TECHNISCHE UNIVERSITÄT
CHEMNITZ

Optimierung der Energie-Effizienz für Algorithmen der Linearen Algebra durch SIMD-Programmierung und AVX-Vektorisierung

Dissertation
zur Erlangung des akademischen Grades

Dr.-Ing

Herr M. Sc. Thomas Jakobs
geboren am 24.Januar 1986 in Erlangen

Fakultät für Informatik
an der Technischen Universität Chemnitz

Gutachter:

1. Prof. Dr. G. Rünger
2. Prof. Dr. J. Keller

Tag der Verteidigung:

10.Dezember 2021

Online: https://nbn-resolving.org/urn:nbn:de:bsz:ch1-qucosa2-770963

Das Werk - ausgenommen das Logo TU Chemnitz und sämtliche Abbildungen - steht unter der Creative Commons Lizenz

Attribution 4.0 International

https://creativecommons.org/licenses/by/4.0/

Zusammenfassung

Neben einer kurzen Ausführungszeit rückt bei der Optimierung von Anwendungen und Algorithmen ein geringer Energieverbrauch der genutzten Rechenressourcen in den Fokus der aktuellen Forschung. Eine hohe Energie-Effizienz von Programmen wird dabei erreicht, indem der Energieverbrauch von Programmen und Technologien reduziert wird, ohne dafür die Ausführungszeit übermäßig zu erhöhen. Im parallelen wissenschaftlichen Rechnen ist der Bedarf an energie-effizienten Programmausführungen vor allem für Algorithmen der linearen Algebra gegeben, die als Unterfunktionen in einer Vielzahl von Anwendungen eingesetzt werden. Die Vektorisierung von Programmen durch die Prozessor- und Instruktionssatzerweiterung AVX zeigt Potenzial zur energie-effizienten Ausführung von Algorithmen der linearen Algebra, wobei die erzielte Energie-Effizienz von der Umsetzung der Implementierung abhängt.

Für die gezeigten Untersuchungen werden drei repräsentativ ausgewählte Algorithmen der linearen Algebra für die Ausführung auf AVX-Vektoreinheiten genutzt. Bei der AVX-Vektorisierung der Algorithmen werden verschiedene Programmvarianten erstellt, mit denen Ausführungszeit und Energieverbrauch bei der Ausführung ermittelt werden. Die Programmvarianten unterscheiden sich dabei unter anderem in der Anwendung von Programmtransformationen, wie Loop Tiling oder einer veränderten Speicherzugriffsstruktur. Zusätzlich wird gezeigt, wie die Umsetzung verschiedener Programmieransätze, wie Autovektorisierung oder unterschiedlicher Instruktionssätze, sowie Implementierungsvarianten durch die Auswahl der verwendeten Instruktionen, die Ausführungszeit und den Energieverbrauch der Programmausführung beeinflussen. Die so erstellten Programmvarianten werden auf modernen Prozessoren verschiedener Architekturfamilien mit unterschiedlichen Ausführungsparametern, wie der eingestellten Prozessorfrequenz, ausgeführt. Die Untersuchungen zeigen, dass sich Ausführungszeit und Energieverbrauch von Programmen durch die Vektorisierung reduzieren lassen. Die Auswahl der Programmtransformationen, des Programmieransatzes und der Ausführungsparameter für die energie-effiziente Ausführung von vektorisierten Programmen kann dabei anwendungsspezifisch aufgrund der Eigenschaften des ausgewählten Algorithmus getroffen werden.

Praktische Informatik, Vektorisierung, AVX, Lineare Algebra, Energie-Effizienz

Inhaltsverzeichnis

Verwendete Symbole V

Verwendete Begriffe VII

1. Einleitung 1

2. SIMD Programmierung mit AVX und Grundlagen des energie-effizienten parallelen Rechnens 3
- 2.1. Vektorisierung . . . 4
  - 2.1.1. Vektorisierung mit AVX . . . 4
  - 2.1.2. Funktionsumfang von AVX . . . 5
  - 2.1.3. Programmierung mit AVX-Intrinsics . . . 6
  - 2.1.4. Automatische Vektorisierung durch den Compiler . . . 7
- 2.2. Energie-Effizientes Rechnen . . . 9
  - 2.2.1. Bedarf für energie-effiziente Programme . . . 9
  - 2.2.2. Begriffsdefinitionen zur energie-effizienten Programmoptimierung 10
  - 2.2.3. Metriken für Energie-Effizienz . . . 11

3. AVX-Vektorisierung der Matrix-Multiplikation 13
- 3.1. Algorithmus zur Matrix-Multiplikation . . . 13
  - 3.1.1. Implementierung der Matrix-Multiplikation . . . 14
  - 3.1.2. Eigenschaften der Matrix-Multiplikation im Hinblick auf eine Vektorisierung . . . 16
- 3.2. Vektorisierung der Matrix-Multiplikation . . . 18
  - 3.2.1. Automatisch vektorisierte Programmvarianten der Matrix-Multiplikation . . . 18
  - 3.2.2. AVX-Implementierungen der Matrix-Multiplikation . . . 20
- 3.3. Untersuchung der Ausführungszeit und des Energieverbrauchs der vektorisierten Matrix-Multiplikation . . . 28
  - 3.3.1. Ausführungsumgebung für die Untersuchung der Programmvarianten . . . 28
  - 3.3.2. Abhängigkeit der Programmvarianten zur Matrixgröße . . . 29
  - 3.3.3. Ausführungszeit der vektorisierten Programmvarianten der Matrix-Multiplikation . . . 30
  - 3.3.4. Energieverbrauch und Leistungsaufnahme der vektorisierten Matrix-Multiplikation . . . 37
- 3.4. Zusammenfassung der Vektorisierung Matrix-Multiplikation . . . 41

4. Vektorisierung der Gauss-Elimination für AVX-Vektoreinheiten 43
- 4.1. Gauss-Elimination zur Lösung linearer Gleichungssysteme . . . 44
  - 4.1.1. Algorithmus zur Gauss-Elimination . . . 45

4.1.2. Implementierung der Gauss-Elimination 46
4.1.3. Eigenschaften des Algorithmus zur Gauss-Elimination im Hinblick auf eine Vektorisierung 49
4.2. AVX-Programmvarianten der Gauss-Elimination 50
4.2.1. Gauss-Elimination mit automatisch vektorisierten Programmvarianten 50
4.2.2. Programmvarianten der Gauss-Elimination mit AVX-Intrinsics . 53
4.3. Untersuchungen zu Ausführungszeit und Energieverbrauch der vektorisierten Gauss-Elimination 64
4.3.1. Ausführungsumgebung für die Untersuchung der Programmvarianten 64
4.3.2. Ausführungszeit in Abhängigkeit zur gewählten Matrixgröße und Anzahl ausgeführter AVX-Intstruktionen der Programmvarianten 65
4.3.3. Ausführungszeit und Energieverbrauch in Abhängigkeit zur Prozessorfrequenz bei automatisch vektorisierten Programmvarianten 68
4.3.4. Ausführungszeit und Energieverbrauch der AVX-Intrinsic-Programmvarianten der Gauss-Elimination 74
4.4. Zusammenfassung der Vektorisierung der Gauss-Elimination 82

5. SIMD Gram-Schmidt Prozess zur QR-Faktorisierung einer Matrix 85
5.1. Modifizierter Gram-Schmidt Prozess zur Vektororthogonalisierung . . . 85
5.1.1. Algorithmus des modifizierten Gram-Schmidt Prozesses 86
5.1.2. Implementierung des modifizierten Gram-Schmidt Prozesses . . 87
5.1.3. Eigenschaften des modifizierten Gram-Schmidt Prozesses im Hinblick auf eine Vektorisierung 89
5.1.4. Programmvarianten durch Transformationen des modifizierten Gram-Schmidt Prozesses 90
5.2. AVX-Programmvarianten des modifizerten Gram-Schmidt Prozesses . . 93
5.2.1. *MgsAvx*: Vektorisierung des modifizierten Gram-Schmidt Prozesses 95
5.2.2. Modifikationen der Datenzugriffsreihenfolge 97
5.2.3. Modifikationen des Speicherzugriffs 99
5.2.4. Modifikationen der SIMD Reduktion 102
5.3. Ausführungszeit und Energieverbrauch der vektorisierten Programmvarianten des Gram-Schmidt Prozesses 103
5.3.1. Ausführungsumgebung für die Untersuchung der Programmvarianten des Gram-Schmidt Prozesses 104
5.3.2. Ausführungszeit der vektorisierten MGS-Programmvarianten . . 104
5.3.3. Energieverbrauch der vektorisierten MGS-Programmvarianten . 111
5.4. SIMD-Programmierung für prozessor-integrierte Grafikprozessoren . . . 113
5.5. Zusammenfassung der Vektorisierung des Gram-Schmidt Prozesses . . . 114

6. Zusammenfassung zur energie-effizienten Vektorisierung der Algorithmen 117

Literatur **119**

A. Anhang: Grundlagen zur AVX-Programmierung **127**
A.1. AVX-Datentypen 127
A.2. Alignment: Ausgerichtete Speicherzugriffe mit Vektordatentypen 128
A.3. Anpassung des Iterationsraums auf die Nutzung von Vektoren 129
A.4. Vektorisierte Reduktion 130
A.5. Liste der verwendeten Intrinsics in dieser Arbeit 131

B. Anhang zur Matrix-Multiplikation **135**

C. Anhang zur Gauss-Elimination **141**

D. Anhang zum Gram-Schmidt Prozess für Vektororthogonalisierung **145**

E. Thesen **153**

Verwendete Symbole

A, B	Eingabematrizen der gezeigten Algorithmen
$a_{i,k}$	Element in Zeile i und Spalte k von Matrix A
$a_{piv,k}$	Pivotelement zur Berechnung von $a_{k,k}$ bei der Gauss-Elimination
$a_{i,\_}$	Zeile i von Matrix A
$a_{\_,i}$	Spalte i von Matrix A
b	Eingabevektor von rechten Seiten der Gleichungssysteme für eine Gauss-Elimination
b_i	Element i von Vektor b
C	Ergebnismatrix der Matrix-Matrix-Multiplikation
i, j, k	Zähl- und Indexvariablen, es gilt $i, j, k \in \mathbb{N}$
J	Joule, Einheit für Energie
L	Matrix der Eliminationsfaktoren bei der Gauss-Elimination
l, m, n	Größen von Matrizen, es gilt $l, m, n \in \mathbb{N}$
Q	Orthogonale Matrix, entsteht aus der Zerlegung $A = Q \cdot R$
R	obere Dreiecksmatrix, entsteht aus der Zerlegung $A = Q \cdot R$
$\mathbb{R}^{m \times n}$	Matrix der Größe $m \times n$ mit reellen Zahlen
s	Sekunden, Zeiteinheit
U	obere Dreiecksmatrix, entsteht aus der Umformung $Ax = b \rightarrow Ux = b'$ in einer Gauss-Elimination
W	Watt, Einheit für Leistung
x	Vektor von Unbekannten, Ergebnis einer Gauss-Elimination
%	Formelzeichen für die Modulo-Operation $a\%b$ entspricht a mod b

Verwendete Begriffe

AVX	Advanced Vector Extensions; Die aktuelle Vektorerweiterung von Intel und AMD Prozessoren. Erweitert um zusätzliche Instruktionen in der AVX2-Erweiterung.
AVX512	Nachfolger der AVX-Erweiterung mit doppelter Registergröße und zusätzlichen Instruktionen.
horizontale Operation	Eine Operation die Werte innerhalb des selben *Vektorregisters* miteinander verrechnet oder vertauscht (vgl. [36, Vol.1 S. 12-1]).
iGPU	Prozessor-integrierter Grafikprozessor in modernen Desktop-Prozessoren.
MGS	Der modifizierte Gram-Schmidt Prozess zur Vektororthogonalisierung
MIMD	„Multiple Instruction Multiple Data“ nach der Klassifikation von Flynn [22]. Ein Rechnersystem bei dem mehrere unabhängige Instruktionsströme mit mehreren unabhängige Datenströme verarbeitet werden.
non-temporal Schreiboperation	Bei Ausführung einer Schreiboperation wird die geschriebene Speicherstelle im Hauptspeicher aktualisiert. Befindet sich diese Speicherstelle im Cache, so wird dieser Eintrag, sowie Kopien der entsprechenden Speicherstelle in anderen Caches, als ungültig markiert (vgl. [36, Vol.1 S. 10-12]). Im Gegensatz zu *Write-Back* Schreiboperationen wird der Speicherblock nicht in den Cache geladen.
serielles Programm	Programm, das ausschließlich *skalare Instruktionen* zur Ausführung bringt und keine *Vektorinstruktionen* enthält.
SIMD	„Single Instruction Multiple Data“ nach der Klassifikation von Flynn [22]. Ein Rechnersystem bei dem mit einem Instruktionsstrom mehrere Datenströme verarbeitet werden.

skalare Instruktion	Instruktion, die zur Berechnung *skalare Variablen* als Operanden nutzt (vgl. [36, Vol.1 S. 10-7], [29, S. 285])
skalare Variable	Einzelne Variable, auch skalarer Wert; entspricht einer klassischen Variable, die nur einen Wert enthält, im Gegensatz zu *Vektorvariablen* (vgl. [36, Vol.1 S. 10-1])
Vektorregister	CPU-Register, dass statt einer *skalaren Variablen* eine *Vektorvariable* enthalten kann.
Vektorvariable	Variable bzw. Operand in dem zur SIMD-Nutzung mehrere *skalare Variablen* abgelegt werden. Die Anzahl der Elemente hängt von der genutzten Technologie und der Datengröße der *skalaren Variablen* ab.
Write-Back Schreiboperation	Bei Ausführung einer Schreiboperation wird die geschriebene Speicherstelle im Cache aktualisiert. Befindet sich diese nicht im Cache, so wird sie in den Cache geladen (vgl. Write-Back-Rückschreibestrategie in [63, S. 84]).

1 Einleitung

Unter die SIMD-Programmierung fallen verschiedene Programmiersprachen und -modelle, die die Datenparallelität der Berechnungen ausnutzen. Meist werden SIMD-Programme auf spezialisierten Prozessoren oder Prozessorteilen ausgeführt, die mehrere Datensätze jeweils mit einer einzigen Instruktion berechnen [29]. Solche Prozessoren finden sich zum Beispiel bei Vektorrechnern, wie dem Cray-1 [17], bei Grafikprozessoren (GPUs), wie dem NVIDIA Tesla [55], oder bei speziellen Prozessorerweiterungen bzw. Vektoreinheiten von Prozessoren, wie dem AVX-Instruktionssatz [56]. Für die verschiedenen Ausführungsmodelle solcher Prozessoren müssen SIMD-Programme auf das jeweilige Ausführungsmodell des genutzten Prozessors angepasst werden (vgl. [63, Abschn. 3.6]).

Bei der Programmierung für Vektoreinheiten werden spezielle Vektorinstruktionen genutzt, die mehrere Datenelemente gleichzeitig verarbeiten. Ein Programm, das vektorisiert wird, muss dementsprechend so umgeschrieben werden, dass die speziellen Vektorinstruktionen genutzt werden. Dies macht die Vektorisierung zu einem aufwändigen und fehleranfälligen Prozess, für den entsprechend ausgebildete Programmierer benötigt werden [71]. Zur Erstellung von vektorisierten Programmen gibt es dabei verschiedene Verfahren: Programmcode kann automatisch durch den Compiler umgewandelt werden oder die entsprechenden Vektorinstruktionen werden explizit im Programmcode genutzt. Die Qualität der Vektorisierung kann dabei, je nach verwendeter Technik, Bibliothek und Abstraktion, variieren [61]. Zusätzlich können bei der Implementierung von vektorisierten Programmen die möglichen Implementierungsvarianten, wie Programmtransformationen [62] oder die Auswahl der Instruktionen [31], die Qualität des Programms beeinflussen.

Bei der Bewertung von Programmen gewinnt die Energie-Effizienz von Programmen neben der Performance (geringe Ausführungszeit) immer mehr an Bedeutung, wie die Beispiele [4, 12, 68, 77] zeigen. Die Eignung der Vektorisierung zur Verbesserung der Energie-Effizienz wurde generell bereits in verschiedenen Arbeiten nachgewiesen, wie beispielsweise in [13, 18, 54]. Bei diesen Arbeiten wird das vektorisierte Programm als eines der verglichenen Programme genutzt und die verschiedenen Implementierungsentscheidungen nicht untersucht. Andere Arbeiten, wie [33, 57], untersuchen die Energie-Effizienz von vektorisierten Programmen auf Spezialprozessoren, deren Aufbau sich von den im wissenschaftlichen Rechnen üblichen Prozessoren unterschiedet.

Algorithmen der linearen Algebra werden in vielen Anwendungen des wissenschaftlichen Rechnens verwendet und tragen dort als Unterfunktionen solcher Anwendun-

gen zur Ausführung der problem-spezifischen Algorithmen bei. Eine Optimierung von Algorithmen der linearen Algebra trägt somit zur Optimierung verschiedener Anwendungen bei. In dieser Arbeit werden drei Algorithmen der linearen Algebra aufgrund der folgenden Eigenschaften im Hinblick auf eine Vektorisierung ausgewählt und untersucht. Die Matrix-Multiplikation wird aufgrund der gleichmäßigen Struktur und der damit verbundenen Anwendung von Schleifentransformationen bei unterschiedlichen Vektorisierungstechniken gewählt. Die Gauss-Elimination bietet durch die Herstellung einer Dreiecksmatrix ein unregelmäßiges Speicherzugriffsmuster, wodurch sich die Gauss-Elimination zur Untersuchung des Einflusses verschiedener Speicherzugriffsinstruktionen eignet. Bei der Berechnung einer QR-Zerlegung durch den modifizierten Gram-Schmidt Prozess werden einzelne Berechnungsschritte nacheinander ausgeführt. Jeder dieser Berechnungsschritte hat dabei unterschiedliche Eigenschaften, wodurch sich verschiedene Programmtransformationen und Vektorisierungsansätze untersuchen lassen. Die untersuchten Algorithmen der linearen Algebra stellen dabei eine Auswahl an Algorithmen dar, deren Eigenschaften die Untersuchungen auf ein breites Spektrum an Algorithmen übertragbar machen.

Fokus der Arbeit

Der Fokus dieser Arbeit liegt auf der Untersuchung von Implementierungsvarianten bei der Vektorisierung in Bezug auf die Ausführungszeit und die Energie-Effizienz. Dazu werden verschiedene Vektorisierungsansätze, wie automatische Vektorisierung, und unterschiedliche AVX-Instruktionssätze untersucht. Zusätzlich wird der Einfluss von Programmtransformationen, wie Schleifen-Blockzerlegung, auf die Vektorisierung der Programme untersucht. Zur Implementierung von vektorisierten Programm können aus den verfügbaren Instruktionen verschiedene Instruktionen für die gleiche Aufgabe ausgewählt werden. Die Wahl der entsprechenden Instruktionen und eine ggf. nötige Programmanpassung dafür werden vor allem für Speicherzugriffe untersucht.

Die Untersuchungen sollen dabei nicht nur eine bestmögliche Implementierung der untersuchten Algorithmen herausstellen, sondern auch eine Übertragung auf andere Algorithmen erlauben. Dazu werden die Einflüsse der Änderungen der erstellten Programmvarianten möglichst isoliert voneinander betrachtet.

Aufbau der Arbeit

Die Untersuchungen dieser Arbeit werden anhand von drei Algorithmen der linearen Algebra durchgeführt. Dazu werden im folgenden Kapitel 2 die Grundlagen der AVX-Programmierung und der Bewertung von energie-effizienten Programmen kurz vorgestellt. Im Anschluss werden die folgenden Algorithmen der linearen Algebra jeweils einzeln vorgestellt und die Messergebnisse der Ausführungszeit und des Energieverbrauchs diskutiert: Die Matrix-Multiplikation wird in Kapitel 3, die Gauss-Elimination in Kapitel 4 und der Gram-Schmidt Prozess zur QR-Zerlegung in Kapitel 5 gezeigt. Die Erkenntnisse werden in Kapitel 6 zusammengefasst.

2

SIMD Programmierung mit AVX und Grundlagen des energie-effizienten parallelen Rechnens

Die SIMD (Single Instruction Multiple Data [22]) Programmierung wird seit den ersten Vektorrechnern, wie dem SOLOMON [6] oder dem Cray-1 [17], für die Lösung rechenintensiver Probleme im High-Performance Computing eingesetzt. Das SIMD Programmiermodell beschreibt die Eigenschaft, dass mehrere Datenelemente parallel zueinander durch die selbe Instruktion berechnet werden. Dies bedeutet, dass die Anwendung von SIMD Programmierung vor allem die Datenparallelität von Algorithmen ausnutzt.

Für die Architektur von SIMD Rechnern und deren Programmierung wurden seit den ersten Vektorrechnern diverse Modelle entwickelt. Tabelle 2.1 zeigt eine Auswahl an Arbeiten, die sich mit der SIMD Ausführung oder Programmierung beschäftigen.

Jahr	Beschreibung	Quelle
1962	Aufbau und Programmierung für den SOLOMON Supercomputer.	[6]
1986	Sprachkonstrukte für Lisp zur Nutzung der Connection Machine.	[73]
1993	Entwicklung der C* Programmiersprache, in der Datenlayouts (*Shapes*) zur besseren Vektorverarbeitung eingesetz werden.	[78]
1996	Schlagen Instruktionen zur Berechnung mehrerer Datenelemente pro Instruktion für die MAX-2 Architektur vor.	[51]
2008	CUDA-Programmier- und Ausführungsmodell für GPUs.	[55]
2012	Entwicklung von Transformationen um PRAM-Programme (in Fork) auf CUDA-GPUs auszuführen.	[10]
2015	Konstrukte für die Sprache C, mit denen die Vektorisierung kompletter Funktionen vereinfacht wird.	[62]
2017	Scalable Vector Extension Programmier- und Ausführungsmodell für ARM CPUs.	[74]

Tabelle 2.1.: Auswahl an Arbeiten, die sich mit der Entwicklung im Bereich der SIMD Datenverarbeitung beschäftigen. Die Arbeiten behandeln Programmier- und/oder Hardwareansätze zur Verbesserung der SIMD Verarbeitung.

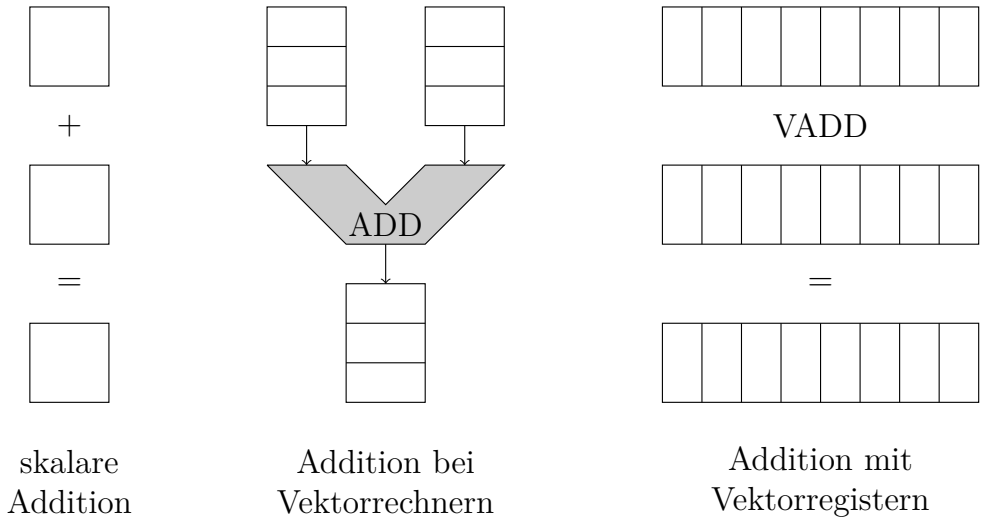

Abbildung 2.1.: Darstellung der Berechnungen bei skalaren Instruktionen (links). Die Abarbeitung der Datenelemente in Vektorrechnern (mitte, nach [63, S.140]) findet nacheinander durch die gleiche Instruktion statt. Bei der Addition mit Vektorregistern (rechts) mit Vektorinstruktionen werden mehrere Elemente mit der selben Instruktion parallel berechnet.

2.1. Vektorisierung

Moderne Prozessoren bieten mit Vektoreinheiten die Möglichkeit SIMD Programme auszuführen. Vektoreinheiten führen spezielle Instruktionen aus, die zusammen mit anderen Anpassungen, wie z.B. größeren Prozessorregistern, die Verarbeitung mehrerer Datenelemente mit der selben Instruktion ermöglichen. Die Ausführung von Programmen auf Vektoreinheiten unterscheidet sich dabei von der Ausführung auf Vektorrechnern: Auf Vektorrechnern werden die Operanden nacheinander durch dieselbe Recheneinheit geleitet, wohingegen bei Vektoreinheiten eine begrenzte Anzahl an Operanden gleichzeitig von einer speziellen Recheneinheit verarbeitet werden. Abbildung 2.1 zeigt die unterschiedliche Berechnungsweise von Instruktionen für skalare Daten (links), für Vektordaten (rechts) und die Berechnungsweise von Vektorprozessoren (mitte).

Bei der Vektorisierung werden gegenüber anderen Arten der parallelen Datenverabeitung keine gesonderten Kontrollflüsse (z.B. Threads) erstellt, die dann unabhängig abgearbeitet werden. Vielmehr wird die Parallelität erreicht, indem simultan mehrere Berechnungen des Programms ausgeführt werden. Dieses Vorgehen schränkt die Einsatzmöglichkeiten der Vektorisierung gegenüber anderen Parallelisierungsverfahren ein, bietet jedoch auch die Möglichkeit solche Verfahren zusätzlich anzuwenden.

2.1.1. Vektorisierung mit AVX

Die Befehlssatzerweiterung *Advanced Vector Extensions* (AVX)[36] ist die aktuelle Vektorerweiterung der meisten Intel und AMD Prozessoren. Die 256-Bit Register der

AVX-Erweiterung fassen dabei bis zu 8 IEEE 32 Bit Gleitkomma-Elemente, welche dann mit der selben Instruktion verarbeitet werden können. AVX ist die Weiterentwicklung der Befehlssatzerweiterungen *Multimedia Extension* (MMX, 64-Bit Register) und der *Streaming SIMD Extension* (SSE, 128-Bit Register). Amiri und Shahbahrami geben in [5] einen Überblick über die Entwicklung und den Funktionsumfang von AVX sowie dessen Erweiterung AVX512 mit 512-Bit Registern.

Bei der Vektorisierung mit AVX werden skalare Daten (klassische, nicht AVX-Datentypen) in spezielle Vektorregister geladen (*engl. loaded/packed*). Die Berechnungen werden im Anschluss auf diesen Vektorregistern durchgeführt (vgl. Abb. 2.1 (rechts)). Nach Abschluss der Berechnungen werden die Daten der Vektorregister wieder in skalare Datentypen zurück gespeichert (*engl. stored/unpacked*).

2.1.2. Funktionsumfang von AVX

Die Befehlserweiterung AVX ist auf Intel-Prozessoren seit der Sandy Bridge Architektur und auf AMD-Prozessoren seit der Bulldozer Architektur (beide 2011) verfügbar. Die Kodierung der AVX-Instruktionen erlaubt es unter anderem die Länge eines Vektors anzugeben und somit die selbe Instruktion für verschiedene Vektorlängen zu nutzen (vgl. VEX Kodierung [36, Vol. 2A, Abschn. 2.3.5]).

In der ersten AVX-Version werden eine Reihe von 256-Bit Befehlen angeboten, die vor allem für die Nutzung mit IEEE `float` und `double` Werten genutzt werden können. Darunter fallen arithmetische Operationen, wie beispielsweise Addition oder Rundung, sowie Instruktionen für Vergleiche, Konvertierungen und logische Operationen. Weiterhin werden Instruktionen angeboten, die für skalare Daten nicht benötigt werden, wie zum Laden und Speichern von Vektoren (*engl. load and store*), Lade- und Speicheroperationen mit Elementauswahl (*engl. masked load/store*), Kopieroperationen mit Elementauswahl (*engl. blend*) und zur Umsortierung von Elementen (*engl. permute*). Eine Auflistung der verfügbaren AVX-Instruktionen findet sich in [36, Vol. 1, Abschn. 14.1-14.2].

Im Gegensatz zu anderen Vektoreinheiten sind die Instruktionen der AVX-Erweiterung als 3-Operanden, und einigen 4-Operanden (z.B. für die Nutzung von Teilvektoren), Instruktionen implementiert. Diese 3-Operanden-Instruktionen bieten gegenüber den 2-Operanden-Instruktionen der Vorgänger-Variante SSE den Vorteil, dass sie eine nicht destruktive Abarbeitung zulassen, wodurch weniger Register, Kopieroperationen und Ladebefehle benötigt werden.

Ab der Prozessorgeneration von Intel Haswell und AMD Bulldozer werden Fused-Multiply-Add (FMA) -Instruktionen angeboten. Die FMA-Instruktionen führen eine Multiplikation und Addition in einem Schritt aus, um die Varianten der Gleichung $d = \pm(a \cdot b) \pm c$ zu lösen. FMA-Instruktionen sind 3-Operanden-Instruktionen und überschreiben einen der übergebenen Operanden mit dem Ergebnis.

