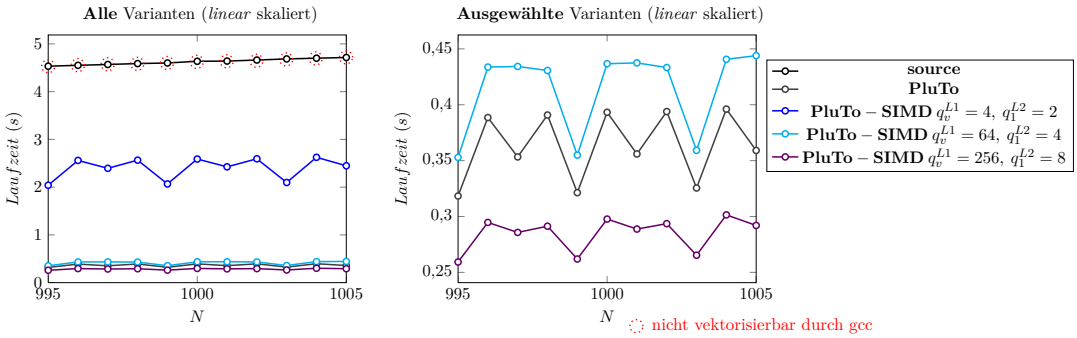


Abbildung 5.10: Korrelationsmatrix **Level-1- und Level-2**-Tiling

Bei der Berechnung der Korrelationsmatrix funktioniert (aufgrund der komplexeren Zugriffsstruktur und Verschachtelung) im originalen Code (*source*) die Autovektorisierung des GCC **in keinem Fall**. Durch das Tiling in Register-konforme Blöcke und das Verlegen der vektorisierbaren Schleife ins Innere einer jeden Verschachtelung, findet die Autovektorisierung bei den optimierten Codes **immer** statt. Ansonsten sind die Ergebnisse zum Einfluss von N auf die Laufzeit dieselben wie bei der Matrix-Multiplikation. In den optimierten Codes ist der jeweils beste Lauf für N (im Vergleich zur Matrix-Multiplikation) um eins nach links verschoben, also nicht bei $N = 996, 1000, 1005$, sondern bei $N = 995, 999, 1004$. Dies liegt daran, dass die Schleifen im Code (vgl. Abbildung 5.6) bei 1, statt wie bei der Matrix-Multiplikation bei 0, beginnen und die Schleifengrenzen bis *einschließlich* N bzw. M laufen. Entsprechend sind die Arrays in den Größe $N + 1$ $(M + 1)$ statt N (M) deklariert.

Fazit der feinskaligen Läufe für beide Codes:

- der GCC vektorisiert den originalen Code der *Matrix-Multiplikation* **nur**, wenn die Matrixbreite (bzw. Anzahl der Iterationen pro Schleife) Register-konform ist
- der GCC vektorisiert den originalen Code der *Berechnung der Korrelationsmatrix* **nie** (auch nicht bei Register-konformer Matrixbreite)
- Tiling durch PluTo-SIMD ermöglicht eine **flächendeckende** Vektorisierung durch den GCC, unabhängig von der Matrixgröße
- die Tile-Quantitäten sollten **nicht zu klein** gewählt werden
- Level-2-Tiling bringt **zusätzliche** Performance-Vorteile
- PluTo-SIMD erzeugt hier bereits mit manuellen Quantitäten $q_v^{L1} = 256$ und $q_1^{L2} = 8$ **schnelleren** Code als PluTo

Grobskaliger Benchmark

Nachdem im letzten Abschnitt gezeigt wurde, dass Tiling mit Register-konformen Quantitäten die Autovektorisierung flächendeckend ermöglicht, soll die Performance von PluTo und PluTo-SIMD nun durch gröbere Skalierungsläufe über N bei quadratischen Matrizen mit verschiedenen manuellen Konfigurationen verglichen werden.

Matrix-Multiplikation

Da im Fall, dass N bereits Register-konform ist, die Autovektorisierung bei der Matrix-Multiplikation auch im Originalcode funktioniert, werden hier für die Matrix-Multiplikation zweier quadratischer Matrizen der Größe $N \times N$ zwei verschiedene Skalierungs-

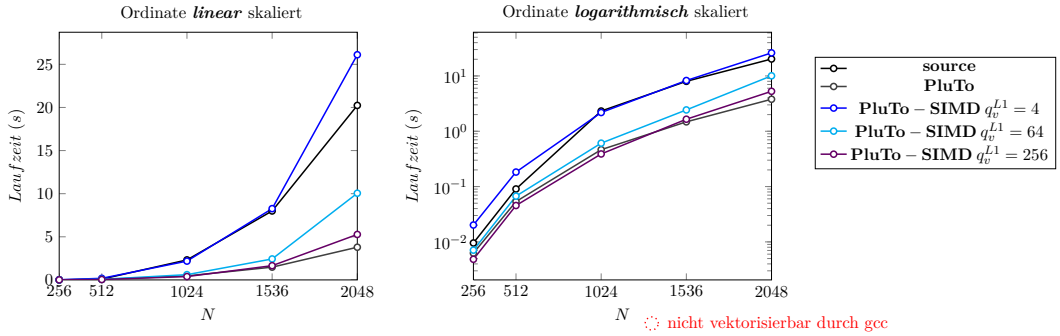


Abbildung 5.11: Matrix-Multiplikation **Level-1**-Tiling **256 etc.**

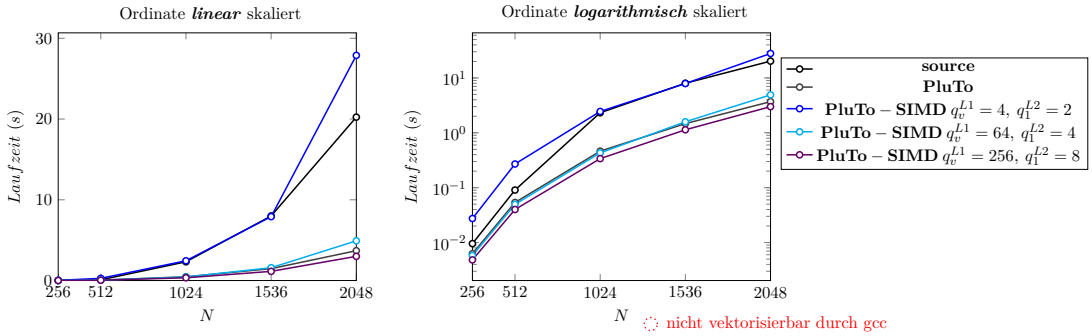


Abbildung 5.12: Matrix-Multiplikation **Level-1- und Level-2**-Tiling **256 etc.**

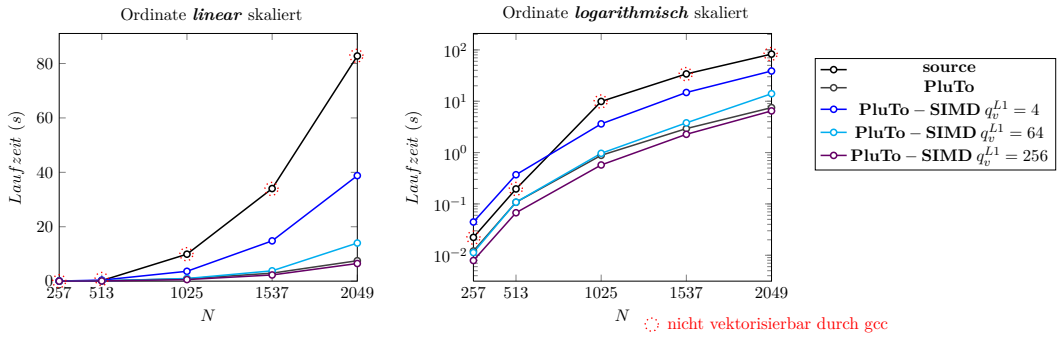


Abbildung 5.13: Matrix-Multiplikation **Level-1**-Tiling **257 etc.**

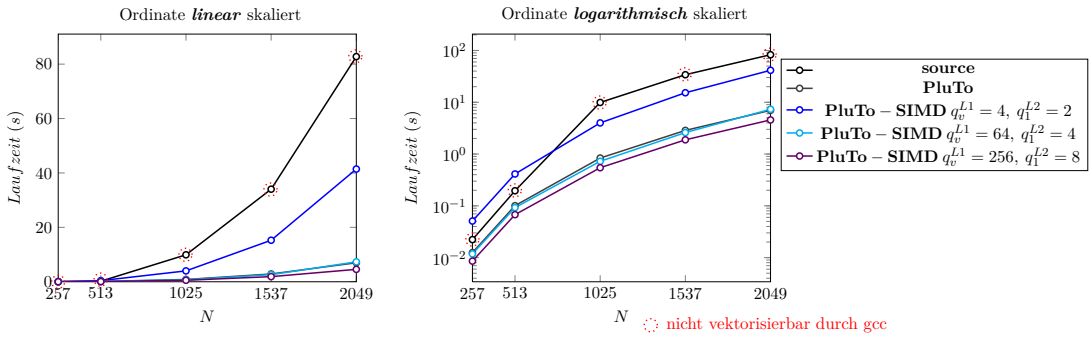


Abbildung 5.14: Matrix-Multiplikation **Level-1- und Level-2**-Tiling **257 etc.**

reihen gemessen und dargestellt, zum einen mit $N = 256, 512, ...$ (Abbildungen 5.11 und 5.12, hier funktioniert bereits im originalen Code die Autovektorisierung des GCC) und zum anderen für $N = 257, 513, ...$ (Abbildungen 5.13 und 5.14, hier funktioniert die Autovektorisierung im originalen Code nicht, genau wie es bei $N = 258, 514, ...$ und $N = 259, 515, ...$ der Fall wäre).

Die Performance-Verhältnisse der Versionen untereinander bleiben im Wesentlichen über das gesamte Spektrum ähnlich. Der Hauptunterschied zwischen den 256er- und den 257er-Versionen ist, dass im 256er-Fall die originale Version (source) durch die Vektorisierung schneller ist als im 257er-Fall.
Bei den 256er-Reihen wird der durch PluTo-SIMD mit den Quantitäten $q_v^{L1} = 4$ bzw. $q_1^{L2} = 2$ optimierte Code sogar langsamer als der originale Code, da in diesem Fall *beide* Versionen durch den GCC vektorisiert werden und die kleinen Quantitäten die Zugriffsstruktur zu sehr 'zerstückeln'. Ab $q_v^{L1} = 64$ bzw. $q_1^{L2} = 4$ sind die PluTo-SIMD Codes allerdings auch hier schneller als die originalen und laufen für $q_v^{L1} = 256$ und $q_1^{L2} = 8$ (im Level-2-Tiling) ebenfalls in allen Fällen schneller als PluTo.

Korrelationsmatrix

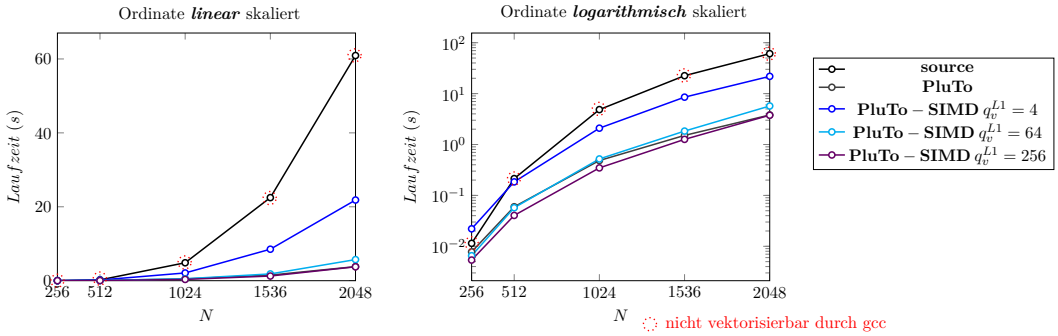


Abbildung 5.15: Korrelationsmatrix **Level-1**-Tiling **256 etc.**

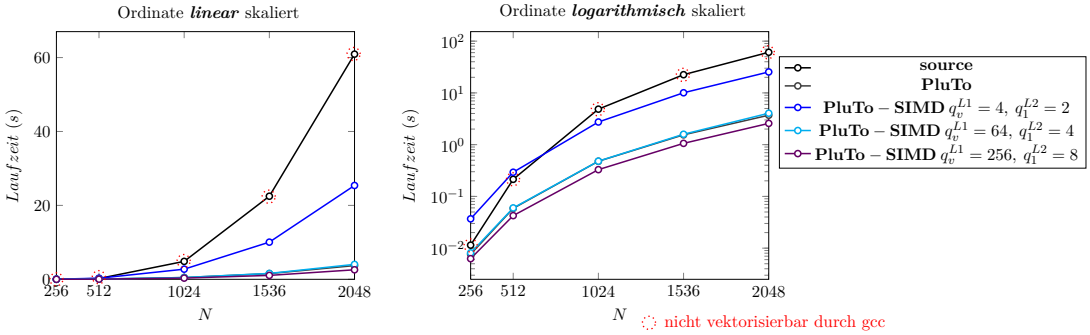


Abbildung 5.16: Korrelationsmatrix **Level-1- und Level-2**-Tiling **256 etc.**

Da der GCC bei der Berechnung der Korrelationsmatrix in **keinem** Fall den originalen Code vektorisieren kann, genügt es in diesem Fall die Skalierungsreihe $N = 256, 512, ...$ zu betrachten (bei $N = 257, 513, ...$ gibt es keinen nennenswerten Unterschied).