Die Erweiterung des AVX-Befehlssatzes unter dem Namen AVX2 ist ab den Intel Haswell (2013) und AMD Excavator (2015) Prozessorgenerationen verfügbar. AVX2 erweitert den AVX-Befehlssatz um Instruktionen zur Berechnung von Ganzzahlwerten (*engl. integer*), die in der ersten Version von AVX nur eingeschränkt verfügbar

sind. Damit ist es möglich die 256-Bit Vektoroperationen nun auch für Ganzzahldatentypen der Längen 8-Bit, 16-Bit, 32-Bit und 64-Bit zu nutzen. Mit AVX2 wird die Möglichkeit der Umsortierung von Elementen, um Instruktionen zur freien Auswahl der Position innerhalb des gesamten 256-Bit Registers, erweitert (*engl. permute*). Zusätzlich werden Instruktionen zum Sammeln nicht zusammenhängend gespeicherter Daten (*engl. gather*) bereit gestellt. Der Gatheroperation wird dabei ein Vektor von Indizes übergeben, anhand dessen beliebige Elemente eines Arrays geladen werden können. Eine Auflistung der verfügbaren AVX2-Instruktionen findet sich in [36, Vol. 1, Abschn. 14.5-14.7].

2.1.3. Programmierung mit AVX-Intrinsics

Für die Programmierung der AVX-Instruktionen werden intrinsische Funktionen (kurz: Intrinsics) vom Compiler bereit gestellt. Diese AVX-Intrinsics kapseln AVX-Assembler Instruktionen in Form einer C-Funktion (vgl. [41]). Die Vektorisierung kann somit in einer höheren Programmiersprache erfolgen, was die Lesbarkeit des Codes erhöht und die Optimierung des Programms vereinfacht [38].

Bei der Programmierung mit AVX-Intrinsics werden zur Unterscheidung von skalaren Datentypen (klassische, nicht AVX-Datentypen) entsprechende Datentypen definiert, die wiederum als Vektoren bzw. Vektordatentypen bezeichnet werden. Ein Beispiel eines solchen Datentyps ist `__m256` der acht `float` Werte enthält. Eine ausführliche Erläuterung der verfügbaren AVX-Datentypen und deren Bedeutung findet sich im Anhang in Abschn. A.1.

Zum Laden und Speichern, sowie für die Durchführung verschiedener Operationen werden die intrinsischen Funktionen nach einem einheitlichen Schema definiert:

```
<type> _mm<size>_<operation>_<type-suffix>( <type> param1,..)
```

- `<type>` ist der Datentyp des Rückgabewertes bzw. der übergebenen Parameter und kann abhängig von der spezifischen Instruktion entweder ein skalarer Datentyp oder ein Vektordatentyp sein.
- `<size>` gibt die Größe in Bits des größten beteiligten Vektors an. Dieser Wert wird genutzt, um zwischen den verschiedenen Technologien und Registergrößen zu unterscheiden.
- `<operation>` drückt die implementierte Operation aus, z.B. `add` für eine Addition.
- `<type-suffix>` spezifiziert die Datentypen der Parameter und gliedert sich in 2 Teile: Der erste Teil beschreibt, ob der Datentyp gepackt (`p`) (*engl. packed*), erweitert gepackt (`ep`) (*engl. extended packed*) oder skalar (`s`) (*engl. scalar*) ist. Der zweite Teil gibt den Datentyp an und kann die folgenden Werte annehmen:

  `s` 32-Bit `float` Werte

  `d` 64-Bit `double` Werte

`i/u#` vorzeichenbehaftete/vorzeichenlose `#`-Bit Ganzzahlwerte, wobei `#` den Werten 8, 16, 32 oder 64 entsprechen kann.

Eine Auflistung und Erklärung der in dieser Arbeit verwendeten AVX-Intrinsics findet sich im Anhang in Abschn. A.5 und eine detaillierte Dokumentation aller Intrinsics in [41] und [36]. Einige AVX-Intrinsics, wie beispielsweise `set` Operationen zum expliziten Setzen der einzelnen Elemente des Vektors, kapseln keine einzelne Assembler-Instruktion, sondern kapseln eine Sequenz von Assembler Instruktionen [41].

Intrinsics zum Laden und Speichern der Vektordaten bekommen als einen der Parameter einen Zeiger auf einen Speicherbereich übergeben, in dem skalare Daten abgelegt sind. Die skalaren Daten müssen dabei unterschiedliche Eigenschaften aufweisen, um mit den entsprechenden Ladeoperationen genutzt werden zu können:

- Die einfachste Variante Daten zu laden ist gegeben, wenn die skalaren Daten direkt hintereinander im Speicher liegen. Diese können dann beispielsweise mit dem loadu Intrinsic geladen werden.
- Beginnen die Daten zusätzlich an einer ausgerichteten (*engl. aligned*) Speicherstelle kann die Operation mit dem load Intrinsic effizienter durchgeführt werden. Die Bedeutung von ausgerichteten Speicherstellen und deren Verwendung in der Vektorisierung ist in Abschn. A.2 erklärt.
- Verteilt im Speicher liegende Daten können seit der AVX2-Erweiterung mit der gather Instruktion geladen werden. Hierzu wird ein Vektor mit Indexwerten an die Ladeoperation übergeben, bei dem jeder Index eine zu ladende Speicherposition angibt.
- Soll nur ein Teil der Elemente eines Vektors geladen werden, bietet die maskload Instruktion eine Möglichkeit die zu ladenden Elemente über eine so genannte *Maske* anzugeben.

Die entsprechenden Speicheroperationen storeu, store und maskstore verhalten sich analog. Für die gather Instruktion gibt es jedoch keine entsprechende Speicheroperation. Die in dieser Arbeit genutzten Intrinsics für Lade- und Speicheroperationen sind in Abschn. A.5 beschrieben. Weitere Intrinsics zum Laden und Speichern von Daten sind in [41] aufgelistet.

2.1.4. Automatische Vektorisierung durch den Compiler

Moderne Compiler sind in der Lage seriellen (nicht vektorisierten) Programmcode durch verschiedene Verfahren in vektorisierte Programme zu transformieren [3]. Bei diesen Verfahren werden durch die Datenabhängigkeitsanalysen des Compilers die Möglichkeiten zur Vektorisierung geprüft und die jeweiligen Instruktionen in Vektorinstruktionen umgewandelt. Zwei Beispiele für seriellen Programmcode, der durch den Compiler vektorisiert werden kann, finden sich bei der Annotation von seriellem Programmcode oder der Nutzung der Intel Cilk Array-Notation.

Annotiertes serielles C-Programm

Der Compiler kann serielle Programme automatisch in (teil-)vektorisierte Programme umwandeln, wofür der Programmcode nicht verändert werden muss. Durch die automatische Umwandlung werden jedoch nicht immer die gleichen Performanceverbesserungen erreicht, die durch eine manuelle Vektorisierung erreicht werden können [71]. Um eine bessere Vektorisierung von seriellen Programmen zu erreichen, können beispielsweise folgende Änderungen vorgenommen werden:

- Es können Datenstrukturen, wie beispielsweise Arrays, so programmiert werden, dass diese sich besser zur Vektorisierung eignen [49].
- Zusätzliche Informationen, zum Beispiel in der Form von `#pragma`s, können in den Programmcode eingebracht werden, um dem Compiler eine bessere Analyse zu ermöglichen [38].

Um solche Änderungen von seriellen Programmen zur automatischen Vektorisierung durchführen zu können, ist oft ein ausreichendes Verständnis von vektorisierten Programmen erforderlich [49].

Unterstützte Vektorisierung durch die Cilk Plus Array-Notation

Eine alternative Schreibweise der Berechnung von Array-Elementen kann dem Compiler ebenfalls die automatische Vektorisierung ermöglichen. Hierzu stellt Intel Cilk Plus die Array-Notation zur Beschreibung der jeweiligen Berechnungen von mehreren Elementen eines Arrays zu Verfügung. Dazu wird der Zugriff auf die einzelnen Elemente eines Arrays wie folgt dargestellt:

$$\text{Array[Startindex:Anzahl:Schrittweite]}$$

Die Array-Notation bestimmt damit die einzelnen Elemente eines Arrays, die für die Berechnung genutzt werden sollen. Die Angabe der Schrittweite kann entfallen, wenn die Schrittweite den Wert 1 hat. Eine Addition von zwei Arrays A und B der Länge 5 kann somit in der folgenden Form dargestellt werden:

$$\text{C[0:5]} = \text{A[0:5]} + \text{B[0:5]}$$

Bei der Nutzung skalarer Werte in einem Statement in Array-Notation wird dieser Wert für alle Elemente des angegeben Arrays genutzt.

Die Array-Notation enthält, anders als der elementweise Zugriff innerhalb einer Schleife, keine Abhängigkeiten der Reihenfolge von Elementzugriffen (siehe auch *forall-Schleife* und *Vektoranweisung* in [63]). Durch diese geänderte Semantik, gegenüber einer `for` Schleife, entfallen Einschränkungen und Datenabhängigkeiten bei der automatischen Vektorisierung.

Der Cilk Plus Programmcode wird dabei durch den Compiler in der gleichen Art und Weise automatisch vektorisiert, kann jedoch zusätzliche oder andere Programmtransformationen vornehmen. Die Unterstützung der Intel Cilk Plus Array-Notation in Intel Compilern ist seit 2018 als *Depricated* gekennzeichnet und wird nicht weiter entwickelt [34].

2.2. Energie-Effizientes Rechnen

Die Zielsetzung des energie-effizienten Rechnens bzw. der energie-effizienten Ausführung von Programmen und Algorithmen ist ein zusätzliches Ziel zur Optimierung von Programmen. Ein Grund für energie-effiziente Optimierung ist, dass moderne Computersysteme in ihrer Funktionsweise zunehmend durch physikalische Grenzen eingeschränkt werden. Einige dieser Grenzen werden direkt oder indirekt durch den Energieverbrauch des Systems beeinflusst.

2.2.1. Bedarf für energie-effiziente Programme

Eines der Hauptprobleme bei der Nutzung von Rechnern ist die erzeugte Wärme der Halbleiterelemente, vor allem der Prozessoren. Im Speziellen lässt sich für Prozessoren ein Wärme-Budget angeben, welches beschreibt wie viel Wärme innerhalb einer bestimmten Zeit abgeführt werden kann. Die erzeugte Wärme wird durch die physikalische Umwandlung aus elektrischer Energie erzeugt und lässt sich direkt auf die zugeführte Energie übertragen. Auf diese Weise kann die Leistungsaufnahme bei einer maximalen theoretischen Auslastung des Prozessors (*engl. auch Thermal Desgin Power (TDP)*) [37] angegeben werden. Die tatsächliche Leistungsaufnahme des Prozessors kann dabei kurzzeitig über der angegeben TDP liegen. Wird die TDP zu lange überschritten, werden Teile des Prozessors abgeschaltet, um Schäden zu vermeiden. Diese Abschaltung von Prozessorteilen wird als „Dark Silicon“ [21] bezeichnet und reduziert damit die theoretisch mögliche Performance eines Prozessors. Eine energie-effiziente Programmausführung kann somit die Leistungsfähigkeit eines Prozessors steigern, da mehr Prozessorteile für längere Zeit genutzt werden können und keine Teilabschaltung erfolgen muss.

Die TDP wird vom Prozessorhersteller unter anderem dazu angegeben, die nötige maximale Kühlleistung für den Prozessor zu bestimmen. Die Kühlung von Rechnersystemen stellt im Bereich von Rechenzentren einen erheblichen Faktor der Betriebskosten dar. Eine Senkung der erzeugten Abwärme der Computersysteme durch energie-effizientere Programmausführung kann in diesem Bereich zusätzliche Kosteneinsparungen (neben den eingesparten Energiekosten) bei der Kühlung erlauben. Bei mobilen Plattformen, die oft auf eine passive Kühlung angewiesen sind, kann die reduzierte Abwärme eine erhöhte Leistung ermöglichen.

Ein zusätzlicher Faktor ist die Menge an Energie, die zur Verfügung steht. Bei einem tragbaren Gerät limitiert die Kapazität der Batterie die Einsatzdauer und den Funktionsumfang des Gerätes. Wird ein solches Gerät mit energiesparenden Programmen genutzt, erhöht sich die Nutzungsdauer und damit die Einsatzmöglichkeit des Geräts. Auch die Menge an Energie, die zu einem bestimmten Zeitpunkt zur Verfügung steht, kann dabei begrenzt sein. Im Fall von Rechenzentren kann beispielsweise die Dimensionierung der Stromversorgung nur einen gewissen Energieverbrauch erlauben. Auch bei batteriebetriebenen Geräten lässt sich die Kapazität der Batterie nur über eine gewisse Zeit abrufen, da chemische Prozesse im Inneren der Batterie die Abgabemenge begrenzen. Die Kombination der genannten Punkte führt zu einem Bedarf

an Optimierungsstrategien für energie-effiziente Programmausführungen.

2.2.2. Begriffsdefinitionen zur energie-effizienten Programmoptimierung

Die gezeigten Beispiele demonstrieren die Notwendigkeit für die energie-effiziente Ausführung von Programmen. Für die Zielsetzung der energie-effizienten Programmierung und Optimierung werden die folgenden Grundbegriffe genutzt:

Energie, auch Energieverbrauch, bezeichnet die aufgenommene Menge elektrischer Energie. Sie wird in Joule (J) oder Kilowattstunden ($\mathrm{kW\,h} = \mathrm{kJ\,h\,s^{-1}}$) angegeben.

Leistung, auch Leistungsaufnahme, gibt die in einem Zeitraum aufgenommene Menge Energie an. Die Leistung wird in Watt ($\mathrm{W} = \mathrm{J\,s^{-1}}$) angegeben. Der Zeitraum über den die Energie aufgenommen wird, ist üblicherweise nicht angegeben.

Laufzeit, Programmlaufzeit oder Ausführungszeit beschreibt die Zeit, die für die Ausführung eines Programms benötigt wird. Je nach Betrachtung kann dieser Wert für das gesamte Programm oder nur einen Teil des Programms genutzt werden.

Der Begriff Energie-Effizienz wird als Sammelbegriff für verschiedene Optimierungen im Zusammenhang mit Energie verwendet. Das spezielle Ziel der Optimierung muss entsprechend der Anforderungen spezifiziert werden. Die Ziele der Optimierung von Programmen können beispielsweise in die folgenden Gruppen eingeteilt werden:

Performance: Ein Programm soll möglichst schnell abgearbeitet sein. Oft bedeutet dies die Ausnutzung aller zur Verfügung stehenden Ressourcen. Die Performance wird als klassisches Optimierungsziel und als Gegensatz zu energiesparenden Optimierungen genutzt.

Energiesparend: Die Ausführung eines Programms soll möglichst wenig Energie verbrauchen. Eine verlängerte Programmlaufzeit wird hierbei in Kauf genommen.

Leistungsbegrenzt: Bei der Ausführung des Programms darf (ggf. im Mittel) zu einem bestimmten Zeitpunkt nicht mehr Leistung verbraucht werden, als vorgegeben ist. Hierbei handelt es sich oft um Umgebungen, in denen die Stromzufuhr begrenzt ist.

Einzelziel-Optimierung mit Grenzparametern: Die Optimierung in Bezug auf die genannten Ziele kann für die anderen Parameter zu einer Verschlechterung führen. Um eine Verbesserung nicht durch untragbaren Aufwand zu erreichen, können für die nicht optimierten Ziele bestimmte Grenzparameter gesetzt werden, wie z.B. dass sich bei energiesparender Ausführung die Laufzeit nur um einen bestimmten Faktor verschlechtern darf.

Multiziel-Optimierung: In der Realität treten die genannten Zielsetzungen oft nicht einzeln auf. Hierzu kann beispielsweise eine Kostenfunktion definiert werden, bei der in die Bewertung des Ergebnisses mehrere Ziele einfließen.

2.2.3. Metriken für Energie-Effizienz

Bei der Optimierung von Programmen im Hinblick auf Energie-Effizienz werden neben der Programmlaufzeit verschiedene Metriken zur Bewertung der Energie-Effizienz genutzt. Die genutzte Metrik soll dabei das Optimierungsziel widerspiegeln und aussagekräftig bezüglich des Grads der Veränderung sein. Oft verwendet werden hierfür die folgenden Begriffe und Kostenfunktionen:

Energie, Energieverbrauch, wird in der physikalischen Einheit Joule (J) oder Kilo Watt Stunden (kWh) angegeben.

Leistungsaufnahme: Für die Leistungsaufnahme wird meist der Energieverbrauch über einen Zeitraum gemessen und im Anschluss durch die vergangene Zeit geteilt $Leistung = \frac{Energie}{Zeit}$. Die Leistungsaufnahme wird in der physikalischen Einheit Watt (W) angegeben.

Energy Delay Product (EDP) bzw. Kostenfunktionen Et^n aus Energie E, Zeit t und Gewichtung n (z.B. [7, 25, 69]). Mit dem Energy Delay Product wird eine Multiziel-Optimierung angestrebt, bei der die Reduktion des Energieverbrauchs nur dann als Verbesserung angesehen wird, sofern die gewichtete Laufzeit dabei nicht überproportional ansteigt.

Energy Delay Sum (EDS) und Energy Delay Distance (EDD) (beide [69]). Vergleichbar mit dem EDP setzen diese Kostenfunktionen Energie E und Laufzeit t in eine Kostenfunktion zur Multiziel-Optimierung um. Die Energy Delay Sum wird mittels $EDS = \alpha E + \beta t$ und die Energy Delay Distance mittels $EDD = \sqrt{E^2 + (\beta t)^2}$ berechnet, wobei $\alpha, \beta \in \mathbb{R}$ zur Gewichtung verwendet werden.

Energy Speedup (ES) und Power Increase Faktor (PI) (beide [67]). Vergleichbar mit dem Speedup zur Bewertung des Verhältnisses von Ausführungszeiten bei paralleler und sequentieller Ausführung, bewertet der Energy Speedup das Verhältnis des Energieverbrauchs bei sequentieller und paralleler Ausführung. Analog wird dies für die Leistungsaufnahme in einem Power Increase Faktor definiert. Weitere Metriken wie Energy per Speedup werden in [67] vorgestellt, Analysiert und deren Einsatz diskutiert.

Instruktionen und Daten pro Energie, wie $FLOP/J$, $FLOPS/W$, $Byte/J$ (z.B. [25, 26, 32]). Diese Kostenfunktionen sollen zeigen wie effektiv ein System oder Algorithmus Energie in ausgeführte Instruktionen bzw. verarbeitete Datenelemente umsetzt.

Die genannten Metriken bieten verschiedene Vor- und Nachteile. Gonzalez und Horowitz führen in [25] das Energy Delay Product zur Vergleichbarkeit der Energie-Effizienz verschiedener Prozessoren ein. Sie argumentieren, dass die Leistungsaufnahme für diesen Zweck nicht geeignet ist, da eine niedrigere Frequenz grundsätzlich zu einer niedrigeren Leistungsaufnahme führt. Auf die gleiche Weise hängt auch der Energieverbauch pro ausgeführter Instruktion hauptsächlich von der angelegten Versorgungsspannung ab. Das EDP gleicht diese Abhängigkeit durch die Mitbetrachtung der Ausführungszeit aus.

Bekas und Curioni zeigen in [7], dass die Verwendung von FLOPS/W eine Reduzierung der Leistung pro Operation begünstigt, jedoch keine Aussage über die Laufzeit bzw. Anzahl der Operationen gibt. Aus diesem Grund schlagen sie eine $f(t) \cdot E$-Metrik vor, bei der die Funktion $f(t)$ je nach Algorithmus und Anwendungsfall spezifiziert werden muss. Für den Fall $f(t) = t$ entspricht die vorgeschlagene Metrik dem EDP.

Roberts et al. argumentieren in [69], dass die Nutzung von Energie, Leistung und Laufzeit die Auswahl der optimalen Programmvariante dem Endnutzer überlässt. Sie zeigen weiter, dass sich instruktionsbasierte Metriken nicht eignen, da die realen Kosten der Programmausführung nicht betrachtet werden. Außerdem argumentieren sie, dass Et^n-Metriken nicht zur Bewertung der energie-effizienten Ausführung von Programmen herangezogen werden dürfen. Die Autoren führen hierzu eine Reihe von Kriterien an, die eine Kostenfunktion erfüllen soll und zeigen, dass Et^n-Metriken nur eines dieser Kriterien erfüllen. Zu einem ähnlichen Schluss kommen auch Sazeides et al. in [72] und Hsu, Feng und Archuleta in [32]. Sazeides et al. zeigen, dass die Optimierung eines Programmteils die Parameter der Kostenfunktion verbessern kann, dabei jedoch ein schlechterer Wert durch das EDP angezeigt wird. Hsu, Feng und Archuleta gehen in Kombination mit FLOPS^n/W Metriken darauf ein, dass die Ergebnisse dieser Kostenfunktionen stark parallele Systeme und Ausführungen bevorteilen.

Zur Bewertung von Programmausführungen schlagen Roberts et al. in [69] zwei neue Kostenfunktionen vor. Die Energy Delay Sum (EDS) soll gegenüber dem bekannten Energy Delay Product die benötigten Eigenschaften zum Vergleich von Programmen bieten. Diese Eigenschaften umfassen dabei unter anderem die Abgegrenztheit (*engl. bounded*), Stabilität (*engl. stable*) und Additivität (*engl. additive*) der erhaltenen Kosten, wodurch gleiche Optimierungen bei unterschiedlichen Ausgangswerten dennoch eine gleich große Änderung der Kosten erzeugen. Roberts et al. fordern außerdem von einer Kostenfunktion, dass sie den Anwender in Richtung der besten Optimierung führt (*engl. directed*). Sie stellen dabei selbst fest, dass sich die *directed* Eigenschaft und die *additive* Eigenschaft widersprechen, weshalb sie zwei Metriken für unterschiedliche Zwecke vorschlagen: Die Energy Delay Sum zum Vergleich von Programmen und die Energy Delay Distance für die Auswahl der nächsten Optimierungsschritte. Zur Berechnung der EDS und der EDD werden zusätzlich Parameter zur Skalierung und Gewichtung der einzelnen Faktoren eingeführt, die jedoch nicht näher bestimmt werden.

Die gezeigten Metriken haben damit jeweils entsprechende Vor- und Nachteile, die sie für den Einsatz in bestimmten Situationen passend machen. Die Auswahl der passenden Metrik muss somit der Endanwender oder Autor einer Studie wählen.

3 AVX-Vektorisierung der Matrix-Multiplikation

Die Matrix-Multiplikation (auch Matrix-Matrix-Multiplikation) wird in verschiedenen Anwendungen des wissenschaftlichen Rechnens als Unterfunktion genutzt. Oft lassen sich komplexere Algorithmen der linearen Algebra mit Hilfe von Matrix-Multiplikationen lösen, wie z.B. die QR-Zerlegung mit Hilfe von Householder-Transformationen [24].

Die Matrix-Multiplikation gehört zu den elementaren Routinen in der linearen Algebra (vgl. BLAS - Basic Linear Algebra Subprograms [9]). Verschiedene Algorithmen und Programmtransformationen zur Matrix-Multiplikation werden beispielsweise in [20] für Vektorprozessoren, in [50] für einen CELL SIMD Prozessor oder in [1] für einen Pentium 3 Prozessor mit SSE-Erweiterung untersucht. In [28] werden die Ausführungszeiten von AVX-Implementierungen der Matrix-Multiplikation bei der Nutzung verschiedener Compiler, der Nutzung von OpenMP-Parallelisierung und Programmtransformation, wie *loop unrolling*, verglichen. Verschiedene Bibliotheken mit Routinen der linearen Algebra, wie Intels MKL [39], ATALS [16] oder LAPACK [19], bieten für die Matrix-Multiplikation performance-optimierte Implementierungen an. Spezielle Algorithmen zur Reduzierung der Anzahl ausgeführter Operationen bei der Multiplikation von Matrizen wurden beispielsweise von Strassen in [75] oder von Winograd in [80] vorgestellt.

Die stetige Forschung an Algorithmen zur Matrix-Multiplikation spiegelt deren breites Einsatzgebiet und deren Bedeutung im wissenschaftlichen Rechnen wieder. Durch die Verwendung als Unterfunktion trägt eine effiziente Implementierung der Matrix-Multiplikation zur Optimierung verschiedener Anwendungen bei.

In diesem Kapitel werden verschiedene Verfahren zur Vektorisierung und der Einfluss von Programmtransformationen auf die vektorisierte Matrix-Multiplikation untersucht. Teile der Untersuchungen wurden bereits in [43] veröffentlicht.

3.1. Algorithmus zur Matrix-Multiplikation

Die Matrix-Multiplikation ist ein Algorithmus zur Multiplikation von zwei Matrizen A und B nach der folgenden Formel:

$$C = A \cdot B \quad \text{mit} \quad A \in \mathbb{R}^{l \times m}, B \in \mathbb{R}^{m \times n} \text{ und } C \in \mathbb{R}^{l \times n} \tag{3.1}$$

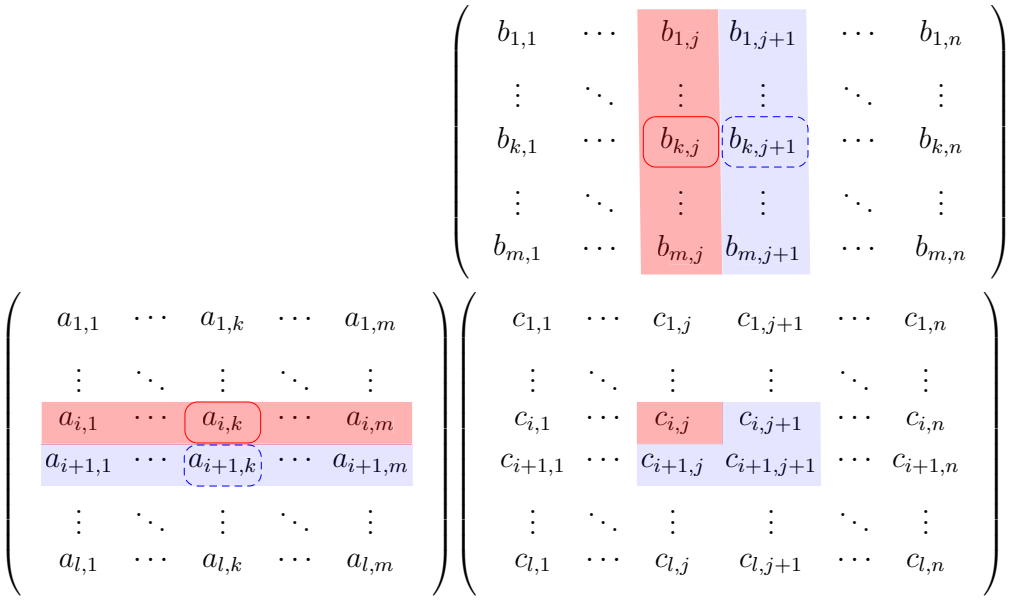

Abbildung 3.1.: Illustration des Bearbeitungsschemas der Matrix-Multiplikation in der *MMScal* Programmvariante. Zur Berechnung des Elements $c_{i,j}$ werden die Elemente der Zeile $a_{i,\_}$ und der Spalte $b_{\_,j}$ (rot hinterlegt) jeweils paarweise ($a_{i,k} \cdot b_{k,j}$, rote Umrandung) genutzt. Die blau hinterlegten Elemente werden bei der Anwendung der Schleifen-Blockzerlegung (Blockgröße = 2) in der *MMScalTile* Programmvariante zusätzlich genutzt.

Zur Berechnung eines Elements $c_{i,j}$ der Matrix C werden die Elemente der Zeile $a_{i,\_}$ der Matrix A und der Spalte $b_{\_,j}$ der Matrix B genutzt. Dazu wird das Skalarprodukt von $a_{i,\_}$ und $b_{\_,j}$ wie folgt berechnet:

$$\begin{aligned} c_{i,j} &= a_{i,\_} \cdot b_{\_,j} \\ &= \sum_{k=1}^{m} a_{i,k} \cdot b_{k,j}; \\ &\quad \text{mit } 1 \leq i \leq l; 1 \leq j \leq n \end{aligned} \tag{3.2}$$

Abbildung 3.1 zeigt die genutzten Elemente der Matrizen A und B (rot) zur Berechnung des Elements $c_{i,j}$.

3.1.1. Implementierung der Matrix-Multiplikation

Für jedes Element $c_{i,j}$ der Ergebnismatrix C wird nach Gleichung 3.2 das Skalarprodukt von Zeile $a_{i,\_}$ und spalte $b_{\_,j}$ berechnet. Algorithmus 3.1 zeigt dafür die Implementierung der seriellen Programmvariante **MMScal** zur Berechnung der Matrix-Multiplikation durch ein Schleifennest mit den folgenden drei Schleifen:

- Die *i-Schleife* (Zeile 2) iteriert über die Zeilen $a_{i,\_}$ der Matrix A und die Zeilen $c_{i,\_}$ der Matrix C.

```
1 // Schleife über die Zeilen a_{i,_}:
2 for i = 0 < l; i+=1
3   // Schleife über die Spalten b_{_,j}:
4   for j = 0 < n; j+=1
5     C[i*n+j] = 0
6     // Schleife über die Elemente der Zeile a_{i,_} und der Spalte b_{_,j}
7     for k = 0 < m; k+=1
8       C[i*n+j] += A[i*m+k] * B[k*n+j] // c_{i,j} = a_{i,_} · b_{_,j}
```

Algorithmus 3.1: MMScal: Algorithmus der Matrix-Multiplikation nach [24], Notation angepasst.

- Die *j-Schleife* (Zeile 4) iteriert über die Spalten $b_{\_,j}$ der Matrix B und der Elemente $c_{i,j}$ der Matrix C.
- Die *k-Schleife* in Zeile 7 iteriert über die Elemente $a_{i,k}$ der Zeile $a_{i,\_}$ und die Elemente $b_{k,j}$ der Spalte $b_{\_,j}$.

Innerhalb dieses Schleifennestes (Zeile 8) wird das Produkt der Elemente $a_{i,k}$ und $b_{k,j}$ (in Abb. 3.1 rote Umrandung) gebildet und auf das Zwischenergebnis für das Element $c_{i,j}$ aufsummiert. Nach Ausführung der inneren Schleife (*k-Schleife*) enthält $c_{i,j}$ das Skalarprodukt der Zeile $a_{i,\_}$ und der Spalte $b_{\_,j}$. Die Elemente der Matrix C werden somit jeweils einzeln und zeilenweise berechnet.

Schleifen-Blockzerlegung der Matrix-Multiplikation

Eine der wichtigsten Optimierungen für Matrix-Multiplikationen ist nach [60] die Schleifen-Blockzerlegung (*engl. loop tiling*). Die Schleifen-Blockzerlegung verringert dabei die Zeitspanne zwischen Datenzugriffen auf die Daten des selben Blockes und erhöht somit die zeitliche Lokalität der Speicherzugriffe (vgl. [63]). In Abb. 3.1 wird die Anwendung der Schleifen-Blockzerlegung mit einer Blockgröße tile_a=tile_b=2 auf die Matrix-Multiplikation dargestellt. Durch die Schleifen-Blockzerlegung werden mehrere Elemente der Ergebnismatrix C innerhalb eines Blockes berechnet. Dazu werden zusätzliche Zeilen und Spalten der Matrizen A und B (in Abb. 3.1 blau) für den jeweiligen Block genutzt. Bei der Matrix-Multiplikation werden in einem Block der Schleifen-Blockzerlegung mit den Blockgrößen tile_a und tile_b die folgenden Elemente der Ergebnismatrix C berechnet:

$$c_{i,j}, ..., c_{i,j+tile\_b}, c_{i+1,j}, ..., c_{i+1,j+tile\_b}, ..., c_{i+tile\_a,j+tile\_b}$$

Algorithmus 3.2 zeigt die Implementierung der Schleifen-Blockzerlegung für beliebige Blockgrößen in der **MMScalTile** Programmvariante. Dazu werden die Iterationsvariablen der äußeren beiden Schleifen des Schleifennestes (Zeile 2 und 4) jeweils um den Faktor der Blockgröße (tile_a bzw. tile_b) inkrementiert. Die auf diese Weise übersprungenen Iterationen der Blöcke werden durch zusätzliche Schleifen innerhalb

```
1  // Schleife über die Blöcke der Zeilen a_{i,_}:
2  for i = 0 < l; i+=tile_a
3    // Schleife über die Blöcke der Spalten b_{_,j}:
4    for j = 0 < n; j+=tile_b
5      // Schleife über die Zeilen a_{i,_} innerhalb der Blöcke:
6      for it = i < i+tile_a; it+=1
7        // Schleife über die Spalten b_{_,j} innerhalb der Blöcke:
8        for jt = j < j+tile_b; jt +=1
9          C[it*n+jt] = 0
10     // Schleife über die Elemente der Zeile a_{i,_} und der Spalte b_{_,j}
11     for k = 0 < m; k+=1
12       // Schleife über die Zeilen a_{i,_} innerhalb der Blöcke:
13       for it = i < i+tile_a; it+=1
14         // Schleife über die Spalten b_{_,j} innerhalb der Blöcke:
15         for jt = j < j+tile_b; jt +=1
16           C[it*n+jt] += A[it*m+k] * B[k*n+jt] // c_{it,jt} = a_{it,_} · b_{_,jt}
```

Algorithmus 3.2: MMScalTile Programmvariante: Implementierung der Matrix-Multiplikation mit Anwendung von Schleifen-Blockzerlegung mit den Blockgrößen tile_a für die Anzahl der Zeilen von Matrix A und tile_b für die Anzahl der Spalten von Matrix B. Veränderte Zeilen im Vergleich zur *MMScal* Programmvariante sind hervorgehoben.

des Schleifennestes ausgeführt. Dazu werden in den Zeilen 6 und 8 zwei Schleifen eingefügt, die alle Elemente des Blockes der Matrix C mit Null initialisieren. Analog dazu werden innerhalb der *k-Schleife* (Zeile 11) zwei Schleifen eingefügt, die für alle Elemente des Blockes der Matrix C die Multiplikation und Addition durchführen. Für jede Iteration der *k-Schleife* wird dabei jeweils das nächste Zwischenergebnis aller Elemente des Blockes der Matrix C bestimmt, sodass nach der Ausführung der *k-Schleife* das Skalarprodukt jedes dieser Elemente berechnet ist.

3.1.2. Eigenschaften der Matrix-Multiplikation im Hinblick auf eine Vektorisierung

Die lesenden Datenzugriffe der Matrix-Multiplikation können ohne Abhängigkeiten und damit parallel durchgeführt werden, wodurch die Vektorisierung auf verschiedene Weisen erfolgen kann. Datenabhängigkeiten entstehen dabei ausschließlich bei der Summenbildung in der innersten Schleife des Schleifennestes. Diese Datenabhängigkeit bietet dabei die Möglichkeit die Nutzung von vektorisierten Reduktionen (vgl. Anhang Abschn. A.4) zu untersuchen.

Der Algorithmus zur Matrix-Multiplikation erlaubt die Untersuchung des Einflusses der Schleifen-Blockzerlegung auf die verschiedenen Vektorisierungsansätze. Die Struktur der Matrix-Multiplikation eignet sich insbesondere für die Untersuchung der Kombination von Schleifen-Blockzerlegung und automatischer Vektorisierung.

Die Speicherzugriffe bei der Matrix-Multiplikation erlauben es, den Zugriff auf aus-

Ansatz	Programmvariante	Programmtransformation	Referenz
—	**MMScal**	keine	Alg. 3.1; Abb. 3.1
	MMScalTile	Schleifen-Blockzerlegung	Alg. 3.2; Abb. 3.1
automatisch	**MMAuto**	Automatisch durch Compiler vektorisiert	S. 18; Alg. 3.3
	MMAutoTile	Automatisch durch Compiler vektorisiert mit Schleifen-Blockzerlegung	S. 18
	MMCilk	Vektorisierung mittels Cilk Array-Notation	S. 19; Alg. 3.4
	MMCilkTile	Vektorisierung mittels Cilk Array-Notation mit Schleifen-Blockzerlegung	S. 20; Alg. 3.5
AVX Intrinsics; spaltenweiser Zugriff auf B	**MMAvxGatherIdx**	Spaltenweiser Zugriff auf Matrix B mittels Gather-Operation mit Index-Vektor	S. 21; Alg. 3.6
	MMAvxGatherVdx	Spaltenweiser Zugriff auf Matrix B mittels Gather-Operation mit vektorisierter Index-Berechnung	S. 22; Alg. 3.7
	MMAvxGatherBase	Spaltenweiser Zugriff auf Matrix B mittels Gather-Operation mit Offset-Vektor	S. 23
	MMAvx-Vred[1,2,4,8,16,i]	Teilweise serielle Berechnung der Reduktionsoperation	S. 23
AVX Intrinsics; zeilenweiser Zugriff auf B	**MMAvxRow**	Zeilenweiser Zugriff auf Matrix B	S. 24; Alg. 3.8; Abb. 3.2
	MMAvxTile	Zeilenweiser Zugriff auf Matrix B mit Schleifen-Blockzerlegung	S. 25; Alg. 3.9; Abb. 3.2
	MMAvxStream	Nutzung von non-temporal Schreiboperationen	S. 26

Tabelle 3.1.: Übersicht über die Programmvarianten der Matrix-Multiplikation. Die angewendeten Programmtransformationen werden kurz zusammengefasst, die detaillierte Beschreibung der Programmvariante ist referenziert.

gerichtete Speicherstellen (vgl. Anhang Abschn. A.2) durch geschickte Wahl der Matrixgrößen auch ohne die Anwendung von Techniken wie *loop splitting* (vgl. Anhang Abschn. A.3) durchzuführen. Dadurch wird die Anzahl der Sprünge und seriellen Berechnungen im Programm reduziert.

```
1 // Schleife über die Zeilen a_{i,_}:
2 for i = 0 < l; i+=1
3  #pragma simd
4    // Schleife über die Spalten b_{_,j}:
5   for j = 0 < n; j+=1
6     C[i*n+j] = 0
7     // Schleife über die Elemente der Zeile a_{i,_} und der Spalte b_{_,j}
8     for k = 0 < m; k+=1
9       C[i*n+j] += A[i*m+k] * B[k*n+j] // c_{i,j} = a_{i,_} · b_{_,j}
```

Algorithmus 3.3: MMAuto: Automatische Vektorisierung der Matrix-Multiplikation. Veränderte Zeilen im Vergleich zur *MMScal* Programmvariante in Alg. 3.1 sind hervorgehoben.

3.2. Vektorisierung der Matrix-Multiplikation

Die Vektorisierung der Matrix-Multiplikation kann auf verschiedene Weisen erfolgen: Automatische Vektorisierung durch den Compiler oder Programmierung mit Hilfe von AVX-Intrinsics. In diesem Abschnitt werden die vektorisierten Programmvarianten der Matrix-Multiplikation und deren Modifikationen vorgestellt.

Tabelle 3.1 zeigt die Programmvarianten der Matrix-Multiplikation, sowie eine kurze Zusammenfassung der Implementierungsaspekte.

3.2.1. Automatisch vektorisierte Programmvarianten der Matrix-Multiplikation

Bei der automatischen Vektorisierung werden durch den Compiler verschiedene Prüfungen auf Datenabhängigkeiten und Programmtransformationen durchgeführt [3]. Die erreichte Qualität der Vektorisierung erreicht dabei oft nicht die Qualität einer manuellen Vektorisierung mit Intrinsics [49]. Zur Untersuchung der Qualität der automatischen Vektorisierung werden daher verschiedene Ansätze zur Formulierung des seriellen Programms auf Basis der Angabe der zu vektorisierenden Schleife und der Intel Cilk Array-Notation untersucht.

3.2.1.1. *MMAuto* Programmvarianten durch automatische Vektorisierung

Für die automatisch vektorisierten Programmvarianten werden die gezeigten seriellen Programmvarianten mit Hilfe eines `#pragma` simd durch den Compiler vektorisiert.

Algorithmus 3.3 zeigt die **MMAuto** Programmvariante. Dabei wird die Vektorisierung über Spalten der Matrix B durchgeführt und damit mehrere Elemente der Matrix C gleichzeitig berechnet. Die zu vektorisierende Schleife (*j-Schleife*) wird dazu durch ein `#pragma` simd (Zeile 3) gekennzeichnet. Die Vektorisierung der inneren Schleife (*k-Schleife*) wird vom Compiler nicht automatisch durchgeführt, da zwischen den einzelnen Iterationen Datenabhängigkeiten bestehen.

```
1 // Schleife über die Zeilen a_{i,_}:
2 for i = 0 < l; i+=1
3   // Schleife über die Spalten b_{_,j}:
4   for j = 0 < n; j+=1
5     C[i*n+j] = __sec_reduce_add(A[i*m:m] * B[j:m:n]) // c_{i,j} = a_{i,_} · b_{_,j}
```

Algorithmus 3.4: MMCilk Programmvariante: Die innere Schleife des Schleifennestes wird durch eine spezielle Cilk Reduktionsfunktion und die Nutzung der Cilk Array-Notation erzetzt. Veränderte Zeilen im Vergleich zur *MMScal* Programmvariante in Alg. 3.1 sind hervorgehoben.

Analog dazu wird die **MMAutoTile** Programmvariante erstellt, indem in der *MMScalTile* Programmvariante aus Alg. 3.2 ein `#pragma simd` vor der *j-Schleife* (Zeile 4) eingefügt wird.

3.2.1.2. Programmvarianten in der Intel Cilk Array-Notation

Bei der Intel Cilk Array-Notation wird die Iteration über ein Array mit einer Schleife durch ein Statement in Array-Notation ersetzt. Dieses Statement gibt die Vorschriften zur Berechnung für jedes Element des Arrays an, wobei eine explizite Reihenfolge der Berechnungen nicht angegeben wird. Die Array-Notation setzt sich dabei wie folgt aus dem Index des ersten Elements, der Anzahl der Elemente und der Schrittweite zwischen zwei Elementen in der Datenstruktur zusammen:

A[Startindex:Anzahl_Elemente:Schrittweite]

Durch die entfallende Reihenfolge bei der Nutzung der Array-Elemente werden mögliche Datenabhängigkeiten entfernt und dem Compiler somit die automatische Vektorisierung erleichtert.

MMCilk: Array-Notation der innersten Schleife

Algorithmus 3.4 zeigt die Implementierung der **MMCilk** Programmvariante. Die äußeren Schleifen werden dabei gegenüber der seriellen *MMScal* Programmvariante nicht verändert. Die innerste Schleife (*k-Schliefe*) wird in der **MMCilk** Programmvariante durch ein Statement in Array-Notation und eine Reduktion ersetzt (Zeile 5). Hierzu wird eine Zeile $a_{i,\_}$ der Matrix A (ausgedrückt durch `a[i*m:m]`) mit einer Spalte $b_{\_,j}$ der Matrix B (ausgedrückt durch `b[j:m:n]`) elementweise multipliziert. Die Elemente des entstehenden Ergebnisvektors werden durch die eingebaute Cilk-Funktion `__sec_reduce_add()` zu einem skalaren Wert aufsummiert, der an die Stelle $c_{i,j}$ geschrieben wird.

```
1  // Schleife über die Blöcke der Zeilen a_{i,_}:
2  for i = 0 < l; i += tile_a
3    // Schleife über die Blöcke der Spalten b_{_,j}:
4    for j = 0 < n; j += tile_b
5      // Temporäres Array für die Elemente der Matrix C initialisieren :
6      sum[0:tile_a*tile_b] = 0
7      // Schleife über die Elemente der Zeile a_{i,_} und der Spalte b_{_,j}
8      for k = 0 < m; k+= 1
9        // Schleife über die Zeilen a_{i,_} innerhalb der Blöcke:
10       for it = i < i+tile_a; it += 1
11         // c_{it,jt} = a_{it,_} · b_{_,jt} für jedes Element des Blockes:
12         sum[(it-i)*tile_b:tile_b] += A[it*m+k] * B[k*n+j:tile_b]
13     // Schleife über die Zeilen a_{i,_} innerhalb der Blöcke:
14     for it = i < i+tile_a; i+=1
15       // Temporäres Array auf Elemente c_{it,jt} der Matrix C schreiben:
16       c[it*n+j:tile_b] = sum[(it-i)*tile_b:tile_b]
```

Algorithmus 3.5: MMCilkTile Programmvariante: Die innere Schleife des Schleifennestes der *MMScalTile* Programmvariante wird durch Statements in der Cilk Array-Notation ersetzt. Veränderte Zeilen im Vergleich zur *MMScalTile* Programmvariante in Alg. 3.2 sind hervorgehoben.

MMCilkTile: Anwendung der Array-Notation auf die Schleifen-Blockzerlegung

Die Implementierung von Programmvarianten mit Schleifen-Blockzerlegung kann mit der Cilk Erweiterung nicht aus der *MMCilk* Programmvariante abgeleitet werden. Stattdessen muss als Ausgangspunkt die serielle *MMScalTile* Programmvariante (Alg. 3.2) genutzt werden.

In Alg. 3.5 werden durch die Statements in Array-Notation die *jt-Schleifen* der *MMScalTile* Programmvariante (Zeilen 8 und 15 aus Alg. 3.2) umgeformt. In der **MMCilkTile** Programmvariante werden somit durch die Statements in Array-Notation mehrere Elemente der Matrix C berechnet. Für die Berechnung der Elemente wird in Zeile 6 ein temporäres Array mit der benötigten Anzahl an Elementen initialisiert.

Innerhalb der *k-Schleife* (Zeile 8) wird für jedes Element des temporären Arrays das Skalarprodukt aus den Zeilen $a_{it,\_}$ der Matrix A und den Spalten $b_{\_,jt}$ der Matrix B gebildet (Zeile 12). Das Array sum enthält nach einem Durchlauf der *it-Schleife* (Zeile 10) das Zwischenergebnis aller tile_a·tile_b Werte. Nach dem vollständigem Durchlauf der *k-Schleife* (Zeile 8) enthält sum das Gesamtergebnis dieser Werte. Diese Ergebnisse werden in *tile_a* Schritten (*it-Schleife* in Zeile 14) der Größe tile_b auf die entsprechende Position der Ergebnismatrix C kopiert (Zeile 16).

3.2.2. AVX-Implementierungen der Matrix-Multiplikation

Im Gegensatz zur automatischen Vektorisierung durch den Compiler, kann der Programmierer bei der Vektorisierung mit AVX-Intrinsics zwischen verschiedenen Imple-

```
1  // Schleife über die Zeilen a_{i,_}:
2  for i = 0 < l; i+=1
3    // Schleife über die Spalten b_{_,j}:
4    for j = 0 < n; j+=1
5      _c = _mm256_setzero_ps() // c_[] = 0,...,0
6      // Schleife über die Elemente der Zeile a_{i,_} und der Spalte b_{_,j}
7      for k = 0 < m; k += 8
8        _a = _mm256_loadu_ps(&A[i,k]) // a_[] = a_{i,k},...,a_{i,k+7}
9        // Setzen der Array-Indizes in ein Integer-Vektorregister:
10       _ind = _mm256_setr_epi32(k*n+j,(k+1)*n+j,...,(k+7)*n+j)
11       // b_[] = b_{k,j},...,b_{k+7,j} :
12       _b = _mm256_i32gather_ps(&B[0], _ind, sizeof(float))
13       _t = _mm256_mul_ps(_a, _b) // a_[] · b_[]
14       _c = _mm256_add_ps(_t, _c) // c_[] = c_[] + a_[] · b_[]
15     C[i*n+j] = addreduce_ps(_c) // c_{i,j} = a_{i,_} · b_{_,j}
```

Algorithmus 3.6: MMAvxGatherIdx Programmvariante: Die innere Schleife der Matrix-Multiplikation wird mit Hilfe von AVX-Intrinsics vektorisiert. Dazu wird ein Index-Vektor für den Zugriff auf die im Speicher verteilt abgelegten Elemente der Spalte $b_{\_,j}$ genutzt. Veränderte Zeilen im Vergleich zur *MMScal* Programmvariante in Alg. 3.1 sind hervorgehoben.

mentierungsvarianten wählen. In diesem Abschnitt werden die Programmvarianten der Vektorisierung mittels AVX-Intrinsics vorgestellt. Bei der Matrix-Multiplikation kann auf die Elemente der Matrix B spaltenweise (vgl. *MMScal*) oder zeilenweise (vgl. *MMScalTile*) zugegriffen werden.

3.2.2.1. AVX-Implementierungen mit spaltenweisem Zugriff auf Matrix B

Bei der Vektorisierung der inneren Schleife der Matrix-Multiplikation wird auf die Elemente der Matrix B spaltenweise zugegriffen. Für die Zugriffe auf die Spalten $b_{\_,j}$ der Matrix B gibt es dabei unterschiedliche Arten, wie die gather Instruktion für den Speicherzugriff auf verteilt abgelegte Daten genutzt werden kann. Zusätzlich kann die Summenbildung durch das Skalarprodukt genutzt werden, um den Einsatz der vektorisierten Reduktion zu untersuchen.

MMAvxGatherIdx: Spaltenweiser Zugriff mittels Index-Vektor

Bei der Vektorisierung der *MMScal* Programmvariante werden die Elemente der Spalte $b_{\_,j}$ der Matrix B genutzt. Aufgrund der zeilenweisen Speicherung von Arrays liegen die nacheinander gelesenen Elemente der Matrix B verteilt im Speicher und müssen zum Laden in eine Vektorvariable gesammelt werden. Die gather Instruktion der AVX2-Erweiterung bietet diese Funktionalität an, wozu die Speicherstelle des Arrays und ein Index-Vektor übergeben werden.

```
1  _n = _mm256_set1_epi32(n)  // n[] = n, ..., n
2  _cnt = _mm256_setr_epi32(0,1,...,7)  cnt[] = 0, ..., 7
3  // Schleife über die Zeilen a_{i,_}:
4  for i = 0 < l; i+=1
5    // Schleife über die Spalten b_{_,j}:
6    for j = 0 < n; j+=1
7      _c = _mm256_setzero_ps()  // c[] = 0, ..., 0
8      _j = _mm256_set1_epi32(j)  // j[] = j, ..., j
9      // Schleife über die Elemente der Zeile a_{i,_} und der Spalte b_{_,j}
10     for k = 0 < m; k += 8
11       _k = _mm256_set1_epi32(k)  // k[] = k, ..., k
12       _ind = _mm256_add_epi32(_k, _cnt)  // ind[] = k[] + cnt[]
13       _ind = _mm256_mul_epi32(_ind, _n)  // ind[] = (k[] + cnt[]) · n[]
14       _ind = _mm256_add_epi32(_ind, _j)  // ind[] = (k[] + cnt[]) · n[] + j[]
15       // b[] = b_{k,j}, ..., b_{k+7,j} :
16       _b = _mm256_i32gather_ps(&B[0], _ind, sizeof(float))
```

Algorithmus 3.7: MMAvxGatherVdx Programmvariante mit vektorisierter Berechnung der Index-Werte für die gather Instruktion. Veränderte Zeilen im Vergleich zur *MMAvxGatherIdx* Programmvariante in Alg. 3.6 sind hervorgehoben. Die Ladeinstruktion für die Werte der Matrix A sowie die Berechnung des Skalarproduktes bleiben unverändert.

Algorithmus 3.6 zeigt die **MMAvxGatherIdx** Programmvariante bei der die gather Instruktion und ein Index-Vektor zum Laden der Elemente der Spalte $b_{\_,j}$ genutzt werden. Vor der Ausführung der innersten Schleife wird dazu ein Vektorregister mit Nullen initialisiert (Zeile 5). Für die Berechnungen innerhalb der *k-Schleife* (Zeile 7) werden mehrere Zeilenwerte $a_{i,k}, ..., a_{i,k+7}$ der Matrix A geladen (Zeile 8). Zusätzlich wird in Zeile 10 ein Vektor mit Indexwerten erstellt der zum Laden der Spaltenwerte $b_{k,j}, ..., b_{k+7,j}$ der Matrix B genutzt wird (Zeile 12). Die geladenen Elemente werden miteinander multipliziert (Zeile 13) und auf einen Vektor mit Zwischenergebnissen addiert (Zeile 14). Die Addition der Teilergebnisse auf das Ergebniselement $c_{i,j}$ wird nach der Ausführung der Schleife (Zeile 15) durch eine vektorisierte Reduktion (vgl. Anhang Abschn. A.4) durchgeführt. Jedes Element $c_{i,j}$ wird auf diese Weise vektorisiert berechnet und nach Ausführung der beiden äußeren Schleifen des Schleifennestes ist die gesamte Ergebnismatrix C berechnet.

MMAvxGatherVdx: Vektorisierte Berechnung des Index-Vektors

Für die Berechnung der Indizes in der *MMAvxGatherIdx* Programmvariante werden pro Durchlauf der *k-Schleife* mehrere Operationen für die serielle Berechnung der nötigen Indexwerte ausgeführt. Jeder der skalaren Index-Werte wird im Anschluss in das Vektorregister geschrieben. Diese Berechnungen verlaufen dabei immer nach dem Muster $k_x \cdot n + j$, wobei k_x jeweils für die Zeilen der Werte $b_{k,j} \ldots, b_{k+7,j}$ steht. Diese Berechnung der Indizes für die gather Instruktion kann ebenfalls vektorisiert werden.

In Alg. 3.7 wird die vektorisierte Index-Berechnung der **MMAvxGatherVdx** Programmvariante gezeigt. Dazu werden die Werte n und $1, \ldots, 8$ vor der Ausführung der Schleifen in zwei konstante Vektoren _n (Zeile 1) bzw. _cnt (Zeile 2) gespeichert. Ein weiterer Vektor _j enthält an jeder Stelle den aktuellen Wert der Iterationsvariablen j (Zeile 8). Innerhalb der *k-Schleife* wird der Wert der Iterationsvariablen k in einen Vektor _k geladen. In den Zeilen 12 bis 14 wird der Index-Vektor entsprechend der folgenden Formel berechnet:

$$\_\text{ind} = (\_\text{k} + \_\text{cnt}) * \_\text{n} + \_\text{j}$$

Die Ladeinstruktionen und die Berechnung des Elements $c_{i,j}$ der Matrix C erfolgt identisch zur *MMAvxGatherIdx* Programmvariante.

MMAvxGatherBase: Gather mit konstantem Offset-Vektor

Die bisher gezeigten Programmvarianten *MMAvxGatherIdx* und *MMAvxGatherVdx* verwenden für die Berechnung der Index-Werte jeweils die Adresse des ersten Elements der Matrix B als Ausgangspunkt. Wird stattdessen die Adresse des ersten zu ladenden Elements der Matrix B genutzt, bleibt der Abstand (Offset) zum nächsten Element der Matrix B gleich. Der erste genutzte Index hat damit den Wert 0 und das nächste benötigte Element $b_{k+1,j}$ hat somit, gegenüber dem Element $b_{k,j}$, den Index n. Dieser Offset ändert sich nicht mit unterschiedlichen Werten für die Iterationsvariablen k oder j und kann somit als konstanter Vektor definiert werden.

Für die Programmvariante **MMAvxGatherBase** wird vor der Ausführung des Schleifennestes ein Vektor _offset mit den Werten $0, n, \ldots, n \cdot 7$ definiert. Die gather Instruktion wird mit den folgenden Parametern aufgerufen:

_b = _mm256_i32gather_ps(&B[k∗n+j], _offset, `sizeof(float)`)

Die Berechnung der Index-Werte in der *MMAvxGatherIdx* Programmvariante (Alg. 3.6 Zeile 10) kann auf diese Weise bei der **MMAvxGatherBase** Programmvariante entfallen.

*MMAvxVred** Programmvarianten mit teilweise serieller Reduktion

Vektorisierte Reduktionsoperationen werden mit Hilfe einer Verkettung von Umsortierinstruktionen implementiert, um die einzelnen Elemente des Vektors miteinander zu verrechnen (vgl. Anhang Abschn. A.4). Bei der vektorisierten Reduktion werden die Elemente des Vektors jeweils so miteinander verrechnet, dass nach jedem Schritt (Umsortieren und Addieren) jeweils halb so viele Elemente verbleiben, wie vor der Ausführung des Schrittes. Die erwarteten hohen Kosten der Umsortierinstruktionen fallen für jeden der genannten Schritte an. Ein Kompromiss zwischen den erhöhten Kosten und der erreichten Parallelität kann durch eine teilweise serielle Berechnung der Reduktionsoperation erreicht werden.

Die **MMAvxVred*** Programmvarianten werden auf Basis der *MMAvxGatherBase* Programmvariante erstellt, wobei * durch die Anzahl der seriell berechneten Elemente

ersetzt wird. Eine vollständig seriell ausgeführte Reduktion wird dabei in eine AVX-Speicherinstruktion und eine Schleife zum Aufsummieren der Elemente implementiert und wird als **MMAvxVred8** bezeichnet. Eine vollständig vektorisierte Reduktion wird demnach als **MMAvxVred1** bezeichnet. Bei den Programmvarianten **MMAvxVred2** und **MMAvxVred4** werden die ersten Schritte der Reduktion wie in der *MMAvxVred1* Programmvariante ausgeführt bis die entsprechende Anzahl an Elementen erreicht wird (vgl. Anhang Alg. A.1). Danach wird mit den verbleibenden Elementen wie bei der seriellen Implementierung verfahren. Bei der Nutzung der AVX512-Erweiterung können doppelt so viele Elemente in einem Vektorregister gespeichert werden, wodurch die vollständig vektorisierte Reduktion der **MMAvxVred16** Programmvariante entspricht. Mit der AVX512-Erweiterung wird zusätzlich ein Intrinsic zur Berechnung einer Reduktion angeboten, das in der Programmvariante **Vredi** genutzt wird.

3.2.2.2. AVX-Implementierungen mit zeilenweisem Zugriff auf Matrix B

Eine alternative Möglichkeit zur Vektorisierung der Matrix-Multiplikation ergibt sich, wenn statt der innersten Schleife (*k-Schleife*) die mittlere Schleife (*j-Schleife*) vektorisiert wird. Durch diese Änderung lässt sich die gather Instruktion und der spaltenweise Zugriff auf die Matrix B vermeiden.

MMAvxRow: Vermeidung von Gather-Operationen durch zeilenweisen Elementzugriff

Bei der Schleifen-Blockzerlegung in der *MMScalTile* Programmvariante werden mehrere Elemente der Matrix B zur Berechnung eines Blockes genutzt. Dieses Vorgehen wird für die **MMAvxRow** Programmvariante bei der Vektorisierung genutzt.

Abbildung 3.2 zeigt das Bearbeitungsschema der **MMAvxRow** Programmvariante. Dabei werden für jedes Element $a_{i,k}$ der Matrix A eine Reihe von Elementen $b_{k,j}, ..., b_{k,j+7}$ der Matrix B geladen (rot hinterlegt). Die Anzahl der Elemente aus Matrix B wird so gewählt, dass diese genau ein Vektorregister füllen. Auf diese Weise werden durch die innerste Schleife des Schleifennestes ebenfalls mehrere Elemente der Matrix C berechnet. Die Vektorisierung des Schleifennestes wird somit entlang der Iterationen der mittleren Schleife (*j-Schleife*) durchgeführt.