Auch hier bleiben die Performance-Verhältnisse untereinander über das gesamte Spektrum ähnlich und das Tiling in zu kleine Quantitäten **kann** sich sogar negativ auswirken (hier nur für $N = 256$).

Fazit der grobskaligen Läufe für beide Codes:

- die Beobachtungen der feinskaligen Läufe gelten **auch im breiten Spektrum** für N
- die **kubische** Laufzeit aller Versionen bleibt sichtbar erhalten (da sich die Iterationsanzahl nicht verändert)
- es ändern sich aber die Zugriffszeiten auf die entsprechenden Speicherhierarchien, wodurch sich die Laufzeiten **reduzieren** lassen
- die von **PluTo** generierten Codes erreichen **Speedups von bis zu 11.89** bei der Matrix-Multiplikation und von **bis zu 17.92** bei der Berechnung der Korrelationsmatrix
- die von **PluTo-SIMD** generierten Codes erreichen **Speedups von bis zu 18.19** bei der Matrix-Multiplikation und von **bis zu 23.67** bei der Berechnung der Korrelationsmatrix

5.2.2 SIMD-spezifisches Tiling mit Cache-spezifischen Quantitäten

Im folgenden Abschnitt soll der Einfluss der Quantitäten q_v^{L1} und q_1^{L2} auf die Laufzeit analysiert werden. Hierzu werden die Testcodes mit festen Parametern M, N, K betrachtet und die Quantitäten q_v^{L1} und q_1^{L2} variiert. Da die Quantitäten in einem großen Bereich untersucht werden und die (*kubische*) Laufzeit der beiden Algorithmen moderat bleiben soll, wurden hier für die Matrix-Multiplikation die Parameter $M = 189$, $N = 139233$ und $K = 189$ gewählt und für die Berechnung der Korrelationsmatrix $M = 11923$ und $N = 89$. Der Parameter der vektorisierten Schleife wurde für diese Benchmarks also sehr groß gewählt, und zwar weil diese Schleife in PluTo-SIMD durch q_v^{L1} geblockt und der Parameter q_v^{L1} hier im Bereich von 1 bis 36864 variiert wird. In entsprechend großen quadratischen Fällen ($M = N = K$ wie in Abschnitt 5.2.1) würden die Laufzeiten sehr groß werden. Die Tendenz für quadratische Fälle wird außerdem in den Benchmarks des vorherigen Abschnittes deutlich.

Der Einfluss der Quantitäten q_v^{L1} und q_1^{L2} soll die Idee des Cache-spezifischen Tiling Algorithmus unterstreichen und die hergeleitete Ermittlung der optimalen Werte bestätigen. Das theoretisch errechnete Optimum der Tile-Quantitäten für $\rho = 100\,\%$ aus PluTo-SIMD ist deshalb in den entsprechenden Grafiken durch gestrichelte vertikale Linien eingezeichnet.

Matrix-Multiplikation Level-1-Tiling

Hier wurde die Laufzeit der Matrix-Multiplikation **bei festen Parametern** gemessen, sprich in allen Fällen wurde **dasselbe** Matrixprodukt berechnet. Die Variation von q_v^{L1} (in Abbildung 5.17 dargestellt zwischen 816 und 36864, bis zu diesem Punkt fällt die Laufzeit monoton) beeinflusst lediglich die Zugriffsstruktur und Berechnungsreihenfolge, sodass die resultierenden Performance-Veränderungen durch die optimierte Zugriffsstruktur und ein anderes Cache-Verhalten zu begründen sind. Es wurde zur Ermittlung der zu messenden q_v^{L1}-Größen ρ von $0\,\%$ bis $1000\,\%$ in Prozentschritten erhöht. Der errechnete *theoretische* Optimalwert beträgt $q_v^{L1} = 4096$ (vgl. Beispiel 4.14). Die dargestellte Kurve in Abbildung 5.17 ist eine Glättung der ermittelten Laufzeiten, wobei für jeden Punkt der Mittelwert aus seiner Laufzeit und der seiner 5 linken und 5 rechten Nachbarn berechnet wurde. Die maximale Streuung der exakten Messzeiten ist als blauer Bereich eingezeichnet.

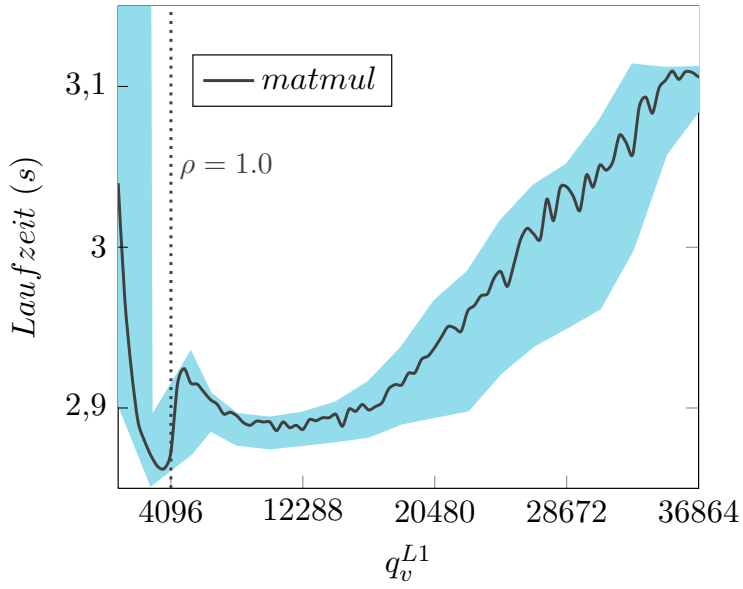


Abbildung 5.17: Matrix-Multiplikation Level-1-Tiling durch PluTo-SIMD

In diesem Bereich wird für PluTo-SIMD mit $2.85\,s$ das *globale* Optimum angenommen,

und zwar bei $q_v^{L1} = 3644$ bzw. $\rho = 0.89$ (89 %). Die Laufzeit des originalen Codes beträgt 15.36 s und die des von PluTo generierten einfachen Tilings 3.83 s.
Somit resultiert der von

- **PluTo** generierte Code in einem **Speedup von 4.01** im default-Setup, der von
- **PluTo-SIMD** mit Cache-spezifischem Tiling erreicht einen **Speedup von 5.39**

(jeweils im Vergleich zum originalen Code).

Bemerkung 5.4. *Die Speedups sind in beiden Fällen deutlich kleiner als im quadratischen Fall, das Verhältnis der Speedups zueinander ist aber gleich. Im quadratischen Fall, wie in den 'grobskaligen Benchmarks mit manuellen Quantitäten', kann der positive Einfluss des Tilings auf die Speicherzugriffe häufiger ausgenutzt werden.*

Modifikationen des Statements aus matmul: ***matmul1, matmul2 und matmul3***

```
1 C[i][j]=C[i][j]+A[i][k]*B[k][j];
2 C[i][j]=C[i][j]+A[i][k]*B[k][j]+D[i][k]*E[k][j];
3 C[i][j]=C[i][j]+A[i][k]*B[k][j]+D[i][k]*E[k][j]+F[i][k]*G[k][j];
```

Abbildung 5.18: Modifikationen des Statements der Matrix-Multiplikation

Um die Ermittlung der Quantitäten weiter zu untersuchen, werden nun die Ergebnisse der 'Modifikationen des Statements' der Matrix-Multiplikation dargestellt. Hierbei handelt es sich um die Multiplikation zweier Matrizen $A \cdot B$ (matmul1) bzw. die Summe zweier solcher Multiplikationen $A \cdot B + D \cdot E$ (matmul2) und die Summe dreier Multiplikationen $A \cdot B + D \cdot E + F \cdot G$ (matmul3). Bei der Ermittlung der Quantität q_v^{L1} verändert sich in den Fällen lediglich der Wert ElPeIt $= ||\mathcal{T}(\mathcal{Z}(i_v))||_{\text{diff}}^{Bl}$ (die Anzahl der Zugriffe der vektorisierten Schleife, in den Codeauszügen zu Beginn des Kapitels **j**), wodurch sich für $\rho = 100\,\%$ die errechneten *theoretischen* Optimalwerte

$$\text{matmul1}: \quad q_v^{L1} = \left\lfloor \frac{\rho * \frac{32*1024}{4}}{2 * \frac{128}{8*4}} \right\rfloor * \frac{128}{8*4} \overset{\rho=1}{=} 4096 \tag{5.1}$$

$$\text{matmul2}: \quad q_v^{L1} = \left\lfloor \frac{\rho * \frac{32*1024}{4}}{3 * \frac{128}{8*4}} \right\rfloor * \frac{128}{8*4} \overset{\rho=1}{=} 2728 \tag{5.2}$$

$$\text{matmul3}: \quad q_v^{L1} = \left\lfloor \frac{\rho * \frac{32*1024}{4}}{4 * \frac{128}{8*4}} \right\rfloor * \frac{128}{8*4} \overset{\rho=1}{=} 2048 \tag{5.3}$$

ergeben. Für die Tests wurde ρ zwischen 50 % und 150 % in 10 %-Schritten erhöht.

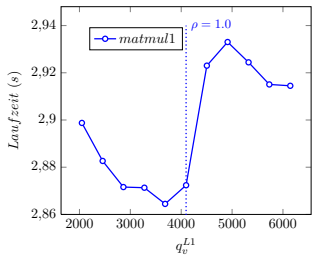


Abbildung 5.19: Matrix-Multiplikation Level-1-Tiling $C = A \cdot B$

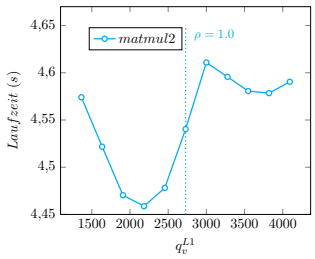


Abbildung 5.20: Matrix-Multiplikation Level-1-Tiling $C = A \cdot B + D \cdot E$

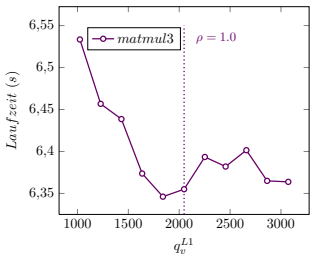


Abbildung 5.21: Matrix-Multiplikation Level-1-Tiling $C = A \cdot B + D \cdot E + F \cdot G$

Die Ergebnisse aus den Abbildungen 5.19, 5.20 und 5.21 bestätigen den Zusammenhang zwischen der Anzahl der Zugriffe und optimaler Tile-Quantität, da bei größerer Menge von Zugriffen (und damit zu ladenden Daten) der Optimalwert der Laufzeit bei einem entsprechend kleinerem q_v^{L1} liegt. Es müssen nämlich entsprechend mehr Daten pro Iteration geladen werden und somit kann nur eine kleinere Anzahl von Iterationen auf einmal in den Level-1-Cache geladen werden.

Um die Verschiebung des Optimalwertes für q_v^{L1} bei den 3 Varianten der Matrix-Multiplikation in einem Diagramm darzustellen, ist die Ordinate (Laufzeit) für jeden Code entsprechend seinem Laufzeitminimum und Laufzeitmaximum skaliert (der entsprechende Bereich der Laufzeit in Sekunden ist in der Legende angegeben).

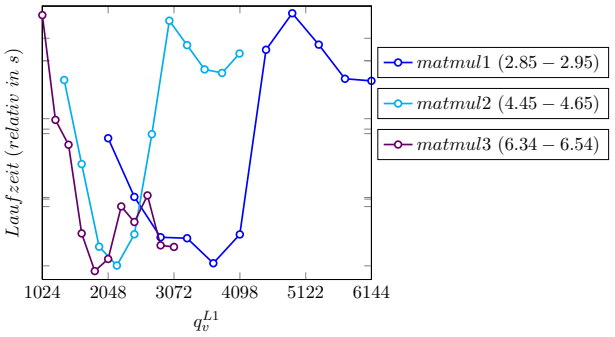


Abbildung 5.22: Matrix-Multiplikationen matmul1, matmul2 und matmul3 Level-1-Tiling

Durch die Ergebnisse bestätigt sich die theoretische Überlegung aus Kapitel 4, dass für ρ aus dem Bereich von $80 - 100\,\%$ der optimale Cachefluss erreicht wird.

Matrix-Multiplikation Level-2-Tiling

In Abschnitt 4.4.1 wurde argumentiert, warum für das Level-2-Tiling in PluTo-SIMD die *erste/äußerste* Schleife zusätzlich geblockt wird. Deshalb wird nun mit $q_v^{L1} = 3684$ ($\rho = 90\,\%$), die Quantität q_x^{L2} für das Level-2-Tiling zwischen 0 und 10 variiert. Im Fall $x = 1$ wurde die *erste* Schleife für das Level-2-Tiling verwendet, bei $x = 2$ die *zweite* und bei $x = 3$ die *dritte*.

Hier zeigt sich, dass nur im Falle $x = 1$ eine (nennenswerte) Verbesserung der Performance stattfindet. In diesem Falle waren die Laufzeiten der 7 Testläufe auch für jeden Wert für q_1^{L2} äußerst konstant, in beiden anderen Fällen eher 'schwankend' (siehe A.2). Dies ist durch die lange Verwendung eines Wertes der Level-2-Steuerschleife zu begründen.
Der theoretisch ermittelte Optimalwert $q_1^{L2} = 8$ ist tatsächlich auch in der Praxis optimal.