Algorithmus 3.8 zeigt die Implementierung der **MMAvxRow** Programmvariante zur Vektorisierung der *j-Schleife*. Für die Berechnungen wird in Zeile 5 ein temporärer Vektor mit Nullen initialisiert. Innerhalb der *k-Schleife* (Zeile 7) wird das Element $a_{i,k}$ an jede Stelle eines Vektorregisters kopiert (Zeile 8). Zusätzlich werden aus der Matrix B die Elemente $b_{k,j}, ..., b_{k,j+7}$ in ein Vektorregister geladen (Zeile 9). Im Anschluss werden die Vektoren miteinander multipliziert (Zeile 10) und auf den temporären Vektor addiert (Zeile 11). Nach Ausführung der *k-Schleife* enthalten die Elemente des temporären Vektors die Ergebnisse der Matrixelemente $c_{i,j}, ..., c_{i,j+7}$, die in Zeile 12 auf die Matrix C geschrieben werden.

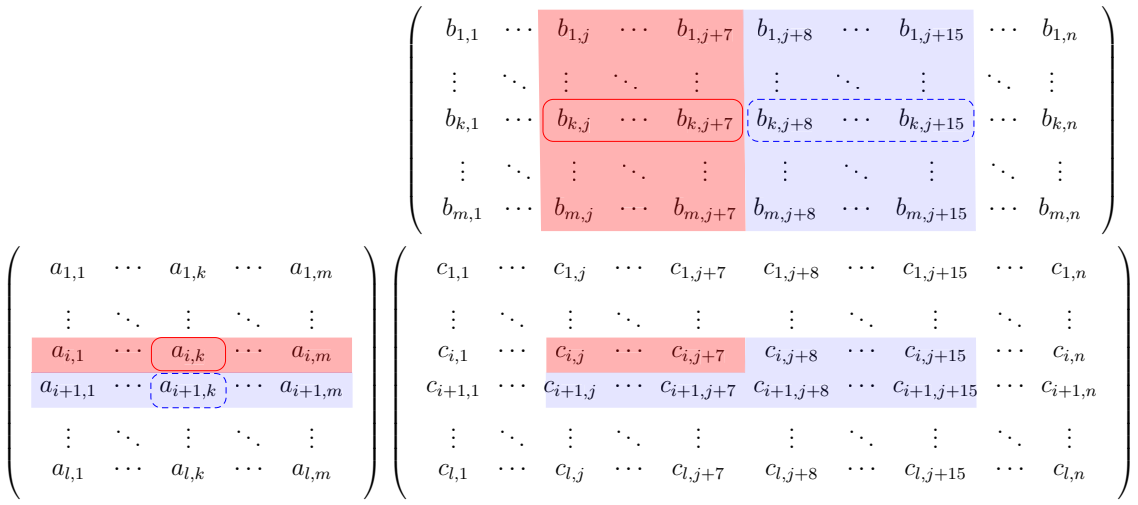

Abbildung 3.2.: Illustration des Bearbeitungsschemas der Matrix-Multiplikation in der *MMAvxRow* Programmvariante. Zur Berechnung der Elemente $c_{i,j}, ..., c_{i,j+7}$ werden die Elemente der Zeile $a_{i,\_}$ und der Spalten $b_{\_,j}, ..., b_{k,j+7}$ (rot hinterlegt) jeweils paarweise genutzt. Die blau hinterlegten Elemente werden bei der Anwendung der Schleifen-Blockzerlegung (Blockgröße = 2) in der *MMAvxTile* Programmvariante zusätzlich genutzt.

MMAvxTile: Zeilenweiser Elementzugriff mit Schleifen-Blockzerlegung

Die Anwendung der Schleifen-Blockzerlegung auf die *MMAvxRow* Programmvariante wird in Abb. 3.2 in blau dargestellt. Für die Schleifen-Blockzerlegung werden wie in der seriellen Programmvariante mehrere Zeilen der Matrix A verwendet. Für Elemente der Matrix B werden, im Vergleich zur *MMScalTile* Programmvariante, nicht nur mehrere einzelne Spalten genutzt, sondern mehrere Vektoren aus Elementen.

Algorithmus 3.9 zeigt die Implementierung der **MMAvxTile** Programmvariante, bei der die Schleifen-Blockzerlegung auf die *MMAvxRow* Programmvariante angewendet wird.

Die *i-Schleife* (Zeile 2) und *j-Schleife* (Zeile 4) werden dazu um die entsprechende Schrittweite tile_a bzw. tile_b erweitert. Die Blockgröße tile_b wird dazu mit der Anzahl der Elemente pro Vektorregister (hier 8) multipliziert um die tatsächliche Anzahl der Matrixelemente der Matrix B abzubilden.

Für die Berechnungen wird statt eines einzigen temporären Vektors _c ein Array von temporären Vektoren der Größe tile_a·tile_b angelegt. Jeder dieser temporären Vektoren wird innerhalb eines Schleifennestes (Zeilen 6 und 8) mit Nullen initialisiert (Zeile 9).

Die Elemente der Zeilen der Matrix A werden innerhalb der *k-Schleife* in ein Array mit tile_a Vektoren geladen (Zeile 14). Analog dazu werden die Vektoren der Spalten der Matrix B in ein Array mit tile_b Vektoren geladen (Zeile 17).

In den Zeilen 19 bis 23 wird für alle Elemente des Blockes das nächste Zwischenergebnis berechnet. Nach vollständiger Abarbeitung der *k-Schleife* werden die Elemente des Blockes auf die entsprechenden Elemente der Matrix C zurück gespeichert (Zeilen 25 bis 29).

```
1  // Schleife über die Zeilen a_{i,_}:
2  for i = 0 < l; i+=1
3    // Schleife über die Blöcke der Spalten b_{_,j}:
4    for j = 0 < n; j+=8
5      _c = _mm256_setzero_ps() // c_[] = 0, ..., 0
6      // Schleife über die Elemente der Zeile a_{i,_} und der Spalten b_{_,j}
7      for k = 0 < m; k+=1
8        _a = _mm256_broadcast_ss(&A[i*m+k]) // a_[] = a_{i,k}, ..., a_{i,k}
9        _b = _mm256_load_ps(&B[k*n+j]) // b_[] = b_{k,j}, ..., b_{k,j+7}
10       _t = _mm256_mul_ps(_a, _b) // a_[] · b_[]
11       _c = _mm256_add_ps(_t, _c) // c_[] = c_[] + a_[] · b_[]
12     _mm256_store_ps(&C[i*n+j], _c) // c_{i,j} = a_{i,_} · b_{_,j}
```

Algorithmus 3.8: MMAvxRow Programmvariante: Angepasste AVX-Implementierung der Matrix-Multiplikation zur Vermeidung von Gather-Operationen. Die Verarbeitung mehrerer Elemente entspricht einer Schleifen-Blockzerlegung mit tile_a=1 und tile_b=8, wie in Abb. 3.2 in Rot dargestellt. Veränderte Zeilen im Vergleich zur *MMScal* Programmvariante in Alg. 3.1 sind hervorgehoben.

MMAvxStream Programmvariante mit non-temporal Schreiboperationen

In der *MMAvxTile* Programmvariante werden die Elemente $c_{i,j}$ der Matrix C ausschließlich geschrieben und nicht gelesen. Bei der Verwendung der Write-Back Rückschreibestrategie (vgl. [63]) muss bei einer Schreiboperation die Speicherstelle von $c_{i,j}$ in den Cache geladen werden, um diese im Anschluss zu überschreiben. AVX bietet mit der stream Instruktion (auch *streaming stores*) eine Schreiboperation mit so genanntem non-temporal Flag an. Hierbei müssen Daten nicht erst in den Cache geladen werden, da durch das non-temporal Flag angegeben wird, dass die Daten nicht gleich wieder gelesen werden (vgl. Annahme der zeitlichen Lokalität in [63]). Zur Implementierung der **MMAvxStream** Programmvariante wird die store Instruktion (Zeile 29) der *MMAvxTile* Programmvariante (Alg. 3.9) durch eine stream Instruktion ersetzt.

3.2.2.3. AVX512-Instruktionen und 256-Bit AVX512-Instruktionen

Die gezeigten AVX-Implementierungen werden zusätzlich für die Nutzung von AVX512 implementiert. Dazu werden statt der 256-Bit Register der AVX-Erweiterung die 512-Bit Register der AVX512-Erweiterung genutzt. Die AVX512-Programmvarianten werden implementiert, indem die Angabe 256 in den Datentypen und Intrinsic-Definitionen durch eine 512 ausgetauscht wird (vgl. Anhang Tabellen A.1 und A.2). Einige Instruktionen werden dabei durch eine spezielle AVX512-Instruktion ersetzt, wie z.B. die broadcast Instruktion durch eine set1 Instruktion (siehe auch Tabelle der verwendeten Intrinsics im Anhang Tab. A.2). Zusätzlich müssen die Schrittweiten der vektorisierten Schleifen auf die neue Vektorgröße (16 Elemente) angepasst werden.

AVX-Instruktionen erlauben durch eine spezielle Kodierung die Angabe der Länge der verwendeten Vektorregister. Bei der AVX-Erweiterung erlaubt die VEX-Kodierung

```
1  // Schleife über die Blöcke der Zeilen a_{i,_}:
2  for i = 0 < l; i+=tile_a
3    // Schleife über die Blöcke der Spalten b_{_,j}:
4    for j = 0 < n; j+=tile_b*8
5      // Schleife über die Zeilen des temporären Arrays _c[tile_a][tile_b]:
6      for it = i < i+tile_a; it+=1
7        // Schleife über die Spalten des temporären Arrays _c[tile_a][tile_b]:
8        for jt = j < j+tile_b; jt+=1
9          _c[it][jt] = _mm256_setzero_ps() // c_[] = 0,...,0
10     // Schleife über die Elemente der Zeile a_{i,_} und der Spalte b_{_,j}
11     for k = 0 < m; k+=1
12       // Schleife über die Zeilen a_{i,_} innerhalb der Blöcke:
13       for it = 0 < tile_a; it+=1
14         _a[it] = _mm256_broadcast_ss(&A[(i+it)*m+k]) //a_[] = a_{i,k},...,a_{i,k}
15       // Schleife über die Spalten b_{_,j} innerhalb der Blöcke:
16       for jt = 0 < tile_b; jt+=1
17         _b[jt] = _mm256_load_ps(&B[k*n+(j+jt*8)]) // b_[] = b_{k,j},...,b_{k,j+7}
18       // Schleife über die Zeilen des temporären Arrays _c[tile_a][tile_b]:
19       for it = 0 < tile_a; it+=1
20         // Schleife über die Spalten des temporären Arrays _c[tile_a][tile_b]:
21         for jt = 0 < tile_b; jt+=1
22           _t = _mm256_mul_ps(_a[it], _b[jt]) // a_[] · b_[]
23           _c[it][jt] = _mm256_add_ps(_t, _c[it][jt]) // c_[] = c_[] + a_[] · b_[]
24     // Schleife über die Zeilen des temporären Arrays _c[tile_a][tile_b]:
25     for it = 0 < tile_a; it+=1
26       // Schleife über die Spalten des temporären Arrays _c[tile_a][tile_b]:
27       for jt = 0 < tile_b; jt+=1
28         // Speichern der Elemente c_{i,j},...,c_{i,j+tile_b·8},c_{i+1,j},...,c_{i+tile_a,j+tile_b·8}:
29         _mm256_store_ps(&C[(i+it)*n+(j+jt*8)], _c[it][jt])
```

Algorithmus 3.9: MMAvxTile Programmvariante: Anwendung der Schleifen-Blockzerlegung auf die *MMAvxRow* Programmvariante. Die Verarbeitung mehrerer Elemente entspricht einer Schleifen-Blockzerlegung, wie in Abb. 3.2 in Blau dargestellt. Veränderte Zeilen im Vergleich zur *MMAvxRow* Programmvariante in Alg. 3.8 sind hervorgehoben.

(vgl. [36, Vol.1, Abschn. 5.13]) ein Verwendung von 128-Bit oder 256-Bit Vektorregistern mit den selben Instruktionen. Analog dazu erlaubt die EVEX-Kodierung (vgl. [36, Vol.1, Abschn. 15.1]) unter anderem die Verwendung von 128-Bit, 256-Bit und 512-Bit Vektorregistern mit den meisten AVX512-Instruktionen.

Durch die EVEX-Kodierung ist es möglich, die meisten AVX512-Instruktionen mit einer Vektorgröße von 256-Bit zu nutzen. Zur Untersuchung der Unterschiede zwischen 256-Bit AVX-Instruktionen und 256-Bit AVX512-Instruktionen werden die AVX-Implementierungen der Matrix-Multiplikation zusätzlich mit 256-Bit AVX512-Instruktionen ausgeführt. Zur Erstellung dieser Varianten werden die Programmvarianten der 256-Bit AVX-Implementierungen mit dem Compiler-Flag für AVX512-Architekturen gebaut.

3.3. Untersuchung der Ausführungszeit und des Energieverbrauchs der vektorisierten Matrix-Multiplikation

In diesem Abschnitt werden die Ausführungszeit und der Energieverbrauch der vektorisierten Matrix-Multiplikation untersucht. Im Folgenden wird dazu die Ausführungsumgebung der Messungen von Ausführungszeit und Energieverbrauch der Programmvarianten vorgestellt. Im Anschluss werden die Messergebnisse der Programmvarianten diskutiert.

3.3.1. Ausführungsumgebung für die Untersuchung der Programmvarianten

Die vorgestellten Programmvarianten der Matrix-Multiplikation werden auf drei Desktop-Prozessoren und einem Server-Prozessor ausgeführt. Tabelle 3.2 zeigt die Spezifikationen der vier genutzten Prozessoren. Die AVX-Instruktionssatzerweiterung wurde erstmals mit der Sandy Bridge Architektur verfügbar. Die Sandy Bridge Architektur bietet dabei keine gather Instruktionen an, weshalb die Programmvarianten mit dieser Instruktion dort nicht ausgeführt werden können. Mit der Haswell Architektur wurde die AVX-Erweiterung um zusätzliche Instruktionen (AVX2) erweitert. Bei der Weiterentwicklung zur Skylake Architektur wurden verschiedene Verbesserungen an der Ausführung von AVX-Instruktionen vorgenommen. Der Skylake XEON Prozessor bietet zusätzlich die Nutzung der AVX512-Erweiterung an, bei der unter anderem die Größe der AVX-Register verdoppelt wurde.

Die gezeigten Programmvarianten werden mit dem Intel Compiler `icc` (Version 17.0.0) mit den folgenden Compilerflags compiliert:

- `-O3`: Nutzt die höchste Optimierungsstufe des Compilers (vgl. [38]).
- `-restrict`: Erlaubt die Nutzung des restrict Schlüsselwortes, mit dem überlappende Speicherzugriffe durch unterschiedliche Zeiger ausgeschlossen werden. Jeder verwendete Zeiger der Programmvarianten ist durch das restrict Schlüsselwort gekennzeichnet.
- `-cilk-serialize`: Deaktiviert Threading bei der Nutzung von Intel Cilk. Die Cilk Array-Notation wird weiterhin vektorisiert.
- `-mavx` (Sandy Bridge), `-march=core-avx2` (Haswell, Skylake Core, AVX2 auf Skylake XEON), `-march=native` (AVX512 auf Skylake XEON): Erzeugt den AVX-Code für die entsprechende AVX-Erweiterung.

Zur Messung des Energieverbrauchs der Programmvarianten werden die prozessorinternen Register des Running Average Power Limits Interfaces (RAPL) verwendet. Die Genauigkeit der ausgelesenen Registerwerte gegenüber anderen Methoden wird

Architektur	Jahr	Prozessor	AVX-Version	Frequenzbereich	LLC
Sandy Bridge	2011	Core i7-2600	AVX	1,5-3,7 GHz	8MB
Haswell	2013	Core i7-4770K	AVX2	0,8-3,5 GHz	8MB
Skylake	2015	Core i7-6700	AVX2	0,8-3,5 GHz	8MB
Skylake	2017	XEON Gold 6130	AVX512	1,0-2,1 GHz	22MB

Tabelle 3.2.: Verwendete Prozessoren zur Ausführung der Programmvarianten der Matrix-Multiplikation mit der jeweiligen Prozessorarchitektur, dem Erscheinungsjahr, dem höchsten unterstützten AVX-Instruktionssatz, des einstellbaren Frequenzbereichs und der Größe des Last-Level-Caches (LLC). Vergleiche [40].

beispielsweise in [66] gezeigt. Zusätzliche Werte, wie die Anzahl von Cache Misses, werden mit der PAPI Bibliothek (Version 5.5) gemessen.

Eine Abhängigkeit zwischen der Prozessorfrequenz und der Ausführungszeit sowie dem Energieverbrauch von Programmen wird unter anderem in [14] gezeigt. Die Prozessorfrequenz kann mit Hilfe des Linux-Tools `cpufreq` auf eine Reihe von vordefinierten Werten eingestellt werden. Der Frequenzbereich ist dabei je nach Prozessor unterschiedlich und ist in Tab. 3.2 angegeben.

3.3.2. Abhängigkeit der Programmvarianten zur Matrixgröße

Die Größe der verwendeten Matrizen bei der Matrix-Multiplikation bestimmt den Speicherbedarf bei der Ausführung der Programmvarianten. Der Speicherbedarf der Matrizen bestimmt, ob diese während der Ausführung der Programme vollständig in den Cache geladen werden können. Daraus kann sich für die Ausführung der Programme ein unterschiedliches Verhalten ergeben, wenn die Matrizen vollständig oder nur teilweise in den Cache geladen werden.

Abbildung 3.3 zeigt die Ausführungszeiten verschiedener Programmvarianten in Abhängigkeit zur gewählten Matrixgröße. Die Ausführungszeiten einiger Programmvarianten haben jeweils einen sprunghaften Anstieg bei einer gewissen Matrixgröße. Bei den Desktop-Prozessoren (Sandy Bridge siehe Anhang Abb. B.2) findet dieser Sprung bei einer Matrixgröße von ca. 1000×1000 Elementen je Matrix statt. Dies entspricht in etwa einem Speicherbedarf von 8MB für die zu lesenden Matrizen und damit der Last-Level-Cache Größe der drei Prozessoren. Beim Skylake XEON Prozessor (Abb. 3.3(rechts)) liegt der Sprung einiger Programmvarianten bei einer Matrixgröße zwischen 1300×1300 und 1600×1600 Elementen, was ebenfalls in etwa der verfügbaren Cache-Größe (22MB) entspricht.

Die *MMCilk* Programmvariante und die *MMAvxGather** Programmvarianten, die an diesen Stellen einen sprunghaften Anstieg der Ausführungszeit zeigen, greifen spaltenweise auf die Matrix B zu. Die Programmvarianten, die einen zeilenweisen Zugriff auf die Matrix haben, zeigen an der entsprechenden Cache-Größe nur einen sehr geringen zusätzlichen Anstieg der Ausführungszeit über den erwarteten Wert hinaus. Ein ähnliches Verhalten zeigt sich in der Anzahl der Last-Level-Cache Misses (Anhang Abb. B.1, die bei den gleichen Matrixgrößen jeweils einen sprunghaften Anstieg

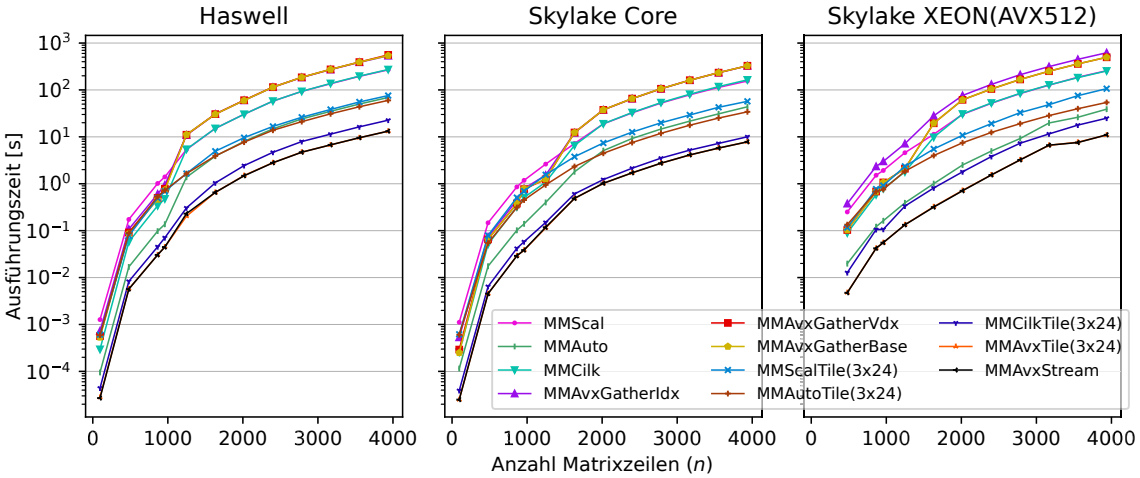

Abbildung 3.3.: Ausführungszeit verschiedener Programmvarianten der Matrix-Multiplikation in Abhängigkeit zur Matrixgröße auf den Haswell (3,5GHz), Skylake Core (3,4GHz) und Skylake XEON (2,0GHz) Prozessoren. Die Werte des Sandy Bridge Prozessors sind im Anhang in Abb. B.2 dargestellt.

zeigen, bei dem die Programmvarianten mit spaltenweisem Zugriff danach eine höhere Anzahl Cache-Misses aufweisen.

Eine Besonderheit stellt die *MMAuto* Programmvariante dar, deren Ausführungszeit den sprunghaften Anstieg nur auf der Sandy Bridge und Haswell Architektur zeigt. Auf den beiden Skylake Prozessoren verhält sich die *MMAuto* Programmvariante vergleichbar mit den Programmvarianten mit zeilenweisem Zugriff auf die Matrix B. Bei der *MMAuto* Programmvariante wird die mittlere (*j-Schleife*) vektorisiert, was zu einem zeilenweisen Zugriff auf Matrix B führt. Für die Sandy Bridge und Haswell Architekturen kann hierbei die unterschiedliche Prozessorarchitektur und ein anderer Microcode im Vergleich zur Skylake Architektur einen Einfluss auf die Ausführung der entsprechenden AVX-Instruktionen haben.

3.3.3. Ausführungszeit der vektorisierten Programmvarianten der Matrix-Multiplikation

In diesem Abschnitt wird die Ausführungszeit der gezeigten Programmvarianten untersucht. Die Untersuchungen werden aufgeteilt nach Programmvarianten mit spaltenweisem Zugriff auf Matrix B, einer teilweise seriell durchgeführten Reduktionsoperation und Programmvarianten mit zeilenweisem Zugriff auf Matrix B.

Programmvarianten mit spaltenweisem Zugriff auf Matrix B

Bei der Vektorisierung der inneren Schleife des Schleifennestes (*k-Schleife*) wird auf die Elemente der Matrix B spaltenweise zugegriffen. Dies bedeutet, dass die aufeinanderfolgend gelesenen Elemente der Matrix B im Speicher verteilt (bzw. nicht nebeneinander) abgelegt sind. Die Betrachtung der Abhängigkeit zur Matrixgröße

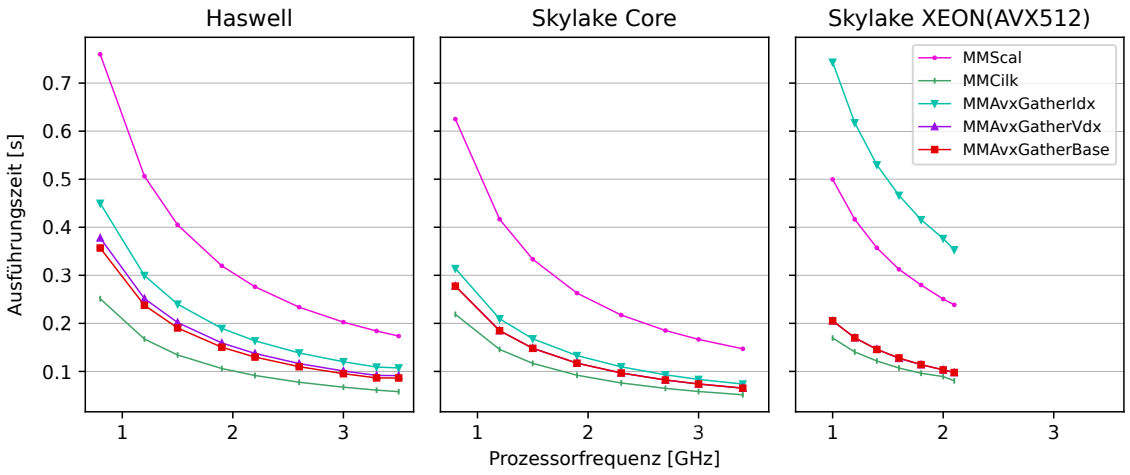

Abbildung 3.4.: Ausführungszeit der Programmvarianten spaltenweisem Zugriff auf Matrix B in Abhängigkeit zur Prozessorfrequenz auf den Haswell und Skylake Prozessoren. Matrixgröße: 480×480.

im vorherigen Abschnitt zeigt, dass sich die Ausführungszeit der Programmvarianten mit spaltenweisem Zugriff bei unterschiedlicher Matrixgröße unterschiedlich verhalten. Aus diesem Grund werden die Betrachtungen für zwei verschiedene Matrixgrößen durchgeführt, die so gewählt sind, dass sie dieses unterschiedliche Verhalten zeigen.

Abbildung 3.4 zeigt die Ausführungszeit der Programmvarianten mit spaltenweisem Zugriff auf Matrix B, wobei die Matrizen vollständig in den Cache geladen werden können. Wie zu erwarten reduziert sich die Ausführungszeit bei höheren Prozessorfrequenzen, da in gleicher Zeit mehr Instruktionen ausgeführt werden.

Die Vektorisierung reduziert bei allen gezeigten Programmvarianten die Ausführungszeit. Die *MMCilk* Programmvariante hat die geringste Ausführungszeit, da der Compiler hierbei zusätzliche Optimierungen, wie *loop unrolling* vornimmt. Bei der Vektorisierung mit AVX-Intrinsics hat die *MMAvxGatherIdx* Programmvariante die höchste Ausführungszeit. Zur Ausführung der gather Instruktion wird dabei ein Vektor mit Indizes gesetzt, wofür jeder Index mit skalaren Instruktionen berechnet wird und anschließend einzeln in das Vektorregister geladen wird.

Eine Reduzierung der Ausführungszeit bringt die vektorisierte Berechnung des Index-Vektors (*MMAvxGatherVdx*). Hierbei werden zwar mehr Vektorinstruktionen ausgeführt, die Operation zum Setzen des Registers, sowie die serielle Berechnung der Indizes entfallen jedoch. Weiter reduzieren lässt sich die Ausführungszeit durch die Nutzung eines Offset-Vektors (*MMAvxGatherBase*), in dem nur die Abstände der entsprechenden Matrixelemente gespeichert sind. Dieser Offset-Vektor wird einmal gesetzt und im Anschluss zusammen mit der Adresse des ersten zu ladenden Elements an die gather Instruktion übergeben.

Auf den beiden Skylake Prozessoren gibt es keinen Unterschied zwischen der *MMAvxGatherVdx* und der *MMAvxGatherBase* Programmvariante. Auf dem Skylake XEON Prozessor hat die *MMAvxGatherIdx* Programmvariante eine höhere Ausführungszeit

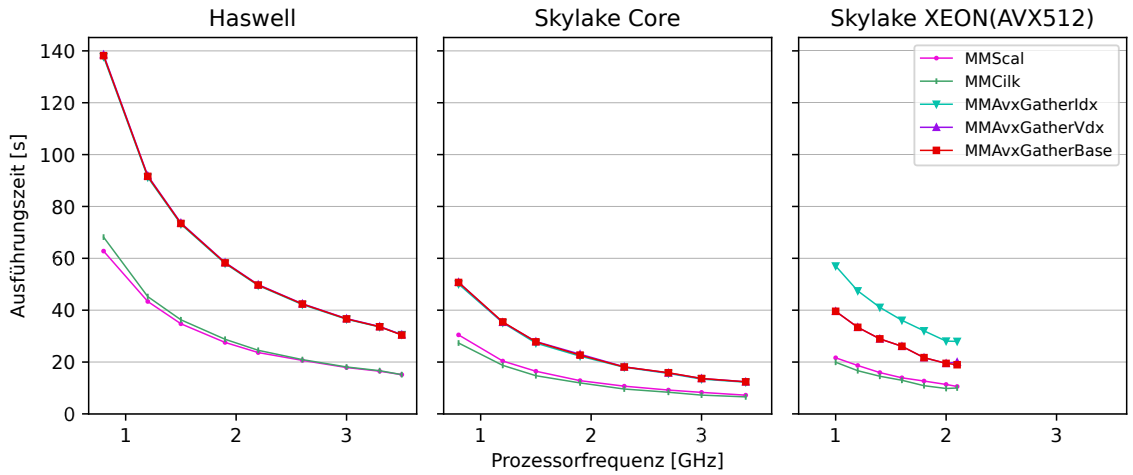

Abbildung 3.5.: Ausführungszeit der Programmvarianten spaltenweisem Zugriff auf Matrix B in Abhängigkeit zur Prozessorfrequenz auf den Haswell und Skylake Prozessoren. Matrixgröße: 1632×1632.

als die serielle Programmvariante. Dies begründet sich durch die genutzten AVX512-Vektoren, für die doppelt so viele Indizes berechnet und gesetzt werden müssen. Außerdem müssen beim Laden der Matrixelemente in ein Vektorregister doppelt so viele verteilt abgelegte Werte geladen werden.

Abbildung 3.5 zeigt die Ausführungszeiten der Programmvarianten mit spaltenweisem Zugriff auf Matrix B bei einer Matrixgröße, bei der nicht alle Werte in den Cache geladen werden können. Für die Haswell und Skylake Core Prozessoren haben damit die *MMAvxGather** Programmvarianten die gleiche Ausführungszeit. Die Ausführung der Programmvarianten hängt bei großen Matrixgrößen verstärkt von Speicheranbindung des Prozessors ab, weshalb die Berechnung der Indizes während der Wartezeiten auf die benötigten Speicherblöcke durchgeführt werden kann. Bei der *MMAvxGatherIdx* Programmvariante auf dem Skylake XEON Prozessor benötigt das Setzen der Indizes weiterhin mehr Zeit, als durch die Speicheroperationen verdeckt werden kann. Die *MMCilk* Programmvariante zeigt bei größeren Matrizen ebenfalls eine erhöhte Ausführungszeit, diese Erhöhung ist jedoch geringer als die der *MMAvxGather** Programmvarianten, sodass stellenweise noch eine Reduzierung der Ausführungszeit gegenüber der seriellen Programmvariante erreicht wird.

Ausführungszeiten der Programmvarianten mit teilweise serieller Reduktion

Die vektorisierte Reduktionsoperation enthält eine Reihe von kostenintensiven Umsortierinstruktionen (vgl. Anhang Abschn. A.4). Um diese Umsortierinstruktionen zu vermeiden, kann die Reduktion seriell, oder teilweise seriell ausgeführt werden. Der Einfluss auf die Ausführungszeit einer teilweise seriellen Reduktionsoperation wird im folgenden untersucht. Durch die verhältnismäßig geringe Anzahl der Reduktionsoperationen im Vergleich zu den restlichen AVX-Instruktionen ist der Einfluss auf die Ausführungszeit nur gering.

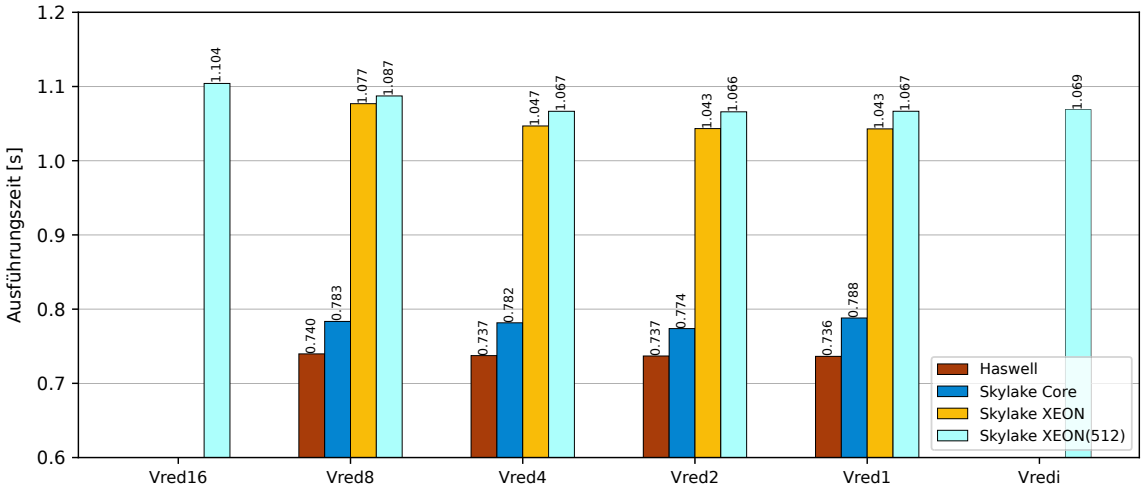

Abbildung 3.6.: Ausführungszeit der *MMAvxVred** Programmvarianten der Matrix-Multiplikation auf der Haswell (3,5GHz), Skylake Core (3,4GHz) und Skylake XEON (2,0GHz) Architektur. Matrixgröße: 960×960.

Abbildung 3.6 zeigt die Ausführungszeiten der *MMAvxVred** Programmvarianten, wobei die Zahl im Namen für die Anzahl an seriell reduzierten Vektorelementen steht. Im Vergleich zu einer seriellen Reduktion (*MMAvxVred8*, bzw. *MMAvxVred16* bei AVX512) reduziert die Vektorisierung der Reduktionsoperation die Ausführungszeit der Programmvariante. Die vollständige Vektorisierung der Reduktion (*MMAvxVred1*) hat eine leicht höhere Ausführungszeit als die Nutzung einer skalaren Addition der letzten beiden Elemente. Gleiches gilt für die Nutzung des AVX512-Reduktions-Intrinsics (*MMAvxVredi*). Für den Skylake Core Prozessor erhöht sich die Ausführungszeit durch die letzte Reduktion sogar im Vergleich zur vollständig seriellen Reduktion. Das beobachtete Verhalten zeigt, dass sich die kostenintensive Umsortierinstruktion für die letzte Addition der Reduktionsoperation nicht mehr lohnt, sodass eine Speicheroperation und eine skalare Addition günstiger sind.

Ausführungszeit von Programmvarianten mit zeilenweisem Zugriff auf Matrix B

Durch die Anwendung von Schleifen-Blockzerlegung auf das Schleifennest wird der spaltenweise Zugriff auf die Elemente der Matrix B so transformiert, dass innerhalb eines Blockes mehrere Elemente der gleichen Zeile der Matrix B genutzt werden können. Auf diese Weise ändert die Schleifen-Blockzerlegung das Zugriffsmuster zu einem zeilenweisen Zugriff auf die Elemente der Matrix B.

Abbildung 3.7 zeigt die Ausführungszeiten der Programmvarianten mit zeilenweisem Zugriff auf die Matrix B bei kleinen Matrizen. Alle angewendeten Vektorisierungsansätze und die Schleifen-Blockzerlegung reduzieren die Ausführungszeit gegenüber der seriellen Ausführung der Matrix-Multiplikation. Bei der Anwendung der Schleifen-Blockzerlegung auf die serielle Programmvariante hat die Nutzung einer größeren Blockgröße (3×24) eine höhere Ausführungszeit als die Nutzung einer kleineren

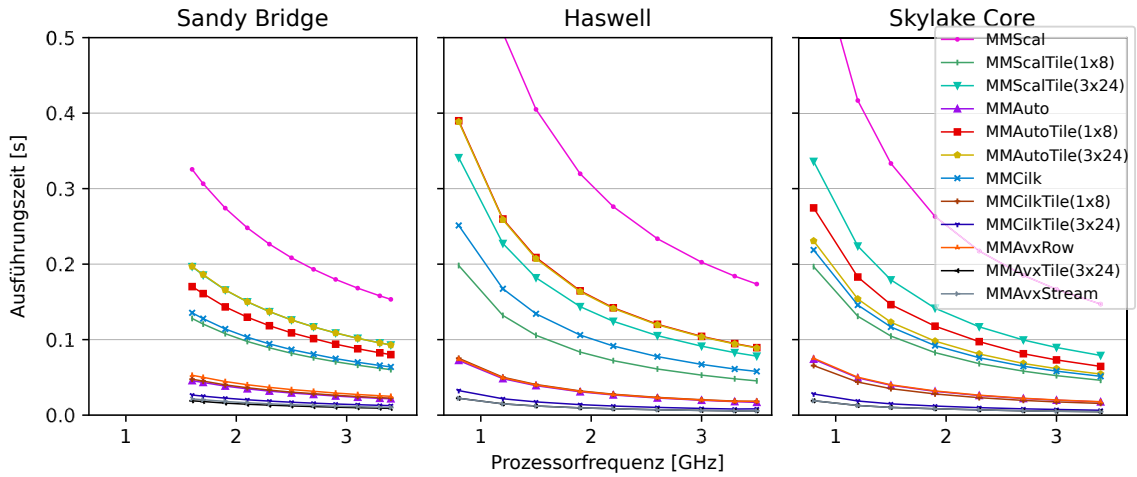

Abbildung 3.7.: Ausführungszeit der Programmvarianten mit zeilenweisem Zugriff auf Matrix B und Schleifen-Blockzerlegung mit unterschiedlichen Blockgrößen auf dem Sandy Bridge, Haswell und Skylake Core Prozessor. Die Blockgröße $(a \times b)$ entspricht der Anzahl der Matrixelemente in jede Dimension. Matrixgröße: 480×480. Die Werte für den Skylake XEON Prozessor sind im Anhang in Abb. B.3 gezeigt.

Blockgröße (1×8). Bei den automatisch vektorisierten Programmvarianten *MMAuto** hat die Programmvariante ohne Schleifen-Blockzerlegung die geringste Ausführungszeit und die Erhöhung der Blockgröße verlängert ebenfalls die Ausführungszeit. Die Speicherzugriffe der *MMAutoTile** Programmvarianten greifen auf verteilt im Speicher liegende Elemente zu, was in der *MMAuto* Programmvariante durch die Vektorisierung der *j-Schleife* nicht nötig ist. Bei der automatischen Vektorisierung der Programmvarianten mit Schleifen-Blockzerlegung erzeugt der Compiler zusätzliche Ladeoperation zum Sammeln der verteilten Elemente.

Bei den *MMCilk** und *MMAvx** Programmvarianten reduziert die Anwendung der Schleifen-Blockzerlegung die Ausführungszeit zusätzlich. Für die *MMCilkTile* Programmvariante mit einer Blockgröße von 1×8 Elementen und für die automatisch vektorisierte *MMAuto* Programmvariante erzeugt der Compiler einen sehr ähnlichen Assembler-Code, wie die Implementierung der *MMAvxRow* Programmvariante. Dementsprechend sind auch die Ausführungszeiten dieser drei Programmvarianten sehr ähnlich. Bei einer Blockgröße von 3×24 Elementen reduziert sich die Ausführungszeit der *MMCilkTile* und der *MMAvxTile* Programmvarianten. Beide Programmvarianten reduzieren die Anzahl der Ladeoperationen im Verhältnis zur Anzahl der damit berechneten Ergebniselemente. Die zusätzliche Nutzung der stream Instruktion (*MMAvxStream*) verändert die Ausführungszeit dabei kaum.

Abbildung 3.8 zeigt die selben Programmvarianten bei der Ausführung mit einer Matrix, bei der die Matrizen nicht mehr vollständig in den Cache geladen werden können. Die erhöhte Ausführungszeit der *MMCilk* Programmvariante ist bereits bei den Programmvarianten mit spaltenweisem Zugriff auf Matrix B besprochen. Die Ausführungszeiten der *MMCilkTile* Programmvarianten erhöht sich dabei nicht im gleichen Maße, da durch den zeilenweisen Zugriff auf Matrix B weniger Speicherzugriffe nötig

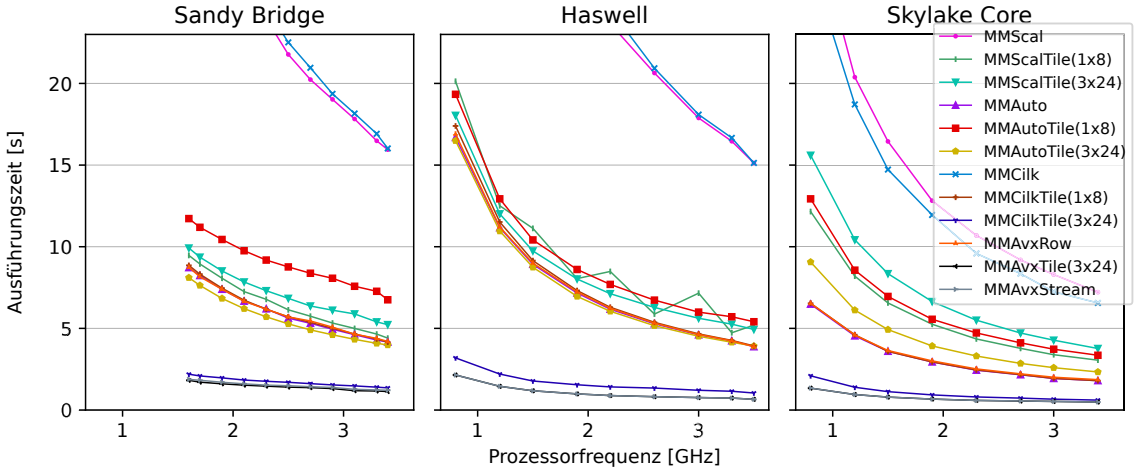

Abbildung 3.8.: Ausführungszeit der Programmvarianten mit zeilenweisem Zugriff auf Matrix B und Schleifen-Blockzerlegung mit unterschiedlichen Blockgrößen auf dem Sandy Bridge, Haswell und Skylake Core Prozessor. Die Blockgröße $(a \times b)$ entspricht der Anzahl der Matrixelemente in jede Dimension. Matrixgröße: 1632×1632. Die Werte für den Skylake XEON Prozessor sind im Anhang in Abb. B.3 gezeigt.

sind. Bei den seriellen Programmvarianten reduziert sich der Unterschied der Ausführungszeit bei verschiedenen Blockgrößen durch die bessere Ausnutzung des Caches. Die *MMAutoTile* Programmvariante mit einer Blockgröße von 3×24 hat, durch den größeren Einfluss der Cache-Ausnutzung, auf dem Sandy Bridge und Haswell Prozessor eine leicht geringere Ausführungszeit als die *MMAuto* Programmvariante ohne Schleifen-Blockzerlegung.

Die Nutzung der größeren Blockgröße von 3×24 Elementen reduziert die Ausführungszeit der *MMCilkTile* und *MMAvxTile* Programmvarianten am meisten, da hierbei mehr Rechenoperationen als Speicherzugriffsoperationen durchgeführt werden.

Ausführungszeit in Abhängigkeit zur Blockgröße

Wie bereits gezeigt, hat die gewählte Blockgröße der Schleifen-Blockzerlegung einen Einfluss auf die Ausführungszeiten der vektorisierten Programmvarianten. Die Auswahl einer passenden Blockgröße ist somit ein wesentlicher Bestandteil bei der Anwendung der Schleifen-Blockzerlegung.

In Abb. 3.9 werden die Ausführungszeiten der *MMAvxTile* Programmvariante mit verschiedenen Blockgrößen und Vektorgrößen auf dem Skylake XEON Prozessor gezeigt. Die *MMAvxRow* Programmvariante (entspricht Blockgröße 1×1) hat bei der Ausführung mit 512-Bit Vektoren aufgrund der größeren Vektoren eine kürzere Ausführungszeit als bei der Ausführung mit 256-Bit Vektoren. Bei der Anwendung der Schleifen-Blockzerlegung in den *MMAvxTile* Programmvarianten verringert sich die Ausführungszeit mit steigender Blockgröße bis zu einem gewissen Punkt.

Bei der Ausführung mit **256-Bit AVX2**-Instruktionen sinkt die Ausführungszeit bis zu einer Blockgröße von 3×3 (entspricht 3×24 Elementen). Eine weitere Ver-

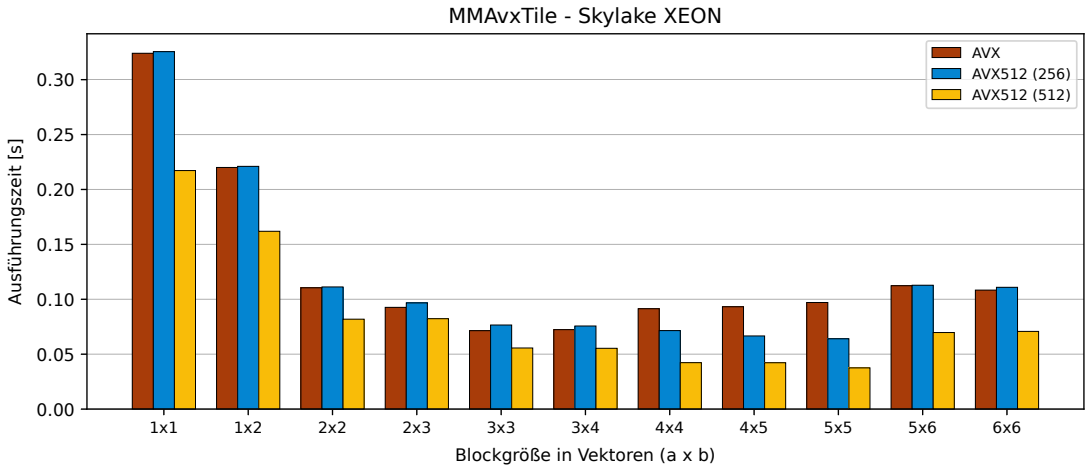

Abbildung 3.9.: Ausführungszeit der Programmvarianten mit Schleifen-Blockzerlegung mit unterschiedlichen Blockgrößen auf dem Skylake XEON Prozessor. Die Blockgröße entspricht der Anzahl der AVX-Vektoren (vgl. Abb. 3.2) in jede Dimension. Matrixgröße: 960×960; Prozessorfrequenz 2,0GHz.

größerung der Blockgrößen führt dabei wieder zu längeren Ausführungszeiten. Bei der Ausführung der inneren (*k-Schleife*) werden bei der 3×3 Variante je drei Vektoren der Matrizen A und B geladen. Zusätzlich werden neun Vektoren für die Zwischenspeicherung der Ergebniselemente der Matrix C genutzt. Die Anzahl der genutzten Vektorregister liegt damit unter den verfügbaren 16 Vektorregistern (vgl. YMM-Register [36]). Durch die Überlappung von Lade- und Rechenoperationen kann der Compiler diesen Effekt auch Teilweise für eine Blockgröße von 3×4 ausnutzen. Ab einer Blockgröße von 4×4 werden einzelne Register zurückgeschrieben um die Berechnungen des Blockes ausführen zu können.

Die Nutzung von **512-Bit AVX512**-Instruktionen erreicht die geringste Ausführungszeit bei einer Blockgröße von 5×5. Die Anzahl der benötigten Vektorregister für die Berechnung eines Blockes ist hierbei 35. Mit der AVX512-Erweiterung wurde unter anderem die Anzahl der verfügbaren AVX-Register auf 32 erhöht (vgl. ZMM-Register [36]). Durch die Überlappung der Lade- und Rechenoperationen können die Berechnungen bei einer Blockgröße von 5×5 ohne Zwischenspeichern der Register durchgeführt werden.

Werden die Programmvarianten mit **256-Bit AVX512**-Instruktionen ausgeführt ist die Ausführungszeit der *MMAvxRow* Programmvariante leicht höher als bei der Nutzung von AVX2-Instruktionen (1×1 in Abb. 3.9). Diese Erhöhung der Ausführungszeit ist ein Hinweis auf die aufwändigere Kodierung der AVX512-Instruktionen. AVX- und AVX2-Instruktionen werden mit dem so genannten VEX-Präfix kodiert [36, Vol.2A Abschn. 2.3]. Dahingegen werden AVX512-Instruktionen mit dem so genannten EVEX-Präfix kodiert [36, Vol.2A Abschn. 2.6]. Der EVEX-Präfix ist größer als der VEX-Präfix, was beim Laden und Ausführen der Instruktionen zu einem geringen Mehraufwand führt.

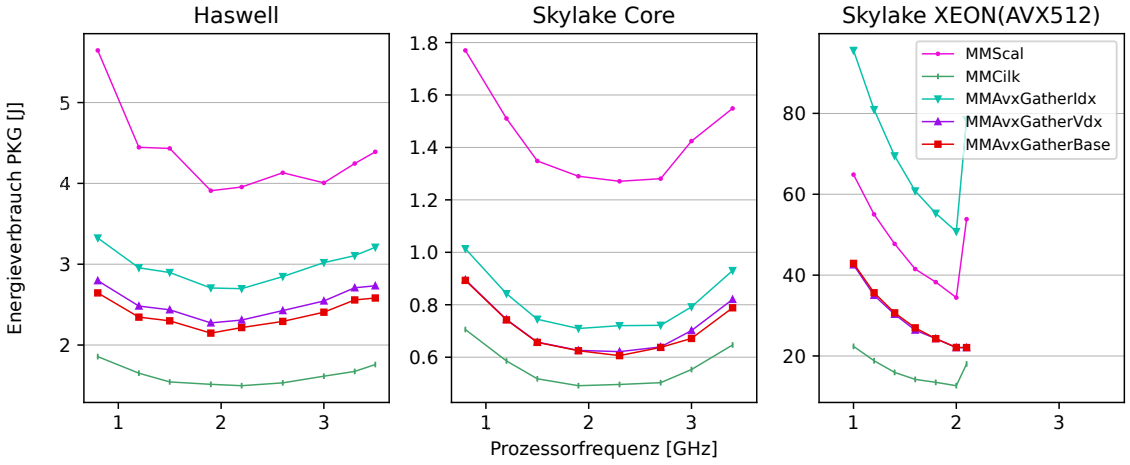

Abbildung 3.10.: Energieverbrauch der Programmvarianten mit spaltenweisem Zugriff auf Matrix B in Abhängigkeit zur Prozessorfrequenz auf den Haswell und Skylake Prozessoren. Matrixgröße: 480×480.

Die Ausführungszeiten der Programmvarianten mit **256-Bit AVX512**-Instruktionen verhalten sich in Abhängigkeit zur Blockgröße ähnlich, wie die Ausführungen mit 512-Bit Instruktionen. Dieses Verhalten erklärt sich durch die Adressierung der Register mit Hilfe des Präfixes: Der VEX-Präfix kann bei der Adressierung maximal 16 Vektorregister adressieren (vgl. [36, Vol.2A Abschn. 2.3.5]) der EVEX-Präfix unterstützt bis zu 32 Vektorregister (vgl. [36, Vol.2A Abschn. 2.6.1]). Da die Blockgröße mit der minimalen Ausführungszeit von der Anzahl der verfügbaren Register abhängt, erreicht die Nutzung der EVEX-Kodierten 256-Bit AVX512-Instruktionen die kürzeste Ausführungszeit bei der gleichen Blockgröße (5×5) wie 512-Bit Instruktionen.

3.3.4. Energieverbrauch und Leistungsaufnahme der vektorisierten Matrix-Multiplikation

Neben einer geringen Ausführungszeit von Programmen ist die Energie-Effizienz ein weiteres Optimierungsziel bei der Erstellung von Programmen. In diesem Abschnitt wird dazu der Energieverbrauch und die Leistungsaufnahme der vorgestellten Programmvarianten der Matrix-Multiplikation diskutiert.

Programmvarianten mit spaltenweisem Zugriff auf Matrix B

Abbildung 3.10 zeigt den Energieverbrauch der Programmvarianten mit spaltenweisem Zugriff auf Matrix B bei kleinen Matrizen. Der Energieverbrauch der Programmausführungen sinkt bis zu einem gewissen Punkt mit steigender Prozessorfrequenz. Bei weiter steigender Prozessorfrequenz steigt der Energieverbrauch der Programmausführungen wieder an. Der geringste Energieverbrauch wird bei einer mittleren Prozessorfrequenz erreicht. Durch die steigende Prozessorfrequenz verkürzt sich die Ausführungszeit, was zu einem geringeren Einfluss des statischen Energieverbrauchs des

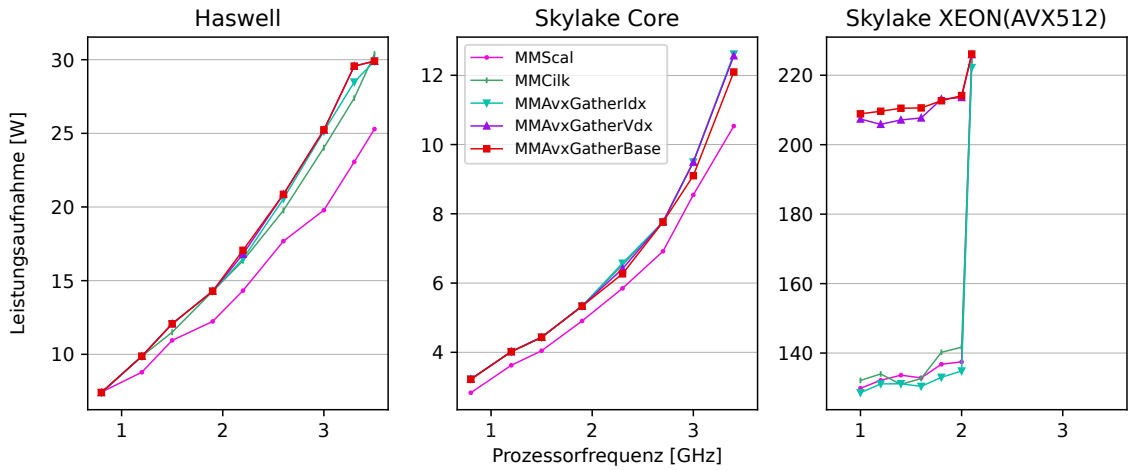

Abbildung 3.11.: Leistungsaufnahme der Programmvarianten mit spaltenweisem Zugriff auf Matrix B in Abhängigkeit zur Prozessorfrequenz auf den Haswell und Skylake Prozessoren. Matrixgröße: 480×480.

Prozessors führt. Zur Erhöhung der Prozessorfrequenz wird die Versorgungsspannung des Prozessors erhöht, was den Energieverbrauch erhöht [65].

Die vektorisierten Programmvarianten haben einen wesentlich geringeren Energieverbrauch, was unter anderem durch die kürzere Ausführungszeit und die damit verbundene kürzere Aktivierung des Prozessors begründet ist. Das Verhältnis der Energieverbrauchswerte der Programmvarianten zueinander entspricht dem Verhältnis der Ausführungszeiten dieser Programmvarianten zueinander, sodass eine kürzere Ausführungszeit auch zu niedrigerem Energieverbrauch führt. Der Energieverbrauch der Programmvarianten mit spaltenweisem Zugriff auf Matrix B bei der Ausführung mit Matrizen, die nicht vollständig in den Cache passen, verhält sich analog (vgl. Anhang Abb. B.4). Um die Reduzierung des Energieverbrauchs mit der Reduzierung der Ausführungszeit ins Verhältnis zu setzten wird die Leistungsaufnahme der Programmvarianten betrachtet. Die Leistungsaufnahme berechnet sich aus dem Energieverbrauch dividiert durch die Ausführungszeit der Programmausführung.

Abbildung 3.11 zeigt die Leistungsaufnahme der Programmvarianten aus Abb. 3.10. Die Leistungsaufnahme der Prozessoren steigt mit steigender Prozessorfrequenz durch die erhöhte Versorgungsspannung an (vgl. [65]). Das Verhältnis der Leistungsaufnahme der Programmvarianten ist invers zum Verhältnis der Energieverbrauchswerte: Eine Programmvariante mit geringerem Energieverbrauch hat eine höhere Leistungsaufnahme. Bei der Vektorisierung der Programmvarianten werden zur Ausführung größere Register und dementsprechend größere Recheneinheiten verwendet. Somit werden bei den vektorisierten Programmvarianten mehr Transistoren in kürzerer Zeit genutzt. In [65] wird dies über eine erhöhte Schaltwahrscheinlichkeit der Transistoren modelliert.

Eine Ausnahme von diesem Verhalten zeigt die *MMCilk* Programmvariante, welche den geringsten Energieverbrauch hat und deren Leistungsaufnahme geringer ist als die der *MMAvxGather** Programmvarianten. Dieser Unterschied lässt sich auf die

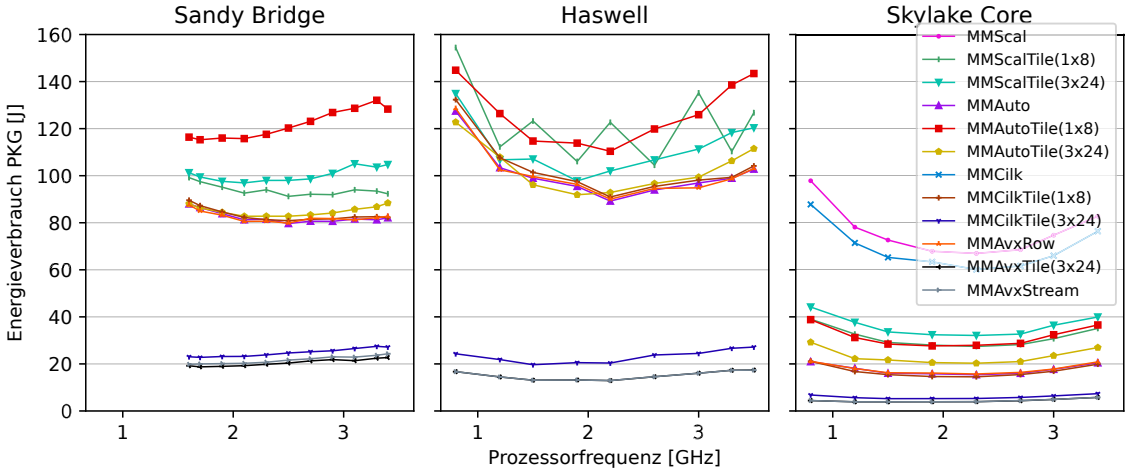

Abbildung 3.12.: Energieverbrauch der Programmvarianten mit zeilenweisem Zugriff auf Matrix B und Schleifen-Blockzerlegung mit unterschiedlichen Blockgrößen auf dem Sandy Bridge, Haswell und Skylake Core Prozessor. Die Blockgröße $(a \times b)$ entspricht der Anzahl der Matrixelemente in jede Dimension. Matrixgröße: 1632×1632. Die Werte für kleine Matrizen sind in im Anhang in Abb. B.7 und für den Skylake XEON Prozessor im Anhang in Abb. B.9 gezeigt.

Nutzung der gather Instruktion zurückführen, die in der *MMCilk* Programmvariante nicht genutzt wird.