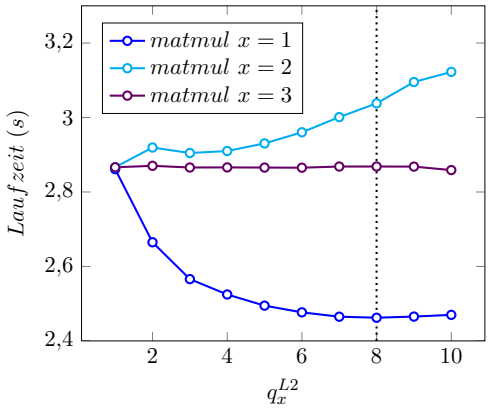


Abbildung 5.23: Einfluss der Wahl der Level-2-Tiling Schleife

Da die Menge der Iterationen in einem resultierenden Level-2-Block $q_v^{L1} * q_1^{L2}$ beträgt und in q_v^{L1} bereits der Faktor ρ (für die nicht vollständige Auslastung des Cache) enthalten ist, muss für q_1^{L2} keine weitere 'Korrektur' bzw. 'Anpassung' vorgenommen werden.

Fazit des SIMD- und Cache-spezifischen Tilings der Matrix-Multiplikation durch PluTo-SIMD für M = 189, N = 139233 und K = 189:

- PluTo-SIMD ermittelt (für $\rho = 90\,\%$) den für die Laufzeit der Standard-Matrix-Multiplikation **optimalen** Wert q_v^{L1} **für das Level-1-Tiling** sowie den **optimalen** Wert q_1^{L2} **für das Level-2-Tiling**
- die Wahl der äußersten Schleife für das Level-2-Tiling liefert die besten Ergebnisse und eine **zusätzliche Performance-Steigerung**
- auch bei den Matrix-Multiplikation ist ein ρ-Wert von 90 % (bzw. in einem Fall 80 %) **optimal**

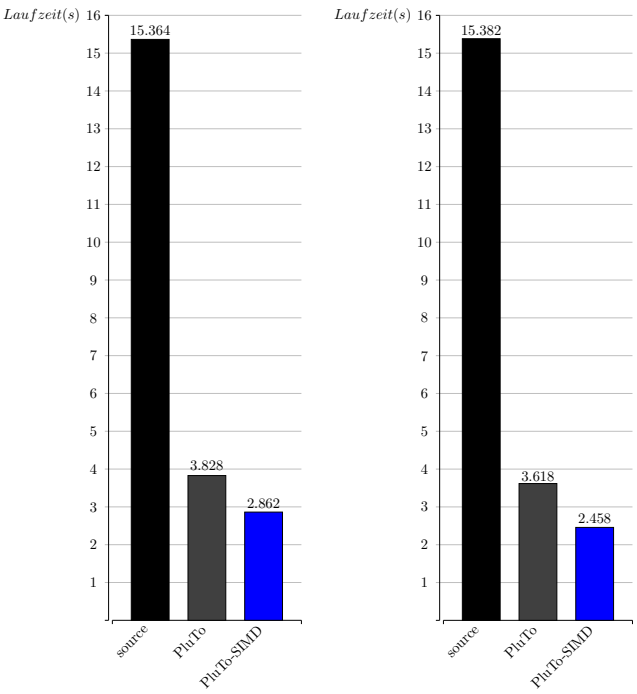


Abbildung 5.24: Level-1- bzw. Level-1- und Level-2-Tiling Resultate der Matrix-Multiplikation

Korrelationsmatrix

Nun werden die Ergebnisse des Cache-spezifischen Tilings für die Berechnung der Korrelationsmatrix gezeigt. Durch die vielen Statements und die Verschachtelung des Codes können hier außerdem Einflüsse der Optionen MAX, MIN und TPS auf die Laufzeit der Berechnung der Korrelationsmatrix ermittelt werden.

Zunächst wird ρ (genau wie bei der Matrix-Multiplikation) von 0 % bis 1000 % in Prozentschritten erhöht. Durch das hier angewandte Statement-spezifische Tiling (TPS) in PluTo-SIMD gibt es für die verschiedenen Statements z. T. verschiedene Tile-Quantitäten. Für $\rho = 1.0$ gilt: $\mathbf{q_v^{L1}}(S1, S2) = 2456$, $\mathbf{q_v^{L1}}(S3) = 7368$, $\mathbf{q_v^{L1}}(S4, S5) = 3684$ und $\mathbf{q_v^{L1}}(S3) = 1$ (vgl. Beispiel 4.16). Für andere Größen von ρ sind die Quantitäten entsprechend der hergeleiteten Formel aus (4.10) im wesentlichen linear dazu skaliert. Dies gilt im Übrigen nicht für $\mathbf{q_v^{L1}}(S3) = 1$, da $S3$ keinen vektorisierten Zugriff enthält und somit nicht geblockt wird. Die Darstellung der Laufzeit in Abbildung 5.25 wurde in gleicher Weise erstellt wie Abbildung 5.17.

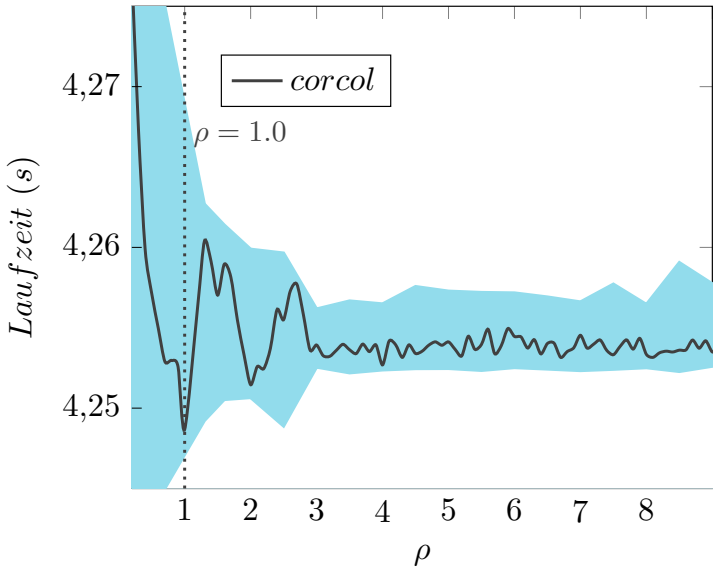


Abbildung 5.25: Berechnung der Korrelationsmatrix Level-1-Tiling durch PluTo-SIMD (TPS)

In diesem Bereich wird für PluTo-SIMD mit 4.247 s das *globale* Optimum angenommen, und zwar bei $\rho = 1.0$ (100 %). Die Laufzeit des originalen Codes beträgt 8.179 s und die des von PluTo generierten einfachen Tilings 4.345 s.
Neben der Optimalkonstellation für $\rho = 1.0$ ist die Performance bei $\rho = 2.0$ und $\rho = 3.0$ *lokal* optimal. Dies ist dadurch zu erklären, dass die generierten Blöcke in diesen Fällen genau in zwei bzw. drei Zügen in den Level-1-Cache passen. Der Optimalwert liegt hier

exakt bei $\rho = 1.0$ (und nicht knapp darunter), was unter anderem dadurch zu begründen ist, dass im gesamten *Scope* des Codes *keine* skalaren Variablen o. Ä. vorkommen.

Somit resultiert der von

- **PluTo** generierte Code in einem **Speedup von 1.88** im default-Setup, der von
- **PluTo-SIMD** mit Cache-spezifischem Tiling erreicht einen **Speedup von 1.93**

(jeweils im Vergleich zum originalen Code).

Bemerkung 5.5. *Auch hier sind die absoluten Speedups in beiden Fällen deutlich kleiner als in den quadratischen Konstellationen des vorherigen Abschnitts mit $M = N$ (Begründung in Bemerkung 5.4). Die Auswertungen an dieser Stelle sollen in erster Linie der Ermittlung des Optimalwertes für die Tile-Quantitäten dienen (Bemerkung 5.2).*

In Abbildung 5.26 sind die resultierende Laufzeiten für die 3 Optionen dargestellt. Bei MIN wird die optimale Laufzeit für $\rho < 1.0$ angenommen (hier bei $\rho = 0.5$). Dies liegt daran, dass in diesem Fall der kleinste Wert für die Zugriffe der Statements (bzw. Statementblöcke) für das Tiling der gesamten Verschachtelung gewählt wird und die Schleifen somit tendenziell in zu große Tiles geblockt werden. Ein klein gewählter Wert für ρ kann dies hier wieder ausgleichen.

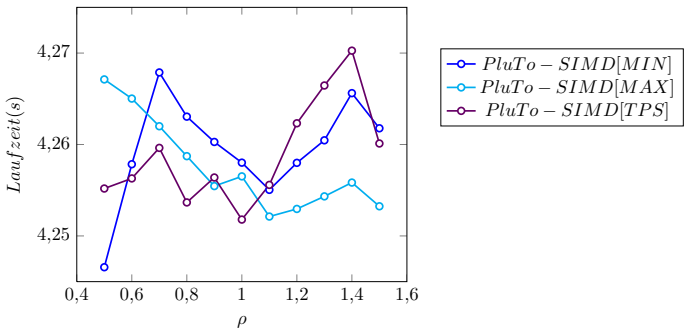


Abbildung 5.26: Einfluss der Optionen MAX, MIN, TPS bei der Berechnung der Korrelationsmatrix

Bei MAX wird das Optimum für $\rho > 1.0$ angenommen (hier bei $\rho = 1.1$). Die Begründung läuft analog zum letzten Abschnitt.
Bei TPS wird die beste Laufzeit für $\rho = 1.0$ erreicht.
Dies bestätigt, dass durch die gezielten Quantitäten für jeden Block (bei TPS) von Statements eine Optimalkonstellation gezielt generiert werden kann. Bei beiden Alternativen (MAX und MIN) kann der Optimalwert aufgrund der Verallgemeinerung einer

Konstellation auf alle Blöcke entsprechend verschoben auftreten. Dies gilt insbesondere bei vielfach verschachtelten Schleifen mit vielen Blöcken, deren vektorisierte Zugriffsanzahlen stark variieren. Gäbe es z. B. einen Statementblock mit nur einem vektorisierten Zugriff und einen anderen mit vielleicht 8 (oder gar 16 etc.) vektorisierten Zugriffen, so würde sich eine Verallgemeinerung durch MIN oder MAX noch deutlich gravierender auswirken.

Fazit des SIMD- und Cache-spezifischen Tilings der Berechnung der Korrelationsmatrix durch PluTo-SIMD für M = 11923 und N = 89 bei ρ= 0.9:
In Abbildung 5.27 sind resultierende Laufzeiten zum Vergleich für $\rho = 0.9$ angegeben. **Alle** Codes können für andere Werte von ρ (aus oben genannten Gründen) geringfügig bessere Performance erreichen, allerdings ist ein durch die theoretische Herleitung und die Ergebnisse der Matrix-Multiplikation motivierter Wert von 90 % für ρ eine sinnvolle Vergleichsgrundlage.

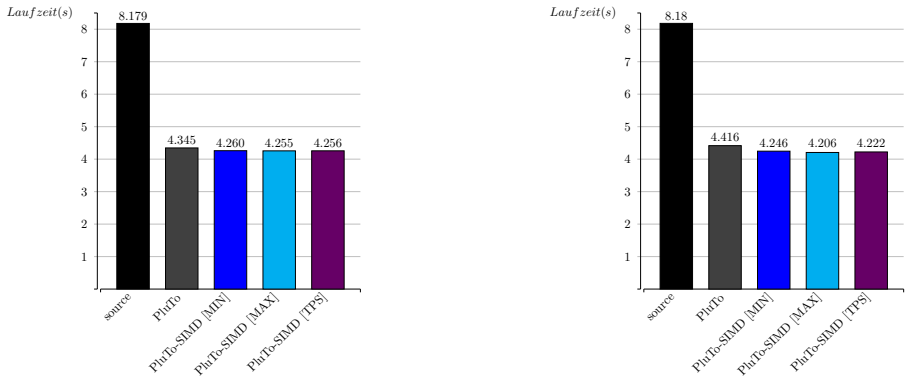


Abbildung 5.27: Level-1- bzw. Level-1- und Level-2-Tiling Berechnung der Korrelationsmatrix

Auch hier sind die Speedups (wie bei der Matrix-Multiplikation) durch PluTo (einfaches Tiling: 1.88*, zweifaches Tiling:* 1.85*) und PluTo-SIMD (Level-1-Tiling:* 1.92*, Level-2-Tiling:* 1.94*), aufgrund der bereits im vergangenen Abschnitt erläuterten Gründen, deutlich geringer als im quadratischen Fall aus dem Abschnitt '**Grobskaliger Benchmark**'. Um die Einflüsse der verschiedenen Optionen in PluTo-SIMD im Cache-spezifischen Tiling aufzuzeigen, eignet sich aber diese Parameterkonstellation aufgrund der nötigen Größe der vektorisierten Schleife bzw. dessen Parameter M.*

- PluTo-SIMD ermittelt (für $\rho = 100\,\%$ bei TPS) den für die Laufzeit der Berechnung der Korrelationsmatrix **optimalen** Wert q_v^{L1} **für das Level-1-Tiling**
- MIN und MAX können hier zwar auch entsprechend gute Ergebnisse wie TPS erzielen, allerdings ist der entsprechende Parameter ρ nicht **einfach bestimmbar**

- das Level-1-Tiling durch PluTo-SIMD erzeugt **in allen Konstellationen MIN, MAX oder TPS** schnelleren Code als das Tiling durch PluTo
- ein zweifaches Tiling durch PluTo **erhöht** hier sogar die Laufzeit durch die Verkomplizierung des Codes und die resultierende Steigerung der Speichersprünge
- das gezielte Level-2-Tiling in PluTo-SIMD bringt **weitere** Performance-Vorteile

5.2.3 Cache-Miss-Rate Benchmark der Matrix-Multiplikation

Durch Messungen mit *PAPI* [BDG+00] und *Cachegrind* [NS03] (vgl. Kapitel 2) konnte die Verbesserung des Cache-Verhaltens durch die Anwendung von PluTo-SIMD (hier: Level-1- und Level-2-Tiling) bestätigt werden. Die Level-1-Cache-Miss-Rate wurde um mehr als 50 % reduziert, die Level-2-Cache-Miss-Rate sogar um mehr als 90 %. PluTo kann in beiden Bereichen ähnlich gute oder sogar leicht bessere Ergebnisse erreichen. Die bereits vielfach angesprochene 'Verkomplizierung' des Codes in PluTo durch das Blocken aller Schleifen erklärt den trotzdem vorhandenen Laufzeitvorteil von PluTo-SIMD und lässt sich an der Anzahl der TLB(*Translation Lookaside Buffer*)-Misses erkennen.
Der ***T**ranslation **L**ookaside **B**uffer* rechnet virtuelle Speicheradressen in physikalische um und bildet somit die Schnittstelle zwischen den abstrakten Datenzugriffen in einem Quellcode und den konkreten Zugriffen auf den Hardwarespeicher. Da in PluTo-SIMD die vektorisierte Schleife in relativ große Tiles geblockt wird, werden Datenzugriffe, die nur von den anderen Schleifenindizes (der nicht vektorisierten Schleifen) abhängen, häufig wiederverwendet. Dadurch müssen die entsprechenden physikalischen Adressen deutlich weniger oft neu berechnet werden. Eine solche Berechnung ist relativ 'teuer' und somit ein wesentlicher Laufzeitfaktor in vielen Codes mit verschachtelten Schleifen [GvdG02]. Durch das optimierte 'lange Ausnutzen' berechneter Speicheradressen in PluTo-SIMD ist der Wert der TLB-Misses (gemessen durch *PAPI*) im Originalcode von **1.342.624.625** durch PluTo-SIMD auf **62.548** verringert worden, PluTo verzeichnet noch **3.074.570** TLB-Misses, was sich in der resultierenden Laufzeit stark niederschlägt.

Bemerkung 5.6. *Die absoluten Messwerte der Counter können und sollen aufgrund verschiedener Einflüsse und Ungenauigkeiten hier nur als Anhaltspunkt und zur Beurteilung starker Unterschiede verwendet werden.*

5.2.4 Laufzeit der Kompilierung durch PluTo und PluTo-SIMD

Die Laufzeiten der Kompilierungen (hier der Berechnung der Korrelationsmatrix) durch PluTo ohne Tiling bzw. mit Tiling oder durch PluTo-SIMD mit Cache-spezifischem Tiling sind, und bleiben auch im zweiphasigen Modell von PluTo-SIMD, relativ gering. Sie sind in Anbetracht der verringerten (und der absoluten) Laufzeit des resultierenden Codes, insbesondere bei mehrfacher Ausführung, vernachlässigbar.

clean: $\mathbf{0.112}\,s$	--tile:	$\mathbf{0.197}\,s$	--tile --simd: $\mathbf{0.219}\,s$
	--tile --l2tile:	$\mathbf{0.238}\,s$	--tile --l2tile --simd: $\mathbf{0.242}\,s$

Zum Vergleich dauert eine Kompilierung des Codes in Maschinencode durch den GCC, wie sie auch im Anschluss an die Optimierungen durch PluTo und PluTo-SIMD durchgeführt wurde, im Standardmodus $\mathbf{0.1}\,s$ und mit $-O3$ $\mathbf{0.25}\,s$.

5.2.5 Schwächen der Methoden mit dem GCC

Um die noch vorhandenen Schwächen der vorgestellten Methode darzustellen, werden im Folgenden die Ergebnisse von drei Codes vorgestellt, bei denen die Performance von PluTo und/oder PluTo-SIMD nicht optimal ist.

Jacobi 1-D

Testcode:

```
1 #pragma scop
2 for(t=1; t<T; t++)
3  for(i=1; i<N-1; i++)
4    a[t][i]=0.333*(a[t-1][i]
         +a[t-1][i-1]+a[t-1][i+1]);
5 #pragma endscop
```

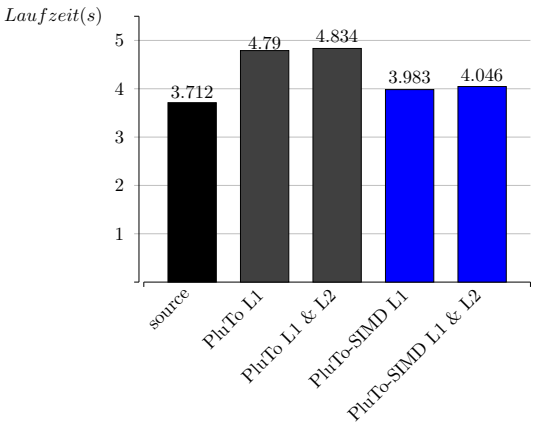


Abbildung 5.28: 1-D Jacobi

Der Jacobi-Code ist ein Beispiel, in welchem der GCC weder durch die Transformation in PluTo noch in PluTo-SIMD profitiert. Dies ist im Wesentlichen dadurch zu begründen, dass der GCC schon in der Lage ist den originalen Code (source) zu vektorisieren, da die vektorisierbare Schleife sich bereits im Innersten der Verschachtelung befindet und für die reine Vektorisierung keine Transformation nötig ist. Allerdings transformiert PluTo (und somit auch PluTo-SIMD) das Statement durch ein Parallelepiped-Tiling, damit eine Blockung des Codes möglich ist. Die resultierende Zugriffsstruktur auf das Array wird dadurch aber komplizierter, da jeder originale Zugriff auf `j` im transformierten Code ein Zugriff durch `t6-t3` (vgl. Abbildung 4.11) wird und somit für die Indizes deutlich mehr gerechnet werden muss.
Durch das Tiling ist der Code parallelisierbar (Multicore), was hier allerdings nicht genutzt wird, da es nicht im Fokus dieser Arbeit steht. Hierdurch könnte die Performance deutlich gesteigert werden, sodass der resultierende Code entsprechend schneller wäre als der originale.

Fazit: Der GCC profitiert hier weder von PluTo noch von PluTo-SIMD, da die vorliegende Codestruktur bereits nativ durch den GCC vektorisiert werden kann und das

(hier *nicht* für Parallelität genutzte) Tiling die Codestruktur verkompliziert. PluTo-SIMD erzeugt durch seine gezielte Vektorisierung und Cache-Anpassung allerdings noch schnelleren (oder '*weniger langsameren*') Code als PluTo.

Weitere BLAS(Basic Linear Algebra Subprograms)-Routinen (GEMVER)

Testcode:

```
1  #pragma scop
2  for(i=0; i<N; i++)
3    for(j=0; j<N; j++)
4      B[i][j]=A[i][j]+u1[i]*v1[j]
                +u2[i]*v2[j];
5
6  for(i=0; i<N; i++)
7    for(j=0; j<N; j++)
8      x[i]=x[i]+beta*B[j][i]*y[j];
9
10 for(i=0; i<N; i++)
11   x[i]=x[i]+z[i];
12
13 for(i=0; i<N; i++)
14   for(j=0; j<N; j++)
15     w[i]=w[i]+alpha*B[i][j]*x[j];
16 #pragma endscop
```

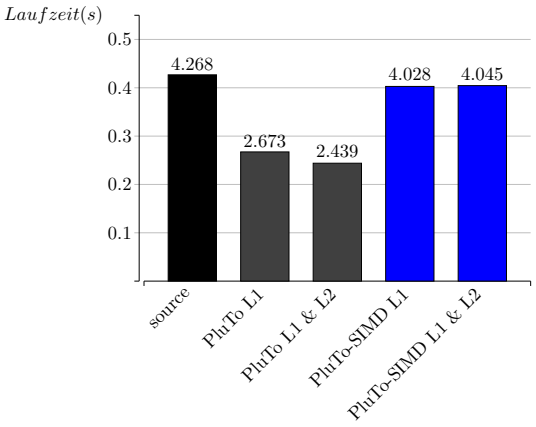


Abbildung 5.29: Weitere BLAS-Routinen (GEMVER)

Der **GE**(neral)**M**(atrix)**VE**(ctor)**R**(outines)-Code beinhaltet Standardroutinen der Linearen Algebra, wie die Vektor-Addition und diverse Matrix-Vektor-Kombinationen. Auch hier können weder PluTo noch PluTo-SIMD zu einer vermehrten Vektorisierung durch den GCC führen (in allen Fällen werden zwei der Schleifen vektorisiert). Da in dem Code aus PluTo sowie PluTo-SIMD u. A. der Zugriff `B[j][i]` bei vektorisierter Schleife `j` vorhanden ist, und somit durch die vektorisierte Schleife auf die äußere Dimension zugegriffen wird, schneidet die Blockung aller Schleifen durch PluTo aufgrund einer höheren Wiederverwendung hier besser ab als die durch PluTo-SIMD.

Fazit: Der GCC kann hier *in keinem Fall* in der Vektorisierung unterstützt werden. Trotzdem verbessern sowohl PluTo als auch PluTo-SIMD die Performance des Codes. Der durch PluTo transformierte Code ist u. A. aufgrund einer höheren Wiederverwendung der Daten im Zugriff auf `B[j][i]` schneller als der durch PluTo-SIMD transformierte.

Dreiecks-Matrix-Multiplikation (STRMM)

Testcode:

```
1 for(i=1; i<N; i++)
2  for(j=0; j<N; j++)
3   for(k=0; k<i; k++)
4    b[j][k] += a[i][k] * b[j][i];
```

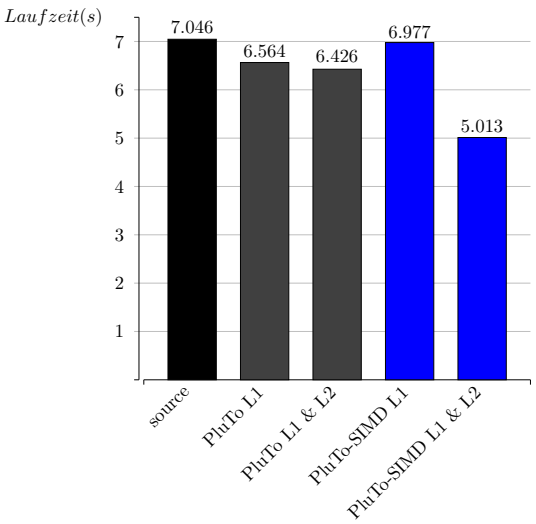


Abbildung 5.30: Matrix-Multiplikation mit unterer Dreiecks-Matrix STRMM

Der **S**(ingle Precision)**TR**(riangular)**M**(atrix)**M**(atrix Multiplication)-Code ist ebenfalls eine BLAS-Funktion und berechnet das Produkt einer beliebigen Matrix und einer Dreiecksmatrix.
Auch hier kann der GCC nicht in der Vektorisierung unterstützt werden. Sowohl der originale Code als auch die transformierten Versionen werden *alle* 'gleich' vektorisiert. Durch PluTo-SIMD mit Level-1-Tiling allein können die durch PluTo vorhandenen Performance-Gewinne nicht erreicht werden. Generiert PluTo-SIMD allerdings sowohl ein Level-1- als auch Level-2-Tiling, so ist der resultierende Code schneller als das original und als *beide* PluTo-Varianten.

Fazit: Bei lediglich Level-1-Tiling ist der Code durch PluTo-SIMD zwar minimal schneller als der Originalcode, erreicht jedoch nicht die Performance von PluTo. Ein Level-1- und Level-2-Tiling durch PluTo-SIMD schlägt hingegen beide PluTo-Varianten.

5.2.6 Vergleich GCC vs. ICC

Um auch die mögliche Unterstützung des ICC durch PluTo-SIMD zu zeigen und die resultierende Performance mit der des GCC zu vergleichen, wurde die Matrix-Multiplikation mit PluTo einfach (PluTo L1) sowie zweifach (PluTo L1 & L2) geblockt. In PluTo-SIMD wurde das SIMD- und Cache-spezifische Tiling vorgenommen. Der ICC wurde, genau wie

der GCC, auf Optimierungsstufe **-O3** verwendet, sodass die automatische Vektorisierung durch SSE sowie weitere Optimierungen aktiviert sind.