Für den **Skylake XEON** Prozessor zeigt sich ein besonderes Verhalten für den Energieverbrauch und die Leistungsaufnahme. Zum Einen sind bei der höchsten einstellbaren Prozessorfrequenz der Energieverbrauch und die Leistungsaufnahme der Programmvarianten stark erhöht (vgl. Abb. 3.10(rechts) und B.4(rechts)). Zum Anderen liegen die resultierenden Werte der Leistungsaufnahme für einige Programmvarianten dauerhaft auf diesem erhöhten Wert (vgl. Abb. 3.11(rechts) und B.5(rechts)). Zusätzlich hat die Programmvariante *MMAvxGatherIdx* bei größeren Matrizen (Abb. B.5) einen Anstieg der Leistungsaufnahme vom niedrigen zum erhöhten Messwert. Dieses Verhalten deutet auf einen Fehler in der Einstellung der Prozessorfrequenz hin, kann jedoch in unabhängigen Messreihen reproduziert werden.

Die Energieverbrauchswerte der **MMAvxVred*** Programmvarianten verhalten sich wie erwartet und sinken bei geringerer Ausführungszeit, die Werte der Leistungsaufnahme steigen dabei entsprechend an. Der Energieverbrauch der *MMAvxVred** Programmvarianten wird im Anhang in Abb. B.6 dargestellt.

Energieverbrauch und Leistungsaufnahme der Programmvarianten mit zeilenweisem Zugriff auf Matrix B

In Abb. 3.12 wird der Energieverbrauch der Programmvarianten mit zeilenweisem Zugriff auf die Elemente der Matrix B dargestellt. Der Energieverbrauch der Programmvarianten verhält sich wie erwartet, sodass eine geringere Ausführungszeit zu einem geringeren Energieverbrauch führt. Eine Ausnahme bildet dabei die *MMAuto-*

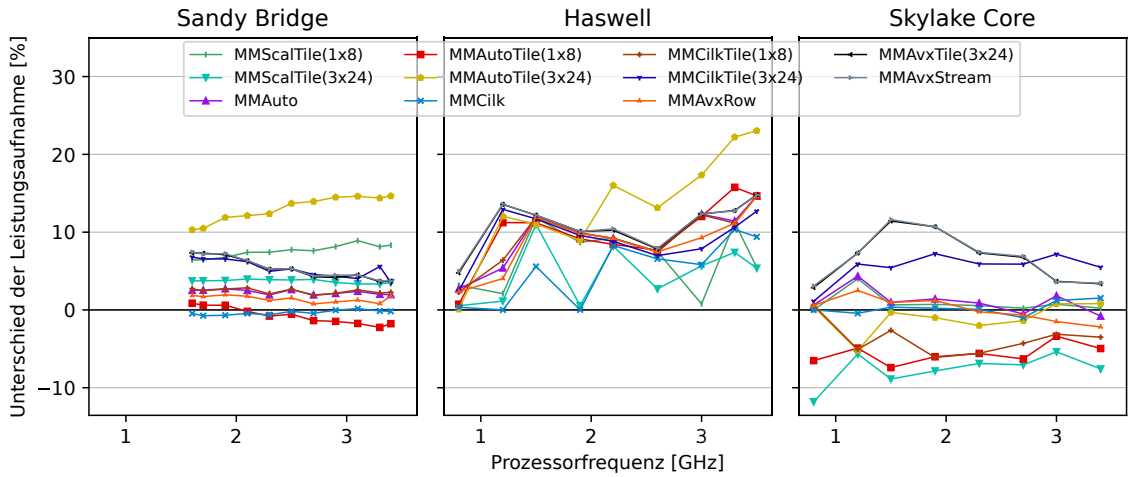

Abbildung 3.13.: Energieverbrauch der Programmvarianten mit zeilenweisem Zugriff auf Matrix B und Schleifen-Blockzerlegung mit unterschiedlichen Blockgrößen auf dem Sandy Bridge, Haswell und Skylake Core Prozessor. Die Blockgröße $(a \times b)$ entspricht der Anzahl der Matrixelemente in jede Dimension. Matrixgröße: 1632×1632. Die Werte für kleine Matrizen sind in im Anhang in Abb. B.8 gezeigt.

Tile Programmvariante mit einer Blockgröße von 3×24, die einen unerwartet höheren Energieverbrauch hat.

Bei den vektorisierten Programmvarianten wird der minimale Energieverbrauch bei einer geringeren Prozessorfrequenz erreicht als bei den seriellen Programmvarianten. Nach [65] begründet sich dies durch die Nutzung von mehr Transistoren (höhere Schaltwahrscheinlichkeit) bei der vektorisierten Berechnung, die zu einer Erhöhung des dynamischen Anteils des Energieverbrauchs führt.

In Abb. 3.13 ist die Leistungsaufnahme der Programmvarianten mit zeilenweisem Zugriff auf Matrix B dargestellt. Auf dem Sandy Bridge und Haswell Prozessor zeigt die Anwendung der Schleifen-Blockzerlegung auf die einzelnen Programmvarianten jeweils den erwarteten Anstieg der Leistungsaufnahme im Verhältnis zum Energieverbrauch der Programmvarianten. Auf dem Skylake Core Prozessor wird durch die Vektorisierung der Energieverbrauch stärker gesenkt, was zu einer Reduktion der Leistungsaufnahme der vektorisierten Programmvarianten führt.

Die Vektorisierung mit Intrinsics in der *MMAvxRow* Programmvariante erreicht auf der Sandy Bridge Architektur eine geringere Leistungsaufnahme als die *MMAuto* Programmvariante mit annähernd gleicher Ausführungszeit und gleichem Energieverbrauch. Auf den Haswell und Skylake Core Prozessoren tritt diese geringere Leistungsaufnahme gegenüber der *MMAuto* Programmvariante nicht auf. Eine Ursache hierfür könnte in der Auswahl der genutzten Instruktionen, wie unterschiedlichen Ladeinstruktionen, bei der automatischen Vektorisierung liegen.

Die Leistungsaufnahme der Programmvarianten aus Abb. 3.13 bei kleineren Matrizen verhält sich wie erwartet, mit Ausnahme der bereits diskutierten *MMAutoTile* Programmvariante, und ist im Anhang in Abb. B.8 dargestellt.

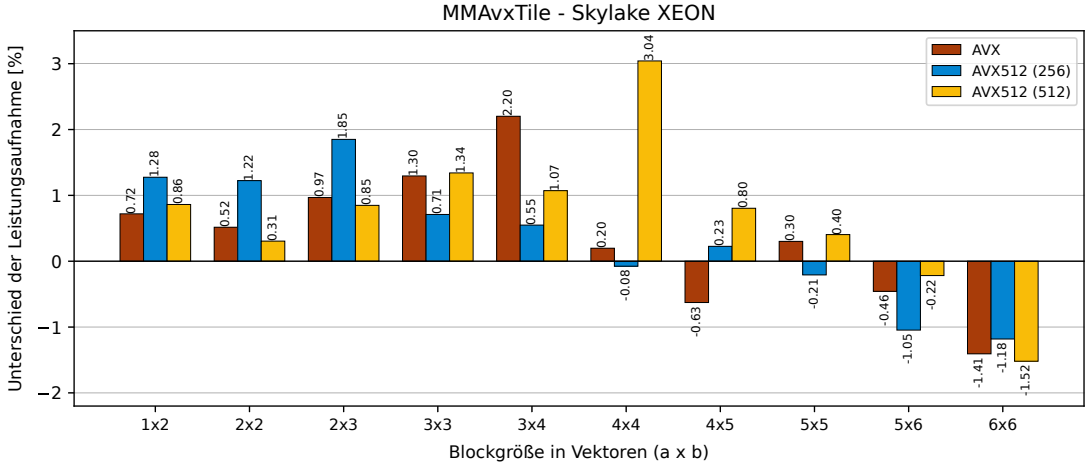

Abbildung 3.14.: Unterschied der Leistungsaufnahme der Programmvarianten mit Schleifen-Blockzerlegung mit unterschiedlichen Blockgrößen im Vergleich zur *MMAvxRow* Programmvariante ohne Schleifen-Blockzerlegung auf dem Skylake XEON Prozessor. Die Blockgröße entspricht der Anzahl der AVX-Vektoren (vgl. Abb. 3.2) in jede Dimension. Matrixgröße: 960×960; Prozessorfrequenz 2,0GHz.

Energieverbrauch und Leistungsaufnahme in Abhängigkeit zur Blockgröße

Die Auswahl der Blockgröße bei der Schleifen-Blockzerlegung hat einen Einfluss auf die Ausführungszeit der Programmausführung. Der Energieverbrauch (Anhang Abb. B.10) verhält sich dabei gegenüber der Ausführungszeit wie erwartet.

Abbildung 3.14 zeigt den Unterschied der Leistungsaufnahme der *MMAvxTile* Programmvariante bei verschiedenen Blockgrößen im Vergleich zur *MMAvxRow* Programmvariante (Blockgröße 1×1). Die Leistungsaufnahme steigt zunächst mit der Blockgröße an, was aus der besseren Ausnutzung der Recheneinheiten durch die Programmvariante folgt. Ab einer Blockgröße, bei der Register für die Berechnung eines Blockes zurück geschrieben werden, sinkt die Leistungsaufnahme der Programmvariante und führt dabei zu einer reduzierten Leistungsaufnahme des Prozessors gegenüber der *MMAvxRow* Programmvariante. Insgesamt wird dadurch für die *MMAvxTile* Programmvariante mit solchen Blockgrößen gegenüber der *MMAvxRow* eine reduzierte Ausführungszeit, ein reduzierter Energieverbrauch und eine reduzierte Leistungsaufnahme erreicht.

3.4. Zusammenfassung der Vektorisierung Matrix-Multiplikation

Die Vektorisierung der Matrix-Multiplikation bietet die Möglichkeit, verschiedene Programmvarianten in Bezug auf den Speicherzugriff, eine vektorisierte Reduktionsoperation und ihrem Verhalten bei unterschiedlicher Ausnutzung des Caches zu untersuchen. Die kürzeste Ausführungszeit und der geringste Energieverbrauch wird dabei mit

der Anwendung der Schleifen-Blockzerlegung auf ein mit AVX-Intrinsics vektorisiertes Programm bei einer auf die Registeranzahl angepassten Blockgröße erreicht.

Die Untersuchungen zeigen, dass die Schleifen-Blockzerlegung eine wichtige Programmtransformation bei der Implementierung der Matrix-Multiplikation ist. Die Anwendung der Schleifen-Blockzerlegung erhöht dabei die Cache-Ausnutzung, stellt einen zeilenweisen Datenzugriff her und macht die Ausführung mit beliebigen Matrixgrößen unabhängig von der Cache-Größe des Prozessors. Die optimale Blockgröße ist dabei Abhängig von der Anzahl der verfügbaren Vektorregister.

Bei der Nutzung einer gather Instruktion führt die Verwendung eines konstanten Offset-Vektors in Kombination mit der Adresse des ersten genutzten Elements zur geringsten Ausführungszeit. Liegen die Daten dabei nicht bereits im Cache, begrenzt die verfügbare Speicherbandbreite diese reduzierte Ausführungszeit. Die dafür benötigte Reduktionsoperation kann bis zu einem gewissen Punkt vektorisiert werden. Die Addition der letzten beiden Elemente bei der Reduktion sollte jedoch seriell ausgeführt werden, um eine der Vektor-Umsortierinstruktionen einzusparen.

Die automatische Vektorisierung der Matrix-Multiplikation erreicht nicht ohne weiteres die Qualität der Vektorisierung mit AVX-Intrinsics. Durch die Anpassung der entsprechenden Ausgangsprogramme auf die Struktur eines vektorisierten Programms kann jedoch eine vergleichbare Vektorisierung erzeugt werden.

Der minimale Energieverbrauch der vektorisierten Programmvarianten liegt bei einer geringeren Prozessorfrequenz als bei der seriellen Programmvariante, da der dynamische Anteil des Energieverbrauchs durch die höhere Prozessorausnutzung ansteigt. Aus dem gleichen Grund erhöht sich die Leistungsaufnahme von vektorisierten Programmen im Vergleich zur seriellen Ausführung.

4

Vektorisierung der Gauss-Elimination für AVX-Vektoreinheiten

Die Gauss-Elimination zur Lösung linearer Gleichungssysteme ist ein wesentlicher Bestandteil in verschiedenen wissenschaftlichen Anwendungen. Beispiele für solche Anwendungen finden sich in der Annäherung der Methode der kleinsten Quadrate oder der Berechnung von Matrixdeterminanten, die wiederum in verschiedenen Verfahren zum Beispiel zur Signalverarbeitung eingesetzt werden. Turner zeigt in „Gauss-Elimination: Workhorse of Linear Algebra“[79] einige der Anwendungen, in denen die Gauss-Elimination eingesetzt wird. Ein Bestandteil der Gauss-Elimination ist die Berechnung einer LU-Faktorisierung. Dongarra, Gustavson und Karp bezeichnen die Berechnung der LU-Faktorisierung als Verfahren, das „vermutlich die meiste Rechnerzeit im wissenschaftlichen Umfeld nutzt“[1][20]. Aus der Nutzung der Gauss-Elimination als Subroutine in den vielseitigen Anwendungen des wissenschaftlichen Rechnens ergibt sich die Notwendigkeit für eine möglichst schnelle und effiziente Ausführung der Subroutine.

Die Optimierung der Gauss-Elimination ist seit den Anfängen des Supercomputing ein Bestandteil wissenschaftlicher Forschungen: Die Datenverteilung und Berechnungsreihenfolge bei der LU-Faktorisierung wird in [20] für Supercomputer mit *Pipeline* Ausführungsmodellen und in [59] für den Hauptspeicher des Cray-1 untersucht. Mit der Weiterentwicklung der Architekturmodelle wurden neue Untersuchungen für Rechnerarchitekturen mit verteiltem Speicher [64] und gemeinsamen Adressraum [11] durchgeführt. In [76] wird die Berechnung der LU-Faktorisierung für Vektoreinheiten von CPUs (SSE-Instruktionserweiterung) implementiert. Die Nutzung von heterogenen Rechnerarchitekturen ist heute vor allem im Bereich des wissenschaftlichen Rechnens weit verbreitet. In [15] wird die Berechnung der LU-Faktorisierung auf Coprozessoren und Grafikkarten in Bezug auf den Energieverbrauch bei der Ausführung untersucht. Zusätzlich bieten Bibliotheken, wie PLASMA [2], LAPACK [19], ATLAS [16] oder Intels MKL [39] performance-optimierte Routinen zur Lösung der Gauss-Elimination an.

Die kontinuierliche Optimierung der Gauss-Elimination für die jeweils verfügbaren Ausführungsmodelle und Rechnerarchitekturen spiegelt deren vielseitigen Einsatz im wissenschaftlichen Rechnen wider. In dieser Arbeit werden diverse Möglichkeiten zur Implementierung der Gauss-Elimination für die Ausführung auf modernen

[1] „probably uses the most computer time in a scientific environment“[20]

AVX-Vektoreinheiten entwickelt. Die Untersuchung von Performance (Ausführungszeit) als auch die Energie-Effizienz der verschiedenen Implementierungsvarianten zeigt die Möglichkeit der Optimierung der Gauss-Elimination durch den Einsatz von AVX-Einheiten auf. Teilaspekte der Untersuchungen in diesem Kapitel sind bereits in [46] und [47] veröffentlicht.

4.1. Gauss-Elimination zur Lösung linearer Gleichungssysteme

Bei der Gauss-Elimination ist ein System von n linearen Gleichungen mit n Unbekannten x_k, deren Koeffizienten $a_{i,k}$ und der rechten Seite b_i, mit $1 \leq i,k \leq n$, gegeben (vgl. [63]):

$$\begin{aligned}
\text{1. Gleichung: } & a_{1,1} \cdot x_1 + a_{1,2} \cdot x_2 + \quad \cdots \quad + a_{1,n} \cdot x_n = b_1 \\
\text{2. Gleichung: } & a_{2,1} \cdot x_1 + a_{2,2} \cdot x_2 + \quad \cdots \quad + a_{2,n} \cdot x_n = b_2 \\
& \qquad\qquad\qquad\qquad \vdots \\
\text{n-te Gleichung: } & a_{n,1} \cdot x_1 + a_{n,2} \cdot x_2 + \quad \cdots \quad + a_{n,n} \cdot x_n = b_n
\end{aligned} \tag{4.1}$$

Das Gleichungssystem wird in insgesamt $n-1$ Schritten so transformiert, dass auf der linken Seite einer der Gleichungen eine einzelne Unbekannte x_n verbleibt. Dazu werden in jedem Schritt k einzelne lineare Gleichungen des Gleichungssystems von anderen Gleichungen des Systems subtrahiert, sodass die Unbekannte x_k aus diesen Gleichungen entfernt wird.

In jedem Schritt k wird für die Koeffizienten der Unbekannten x_k der i-ten Gleichung, mit $k < i$, der Eliminationsfaktor $l_{i,k}$ durch Gleichung 4.2 bestimmt:

$$l_{i,k} = \frac{a_{i,k}}{a_{k,k}} \tag{4.2}$$

Die k-te Gleichung wird mit dem Eliminationsfaktor $l_{i,k}$ multipliziert und für $k < i$ von der jeweils i-ten Gleichung subtrahiert. Auf diese Weise entstehen in den Gleichungen $k+1$ bis n für die Unbekannte x_k als Koeffizienten Nullen. Durch Wiederholung des Schrittes beginnend mit der Zeile $k+1$ reduziert sich die Anzahl der Nicht-Null Koeffizienten bis auf einen einzigen verbleibenden Wert in der n-ten Gleichung. Dieser Prozess wird als Vorwärtselimination bezeichnet und ist lösbar, wenn die Determinante der Koeffizienten-Matrix A nicht Null ist und die Koeffizienten-Matrix A nicht singulär ist [24, S.111].

Im Anschluss werden die Werte der Unbekannten x_n bis x_1 durch die sogenannte Rückwärtssubstitution berechnet. Hierzu wird mit der n-ten Gleichung der Wert für x_n berechnet. Durch anschließendes Einsetzen dieses Wertes von x_n in der $n-1$-ten Gleichung, verbleibt nur noch die Unbekannte x_{n-1} in dieser Gleichung. Da durch die Vorwärtselimination in jeder Gleichung k genau eine Unbekannte x_k mehr enthalten ist, als in der Gleichung $k+1$, können die bekannten Werte für die Unbekannten

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & & & \vdots \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \vdots \\ a_{n,n} & \cdots & \cdots & \cdots & a_{n,n} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ \vdots \\ b_n \end{pmatrix}$$

$$\Big\downarrow \text{LU-Faktorisierung}$$

$$\begin{pmatrix} 0 & 0 & \cdots & \cdots & 0 \\ l_{2,1} & 0 & & & \vdots \\ \vdots & l_{3,2} & 0 & & \vdots \\ \vdots & & \ddots & \ddots & \vdots \\ l_{n,1} & \cdots & \cdots & l_{n,n-1} & 0 \end{pmatrix} \cdot \begin{pmatrix} u_{1,1} & u_{1,2} & u_{1,3} & \cdots & u_{1,n} \\ 0 & u_{2,2} & u_{2,3} & \cdots & u_{2,n} \\ \vdots & 0 & u_{3,3} & \cdots & u_{3,n} \\ \vdots & & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & 0 & u_{n,n} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ \vdots \\ b_n \end{pmatrix}$$

Abbildung 4.1.: Illustration des der Transformationen einer LU-Faktorisierung zur Umformung eines linearen Gleichungssystems der Form $Ax = b$ in die Form $LUx = b$ bei der L eine untere Dreiecksmatrix und U eine obere Dreiecksmatrix ist.

$x_{k+1} \dots x_n$ jeweils in die k-te Gleichung eingesetzt werden. Das Einsetzen der bereits bekannten Werte für die Unbekannten wird für die Gleichungen x_n bis x_1 wiederholt, sodass die Werte für alle Unbekannten bestimmt sind.

4.1.1. Algorithmus zur Gauss-Elimination

Das lineare Gleichungssystem aus Gleichung 4.1 lässt sich als Matrix-Vektor Produkt der folgenden Form darstellen:

$$Ax = b, \text{ mit } A \in \mathbb{R}^{n \times n} \text{ und } x, b \in \mathbb{R}^n \tag{4.3}$$

Die Koeffizienten der einzelnen Gleichungen des linearen Gleichungssystems werden jeweils in den Zeilen der Matrix A abgelegt. Bei der Vorwärtselimination wird die Matrix A in eine obere Dreiecksform überführt, sodass $Ux = b'$ gilt, wobei U die Matrix in oberer Dreiecksform ist und b' die rechte Seite nach der Vorwärtselimination.

Die Faktoren $l_{i,k}$, die zur Berechnung dieser Überführung genutzt werden, werden in einer unteren Dreiecksmatrix L gespeichert, sodass folgendes gilt:

$$Ax = LUx = b, \text{ mit } A, L, U \in \mathbb{R}^{n \times n} \text{ und } x, b \in \mathbb{R}^n \tag{4.4}$$

Die Aufteilung in die beiden Matrizen L und U wird auch als LU-Faktorisierung bezeichnet. Abbildung 4.1 zeigt die Struktur der Matrix A, sowie die Matrizen L und U der LU-Faktorisierung.

Für die Rückwärtssubstitution wird das System $Ux = b'$ zeilenweise für die Zeilen n bis 1 aufgelöst. Die Elemente des Vektors x werden dabei in der Reihenfolge $x_n, x_{n-1} \dots x_1$ nach der folgenden Formel berechnet:

$$x_k = \frac{1}{u_{k,k}} \left(b'_k - \sum_{j=k+1}^{n} u_{k,j} \cdot x_j \right) \tag{4.5}$$

Für die n-te Zeile enthält die Summe keine Summanden, wodurch x_n direkt aus b'_k und $u_{n,n}$ bestimmt werden kann. In jeder weiteren Zeile kommt durch die Dreiecksform der Matrix U jeweils eine Unbekannte in der Summenformel hinzu, sodass pro Zeile eine Unbekannte berechnet werden kann. Nach n Iterationen sind alle Werte des Vektors x bestimmt und das lineare Gleichungssystem gelöst.

Bei der Berechnung der Eliminationsfaktoren $l_{i,k}$ muss der Koeffizient $a_{k,k}$ einen Wert ungleich Null haben, damit die Berechnung $\frac{a_{i,k}}{a_{k,k}}$ zulässig ist. Zusätzlich kann die Berechnung der Eliminationsfaktoren $l_{i,k}$ zu sehr großen Werten führen. Diese großen Eliminationsfaktoren können im Anschluss bei der Berechnung der Elemente der i-ten Zeile der Matrix A ($a_{i,\_}$) zu Rundungsfehlern führen. Golub und Van Loan führen in [24, S. 122ff] den Beweis hierzu. Der Rundungsfehler kann verringert werden, wenn für die Eliminationsfaktoren kleinere Werte errechnet werden. Um sicher zu stellen, dass immer der kleinstmögliche Eliminationsfaktor berechnet wird und der Wert von $a_{k,k}$ nicht Null ist, kann eine Pivotisierung der entsprechenden Matrixzeilen erfolgen. Dazu wird der größte Koeffizient $a_{piv,k}$ mit der folgenden Eigenschaft gesucht:

$$a_{piv,k} = max(a_{i,k}), \text{ mit } k \leq i \leq n \tag{4.6}$$

Im Fall $piv \neq k$ werden die beiden Zeilen $a_{piv,\_}$ und $a_{k,\_}$, sowie die Elemente b_{piv} und b_k der rechten Seite, miteinander getauscht. Die Berechnung von $l_{i,k}$ ergibt nun den kleinstmöglichen Wert für alle Zeilen, da für jedes $l_{i,k}$ folgendes gilt:

$$a_{i,k} \leq a_{k,k}, \text{ mit } k < i \leq n$$

4.1.2. Implementierung der Gauss-Elimination

Zur Implementierung der Gauss-Elimination wird die Matrix Darstellung des Gleichungssystems gewählt. Für die Vorwärtselimination entsteht ein Schleifennest mit den folgenden drei Schleifen:

- Iteration entlang der Diagonalen der Matrix A
- Iteration über die Zeilen $a_{i,\_}$ unterhalb des Diagonalelements $a_{k,k}$
- Iteration über die Elemente der Zeilen $a_{i,\_}$

Das Bearbeitungsschema der Vorwärtselimination ist in Abb. 4.2 für den Schritt k, $1 \leq k < n$ dargestellt. Die äußerste Schleife des Schleifennestes iteriert entlang der

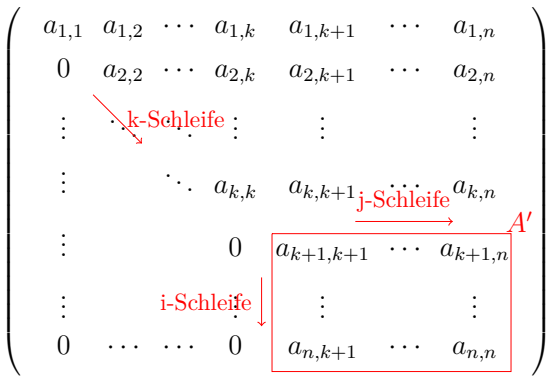

Abbildung 4.2.: Illustration des Bearbeitungsschemas der Matrix durch die Gauss-Elimination. Die *k-Schleife* in Zeile 2 von Alg. 4.1 iteriert entlang der Diagonalelemente $a_{k,k}$, die *j*- und *i-Schleifen* iterieren über jedes Element $a'_{i,j}$ mit $k < i, j \leq n$ der Teilmatrix unter dem Diagonalelement.

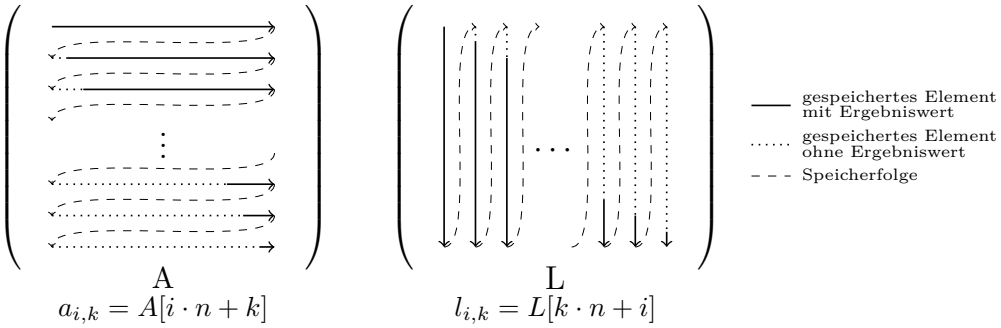

Abbildung 4.3.: Reihenfolge der Matrixelemente bei Abspeicherung in eindimensionalen Arrays für die Matrizen A und L. Die Pfeilrichtung zeigt die Folge der Elemente im Speicher. Der eindimensionale Index wird durch den Zeilen- und Spaltenindex der jeweiligen Matrix berechnet. Ergebniswerte der LU-Faktorisierung werden hervorgehoben.

Diagonalen der Matrix und die beiden inneren Schleifen iterieren über die Elemente der gezeigten Teilmatrix A'.

Abbildung 4.3 zeigt die Ablage der Matrixelemente im Speicher, sowie die Indexberechnungen der beiden Matrizen A und L. Beide Matrizen werden als eindimensionale Arrays im Speicher abgelegt. Der entsprechende zwei-dimensionale Elementindex (i, k) wird mit dem Zeilenindex i, dem Spaltenindex k und der Zeilengröße n in einen eindimensionalen Indexzugriff umgewandelt:

$$\begin{aligned} a_{i,k} &= A[i \cdot n + k] \\ &\quad \text{mit } 1 \leq i, k \leq n \end{aligned} \tag{4.7}$$

Die Matrix L wird spaltenweise (*engl. column major*) abgespeichert, um einen effizienten Zugriff auf die Datenelemente durch die vektorisierten Programmvarianten zu ermöglichen. Der zwei-dimensionale Elementindex (i, k) errechnet sich dabei mit dem

```
1  // Schleife über die Diagonalelemente a_{k,k}:
2  for k = 0 < n-1; k+=1
3    Pivotsuche und Zeilentausch
4    // Schleife über die Zeilen unterhalb des Diagonalelements:
5    for i = k+1 < n; i+=1
6      L[k*n+i] = A[i*n+k] / A[k*n+k] // l_{i,k} = a_{i,k}/a_{k,k}
7      // Schleife über die Elemente a_{i,_} der i-ten Zeile:
8      for j = k+1 < n; j+=1
9        A[i*n+j] = A[i*n+j] - A[k*n+j] * L[k*n+i] // a_{i,j} = a_{i,j} - l_{i,k} · a_{k,j}
10     b[i] = b[i] - b[k] * L[k*n+i] // b_i = b_i - l_{i,k} · b_k
11 Rückwärtssubstitution
```

Algorithmus 4.1: GaussScal Programmvariante: Algorithmus der Gauss- Elimination in C-Pseudocode. Die Matrix A wird in eine obere Dreiecksform transformiert, sodass durch die Rückwärtssubstitution die Werte der Unbekannten $x_{i,k}$ bestimmt werden können. Der Indexzugriff auf die Matrix L ist auf die spaltenweise Abspeicherung angepasst. Nach [24], Notation angepasst.

Zeilenindex i, dem Spaltenindex k und der Spaltengröße n:

$$l_{i,k} = L[k \cdot n + i] \quad \text{mit } 1 \leq i, k \leq n \tag{4.8}$$

Algorithmus 4.1 zeigt den Algorithmus zur Gauss-Elimination und implementiert das Schleifennest mit den folgenden drei Schleifen:

- Die *k-Schleife* in Zeile 2 iteriert schrittweise über die Diagonalelemente $a_{k,k}$ der Matrix A.
- Die *i-Schleife* in Zeile 5 iteriert über die Zeilen der Teilmatrix A'.
- Die *j-Schleife* in Zeile 8 iteriert über die Elemente der Zeilen.