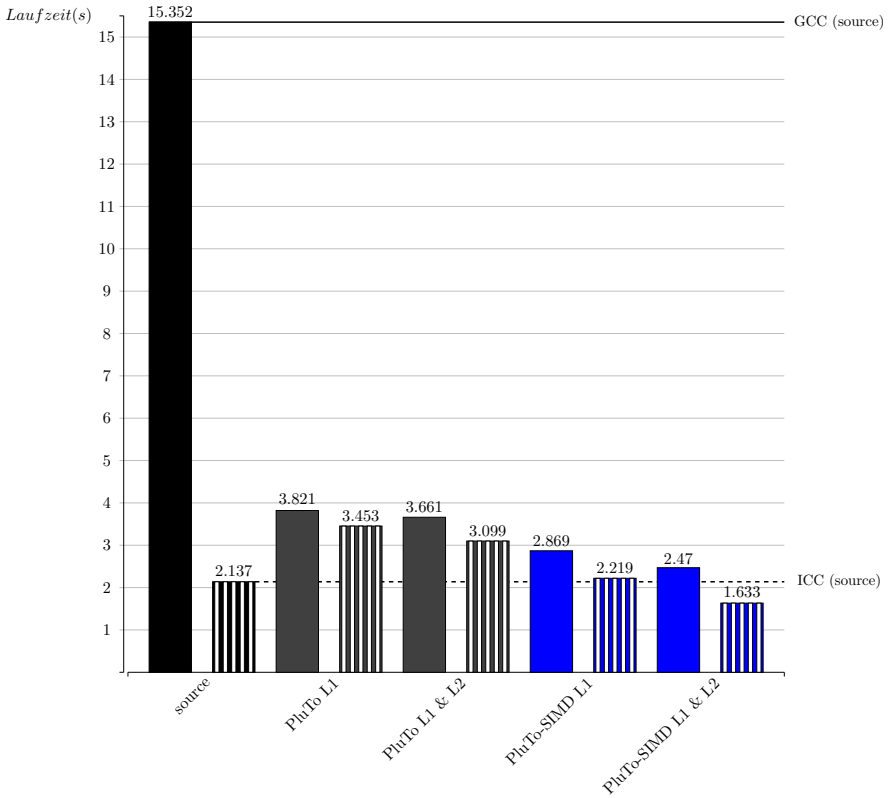


Abbildung 5.31: Matrix-Multiplikation kompiliert mit GCC (ausgefüllt) und ICC (schraffiert)

Die Ergebnisse aus Abbildung 5.31 zeigen, dass die grundsätzliche Performance des durch den ICC kompilierten Codes deutlich besser ist als die GCC-Versionen (Originalcode GCC: 15.352 s, ICC: 2.137 s). Dies liegt unter anderem daran, dass der ICC auch den originalen Code in jedem Fall (unabhängig von M, N, K) vektorisieren kann und weitere Optimierungen vornimmt.
Die GCC-Variante kann sowohl durch PluTo als auch durch PluTo-SIMD deutlich verbessert werden, sodass die durch PluTo-SIMD optimierte und mit dem GCC kompilierte Version (die beste GCC-Variante) fast an die Laufzeit des ICC-kompilierten Originalcodes heranreicht (GCC mit PluTo-SIMD: 2.47 s, ICC original: 2.137 s). Allerdings kann **auch** die Performance eines ICC-kompilierten Codes durch PluTo-SIMD mit einem SIMD- und Cache-spezifischen Level-1- und Level-2-Tiling verbessert werden, sodass die Laufzeit in der besten Konstellation (PluTo-SIMD + ICC) auf 1.633 s verringert werden kann. Das Tiling durch PluTo verschlechtert die Ergebnisse des ICC, womöglich da es dessen Optimierungen z. T. verhindert.

5.2.7 Anwendung auf MPEG4 oder Motion-Tracking

Zuletzt soll ein Codeabschnitt durch PluTo-SIMD für den GCC optimiert werden, wie er in vielen Algorithmen der Videoverarbeitung und Videokompression vorkommt. Der Algorithmus berechnet den 'Unterschied' zwischen Blöcken in zwei aufeinanderfolgen Bildern und wird beispielsweise verwendet, um Objekte in Videos zu verfolgen [RDVB10; SCE03; SB08] oder (in diversen Varianten) um Videos mit dem MPEG4-Standard zu komprimieren [ATKA03]. Alternative Strategien zur Berechnung des Unterschiedes (neben der hier analysierten 'Summe der Differenzen') sind z. B. die 'Summe der absoluten Differenzen' oder die 'Summe der quadrierten Differenzen'.

Testcode:

```
1 #pragma scop
2 for(y_s=0; y_s<N; y_s++) {
3  for(x_s=0; x_s<N; x_s++) {
4   sod[y_s][x_s] = 0.0;
5   for(y_p=0; y_p<M; y_p++) {
6    for(x_p=0; x_p<M; x_p++) {
7     sod[y_s][x_s]=sod[y_s][x_s]+curr[y_p][x_p]-prev[y_s+y_p][x_s+x_p];
8    }
9   }
10  }
11 }
12
13 for(y_s=0; y_s<N; y_s++) {
14  for(x_s=0; x_s<N; x_s++) {
15   result[y_s][x_s]=sod[y_s][x_s];
16  }
17 }
18 #pragma endscop
```

Abbildung 5.32: Berechnung der 'Summe der Differenzen'

Dieser Code wird im Folgenden zum einen für $M, N \in \{308, 408, 508, 608, 708\}$ (nativ Register-konform) und zum anderen mit $M, N \in \{309, 409, 509, 609, 709\}$ analysiert. Der GCC kann den originalen Code, auch falls M und N Register-konform sind, in *keinem Fall* vektorisieren. Durch die Transformation mittels PluTo und PluTo-SIMD wird der Code in *jedem Fall* durch den GCC vektorisierbar. Aufgrund der vierfachen Verschachtelung der Schleifen hat der Code bereits bei den hier gewählten Parametern eine relativ hohe Laufzeit.

Bemerkung 5.7. *Liegen (wie hier) kleine Programmparameter vor, so ist es zur Verwiklichung eines Tilings (hier u. A. zur Generierung Register-konformer Schleifen) ratsam ein SIMD-spezifisches Tiling mit manuellen Tile-Quantitäten vorzunehmen, da das Cache- und SIMD-spezifische Tiling hier Cache-angepasste Parameter in der Größenordnung* $> 10^3$ *berechnet. Für* $M, N < 10^3$ *würde das Tiling also keine Blockung vornehmen, da der gesamte Indexraum in* ***ein*** *Tile passt (vgl. Bemerkung 4.2). Für größere Programmparameter wäre das SIMD- und Cache-spezifische Tiling wiederum sinnvoll.*

Für die hier gewählten Problemgrößen wurde PluTo-SIMD mit $q_v^{L1} = 256$ und $q_1^{L2} = 8$ konfiguriert. Um den sogar negativen Einfluss des Tilings aller Schleifen in PluTo zu beleuchten, wurden hier darüber hinaus Läufe mit PluTo-SIMD und $q_v^{L1} = 32$ gemessen. Das resultierende Tiling für $q_v^{L1} = 32$ (und $q_{i \neq v}^{L1} = 1$, vgl. (4.1)) *entspräche* einem PluTo-Tiling *nur einer* Dimension (da PluTo **alle** Schleifen mit $q_i^{L1} = 32$ blockt).

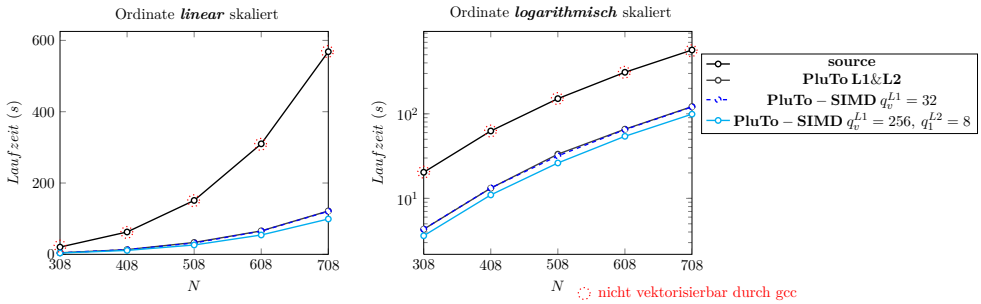


Abbildung 5.33: 'Summe der Differenzen' **308er**

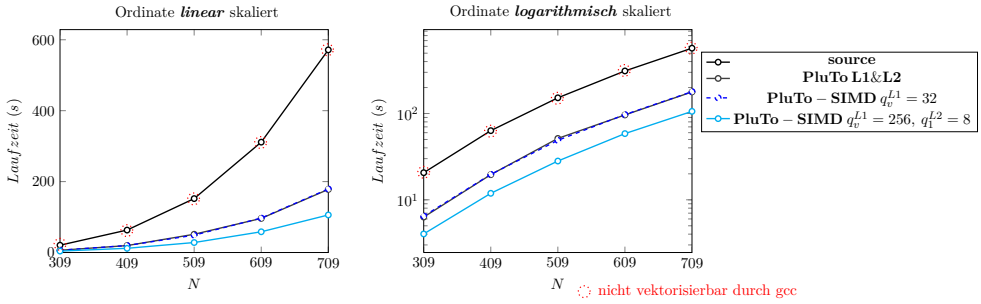


Abbildung 5.34: 'Summe der Differenzen' **309er**

Die 308*er*-Läufe haben in allen Fällen kürzere Laufzeiten als die 309*er*-Läufe, da der 'Rest' einer jeden Schleife nicht mehr Register-konform ist und somit seriell bearbeitet wird. Durch das Tiling *aller* Schleifen in PluTo sind die Auswirkungen bereits deutlich gravierender als in den anderen betrachteten Codes. Hierdurch bleiben vierdimensionale 'Rest-Tiles' übrig, die seriell abgearbeitet werden. Außerdem sind die Tiles durch die

Blockung *aller* Schleifen deutlich zu groß für den Level-1-Cache.
Der Code durch PluTo ist durch die Vektorisierung der vollständigen Tiles und ein verbessertes Zugriffsmuster für den Level-2-Cache (für diesen sind die Tiles noch passend) auch deutlich schneller als der originale Code, allerdings bringt ein SIMD-spezifisches Tiling durch PluTo-SIMD in kleine 32*er*-Blöcke (das entspricht quasi *einer* Dimension eines PluTo-Tilings) z. T. bereits bessere Performance als das volldimensionale PluTo-Tiling. Ein **PluTo-SIMD** Tiling mit manuellen Tile-Quantitäten $q_v^{L1} = 256$ und $q_1^{L2} = 8$ erzielt für $M = N = 709$ einen **Speedup von 5.4** (im Vergleich zum originalen Code), wobei **PluTo** lediglich einen **Speedup von 3.2** erreicht. Das Tiling durch PluTo-SIMD in kleine 32*er*-Blöcke erzielt einen **Speedup von 3.19**. Für $M = N = 708$ ist das SIMD-spezifische Tiling durch PluTo-SIMD in kleine 32*er*-Blöcke sogar schneller als PluTo-Tiling. Dies zeigt, dass das Tiling der drei weiteren Dimensionen in PluTo sich hier **negativ** auf die Laufzeit auswirkt.

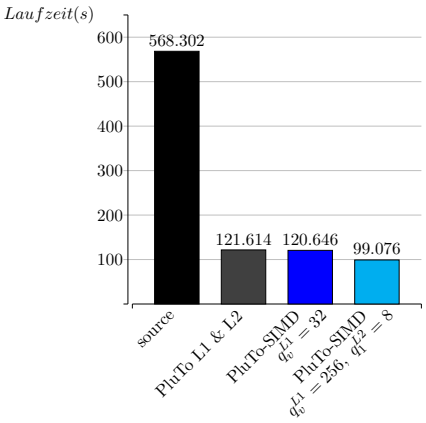


Abbildung 5.35: 'Summe der Differenzen' **N = 708**

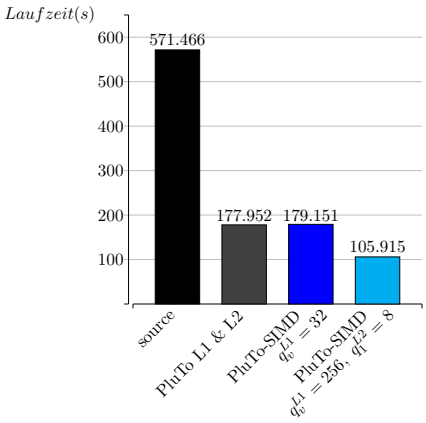


Abbildung 5.36: 'Summe der Differenzen' **N = 709**

Besonders bei vielfach verschachtelten Schleifen (hier: vierfach) kann das statische Tiling aller Schleifen von PluTo (im Gegensatz zum gezielten Tiling weniger Schleifen) also sogar negative Einflüsse auf die Performance haben. Die trotz alledem gute Performance von PluTo ist durch die ermöglichte Vektorisierung zu begründen.
Je mehr Schleifen verschachtelt sind, desto schlechter passen die Tiles von PluTo in den Cache und desto schlechter performt dessen Tiling. PluTo-SIMD passt den Code durch das gezielte Tiling nur einer (bzw. zweier) Schleifen unabhängig von der Verschachtelungstiefe an den Cache an. Außerdem passt PluTo-SIMD das Tiling nicht nur an verschiedene Codes (mit verschiedenen Zugriffsstrukturen und Verschachtelungstiefen) an, sondern auch an sich in Zukunft ändernde Cache-Größen und Vektoreinheiten.