Innerhalb der *k-Schleife* beginnt jeder Schritt mit einer Pivotsuche, bei der das größte Element $a_{piv,k}$ unter dem Diagonalelement $a_{k,k}$ gesucht wird. Im Anschluss wird ein Zeilentausch der k-ten Zeile $a_{k,\_}$ mit der Zeile des Pivotelements $a_{piv,\_}$ durchgeführt. Innerhalb der *i-Schleife* wird für jede Zeile $a_{i\_}$ mit $i > k$ der Eliminationsfaktor $l_{i,k}$ (Zeile 6) berechnet und in der Matrix L abgelegt.

Die *i-* und *j-Schleife* iterieren über die Elemente der Teilmatrix $A' \in \mathbb{R}^{n-k \times n-k}$ aus Abb. 4.2. Mit dem Eliminationsfaktor $l_{i,k}$ werden die Elemente der Zeile $a_{i,\_}$ mit Hilfe der *j-Schleife* in Zeile 8 berechnet. Zusätzlich wird in Zeile 10 das Element b_i des Vektors b auf die gleiche Weise neu berechnet. Nach $n-1$ Schritten (Iterationen der *k-Schleife*) ist die Matrix A in die obere Dreiecksmatrix U transformiert, sowie die untere Dreicksmatrix L aus Eliminationsfaktoren berechnet. Die Matrix A wird in der Implementierung überschrieben und die Null-Elemente des unteren Dreiecks der Matrix A werden nicht explizit berechnet.

Im Anschluss wird die Rückwärtssubstitution ausgeführt, bei der die Werte für x berechnet werden (vgl. Anhang Alg. C.1).

4.1.3. Eigenschaften des Algorithmus zur Gauss-Elimination im Hinblick auf eine Vektorisierung

Der Algorithmus der Gauss-Elimination hat einige Eigenschaften die im Folgenden im Hinblick auf eine Vektorisierung betrachtet werden. Durch die Struktur der Gauss-Elimination, in der die einzelnen Zeilen der Matrix A nacheinander bearbeitet werden, bietet sich eine entsprechende Vektorisierung entlang dieser Berechnung an. Mit der Vektorisierung werden dabei mehrere Elemente der Zeile parallel berechnet, bevor die nächste Zeile berechnet wird.

Bei der Gauss-Elimination werden die Ergebnisse der Berechnungen an die gleiche Stelle zurück geschrieben, von der die Eingabewerte der Matrix A gelesen wurden. Durch dieses Vorgehen reduziert sich die Menge der Datenelemente, die zwischen Speicher und CPU transferiert werden, im Vergleich zu einem Algorithmus bei dem dies nicht möglich ist. Somit wird für die jeweiligen Berechnungen nur eine geringe Anzahl an Speichertransferoperationen benötigt. Dies kann sich positiv auf die Ausführungszeit der vektorisierten Programmvarianten auswirken, da bei der Vektorisierung typischerweise mehr Daten in gleicher Zeit verarbeitet werden.

Bei der Vektorisierung von Programmen hat die Speicheradresse des zugegriffenen Datenelements einen Einfluss auf die nutzbaren AVX-Instruktionen. Für einen effizienten Datenzugriff sollten die Datenelemente an einer ausgerichteten (*engl. aligned*) Speicherstelle geladen bzw. gespeichert werden (vgl. Anhang Abschn. A.2).

Mit der Iteration der *k-Schleife* ändert sich jeweils das erste betrachtete Element der Zeilen von Matrix A. Durch die zeilenweise Vektorisierung der Gauss-Elimination bedeutet dies, dass keine Annahmen über die Eigenschaften der Speicherstelle in Bezug auf die Ausrichtung (*engl. Alignment*) getroffen werden können. Sollen für die Vektorisierung der Gauss-Elimination ausgerichtete Speicherstellen genutzt werden, so muss das entsprechende Alignment für jeden Zugriff auf eine Zeile der Matrix explizit hergestellt werden.

Die *i-* und *j-Schleife* beginnen die Berechnung der Teilmatrix A' aus Abb. 4.2 jeweils beim Wert $k+1$. Die Iterationen der *k-Schleife* verkleinern somit den Iterationsraum der inneren Schleifen nach jedem Schritt k. Dadurch ergibt sich in jedem Schritt k eine unterschiedliche Anzahl verwendeter Matrixelemente. Die Anzahl der Matrixelemente, die in jeder Zeile verwendet werden, ist somit nicht in jedem Schritt durch die Anzahl der Datenelemente eines Vektorregisters teilbar. Dies erfordert die gesonderte Behandlung der letzten Matrixelemente einer Zeile (vgl. Anhang Abschn. A.3). Zusammen mit dem fehlenden Alignment lassen sich dadurch verschiedene Lade- und Speicheroperationen hinsichtlich ihrer Ausführungseigenschaften untersuchen.

Ansatz	Variante	Implementierungsaspekte	Referenz
—	**GaussScal**	Nicht vektorisierte (serielle) Basisvariante	Alg. 4.1; Abb. 4.2
automatisch	**GaussAuto**	Automatisch durch Compiler vektorisiert	Alg. 4.2
	GaussCilk	Vektorisierung mittels Cilk Array-Notation	Alg. 4.3; Abb. 4.4
AVX Intrinsics	**GaussStoreU**	Basisvariante mit AVX-Intrinsics	Alg. 4.4; Abb. 4.5
	GaussRem	Berechnung der letzten Matrixelemente der Zeile durch skalare Instruktionen	Alg. 4.5; Abb. 4.5
	GaussMaskload	Nutzung von maskierten Ladeoperationen	Alg. 4.6; Abb. 4.5
	GaussStore	Anpassung des Iterationsraumes zur Nutzung von Ladeoperationen an ausgerichteten Speicherstellen	Alg. 4.7; Abb. 4.6
	GaussStream	Nutzung von non-temporal Schreiboperationen	Alg. 4.8; Abb. 4.6

Tabelle 4.1.: Übersicht über die Programmvarianten der Gauss-Elimination. Die Programmvarianten sind nach dem verwendeten Vektorisierungsansatz Gruppiert und die wichtigsten Implementierungsaspekte werden kurz zusammengefasst. Die Beschreibung der Programmvariante und das verwendete Bearbeitungsschema sind referenziert.

4.2. AVX-Programmvarianten der Gauss-Elimination

Für die **GaussScal** Programmvariante der Gauss-Elimination wird der Pseudocode aus Alg. 4.1 übersetzt. Die **GaussScal** Programmvariante wird dabei zum Vergleich einer seriellen (nicht vektorisierten) mit den vektorisierten Programmvarianten genutzt.

Tabelle 4.1 zeigt die Programmvarianten der Gauss-Elimination, die in diesem Kapitel erstellt und untersucht werden. Die vektorisierten Programmvarianten werden in diesem Abschnitt vorgestellt.

4.2.1. Gauss-Elimination mit automatisch vektorisierten Programmvarianten

Moderne Compiler sind in der Lage Programmcode automatisch für die Nutzung von Vektorerweiterungen, wie AVX, zu transformieren. Die Vektorisierung kann dabei auf verschiedene Weisen im Programmcode unterstützt bzw. angegeben werden. Im Folgenden werden zwei Möglichkeiten hierfür an Programmvarianten der Gauss-Elimination gezeigt.

```
1 // Signalisiert  dem Compiler die j-Schleife zu vektorisieren
2  #pragma simd
3     // Schleife  über die  Elemente a_{i,_}  der i-ten Zeile:
4     for j = k+1 < n; j+=1
5       A[i*n+j] = A[i*n+j] - A[k*n+j] * L[k*n+i] // a_{i,j} = a_{i,j} - l_{i,k} · a_{k,j}
```

Algorithmus 4.2: GaussAuto Programmvariante: Zur Unterstützung der automatischen Vektorisierung durch den Compiler wird vor der *j-Schleife* ein `#pragma simd` eingefügt.

4.2.1.1. Die automatisch generierte *GaussAuto* Programmvariante

Algorithmus 4.2 zeigt die Anpassung für die automatische Vektorisierung in der **GaussAuto** Programmvariante. Die **GaussAuto** Programmvariante wird aus der *GaussScal* Programmvariante abgeleitet. Hierzu wird vor der *j-Schleife* in Zeile 4 ein `#pragma` simd (Zeile 2) eingefügt. Dieses `#pragma` simd zeigt dem Compiler welche der Schleifen des Schleifennestes vektorisiert werden soll. Die Vektorisierung der Berechnung erfolgt dadurch entlang der Zeilen von Matrix A.

4.2.1.2. Die *GaussCilk* Programmvariante in Cilk Plus Array-Notation

Die Intel Cilk Plus Erweiterung zur Programmiersprache C stellt die Array-Notation zur Verfügung. Bei der Array-Notation wird der Zugriff auf die Elemente eines Arrays in der folgenden Form dargestellt:

A[Startindex:Anzahl_Elemente]

Der Intel-C-Compiler wandelt das Statement in Array-Notation automatisch so um, dass die so bestimmten Elemente des Arrays genutzt werden. Die Array-Notation hat dabei eine ähnliche Semantik wie eine entsprechende Schleife, die über das Array iteriert und jedes einzelne Element nutzt. Bei der Bearbeitung durch eine Schleife ist jedoch durch die Iterationsfolge eine Reihenfolge der Elementzugriffe vorgegeben. Diese Reihenfolge entfällt bei der Array-Notation, wodurch die einzelnen Elemente des Statements in Array-Notation unabhängig voneinander berechnet werden. Die daraus resultierende geringere Anzahl an Datenabhängigkeiten erlaubt dem Compiler weitreichendere Programmtransformationen vorzunehmen, als bei der Vektorisierung einer Schleife. Der Compiler vektorisiert den C/Cilk Programmcode automatisch für den verfügbaren Vektorinstruktionssatz, wie beispielsweise AVX.

Algorithmus 4.3 zeigt die Implementierung der **GaussCilk** Programmvariante. Zur Erstellung der **GaussCilk** Programmvariante mit Intel Cilk Plus wird die *j-Schleife* des Algorithmus (Zeilen 8 und 9 aus Alg. 4.1) entfernt und die entsprechenden Berechnungen durch ein Statement in Array-Notation ersetzt. Die Zeilen 2 bis 6 bleiben unverändert. Die einzelnen Elemente der i-ten Zeile von werden in Alg. 4.3 durch die Array-Notation berechnet. Dadurch erfolgt die Vektorisierung entlang der Zeilen von Matrix A.

```
1  // Schleife über die Diagonalelemente a_{k,k}:
2  for k = 0 < n-1; k+=1
3    Pivotsuche und Zeilentausch
4    // Schleife über die Zeilen unterhalb des Diagonalelements:
5    for i = k+1 < n; i+=1
6      L[k*n+i] = A[i*n+k] / A[k*n+k] // l_{i,k} = a_{i,k}/a_{k,k}
7      // Elemente a_{i,_} der i-ten Zeile:
8      si = i*n+k+1;        // Startindex Zeile i
9      sk = k*n+k+1;        // Startindex Zeile k
10     lng = n - (k + 1); // Anzahl Elemente pro Zeile
11     A[si:lng] = A[si:lng] - A[sk:lng] * L[k*n+i] // a_{i,_} = a_{i,_} - l_{i,k} · a_{k,_}
12   // b_ = b_ - l_{_,k} · b_k
13   b[k+1:n-k-1] = b[k+1:n-k-1] - b[k] * L[(k+1)*n+k:n-k-1] :
14 Rückwärtssubstitution
```

Algorithmus 4.3: GaussCilk Programmvariante: Cilk Plus Implementierung einer Gauss-Elimination. Die Variablen si und sk geben den Index des ersten betrachteten Elements der i-ten bzw. k-Zeile an. Die Berechnung von $a_{i,\_}$ ist in der Array-Notation angegeben und wird durch den Compiler vektorisiert. Die veränderten Programmzeilen im Vergleich zur *GaussScal* Programmvariante in Alg. 4.1 sind hervorgehoben.

Das Statement in Array-Notation wird in jedem Schritt k für jede Matrixzeile i ausgeführt. Für die Umwandlung der *j-Schleife* in ein Statement in Array-Notation werden zuerst die benötigten Array-Indizes und -Längen berechnet:

- Der Index des Elements $a_{i,k+1}$ der Matrix A wird in Zeile 8 berechnet und in si gespeichert. Der Index si entspricht dem Index des ersten Elements $a_{i,j}$, mit $j = k + 1$, der entfernten *j-Schleife*.
- In Zeile 9 wird der Index des Elements $a_{k,k+1}$ analog dazu berechnet und in sk gespeichert.
- In Zeile 10 wird für die Array-Notation zusätzlich die Anzahl der zu berechnenden Elemente der Matrixzeile berechnet und in die Variable lng gespeichert.

In Zeile 11 werden die Elemente $a_{k,k+1}$ bis $a_{k,n}$ jeweils mit dem Eliminationsfaktor $l_{i,k}$ der Zeile i multipliziert. Im Anschluss werden die Zwischenergebnisse von den Elementen $a_{i,k+1}$ bis $a_{i,n}$ subtrahiert und an die Stelle der ursprünglichen Elemente $a_{i,k+1}$ bis $a_{i,n}$ abgespeichert.

Die Berechnung des Vektors b wird in Alg. 4.3 aus der *i-Schleife* heraus genommen und als Array-Notation innerhalb der *k-Schleife* formuliert. Durch die spaltenweise Speicherung von Matrix L kann bei der Berechnung von b in Zeile 13 auf ein zusammenhängendes Stück Speicher zugegriffen werden.

Abbildung 4.4 zeigt die genutzten Vektoren der Matrix A. Vergleichbar zum Bearbeitungsschema der *GaussScal* Programmvariante in Abb. 4.2 werden die genutzten

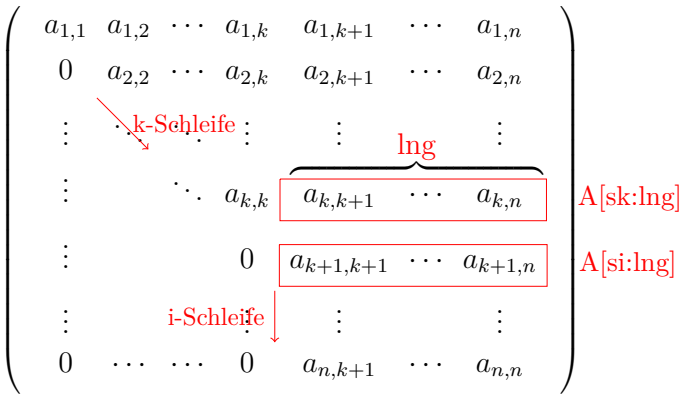

Abbildung 4.4.: Bearbeitungsschema der Matrix A für die **GaussCilk** Programmvariante in Schritt k mit $i = k + 1$. Die Zeilen der Matrix A sind in Array-Notation angegeben: A[sk:lng] bezeichnet die Matrixelemente $a_{k,k+1}$ bis $a_{k,n}$ und A[si:lng] bezeichnet die Matrixelemente $a_{i,k+1}$ bis $a_{i,n}$. Der Index des ersten Elements wird in sk bzw. si gespeichert und die Variable lng enthält die Anzahl der Elemente der beiden Vektoren. Im Unterschied zum Bearbeitungsschema der *GaussScal* Programmvariante in Abb. 4.2 wird die *j-Schleife* durch die Nutzung von Statements in Array-Notation ersetzt.

Matrixelemente für Schritt k gezeigt. Die Elemente von A[sk:lng] und A[si:lng] werden zur Berechnung der $i-$ten Zeile, mit $i = k + 1$, genutzt. Dabei entspricht A[sk:lng] den Elementen $a_{k,k+1}$ bis $a_{k,n}$ und A[si:lng] den Elementen $a_{i,k+1}$ bis $a_{i,n}$ der Matrix A. Da durch die Array-Notation alle benötigten Elemente der Matrixzeile angegeben werden, entfällt die *j-Schleife*.

4.2.2. Programmvarianten der Gauss-Elimination mit AVX-Intrinsics

Die automatische Vektorisierung durch den Compiler ermöglicht eine einfache Erstellung von SIMD Programmen. Die Fähigkeiten des Compilers sind jedoch begrenzt, sodass manche Probleme und Strukturen nicht automatisch vektorisiert werden können [71]. Bei der expliziten Erstellung von vektorisierten Programmen können intrinsische Funktionen (kurz: Intrinsics) genutzt werden, die dem Programmierer die Möglichkeit bieten, die einzelnen Instruktionen des SIMD Programms direkt anzugeben. Intrinsische Funktionen sind Funktionen, die durch den Compiler bereit gestellt werden. Im Fall von AVX kapseln intrinsische Funktionen meist einen entsprechenden AVX-Assembler Befehl in Form einer C-Funktion (vgl. [41]). Bei der Erstellung von SIMD Programmen mit Hilfe von Intrinsics werden die Berechnungen des Programms konkret in die Operationen auf den Vektorregistern zerlegt.

Die AVX-Intrinsics werden über Header-Dateien durch den Compiler zur Verfügung gestellt und werden explizit im Programmcode angegeben, um die entsprechende Berechnung durchzuführen. Die Intrinsics haben dabei die folgende allgemeine Form:

mm<size><function>_<type>()

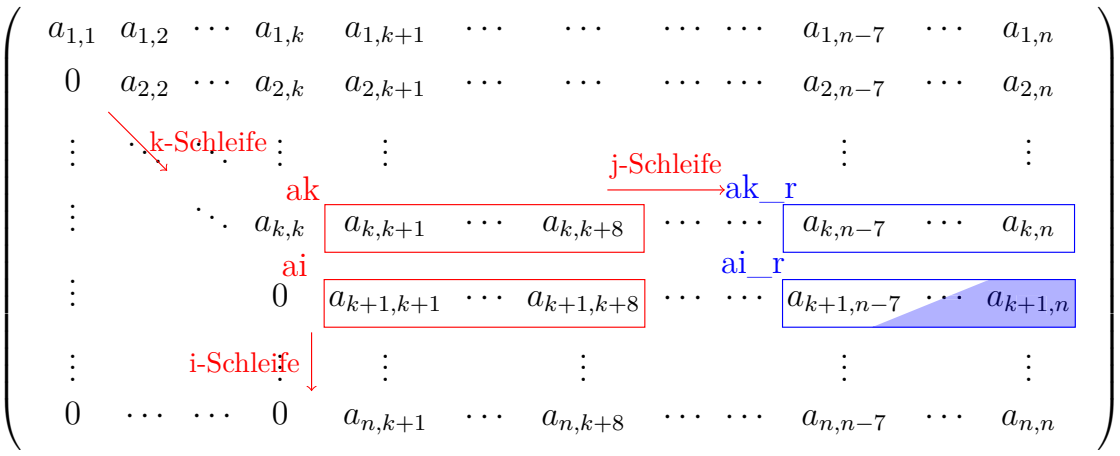

Abbildung 4.5.: Bearbeitungsschema der Matrix A für die Programmvarianten **GaussStoreU**, **GaussMaskload** und **GaussRem** in Schritt k mit $i = k + 1$ und $j = k + 1$. Die Iterationen der *j-* und *i-Schleife* und deren genutzte Vektorvariablen ak und ai sind in rot hervorgehoben.
Die Berechnung der letzen Matrixelemente einer Zeile wird gesondert behandelt und ist in blau hervorgehoben. Dazu werden die letzten 8 Elemente der Zeilen in die Vektorvariablen ak_r und ai_r geladen und für die Berechnung genutzt. Bei der Speicherung werden nur die bisher nicht berechneten Elemente (blaue Füllung) in den Speicher geschrieben.

Zur Unterscheidung der verschiedenen Registergrößen repräsentiert <size> die Anzahl der Bits (128, 256, 512), die für diese Operation genutzt werden. Die nutzbare Größe ist abhängig von der Registergröße der verfügbaren Prozessortechnologie. Als <function> wird ein repräsentativer Name der ausgeführten Operation genutzt, wie add für eine Addition. Die Datentypen, auf denen eine intrinsische Funktion operiert, werden durch <type> angegeben. Ein Beispiel für <type> ist der Suffix _ps, der für einen Vektor aus `float` Zahlen (*engl. packed single precision*) steht. Eine Auflistung der in dieser Arbeit verwendeten intrinsischen Funktionen, sowie deren Bedeutung findet sich im Anhang in Abschn. A.5.

Zur Nutzung der intrinsischen Funktionen werden Vektordatentypen bereit gestellt, wie der __m256 Datentyp für einen Vektor mit 8 `float` Werten. Die skalaren Daten werden in einen solchen Vektordatentyp geladen (*engl. loaded/packed*), damit im Anschluss damit Rechenoperationen durchgeführt werden können. Nach Abschluss der Berechnungen werden die Daten aus den Vektordatentypen wieder auf skalare Speicherstellen gespeichert (*engl. stored/unpacked*).

Für die AVX Implementierungen wird die Vektorisierung entlang der einzelnen Zeilen der Matrix A durch die explizite Zerlegung mit Hilfe von intrinsischen Funktionen umgesetzt.

4.2.2.1. Die *GaussStoreU* Programmvariante

Bei der **GaussStoreU** Programmvariante wird die serielle Gauss-Elimination der *GaussScal* Programmvariante mit Intrinsics vektorisiert. Bei der Implementierung werden keine Optimierungen zur Nutzung von ausgerichteten Speicherzellen vorgenommen und ausschließlich Lade- und Speicheroperationen für unausgerichtete Speicherzellen (loadu und storeu) genutzt.

Das Bearbeitungsschema der **GaussStoreU** Programmvariante wird in Abb. 4.5 für den Schritt k, mit $i = k+1$, dargestellt. Innerhalb der *j-Schleife*, die über die Elemente der Zeile i der Matrix A iteriert, wird statt einem skalaren Wert jeweils ein Vektor mit 8 `float` Werten geladen. Die Zeile wird somit in Blöcke mit je 8 Matrixelementen zerhackt und die *j-Schleife* iteriert über diese Blöcke anstatt der einzelnen Matrixelemente. Auf diese Weise werden zu Beginn der *j-Schleife* die beiden in Abb. 4.5 rot hervorgehobenen Vektoren ak und ai geladen. Die Berechnungen werden im Folgenden mit diesen 8 `float` Werten durchgeführt. In der darauf folgenden Iteration der *j-Schleife* wird der nächste Block mit 8 `float` Werten geladen und berechnet.

Der letzte Vektor jeder Zeile muss gesondert behandelt werden, da die Anzahl der genutzten Matrixelemente pro Zeile nicht in jedem Schritt k durch 8 teilbar ist. Hierfür wird statt des nächsten Blockes mit 8 `float` Werten ein Block mit den letzten 8 `float` Werten (blauer Rahmen in Abb. 4.5) der Zeile geladen. Dieser Block der letzten 8 Werte kann dabei bereits betrachtete Matrixelemente enthalten. Die Berechnungen werden wie vorher auf den beiden Vektoren ausgeführt. Bei der Abspeicherung der Werte werden jedoch nur solche Werte abgespeichert, deren Matrixelement nicht bereits in einem vorherigen Schritt berechnet wurden (blaue Füllung von ai_r in Abb. 4.5). Dieses Vorgehen wird gewählt, damit die Ladeoperationen auf allen 8 Elementen des Blockes ausgeführt werden kann, ohne dabei auf uninitialisierten Speicher zuzugreifen.

Erläuterung zu Algorithmus 4.4

Das beschriebene Bearbeitungsschema wird mit AVX-Intrinsics implementiert und ist in Alg. 4.4 dargestellt. Bei der Implementierung werden jeweils 8 Elemente der i-ten Zeile simultan berechnet.

- Die Zeilen 2 bis 7 werden nicht verändert.
- Das seriell berechnete $l_{i,k}$ (Zeile 7) wird in Zeile 8 für die nachfolgenden Berechnungen mittels einer Broadcast-Instruktion an alle Stellen der Vektorvariablen lv kopiert.
- Im Schleifenrumpf der *j-Schleife* in Zeile 11 werden 8 `float` Werte simultan berechnet (vgl. Abb. 4.2).
- In den Zeilen 12 und 13 werden die Daten der Vektorvariablen `ak` und `ai` geladen. Die Vektorvariable ak enthält damit die Werte der Matrixelemente $a_{k,j} \dots a_{k,j+7}$ (kurz $a_{k,[]}$) und ai die Werte $a_{i,j} \dots a_{i,j+7}$ (kurz $a_{i,[]}$).

```
1  // Schleife über die Diagonalelemente a_{k,k}:
2  for k = 0 < n-1; k+=1
3    Pivotsuche und Zeilentausch
4
5    // Schleife über die Zeilen unterhalb des Diagonalelements:
6    for i = k + 1 < n; i+=1
7      L[k*n+i] = A[i*n+k] / A[k*n+k]; // l_{i,k} = a_{i,k}/a_{k,k}
8      lv = _mm256_broadcast_ss(&L[k*n+i]);
9
10     // Schleife über die Blöcke a_{i,[]} mit je 8 Elementen der i-ten Zeile:
11     for j = k + 1 < n - 8; j+=8
12       __m256 ak = _mm256_loadu_ps(&A[k*n+j]); // a_{k,[]} = a_{k,j} ... a_{k,j+7}
13       __m256 ai = _mm256_loadu_ps(&A[i*n+j]); // a_{i,[]} = a_{i,j} ... a_{i,j+7}
14       ak = _mm256_mul_ps(ak, lv); // a_{k,[]} = a_{k,[]} · l_{i,k}
15       ai = _mm256_sub_ps(ai, ak); // a_{i,[]} = a_{i,[]} - a_{k,[]}
16       _mm256_storeu_ps(&A[i*n+j], ai); // a_{i,j} ... a_{i,j+7} = a_{i,[]}
17
18     // Verbleibende Elemente der i-ten Zeile für n - (k+1) % 8 != 0:
19     if j < n
20       // Hilfsvariable mit 256 Null-Bits gefolgt von 256 Eins-Bits:
21       int loadmask = {0,0,0,0,0,0,0,0,-1,-1,-1,-1,-1,-1,-1,-1};
22       // Laden der Maske, sodass n - (k+1) % 8 Elemente mit 1-Bits markiert werden:
23       _mm256i mask = _mm256_loadu_si256((__m256i*)&loadmask[n-j]);
24       j = n - 8; // Zeilenindex für die letzen 8 Elemente der Zeile setzen
25       __m256 ak_r = _mm256_loadu_ps(&A[k*n+j]); // a_{k,[]} = a_{k,n-7} ... a_{k,n}
26       __m256 ai_r = _mm256_loadu_ps(&A[i*n+j]); // a_{i,[]} = a_{i,n-7} ... a_{i,n}
27       ak_r = _mm256_mul_ps(ak_r, lv);     // a_{k,[]} = a_{k,[]} · l_{i,k}
28       ai_r = _mm256_sub_ps(ai_r, ak_r); // a_{i,[]} = a_{i,[]} - a_{k,[]}
29       // Abspeichern der n - (k+1) % 8 Werte a_{i,n-((n-(k+1)))%8)+1} ... a_{i,n} aus a_{i,[]}:
30       _mm256_maskstore_ps(&A[i*n+j], mask, ai_r);
31
32   b auf ähnliche Weise berechnen
33 Rückwärtssubstitution
```

Algorithmus 4.4: GaussStoreU Programmvariante: AVX-Implementierung der Gauss-Elimination. Für die Berechnung der *j-Schleife* werden Teilstücke der Zeile mit jeweils 8 `float` Werten geladen und berechnet.
Eine gesonderte Bearbeitung für die Fälle in denen die Iterationsanzahl der *j-Schleife* nicht durch 8 teilbar ist (d.h. $n - (k+1)\%8 \neq 0$) ist nötig. Hierfür werden die letzten 8 Elemente der Zeile geladen und die entsprechenden Berechnungen durchgeführt. Bei der Abspeicherung werden jedoch nur die bisher unbetrachteten $n-(k+1)\%8$ Elemente mit Hilfe einer maskierten Speicheroperation geschrieben.

- In Zeile 14 wird die Berechnung von $a_{i,j} - l_{i,k} \cdot a_{k,j}$ durchgeführt, indem die Elemente von ak mit dem Vektor lv multipliziert ($a_{k,[]} \cdot l_{i,k}$) und die Variable ak damit überschrieben werden.

- Im Anschluss (Zeile 15) wird das Zwischenergebnis in ak von ai subtrahiert ($a_{i,[]} - a_{k,[]}$) und wieder in ai gespeichert, sodass der alte Wert überschrieben wird.

- In Zeile 16 werden die neuen Werte $a_{i,[]}$ in ai auf die Matrixelemente $a_{i,j} \ldots, a_{i,j+7}$ zurück geschrieben.

Die nächste Iteration der *j-Schleife* beginnt auf dem nächsten unbearbeiteten Element der Matrixzeile, was durch die Anpassung der Inkrementoperation der Schleife in Zeile 11 auf +=8 gewährleistet ist.