6 Resümee & Ausblick

Resümee: In dieser Arbeit wurden Methoden zur Transformation von Quellcodes entwickelt und implementiert, die als Vorstufe einer End-Kompilierung durch Compiler wie den GCC oder ICC eingesetzt werden können. Der Fokus lag hierbei auf der Entwicklung von Codeoptimierungen für eine Vektorisierung von Schleifen in C-Codes durch ein semi-automatisches Framework.
Diese Methoden sind im auf PluTo [Bona] aufbauenden Framework PluTo-SIMD implementiert worden. Ziel der Transformationen durch PluTo-SIMD ist eine bessere Nutzung von SIMD-Architekturen, z. B. den SSE-Einheiten aktueller x86-Prozessoren, die Prinzipien lassen sich aber auch auf andere Hardware-Architekturen mit vektorbasierten Befehlssätzen übertragen.
Das durch PluTo-SIMD generierte Register-konforme und Cache-angepasste Tiling von Quellcodes ermöglicht eine Hardware-bezogene Optimierung gezielt für die Vektorisierung. Durch das Register-konforme (SIMD-spezifische) Tiling wird eine von den Programmparametern unabhängige Vektorisierung durch den GCC ermöglicht. Die automatische Cache-Anpassung eines solchen Tilings wird durch die in PluTo-SIMD implementierte Zugriffsanalyse möglich und ermittelt in den getesteten Codes optimale Tile-Quantitäten für das SIMD- und Cache-spezifische Tiling. PluTo-SIMD erreicht durch die Transformation (im Vergleich zum Originalcode) in den getesteten Fällen mit dem GCC erhebliche Speedups bis zu einem Faktor von 20. Auch für die Kompilierung mit dem ICC ermöglicht die Cache-Anpassung in PluTo-SIMD, trotz der bereits sehr ausgereiften Optimierungsfunktionalität des ICC, deutliche Performance-Gewinne.
Die semi-automatisierte Arbeitsweise des Frameworks erlaubt neben der Analyse des Codes und der Transformation für die Vektorisierung auch die Anpassung an die vorliegende Hardware ohne das Eingreifen des Users. Eine explizite Programmierung von Codes für SSE durch *Intrinsics* oder gar *Assembler-Code* kann dem User oftmals erspart werden, da der End-Compiler bei einer entsprechenden Vorverarbeitung des Codes durch PluTo-SIMD diesen selbstständig vektorisieren kann. Dies erlaubt eine Code-Optimierung ohne besondere Kenntnis der Thematik und würde eine Einbettung der Methode in einen klas-

sischen End-Compiler grundsätzlich gestatten.
Darüber hinaus konnte gezeigt werden, dass eine Modifikation der lexikographischen Minimierung in der Optimierungsphase von PluTo die Performance der resultierenden Codes verbessern kann.
Insgesamt wird deutlich, dass aktuelle Compiler von der vorgeschalteten Codeoptimierung häufig profitieren können bzw. deren gesamtes Potenzial durch solche Optimierungen z. T. erst ausgeschöpft werden kann.

Ausblick: Durch die vorhandene und gut erweiterbare mathematische Funktionalität in PluTo sind in Zukunft auch zusätzliche Erweiterungen der IP-Lösungsphase vorgesehen.
Die Einbettung der Source-To-Source-Transformationen direkt in einen End-Compiler (wie den GCC) wäre für die Optimierung ein weiterer Vorteil, da dann z. B. die parameterabhängige Optimierung (siehe Abschnitt 4.3.1) bei zur Compile-Zeit definierten Programmparametern automatisch möglich wäre.
Generell könnte in Zukunft durch das Framework entschieden werden, welche Methode des Tilings (alle Schleifen, SIMD-spezifisch, SIMD- und Cache-spezifisch, gar keines) aufgrund der vorliegenden Codestruktur und in Abhängigkeit von den Parametern die wahrscheinlich beste wäre, um diese im Anschluss durchzuführen.
Die Verbindung von Vektorisierung und Multicore-Beschleunigung ist im Framework bereits möglich und kann in Zukunft näher beleuchtet werden.
Für die Schwächen von PluTo-SIMD (siehe Abschnitt 5.2.5) sollen ebenfalls entsprechende Modelle entwickelt werden.
Auch eine Vertauschung der Schleifen, sodass die innerste Schleife einer Verschachtelung möglichst durch innerste Array-Dimensionen läuft, wäre ein, in Hinblick auf die Optimierung der Speicherzugriffe, lohnender Ansatz.
Erste Tests einer Methode speziell für Codes mit Abhängigkeiten in mehreren Dimensionen wurden bereits durchgeführt. Besonders in solchen Fällen *kann* ein Tiling weiterer Schleifen (neben der vektorisierten) vorteilhaft sein. Eine Möglichkeit der Erweiterung des SIMD- und Cache-spezifischen Tilings auf mehrere oder gar alle Schleifen ist, die Schleifen in zwei Kategorien zu unterteilen, nämlich der vektorisierten Schleife **VEC(v)** und den nicht vektorisierten Schleifen **non-VEC(nv)**. In solch einem Fall resultieren Tile-Quantitätenvektoren der Form $\mathbf{q}^{L1} = (q_{nv}^{L1}\ ,\ \ldots\ ,\ q_{nv}^{L1}\ ,\ q_{\mathbf{v}}^{L1}\ ,\ q_{nv}^{L1}\ ,\ \ldots\ ,\ q_{nv}^{L1})$, wobei q_v^{L1} wiederum eine Register-konforme Quantität ist und die q_{nv}^{L1} beliebig sind. Die zu ladende Anzahl an Elementen eines Tiles müsste wiederum an den Cache angepasst werden. Daraus resultierende Tile-Quantitäten sind durch das mehrdimensionale Tiling relativ klein und ermöglichen, durch die Aufteilung auf mehrere Dimensionen, somit eine Anpassung an den Cache schon bei entsprechend kleinen Programmparametern.

Erste Beispielmessungen für die Matrixmultiplikation zeigen, dass auch für diese Methode Cache-abhängige globale Laufzeit-Optima für die Quantitäten vorliegen (siehe Abbildung 6.1, die eingezeichnete Ebene im rechten Bild entspricht der PluTo-default-Laufzeit).

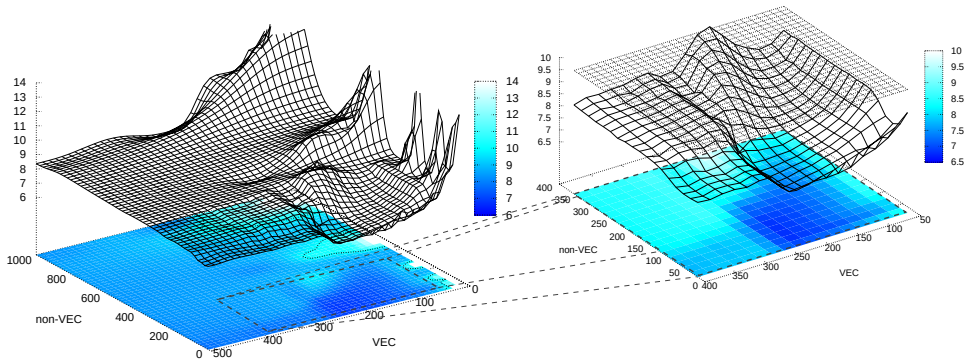


Abbildung 6.1: SIMD- und Cache-spezifisches Tiling für alle Schleifen [Laufzeiten (in s)]

Darüber hinaus wäre eine Wahl der zu blockenden Dimensionen abhängig von der Zugriffsstruktur des vorliegenden Codes möglich, sodass bei einer d-fach verschachtelten Schleife genau die d' Dimensionen ($1 \leq d' \leq d$) geblockt werden, in denen Datenzugriffe vorhanden sind. In diesem Kontext wäre es auch denkbar, die Generierung der Tiling-Hyperebenen durch die IPs in einer Weise durchzuführen, dass direkt ein passendes d'-dimensionales Tiling gesucht wird. Aktuell kann PluTo nur dann ein Tiling einer verschachtelten Schleife generieren, wenn *alle* Dimensionen dieser gültig blockbar sind. Dadurch wäre für einen Fall wie in Abbildung 3.27, in dem kein volldimensionales (hier zweidimensionales) Tiling gültig ist, ein $d' = 1$-dimensionales Tiling möglich.
Erweiterungen des PluTo-Frameworks für weitere parallele Architekturen sind bereits in der Entwicklung, eine BETA-Version für die Generierung von CUDA-Code ist auf der PluTo-Homepage [Bona] erhältlich.

A Appendix

A.1 Arbeitsweise für mehrere Statements

Die Erweiterung auf beliebige Statements wird analog zu Abschnitt 3.9.2 dargestellt. Details, welche an dieser Stelle ausgelassen wurden, sind entsprechend in 3.9.2 zu finden.

Im Folgenden sei

$$\delta_e^k(\vec{s},\vec{t}) = \phi_{S_j}^k(\vec{t}) - \phi_{S_i}^k(\vec{s}) \quad .$$

Gültigkeit In PluTo werden die Tiling-Hyperebenen ϕ_S^k (für Statement S) für jedes Level k iterativ gesucht, wobei für jede solche Hyperebene nach Lemma 3.40 die *Gültigkeitsbedingung* (A.1) gelten muss

$$\delta_e^k(\vec{s},\vec{t}) = \phi_{S_j}^k(\vec{t}) - \phi_{S_i}^k(\vec{s}) = \mathbf{h_k^{S_j}} \cdot \vec{t} - \mathbf{h_k^{S_i}} \cdot \vec{s} \geq 0 \quad \forall(\vec{s},\vec{t}) \in \mathcal{P}_e \tag{A.1}$$

Optimalität Die Ungleichung zur *Volumenbeschränkung* (Kommunikationsvolumen und Wiederverwendung) für Level k in PluTo lautet allgemein

$$\delta_e^k(\vec{s},\vec{t}) = \phi_{S_j}^k(\vec{t}) - \phi_{S_i}^k(\vec{s}) = \mathbf{h_k^{S_j}} \cdot \vec{t} - \mathbf{h_k^{S_i}} \cdot \vec{s} \leq \mathbf{u} \cdot \vec{p} + w = \nu(\vec{p}) \quad \forall(\vec{s},\vec{t}) \in \mathcal{P}_e \tag{A.2}$$

wobei $\vec{p}$ der N_p-dimensionale Vektor der Programmparameter bei **nicht-gleichförmigen Abhängigkeiten** ist, da der Abhängigkeitsvektor dann nicht konstant ist und ggf. nur über die Parameter abgeschätzt werden kann.

Linearisierung bei nicht-gleichförmigen Abhängigkeiten Um eine Ungleichung für eine Kante $e \in E$ aus dem DDG bei nicht-gleichförmigen Abhängigkeiten zu linearisieren,

wendet man das affine Farkas Lemma an, denn somit gilt nach (A.2)

$$\begin{aligned} \delta_e^k(\vec{s},\vec{t}) = \quad & \phi_{S_j}^k(\vec{t}) - \phi_{S_i}^k(\vec{s}) && \leq \mathbf{u}\cdot\vec{p}+w \quad \forall(\vec{s},\vec{t}) \in \mathcal{P}_e \\ \Rightarrow \quad & \mathbf{u}\cdot\vec{p}+w-(\phi_{S_j}^k(\vec{t}) - \phi_{S_i}^k(\vec{s})) && \geq 0 \qquad\quad \forall(\vec{s},\vec{t}) \in \mathcal{P}_e \quad . \end{aligned}$$

Daraus folgt mit Lemma 2.13:

$$\Rightarrow \quad \mathbf{u}\cdot\vec{p}+w-(\phi_{S_j}^k(\vec{t}) - \phi_{S_i}^k(\vec{s})) \quad \equiv \lambda_0 + \sum_{l=1}^{M_e} \lambda_l {\mathcal{P}_e}^l \qquad\qquad \lambda_l \geq 0 \qquad (A.3)$$

${\mathcal{P}_e}^l$ ist die LHS der l-ten Zeile des $\mathcal{P}_e$ beschreibenden Ungleichungssystems (also für das Ungleichungssystem in der Form $A\cdot x+b\geq 0$ die LHS von $a_i\cdot x+b_i\geq 0$, vgl. Lemma 2.13) und M_e die Anzahl der Ungleichungen dieses Systems.

Das System in (A.3) kann nun durch Koeffizientenvergleich und anschließende Anwendung der Gausselimination und Elimination der λ_i durch FME (wegen der Ungleichungen für λ_i) gelöst werden, was in einer Menge von Ungleichungen resultiert, welche lediglich von den $\{h_{k,1}^{S_i},\ldots,h_{k,d}^{S_i}\}$ $\forall S_i\in\mathcal{S}$ und $\{u_1,\ldots,u_{N_p},w\}$ abhängen und somit eine für das Ungleichungssystem kompatible Form haben.

Ein Beispiel hierzu findet sich auch in [Bon08, Bsp. 3.11.1].

Nulllösung Die *Nulllösung* kann verhindert werden, indem man lediglich positive Koeffizienten $h_{k,i}$ zulässt und

$$\sum_{i=1}^{d} h_{k,i}^S \geq 1 \quad \forall S\in\mathcal{S} \qquad (A.4)$$

zum System hinzuzufügen.

Linear unabhängige Lösungen Da lediglich linear unabhängige Hyperebenen zu den bisher gefundenen betrachtet werden sollen, kann der *Orthogonalraum* zu diesen gebildet werden, da durch ihn linear abhängige Lösungen ausgeschlossen werden können.

Lemma A.1. *Sei H_S die Matrix, welche zeilenweise Hyperebenen(-Normalen) enthält. Dann bilden die Zeilen von*

$$H_S^{\perp} = \mathcal{I} - H_S\left(H_S^T\cdot H_S\right)^{-1} H_S^T \qquad (A.5)$$

den Orthogonalraum zu H_S (bzgl. des Standardskalarproduktes). [Pen55; LP92; BRS07]

Definition A.2. *$H_S^{k\perp}$ ist der Orthogonalraum zu den Hyperebenen $\{\phi_S^1, \ldots, \phi_S^{k-1}\}$.*

Ein linear unabhängiger Vektor ($\neq \vec{0}$) darf nicht im Kern des Orthogonalraumes liegen, muss also mindestens eine Nicht-Nullkomponente im Orthogonalraum besitzen. [Bon08, Kap. 3.9]
Da auch die Bedingung $H_S^{\perp} \cdot \mathbf{h_k^S} \neq \vec{0}$ keinen konvexen Raum bildet, müsste $H_S^{i\perp} \cdot \mathbf{h_k^S} > 0$ **oder** $H_S^{i\perp} \cdot \mathbf{h_k^S} < 0$ für mindestens eine Zeile i von $H_S^{\perp}$ gelten. Da die Kombinationsmöglichkeiten für solch eine Überprüfung enorm schnell anwachsen, wird in PluTo auch in diesem Fall eine vereinfachende Annahme getroffen, nämlich dass alle Ungleichungen *positiv* sind und somit nur gefordert werden muss, dass **mindestens eine** mit '≥ 1' erfüllt wird während alle '≥ 0' sind, wodurch lediglich $H_S^{k\perp} \cdot \mathbf{h_k^S}^T \geq \vec{0}$ und $(1 \ldots 1) \cdot (H_S^{k\perp} \cdot \mathbf{h_k^S}^T) \geq 1$ hinzugefügt werden müssen.

Zielfunktion Unter den gegebenen **Restriktionen** soll nun der Vektor $(\mathbf{u}, w, \mathbf{h_k^{S_1}}, \ldots, \mathbf{h_k^{S_T}})$ lexikographisch minimiert werden.

$$\min_{\prec}(u_1, \ldots, u_{N_p}, w, \quad h_{k,1}^{S_1}, \ldots, h_{k,d}^{S_1} \quad , \ldots, \quad h_{k,1}^{S_T}, \ldots, h_{k,d}^{S_T}) \tag{A.6}$$

Bemerkung A.3. *Um auf diese Weise gefundene Hyperebenen verschiedener Statements zu **verschmelzen**, müssen Strategien und Bedingungen ermittelt werden, wann dies überhaupt möglich ist. Miteinander verschmolzene Hyperebenen entsprechen verschmolzenen Schleifenköpfen, sodass mehrere Statements in einer verschachtelten Schleife ggf. auf verschiedenen Tiefen stehen (wie in Listing 3.3). Ist eine solche Verschmelzung von Schleifen bzw. Hyperebenen nicht möglich, wird eine **skalare Dimension** eingefügt (welche durch eine natürliche Zahl z charakterisiert ist). Diese sagt aus, dass die verschachtelten Schleifen (bzw. Hyperebenen) folgender Dimensionen nicht mit den Dimensionen anderer z'-Skalar-Dimensionen ihrer Statements verschmolzen werden dürfen. Somit werden die entsprechenden Schleifenblöcke nacheinander ausgeführt.*

Details zu dieser Vorgehensweise und eine ausführliche Erläuterung der Begriffe finden sich in [Bon08, Kap. 4].

Resultierendes System Aus diesen Elementen resultiert das zu lösende *Optimierungsproblem* für Statements $S_1, \ldots, S_T$ auf Level k ($\mathbf{h_k^{S_i}} = (h_{k,1}^{S_i}, \ldots, h_{k,d}^{S_i})$) mit d-dimensionalem Indexraum:

$$
\begin{array}{lrll}
\min_{\prec} & (\mathbf{u}, w, \mathbf{h_k^{S_1}}, \ldots, \mathbf{h_k^{S_T}}) & & \\
\text{s.t.} & \phi_{S_j}(\vec{t}) - \phi_{S_i}(\vec{s}) & \geq 0 & \forall e^{S_i \to S_j} \\
 & \phi_{S_j}(\vec{t}) - \phi_{S_i}(\vec{s}) & \leq \nu(\vec{p}) & \forall e^{S_i \to S_j} \\
 & \mathbf{h_k^S} & \neq \vec{0} & \forall S \in \mathcal{S} \\
 & H_S^{k\perp} \cdot \mathbf{h_k^S}^T & \neq \vec{0} & \forall S \in \mathcal{S} \\
 & \mathbf{h_k^S} & \in \mathbb{Z}^d & \forall S \in \mathcal{S}
\end{array}
$$

$\downarrow$ bzw. in PluTo $\downarrow$

$$
\begin{array}{lrlll}
\min_{\prec} & (\mathbf{u}, w, \mathbf{h_k^{S_1}}, \ldots, \mathbf{h_k^{S_T}}) & & & \\
\text{s.t.} & \phi_{S_j}(\vec{t}) - \phi_{S_i}(\vec{s}) & \geq 0 & \forall e^{S_i \to S_j} & (A.1) \\
 & \phi_{S_j}(\vec{t}) - \phi_{S_i}(\vec{s}) & \leq \nu(\vec{p}) & \forall e^{S_i \to S_j} & (A.2) \\
 & (1 \ldots 1) \cdot \mathbf{h_k^S}^T & \geq 1 & \forall S \in \mathcal{S} & (A.4) \\
 & H_S^{k\perp} \cdot \mathbf{h_k^S}^T & > \vec{0} & \forall S \in \mathcal{S} & (A.5) \\
 & \mathbf{h_k^S} & \in \mathbb{Z}^{+d} & &
\end{array}
$$

Für die gesuchte Tiling-Matrix H_S für Statement S ist $\mathbf{h_k^S}$ die k-te Tiling-Hyperebene bzw. die k-te Zeile. (A.1) und (A.4) werden ggf. (bei nicht-gleichförmigen Abhängigkeiten) durch Farkas linearisiert und (A.5) mit '>' angegeben.

A.2 Wahl der Level-2-Tile Schleife

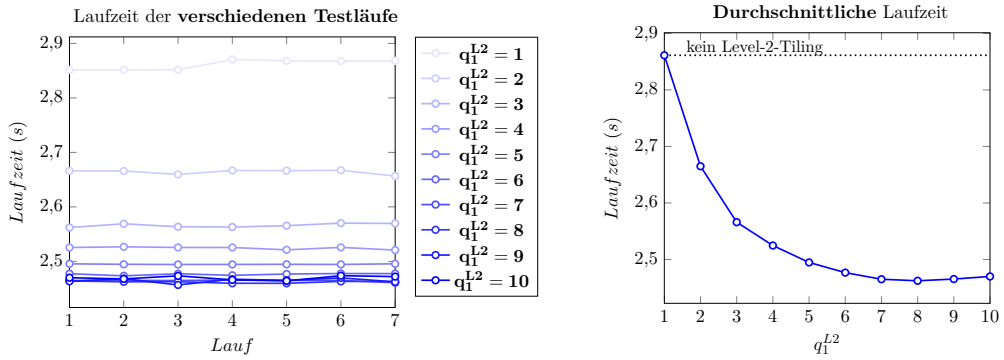


Abbildung A.1: PluTo-SIMD mit Level-2-Tiling der **ersten (äußersten)** Schleife (matmul)

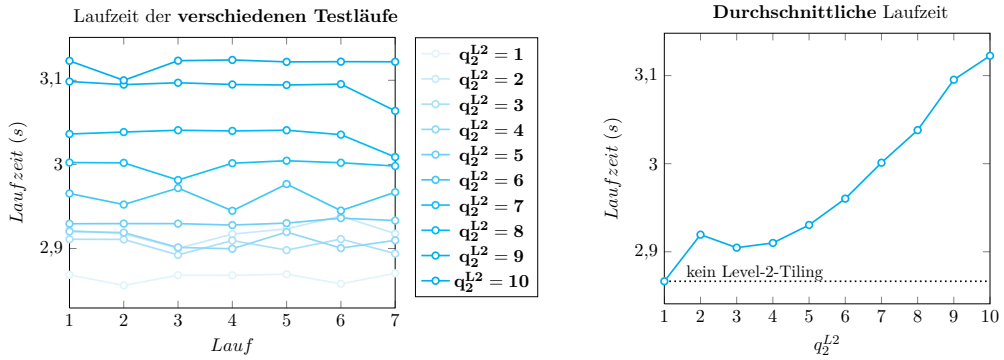


Abbildung A.2: PluTo-SIMD mit Level-2-Tiling der **zweiten (mittleren)** Schleife (matmul)

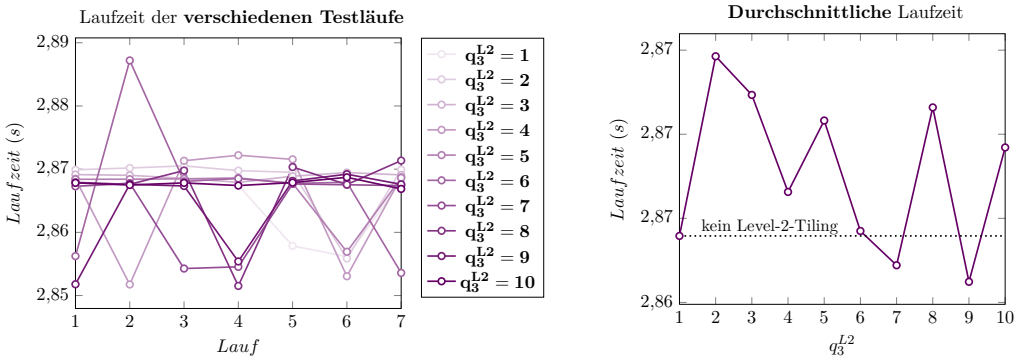


Abbildung A.3: PluTo-SIMD mit Level-2-Tiling der **dritten (innersten)** Schleife (matmul)

Abbildungsverzeichnis

2.1 2-D Linearkombination, Affinkombination und Konvexkombination 7
2.2 Konvexe Menge 8
2.3 Nichtkonvexe Menge 8
2.4 Konvexe Hülle 8
2.5 Konische Hülle 8
2.6 Pyramide als Polyeder im $\mathbb{R}^3$ 11
2.7 Farkas Lemma - geometrisch 13
2.8 Maximierung einer linearen Zielfunktion 15
2.9 Der Simplex-Algorithmus 18
2.10 Berechnung der ersten M Fibonacci-Zahlen 23
2.11 Zweioperandige SSE-Instruktion auf 4 Elementen (float) 25
2.12 Cache-Hierarchie des Intel® Xeon® X5650 26
2.13 Phasen eines Compilers [1] 27

3.1 Quellcodeausschnitt mit zugehörigem Abhängigkeitsgraph (**DDG**) 30
3.2 Perfekt verschachtelte Schleife 31
3.3 Nicht-perfekt verschachtelte Schleife 31
3.4 Gleichförmige Abhängigkeiten in einem 2-D Indexraum 33
3.5 Nicht-gleichförmige Abhängigkeiten in einem 2-D Indexraum 33
3.6 Abhängigkeitspolyeder für beide Abhängigkeiten 35
3.7 Schleifenresiduum mit negativem Zyklus 38
3.8 Abhängigkeitstransformation 40
3.9 Beispiele für unimodulare Transformationsmatrizen 43
3.10 Unimodulare Transformation 43
3.11 Nicht-unimodulare Transformation 43
3.12 Außen-Parallele Schleife 46
3.13 Innen-Parallele Schleife 47
3.14 Verzögert Parallele Schleife 47
3.15 Transformation bei verzögerter Parallelität 48

3.16 Ausgangspolyeder . . . 48
3.17 Gültige Transformation . . . 49
3.18 Nicht gültige Transformation . . . 49
3.19 Native innere Parallelität . . . 49
3.20 Transformation für äußere Parallelität . . . 49
3.21 Schleife für Space-Time-Mapping . . . 52
3.22 Gegebene Parallelität: theoretisch zeitoptimal aber hohe Kommunikation 54
3.23 Feautrier: theoretisch zeitoptimal aber schlechte Datenwiederverwendung 54
3.24 Mögliche Kombination der Vorteile beider anderer Versionen (manuell erstellt) . . . 54
3.25 Codeveränderung eines rechteckigen Tilings . . . 56
3.26 Iterationsänderung eines rechteckigen Tilings . . . 57
3.27 Kein 2-D Tiling möglich . . . 62
3.28 Tiling legal (grün) und illegal (rot) . . . 62
3.29 Tiling illegal **ohne** Schleifenumkehr . . . 62
3.30 1-D perfekt verschachtelter Jacobi . . . 64
3.31 Hyperebenen $\phi^1 = (1,0)$, $\phi^2 = (1,1)$ . . . 64
3.32 Hyperebenen $\phi^1 = (1,1)$, $\phi^2 = (1,0)$ . . . 64
3.33 Hyperebenen $\phi^1 = (2,1)$, $\phi^2 = (1,0)$ . . . 64
3.34 Kommunikationsvolumen für den 1-D Jacobi . . . 68
3.35 Ablauf eines Transformationsprozesses via PluTo . . . 71

4.1 Originale Schleifen (links) und in 32-er Blöcke eingeteilte Schleifen (rechts) 85
4.2 Zusammenfassung der Command-Line-Optionen in PluTo-SIMD . . . 86
4.3 Überspringen der Dummy-Variable im Lösungsvektor . . . 88
4.4 Zweifaches Tiling nach PluTo (default) . . . 93
4.5 Tiling nach PluTo-SIMD für zwei Cache Level mit j vektorisierbar . . . 98
4.6 Tiling nach PluTo-SIMD für zwei Cache Level mit i vektorisierbar . . . 98
4.7 Visualisierung zu Abbildung 4.5 (links) und Abbildung 4.6 (rechts) . . . 98
4.8 Konfigurationsdatei für das Cache- und SIMD-angepasste Tiling . . . 107
4.9 Standard Matrix-Multiplikation für $\alpha = 1$ . . . 110
4.10 Durch PluTo-SIMD Cache-spezifisch transformierte Matrix-Multiplikation 111
4.11 1-D Jacobi, vgl. Abbildung 3.30 . . . 111
4.12 Durch PluTo-SIMD Cache-spezifisch transformierter Jacobi . . . 112
4.13 Berechnung der Korrelationsmatrix . . . 113

4.14 Durch PluTo-SIMD Cache-spezifisch transformierte Berechnung der Korrelationsmatrix . . . 115
4.15 Flowchart zum Ablauf von PluTo-SIMD; Exponent 1 und 2 ≙ Phase 1 und 2 . . . 116

5.1 Code mit zwei nicht-gleichförmigen Abhängigkeiten . . . 118
5.2 **Ohne** Tausch der Parameter . . . 119
5.3 **Mit** Tausch der Parameter . . . 119
5.4 Laufzeit der *extremsten* Konfigurationen im Vergleich . . . 119
5.5 Original-Code der Matrix-Multiplikation . . . 120
5.6 Original-Code der Berechnung der Korrelationsmatrix . . . 121
5.7 Matrix-Multiplikation **Level-1**-Tiling . . . 123
5.8 Matrix-Multiplikation **Level-1- und Level-2**-Tiling . . . 123
5.9 Korrelationsmatrix **Level-1**-Tiling . . . 124
5.10 Korrelationsmatrix **Level-1- und Level-2**-Tiling . . . 124
5.11 Matrix-Multiplikation **Level-1**-Tiling **256 etc.** . . . 126
5.12 Matrix-Multiplikation **Level-1- und Level-2**-Tiling **256 etc.** . . . 126
5.13 Matrix-Multiplikation **Level-1**-Tiling **257 etc.** . . . 126
5.14 Matrix-Multiplikation **Level-1- und Level-2**-Tiling **257 etc.** . . . 126
5.15 Korrelationsmatrix **Level-1**-Tiling **256 etc.** . . . 127
5.16 Korrelationsmatrix **Level-1- und Level-2**-Tiling **256 etc.** . . . 127
5.17 Matrix-Multiplikation Level-1-Tiling durch PluTo-SIMD . . . 129
5.18 Modifikationen des Statements der Matrix-Multiplikation . . . 130
5.19 Matrix-Multiplikation Level-1-Tiling $C = A \cdot B$ . . . 131
5.20 Matrix-Multiplikation Level-1-Tiling $C = A \cdot B + D \cdot E$ . . . 131
5.21 Matrix-Multiplikation Level-1-Tiling $C = A \cdot B + D \cdot E + F \cdot G$ . . . 131
5.22 Matrix-Multiplikationen matmul1, matmul2 und matmul3 Level-1-Tiling . 131
5.23 Einfluss der Wahl der Level-2-Tiling Schleife . . . 132
5.24 Level-1- bzw. Level-1- und Level-2-Tiling Resultate der Matrix-Multiplikation 132
5.25 Berechnung der Korrelationsmatrix Level-1-Tiling durch PluTo-SIMD (TPS) 133
5.26 Einfluss der Optionen MAX, MIN, TPS bei der Berechnung der Korrelationsmatrix . . . 134
5.27 Level-1- bzw. Level-1- und Level-2-Tiling Berechnung der Korrelationsmatrix . . . 135
5.28 1-D Jacobi . . . 138
5.29 Weitere BLAS-Routinen (GEMVER) . . . 139

5.30 Matrix-Multiplikation mit unterer Dreiecks-Matrix STRMM 140
5.31 Matrix-Multiplikation kompiliert mit GCC (ausgefüllt) und ICC (schraffiert) 141
5.32 Berechnung der 'Summe der Differenzen' 142
5.33 'Summe der Differenzen' **308er** 143
5.34 'Summe der Differenzen' **309er** 143
5.35 'Summe der Differenzen' **N = 708** 144
5.36 'Summe der Differenzen' **N = 709** 144

6.1 SIMD- und Cache-spezifisches Tiling für alle Schleifen [Laufzeiten (in s)] . 147

A.1 PluTo-SIMD mit Level-2-Tiling der **ersten (äußersten)** Schleife (matmul) 153
A.2 PluTo-SIMD mit Level-2-Tiling der **zweiten (mittleren)** Schleife (matmul) 153
A.3 PluTo-SIMD mit Level-2-Tiling der **dritten (innersten)** Schleife (matmul) 153

Literaturverzeichnis

[Aig06] M. Aigner. *Diskrete Mathematik*. Aufbaukurs Mathematik. Vieweg, 2006.

[ASLU08] A. V. Aho, R. Sethi, M. S. Lam, and J. D. Ullmann. *Compiler: Prinzipien, Techniken und Werkzeuge*. Pearson Studium. Addison Wesley Verlag, 2008.

[ATKA03] M. Abbas, B. Talha, S. Khan, and A. Abbas. A motion estimation chip for block based mpeg-4 video applications. In *Multi Topic Conference, 2003. INMIC 2003. 7th International*, pages 253 – 257, December 2003.

[Ban94] U. K. Banerjee. *Loop Parallelization*. Loop transformations for restructuring compilers. Kluwer Academic Publishers, Norwell, MA, USA, 1994.

[Ban97] U. Banerjee. *Dependence analysis*. Number Bd. 3 in Loop transformations for restructuring compilers. Kluwer Academic Publishers, 1997.

[Bas] C. Bastoul. ClooG: Chunky Loop Generator.

[Bas08] C. Bastoul. Extracting polyhedral representation from high level languages. Technical report, LRI, Paris-Sud University, 2008. Related to the Clan tool.

[BDG+00] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14:189–204, 2000.

[BHRS08] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[Bona] U. Bondhugula. PluTo: An automatic parallelizer and locality optimizer for multicores.

[Bonb] U. Bondhugula. PluTo: An automatic parallelizer and locality optimizer for multicores.

[Bon08] U. Bondhugula. *Effective automatic parallelization and locality optimization using the polyhedral model*. PhD thesis, Columbus, OH, USA, 2008. Adviser-Sadayappan, P.

[BPM04] I. D. Baxter, C. Pidgeon, and M. Mehlich. Dms: Program transformations for practical scalable software evolution. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 625–634, Washington, DC, USA, 2004. IEEE Computer Society.

[Bro71] W. S. Brown. On euclid's algorithm and the computation of polynomial greatest common divisors. In *Proceedings of the second ACM symposium on Symbolic and algebraic manipulation*, SYMSAC '71, pages 195–211, New York, NY, USA, 1971. ACM.

[BRS07] U. Bondhugula, J. Ramanujam, and P. Sadayappan. Automatic mapping of nested loops to fpgas. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, pages 101–111, New York, NY, USA, 2007. ACM.

[BT97] D. Bertsimas and J. N. Tsitsiklis. *Introduction to linear optimization*. Athena Scientific series in optimization and neural computation. Athena Scientific, 1997.

[Cor01] T. H. Cormen. *Introduction to algorithms*. MIT electrical engineering and computer science series. MIT Press, 2001.

[Dan66] G. B. Dantzig. *Lineare Programmierung und Erweiterungen; ins Deutsche übertragen und bearbeitet von Arno Jaeger*. Springer-Verlag, 1966.

[DR95] M. Dion and Y. Robert. Mapping affine loop nests: new results. In *Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking*, HPCN Europe '95, pages 184–189, London, UK, 1995. Springer-Verlag.

[Dun90] R. Duncan. A survey of parallel computer architectures. *Computer*, 23(2):5–16, February 1990.

[DV96] A. Darte and F. Vivien. Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. Technical report, 1996.

[Fea] P. Feautrier. PipLib: Parametric Integer Programming Library.

[Fea92a] P. Feautrier. Some efficient solutions to the affine scheduling problem: I. one-dimensional time. *Int. J. Parallel Program.*, 21:313–348, October 1992.

[Fea92b] P. Feautrier. Some efficient solutions to the affine scheduling problem: Ii. multidimensional time. *Int. J. Parallel Program.*, 21:389–420, 1992.

[Fly72] M. J. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9):948 –960, September 1972.

[GA05] S.I. Gass and A. Assad. *An annotated timeline of operations research: an informal history*. International series in operations research & management science. Kluwer Academic Publishers, 2005.

[GL96] M. Griebl and C. Lengauer. The loop parallelizer loopo. In *Proc. Sixth Workshop on Compilers for Parallel Computers, volume 21 of Konferenzen des Forschungszentrums Jülich*, pages 311–320. Forschungszentrum, 1996.

[Gri04] M. Griebl. Automatic parallelization of loop programs for distributed memory architectures, 2004.

[GvdG02] K. Goto and R. van de Geijn. On reducing tlb misses in matrix multiplication, 2002.

[hta] http://docs.python.org/library/2to3.html *(Python)*.

[htb] http://gcc.gnu.org/ *(GCC)*.

[huct] http://software.intel.com/en-us/articles/intel-compilers/ *(ICC)*.

[Int99] Intel architecture software developer's manual, volume 1 – basic architecture. *Intelligence/sigart Bulletin*, 1999.

[Int11] Intel. Intel processor identification and the cpuid instruction. Technical report, 2011.

[IT88] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 319–329, New York, NY, USA, 1988. ACM.

[Jim99] M. Jimenez. *Multilevel Tiling for Non-Rectangular Iteration Spaces*. PhD thesis, 1999.

[JRR97] M. Jünger, G. Reinelt, and G. Rinaldi. The traveling salesman problem. In M. DellÁmico, F. Maffioli, and S. Martello, editors, *Annotated bibliographies in combinatorial optimization*, pages 199–221. John Wiley & Sons, 1997.

[Jä96] K. Jänich. *Lineare Algebra*. Springer-Lehrbuch. Springer, 6. edition, 1996.

[Kan39] L. V. Kantorovich. Mathematical methods of organizing and planning production. 1939.

[KM72] V. Klee and G. J. Minty. How good is the simplex algorithm? In O. Shisha, editor, *Inequalities*, volume III, pages 159–175. Academic Press, New York, 1972.

[Kön04] K. Königsberger. *Analysis 2.* Springer-Lehrbuch. Springer, 2004.

[LCF+03] J. Lu, H. Chen, R. Fu, W.-C. Hsu, B. Othmer, P.-C. Yew, and D.-Y. Chen. The performance of runtime data cache prefetching in a dynamic optimization system. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 180–, Washington, DC, USA, 2003. IEEE Computer Society.

[LL97] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Parallel Computing*, pages 201–214. ACM Press, 1997.

[LP92] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. Technical report, Ithaca, NY, USA, 1992.

[Muc97] S. S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[Net04] N. Nethercote. *Dynamic Binary Analysis and Instrumentation.* PhD thesis, 2004. It's related with Valgrind framework.

[NS03] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *In Third Workshop on Runtime Verification (RV'03*, 2003.

[NW88] G. L. Nemhauser and L. A. Wolsey. *Integer and combinatorial optimization.* Wiley-Interscience, New York, NY, USA, 1988.

[Pen55] R. Penrose. A generalized inverse for matrices. *Proceedings of the Cambridge Philosophical Society*, 51:406–413, 1955.

[Ram95] J. Ramanujam. Beyond unimodular transformations. *Journal of Supercomputing*, 9:365–389, 1995.

[RDVB10] W. J. Ryan, A. T. Duchowski, E. A. Vincent, and D. Battisto. Match-moving for area-based analysis of eye movements in natural tasks. In *Proceedings of the 2010 Symposium on Eye-Tracking Research & Applications*, ETRA '10, pages 235–242, New York, NY, USA, 2010. ACM.

[SB08] K. Schöffmann and L. Böszörmenyi. Fast segmentation of h.264/avc bitstreams for on-demand video summarization. In *Proceedings of the 14th inter-*

national conference on Advances in multimedia modeling, MMM'08, pages 265–276, Berlin, Heidelberg, 2008. Springer-Verlag.

[SCE03] E. Salvador, A. Cavallaro, and T. Ebrahimi. Spatio-temporal shadow segmentation and tracking. In *in Proc. of Visual Communications and Image Processing*, pages 389–400, 2003.

[Sch86] A. Schrijver. *Theory of linear and integer programming:*. Wiley, 1986.

[Sch10] A. Schrijver. Theory of linear and integer programming - course. 2010.

[Sho81] R. Shostak. Deciding linear inequalities by computing loop residues. *J. ACM*, 28:769–779, October 1981.

[TJYD09] D. Terpstra, H. Jagode, H. You, and J. Dongarra. Collecting performance data with papi-c. 2009.

[Xue97] J. Xue. Unimodular transformations of non-perfectly nested loops. *Parallel Computing*, 22:1621–1645, 1997.

[Xue00] J. Xue. *Loop tiling for parallelism*. Kluwer international series in engineering and computer science. Kluwer Academic, 2000.