Die Anzahl der Matrixelemente in der i-ten Zeile ist nicht in jedem Schritt k durch 8 teilbar. Somit muss die Iteration der *j-Schleife* über die Blöcke der Größe abgebrochen werden, obwohl noch unberechnete Matrixelemente verbleiben können. Diese verbleibenden Elemente der ursprünglichen *j-Schleife* vor dem Zerhacken in Blöcke (auch Schleifenrest) werden gesondert behandelt. In den Zeilen 19 bis 30 in Alg. 4.4 wird der Schleifenrest der *j-Schleife* berechnet. Wie beschrieben werden dazu die letzten 8 Elemente der Matrixzeile genutzt, wodurch der Schleifenrest die folgenden Matrixelemente umfasst:

$$a_{i,n-rem+1} \ldots a_{i,n}, \text{ mit } rem = n - (k+1)\%8 \tag{4.9}$$

Schleifenreste können bei der AVX-Programmierung mit maskierten Instruktionen berechnet werden. Maskierte Lade- oder Speicheroperationen werden genutzt, um Elemente von der Verarbeitung durch die Lade- oder Speicheroperation auszuschließen. Somit können durch maskierte Operationen Matrixelemente, die nicht mehr in der Zeile enthalten sind, beim Laden und Speichern übersprungen werden.

In Alg. 4.4 wird eine maskierte Instruktion zum Speichern der berechneten Matrixelemente genutzt. Dazu werden die letzten 8 `float` Werte der Zeile, unabhängig von der tatsächlich benötigten Anzahl, in den Zeilen 25 und 26 in die Vektorvariablen ak_r und ai_r geladen und anschließend berechnet (vgl. blaues Rechteck in Abb. 4.5). Die Berechnung der Werte in den Zeilen 27 und 28 wird unverändert auf allen 8 Elementen der Vektorvariablen durchgeführt. Zum Speichern der Ergebnisse in Zeile 30 wird eine Maske verwendet, sodass nur die benötigten Elemente abgespeichert und die übrigen Werte verworfen werden.

Die benötigte Maske wird in den Zeilen 21 und 23 erstellt. Elemente des Vektorregisters, die nicht gespeichert werden sollen, werden in der Maske an der entsprechenden Stelle des Elements durch Nullen gekennzeichnet. Analog werden zu speichernde Vektorelemente durch einen Wert ungleich Null in der Maske gekennzeichnet. Für die Erstellung der Maske wird in Zeile 21 eine Hilfsvariable loadmask in Form eines Arrays mit 256 Null-Bits und 256 Eins-Bits erstellt. Beim Laden der Maske in Zeile 23 wird der Startpunkt der Ladeoperation im Array so gewählt, dass für die letzten $n - (k+1)\%8$ Matrixelemente Eins-Bits in die Maske geladen werden. Die Anzahl

```
1  // Schleife über die Blöcke a_{i,[]} mit je 8 Elementen der i-ten Zeile:
2  for j = k + 1 < n - 8; j+=8
3    __m256 ak = _mm256_loadu_ps(&A[k*n+j]); // a_{k,[]} = a_{k,j} ... a_{k,j+7}
4    __m256 ai = _mm256_loadu_ps(&A[i*n+j]); // a_{i,[]} = a_{i,j} ... a_{i,j+7}
5    ak = _mm256_mul_ps(ak, lv); // a_{k,[]} = a_{k,[]} · l_{i,k}
6    ai = _mm256_sub_ps(ai, ak); // a_{i,[]} = a_{i,[]} - a_{k,[]}
7    _mm256_storeu_ps(&A[i*n+j], ai); // a_{i,j} ... a_{i,j+7} = a_{i,[]}
8
9  // Verbleibende Elemente der i-ten Zeile für n - (k+1) % 8 != 0:
10 for j = j < n; j+=1
11   A[i*n+j] = A[i*n+j] - A[k*n+j] * L[k*n+i] // a_{i,j} = a_{i,j} - l_{i,k} · a_{k,j}
```

Algorithmus 4.5: GaussRem Programmvariante: Die Berechnung der letzten Matrixelemente der i-ten Zeile wird seriell durchgeführt. Die dafür benötigte Schleife beginnt die Iteration beim nächsten Index j der i-ten Zeile, der nicht mehr in der blockweisen Abarbeitung enthalten ist.
Dargestellt ist die innerste Schleife des Schleifennestes. Veränderte Zeilen im Vergleich zur *GaussStoreU* Programmvariante in Alg. 4.4 sind hervorgehoben.

der mit Eins-Bits markierten Matrixelemente wird dabei durch die Verschiebung des Index der Ladeoperation aus der Hilfsvariable loadmask erzeugt. Je mehr Matrixelemente gespeichert werden sollen, desto höher der Index für die Ladeoperation und desto mehr Eins-Bits werden dabei geladen.

Die Berechnung der $n-(k+1)$ Elemente der rechten Seite b des Gleichungssystems erfolgt auf ähnliche Weise und wird im Anhang in Alg. C.2 gezeigt. Dazu werden die Elemente $b_i \dots b_{i+7}$, mit $i \geq k+1$, jeweils in Blöcken der Größe 8 geladen und entsprechend der Formel $b_i - (b_k \cdot l_{i,k})$ neu berechnet.

4.2.2.2. Die *GaussRem* Programmvariante: Skalare Berechnung der letzten Elemente der Matrixzeile

Instruktionen, die für das Laden und Speichern der Vektorelemente eine Maske nutzen, um nur einen Teil der Vektorelemente zu bearbeiten (kurz: maskierte Instruktionen), haben bei der Ausführung höhere Ausführungskosten als die entsprechenden unmaskierten Operationen. Die von Intel angegeben Werte für Durchsatz und Latenz der Instruktionen in [41] sind dabei vor allem auf älteren Prozessorarchitekturen um das doppelte bis vierfache (Durchsatz in Zyklen pro Instruktion) bzw. achtfache (Latenz) höher als für unmaskierte Instruktionen. In der *GaussStoreU* Programmvariante werden die letzten Matrixelemente der Zeile mit Hilfe von maskierten Speicheroperationen vektorisiert. Zur Untersuchung des Einflusses von maskierten Speicheroperation wird die Programmvariante **GaussRem** erzeugt, die die letzten Matrixelemente der Zeile durch serielle Berechnungen durchführt.

Algorithmus 4.5 zeigt die Implementierung der innersten Schleife der **GaussRem** Programmvariante. Die Berechnungen in den Zeilen 2 bis 7 werden dabei im Vergleich

```
1  // Hilfsvariable mit 256 Null-Bits gefolgt von 256 Eins-Bits:
2  int loadmask = {0,0,0,0,0,0,0,0,-1,-1,-1,-1,-1,-1,-1,-1};
3  // Schleife über die Blöcke a_{i,[]} mit je 8 Elementen der i-ten Zeile:
4  for j = k + 1 < n - 8; j+=8
5    // Laden der Maske, sodass 8 Elemente mit 1-Bits markiert werden:
6    __m256i mask = _mm256_loadu_si256((__m256i*)&loadmask[8]);
7    __m256 ak = _mm256_maskload_ps(&A[k*n+j], mask); //a_{k,[]} = a_{k,j} ... a_{k,j+7}
8    __m256 ai = _mm256_maskload_ps(&A[i*n+j], mask); //a_{i,[]} = a_{i,j} ... a_{i,j+7}
9    ak = _mm256_mul_ps(ak, lv); // a_{k,[]} = a_{k,[]} · l_{i,k}
10   ai = _mm256_sub_ps(ai, ak); // a_{i,[]} = a_{i,[]} - a_{k,[]}
11   _mm256_storeu_ps(&A[i*n+j], ai); // a_{i,j} ... a_{i,j+7} = a_{i,[]}
12
13 // Verbleibende Elemente der i-ten Zeile für n - (k + 1) % 8 != 0:
14 if j < n
15   // Laden der Maske, sodass n - (k + 1) % 8 Elemente mit 1-Bits markiert werden:
16   __m256i mask = _mm256_loadu_si256((__m256i*)&loadmask[n-j]);
17   j = n - 8; // Zeilenindex für die letzen 8 Elemente der Zeile setzen
18   __m256 ak_r =_mm256_maskload_ps(&A[k*n+j], mask); //a_{k,[]} = a_{k,n-7} ... a_{k,n}
19   __m256 ai_r =_mm256_maskload_ps(&A[i*n+j], mask); //a_{i,[]} = a_{i,n-7} ... a_{i,n}
20   ak_r = _mm256_mul_ps(ak_r, lv);      // a_{k,[]} = a_{k,[]} · l_{i,k}
21   ai_r = _mm256_sub_ps(ai_r, ak_r); // a_{i,[]} = a_{i,[]} - a_{k,[]}
22   // Abspeichern der n - (k + 1) % 8 Werte a_{i,n-((n-(k+1)))%8)+1} ... a_{i,n} aus a_{i,[]}:
23   _mm256_maskstore_ps(&A[i*n+j], mask, ai_r);
```

Algorithmus 4.6: GaussMaskload Programmvariante: Die Ladeoperationen sind durch maskierte Ladeoperationen ausgetauscht. Bei der Berechnung der Blöcke wird die Maske so gewählt, dass alle Elemente geladen werden. Für die letzten Matrixelemente der i-ten Zeile wird die Maske der Speicheroperation verwendet.
Dargestellt ist die innerste Schleife des Schleifennestes. Veränderte Zeilen im Vergleich zur *GaussStoreU* Programmvariante in Alg. 4.4 sind hervorgehoben.

zur *GaussStoreU* Programmvariante nicht verändert. Die Berechnung der letzten Elemente der Matrixzeile wird in der **GaussRem** Programmvariante jedoch nicht vektorisiert. Dazu werden die Berechnungen in den Zeilen 19 bis 30 aus Alg. 4.4 verändert. Anstelle der `if` Abfrage in Zeile 19 aus Alg. 4.4 wird in der **GaussRem** Programmvariante die folgende `for` Schleife in Zeile 10 in Alg. 4.5 eingefügt:

```
for j = j < n; j+=1
```

Die erste Iteration dieser Schleife entspricht dabei der nächsten Iteration der *j-Schleife* in Zeile 2, die aufgrund der Blockgröße nicht mehr ausgeführt werden kann.

Im Schleifenkörper der neuen Schleife wird in Zeile 11 die serielle Berechnung der letzten Matrixelemente der i-ten Zeile eingefügt.

4.2.2.3. Die *GaussMaskload* Programmvariante: Maskierte Ladeoperationen

Der Einfluss von maskierten Instruktionen auf Ausführungszeit und Energieverbrauch im Vergleich zu unmaskierten Instruktionen wird mit Hilfe einer weiteren Modifikation der *GaussStoreU* Programmvariante aus Alg. 4.4 untersucht. Die Anzahl der maskierten Instruktionen, die in der *GaussStoreU* Programmvariante ausgeführt werden, ist im Vergleich zur Anzahl der übrigen Instruktionen sehr gering. Zur Untersuchung von Unterschieden bei Laufzeit und Energieverbrauch zwischen maskierten und unmaskierten Instruktionen wird die Anzahl der maskierten Instruktionen in der **GaussMaskload** Programmvariante erhöht.

Algorithmus 4.6 zeigt die innerste Schleife des Schleifennestes der **GaussMaskload** Programmvariante. Hierbei werden die Ladebefehle in den Zeilen 7, 8, 18 und 19 durch maskierte Ladebefehle ersetzt.

Durch die maskierten Ladeinstruktionen in den Zeilen 7 und 8 sollen weiterhin 8 Matrixelemente des Blockes geladen werden. Dazu wird in Zeile 6 eine Maske geladen, bei der alle Elemente des Vektorregisters Eins-Bits enthalten. Dies wird erreicht, indem die Hilfsvariable loadmask (in Zeile 2) so verwendet wird, dass in die Maske die rechten 8 Elemente (8 mal 32 Eins-Bits) geladen werden.

In den Zeilen 18 und 19 wird die bereits erstellte Maske aus Zeile 16 für die Ladebefehle genutzt. Diese Maske wird bereits für die Speicheroperation in Zeile 23 genutzt und markiert ausschließlich die bisher nicht berechneten Matrixelemente der i-ten Zeile. Bei der Verwendung dieser Maske in den Ladeoperationen werden die markierten (Eins-Bits) Matrixelemente in das Vektorregister geladen. Matrixelemente, die nicht zum Laden markiert sind (Null-Bits), werden durch die maskload Instruktion nicht geladen und stattdessen mit dem Wert 0 initialisiert.

4.2.2.4. Die *GaussStore* Programmvariante mit ausgerichteten Speicherzugriffen

In den bisher gezeigten AVX-Programmvarianten der Gauss-Elimination wird der Speicherzugriff auf die Elemente der Matrix A entsprechend der Schleifengrenzen organisiert. Dies hat zur Folge, dass die entsprechenden Speicherzugriffe für Lade- und Speicheroperationen auf unausgerichteten Speicherstellen durchgeführt werden. Werden Lade- oder Speicheroperationen auf ausgerichtete (*engl. aligned*) Speicherstellen ausgeführt, befinden sich alle benötigten Elemente in der selben Cache-Zeile (vgl. Anhang Abschn. A.2). Auf diese Weise kann die Operation effizienter erfolgen, da die Daten nicht aus mehreren Cache-Zeilen zusammengetragen werden müssen. Im Fall von AVX wird demnach ein Alignment von 256 Bit benötigt, damit die Daten eines 256 Bit Vektors in der selben Cache-Zeile liegen.

Für die **GaussStore** Programmvariante soll der Speicherzugriff auf ausgerichtetem Speicher erfolgen, wodurch das Bearbeitungsschema der AVX Variante angepasst wird. Für die Nutzung von Lade- und Speicheroperationen für ausgerichteten Speicher werden die ersten Iterationen der Schleife mittels *loop peeling* von der Schleife abgetrennt. Auf diese Weise beginnt die Schleife die Iteration über die Blöcke mit einem

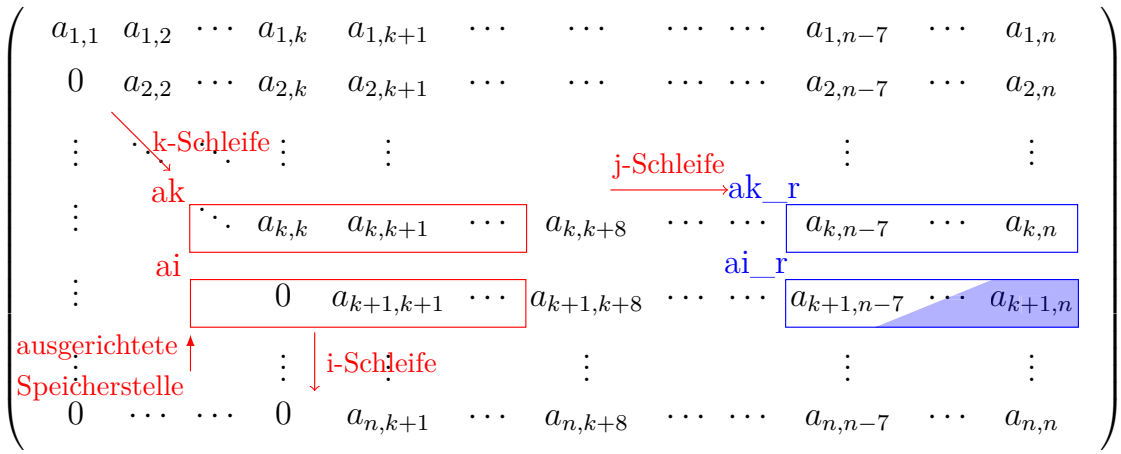

Abbildung 4.6.: Bearbeitungsschema der Matrix A für die Programmvarianten **GaussStore** und **GaussStream** in Schritt k mit $i = k + 1$ und $j = k + 1$. Die Iterationen der *j*- und *i-Schleife* und deren genutzte Datenvektoren werden in rot hervor gehoben. Das erste Element der Vektorvariablen ak und ai wird dabei so gewählt, dass es an einer ausgerichteten Speicherstelle abgelegt ist. Dadurch verschiebt sich der Beginn des ersten Blockes nach links.
Die Berechnung der letzten Matrixelemente der i-ten Zeile wird in blau hervorgehoben und ist identisch zur Bearbeitung in Abb. 4.5.

Element an einer ausgerichteten Speicherstelle.

Bei der Gauss-Elimination kann die Berechnung der abgetrennten Matrixelemente ebenfalls vektorisiert werden: Dazu wird der Beginn der ersten Vektorvariablen der i-ten Zeile so gewählt, dass der Wert $a_{i,k+1}$ in dieser Vektorvariablen enthalten ist, diese jedoch als erstes Element ein Matrixelement an einer ausgerichteten Speicherstelle enthält. Die zusätzlich berechneten Werte werden im Ergebnis ignoriert, da diese im weiteren Verlauf als Null angenommen und nicht mehr gelesen werden.

Das Bearbeitungsschema der **GaussStore** Programmvariante wird in Abb. 4.6 dargestellt. Als Startpunkt für das Laden an einer ausgerichteten Speicherzelle werden die Matrixelemente der Vektorvariable ai gewählt, da diese sowohl geladen als auch gespeichert werden. Die Indizes der Ladeoperationen von ak und ai werden dabei um einen Offset nach vorne verschoben, sodass sich das erste geladene Element an einer ausgerichteten Speicherstelle befindet.

Die Matrix A wird dafür so im Speicher alloziert, dass das erste Matrixelement $a_{1,1}$ an einer ausgerichteten Speicherstelle abgelegt ist. Weiterhin ist das Alignment immer ein Vielfaches der Elementgröße, hier: 32-Bit pro `float` bei 256-Bit Alignment. Somit ist es ausreichend einen Indexzugriff für das eindimensionale Array zu wählen, der durch 8 teilbar ist, um ein Matrixelement zu finden, dass an einer ausgerichteten Speicherstelle abgelegt ist.

Algorithmus 4.7 zeigt die Implementierung der Schleife über die Blöcke der Matrixzeile in der **GaussStore** Programmvariante. Der Index für den Zugriff auf das eindimensionale Array für das Matrixelement $a_{i,k}$ wird in Zeile 1 berechnet. Im Anschluss wird durch eine Integer-Division (Zeile 2) und eine Multiplikation (Zeile 3) der

```
1  idx = i * n + k + 1 // Index des ersten Elements a_{i,j} = a_{i,k+1}
2  idx /= 8; // Integer-Division entspricht ⌊idx/8⌋
3  idx *= 8; // idx durch 8 teilbar und idx ≤ i·n+k+1 < idx+8
4  peel = idx - i * n // Rückrechnung für j als Spaltenindex
5
6  // Schleife über die Blöcke a_{i,[]} mit je 8 Elementen der i-ten Zeile:
7  for j = peel < n - 8; j+=8
8    __m256 ak = _mm256_loadu_ps(&A[k*n+j]); // a_{k,[]} = a_{k,j} ... a_{k,j+7}
9    __m256 ai = _mm256_load_ps(&A[i*n+j]); // a_{i,[]} = a_{i,j} ... a_{i,j+7}
10   ak = _mm256_mul_ps(ak, lv); // a_{k,[]} = a_{k,[]} · l_{i,k}
11   ai = _mm256_sub_ps(ai, ak); // a_{i,[]} = a_{i,[]} - a_{k,[]}
12   _mm256_store_ps(&A[i*n+j], ai); // a_{i,j} ... a_{i,j+7} = a_{i,[]}
```

Algorithmus 4.7: GaussStore Programmvariante: Programmausschnitt zur Anpassung der Indexberechnung, sodass der Spaltenindex j ein Matrixelement angibt, dass an einer ausgerichteten Speicherstelle abgelegt ist. Eine ausgerichtete Speicherstelle wird gefunden, indem ein Index für den eindimensionalen Array-Zugriff gewählt wird, der durch 8 teilbar ist. Die Lade- und Speicheroperation der Vektorvariable ai wird durch Instruktionen für ausgerichteten Speicher ausgetauscht.
Dargestellt ist ein Ausschnitt der innersten Schleife des Schleifennestes. Die Berechnung der letzten Matrixelemente der i-ten Zeile ist identisch zu Alg. 4.4. Veränderte Zeilen im Vergleich zur *GaussStoreU* Programmvariante in Alg. 4.4 sind hervorgehoben.

nächstkleinere, durch 8 teilbare Index erhalten. Für die Ausführung der *j-Schleife* wird der eindimensionale Index in Zeile 4 wieder in einen Spaltenindex umgewandelt. Der Offset zum jeweiligen Element $a_{i,k+1}$ ändert sich dabei in jeder Iteration der *i-Schleife*, da die Anzahl der Matrixelemente pro Zeile nicht zwingend durch 8 teilbar ist.

Dieses Vorgehen berechnet zusätzliche Elemente, die jedoch nach der Vorwärtselimination als Null betrachtet werden und damit nicht weiter genutzt werden. Durch diese Änderung kann die Ladeoperation in Zeile 9 von Alg. 4.7 durch eine ausgerichtete Ladeoperation (_mm256_load_ps) ersetzt werden. Zusätzlich wird die Speicheroperation in Zeile 12 durch eine ausgerichtete Speicheroperation ersetzt.

Das Alignment der Matrixelemente der Vektorvariablen ak kann auf diese Weise nicht sicher gestellt werden. Aus diesem Grund wird die Ladeoperation der Vektorvariablen ak nicht durch eine Ladeoperation für ausgerichteten Speicher ersetzt.

Die Ladeoperation für die letzten Matrixelemente der i-ten Matrixzeile findet mit Element $a_{i,n-7}$ statt, um die Berechnung wie in Alg. 4.4 gezeigt durchzuführen. Daher wird die Berechnung dieser Elemente ebenfalls nicht auf die Nutzung von ausgerichteten Speicherstellen angepasst.

```
1  idx = i * n + k + 1 // Index des ersten Elements a_{i,j} = a_{i,k+1}
2  idx /= 8; // Integer-Division entspricht ⌊idx/8⌋
3  idx *= 8; // idx durch 8 teilbar und idx ≤ i·n+k+1 < idx+8
4  peel = idx - i * n // Rückrechnung für j als Spaltenindex
5
6  // Schleife über die Blöcke a_{i,[]} mit je 8 Elementen der i-ten Zeile:
7  for j = peel < n - 8; j+=8
8    __m256 ak = _mm256_loadu_ps(&A[k*n+j]); // a_{k,[]} = a_{k,j} ... a_{k,j+7}
9    __m256 ai = _mm256_load_ps(&A[i*n+j]); // a_{i,[]} = a_{i,j} ... a_{i,j+7}
10   ak = _mm256_mul_ps(ak, lv); // a_{k,[]} = a_{k,[]} · l_{i,k}
11   ai = _mm256_sub_ps(ai, ak); // a_{i,[]} = a_{i,[]} - a_{k,[]}
12   _mm256_stream_ps(&A[i*n+j], ai); // a_{i,j} ... a_{i,j+7} = a_{i,[]}
```

Algorithmus 4.8: GaussStream Programmvariante: Die Ladeoperation der Vektorvariable ai wird wie in der *GaussStore* Programmvariante aus Alg. 4.7 durch eine Instruktion für ausgerichteten Speicher realisiert. Die Speicheroperation der Vektorvariable ai wird durch eine non-temporal Schreiboperation (auch *streaming store*) implmentiert.
Veränderte Zeilen im Vergleich zur *GaussStore* Programmvariante in Alg. 4.7 sind hervorgehoben.

4.2.2.5. Die *GaussStream* Programmvariante mit non-temporal Schreiboperationen

Das Speichern der berechneten Elemente in Zeile 12 wird in der *GaussStore* Programmvariante in Alg. 4.7 als normale Write-Back Schreiboperation ausgeführt. Bei Write-Back Operationen werden die zu schreibenden Daten in den Cache geschrieben und erst zu einem späteren Zeitpunkt in den Hauptspeicher zurückgeschrieben. Die in Zeile 12 geschriebenen Daten werden jedoch erst nach einem kompletten Durchlauf der *i-Schleife* wieder benötigt, weshalb diese bei großen Matrizen bereits aus dem Cache zurückgeschrieben wurden und eine Wiedernutzung damit entfällt.

Eine weitere Möglichkeit der Rückschreibestrategie sind non-temporal Schreiboperationen mit denen die Daten nicht erst im Cache zwischengespeichert werden, sondern gleich in den Hauptspeicher geschrieben werden. Dazu werden alle Kopien des Speicherblocks, in dem die Daten abgelegt sind, in allen Caches für ungültig erklärt und der Speicherblock direkt im Hauptspeicher überschrieben. AVX bietet mit dem Intrinsic _mm256_stream_ps eine Möglichkeit solche non-temporal Schreiboperationen (auch *streaming stores*) bei ausgerichteten Speicherzugriffen auszuführen. In der **GaussStream** Programmvariante in Alg. 4.8 wird die Schreiboperation in Zeile 12 durch eine non-temporal Schreiboperation ersetzt.

Architektur	Jahr	Prozessor	AVX-Version	Frequenzbereich	LL-Cache
Sandy Bridge	2011	Core i7-2600	AVX	1,5-3,7 GHz	8 MB
Haswell	2013	Core i7-4770K	AVX2	0,8-3,5 GHz	8 MB
Skylake	2015	Core i7-6700	AVX2	0,8-3,5 GHz	8 MB

Tabelle 4.2.: Verwendete Prozessoren zur Ausführung der SIMD-Programmvarianten der Gauss-Elimination mit dem Erscheinungsjahr, dem höchsten unterstützten AVX- Instruktionssatz, des einstellbaren Frequenzbereichs und der Größe des Last-Level Caches (LL-Cache). Vergleiche [40].

4.3. Untersuchungen zu Ausführungszeit und Energieverbrauch der vektorisierten Gauss-Elimination

Die vektorisierten Programmvarianten werden im Folgenden im Hinblick auf deren Ausführungszeit und Energieverbrauch untersucht. Dazu werden die Programmvarianten auf Systemen mit verschiedenen Prozessorarchitekturen ausgeführt und verschiedene Ausführungsparameter, wie Ausführungszeit und Energieverbrauch, gemessen.

4.3.1. Ausführungsumgebung für die Untersuchung der Programmvarianten

Zur Ausführung der vektorisierten Programmvarianten werden drei Prozessoren aus verschiedenen Architekturfamilien gewählt. Die drei gewählten Prozessoren unterscheiden sich in der zugrunde liegenden Prozessorarchitektur sowie dem unterstützten AVX-Instruktionssatz. Die Spezifikationen der verwendeten Prozessoren aus der Sandy Bridge, Haswell und Skylake Architektur werden in Tab. 4.2 gezeigt. Der AVX-Instruktionssatz wurde erstmalig in der Sandy Bridge Architektur eingeführt und in der Haswell Architektur um weitere Instruktionen zum AVX2-Instruktionssatz erweitert [36]. Bei der Weiterentwicklung zur Skylake Architektur wurden verschiedene Aspekte der Hardware in Bezug auf die Ausführung von AVX-Instruktionen weiter verbessert [58].

Mit dem Linux-Tool `cpu-freq` lässt sich die Prozessorfrequenz auf einen festen Wert einstellen. Die wählbaren Prozessorfrequenzen variieren dabei je nach Prozessorarchitektur und sind ebenfalls in Tab. 4.2 aufgeführt. Die Prozessorfrequenz (auch Taktrate) hat einen Einfluss auf die Ausführungszeit und den Energieverbrauch von Programmen [81]. Bei vektorisierten Programmen zeigt sich ein ähnlicher Einfluss der Prozessorfrequenz, es zeigt sich jedoch in einigen Fällen eine abweichendes Verhalten der vektorisierten gegenüber der seriellen Programmausführung.

Bei der Ausführung der Programme wird der Energieverbrauch des Prozessors über die Model Specific Register (MSR) gemessen, die über das Running Average Power Limits Interface (RAPL) der Prozessoren verfügbar sind [36]. Zusätzliche Laufzeitparameter werden mit der PAPI-Bibliothek (Version 5.5) gemessen. Jede Programmva-

riante wird zehn mal ausgeführt und der Mittelwert der gemessenen Werte gebildet.

Die Programmvarianten werden mit der Intel C++ Compiler Suite (Version 17.0.0) compiliert. Der Intel Compiler verwendet die GNU Compiler Collection 4.9.0. Wenn für die entsprechende Architektur AVX2 Fused-Multply-Add (FMA) Befehle verfügbar sind, ersetzt der Compiler automatisch passende Konstrukte mit FMA-Instruktionen. Für den Compileraufruf werden die folgenden Compiler-Optionen genutzt:

- `-O3` Aktiviert die höchste Optimierungsstufe des Compilers und erlaubt den Einsatz verscheidender Optimierungsmethoden. Eine vollständige Liste der Optimierungen findet sich in [38].
- `-restrict` Aktiviert die Nutzung des restrict Schlüsselwortes. Dieses Schlüsselwort zeigt an, dass die Speicherregion eines Pointers nur von diesem Pointer genutzt wird und kein anderer Pointer auf die gleiche Speicherregion zugreift (Pointer-alias). Hierdurch kann der Compiler bestimmte Optimierungen besser durchführen. Insbesondere ist dies für die automatische Vektorisierung erforderlich. Alle Speicherbereiche der Programmvarianten sind mit dem restrict Schlüsselwort gekennzeichnet.
- `-cilk-serialize` Schaltet Threading-Transformationen von Intel Cilk Plus aus. Auf diese Weise werden durch Cilk nur Transformationen an Code in der „Array-Notation“ durchgeführt.
- `-mavx` Erzeugt Programmcode für die AVX-Instruktionssatzerweiterung. Wird nur auf der Sandy Bridge Architektur verwendet.
- `-march=core-avx2` Erzeugt Programmcode für die AVX2-Instruktionssatzerweiterung. Wird auf der Haswell und Skylake Architektur verwendet.

4.3.2. Ausführungszeit in Abhängigkeit zur gewählten Matrixgröße und Anzahl ausgeführter AVX-Intstruktionen der Programmvarianten

Die in Abschn. 4.2 vorgestellten Programmvarianten werden in der beschriebenen Ausführungsumgebung ausgeführt. Zur späteren Bewertung der Unterschiede von Ausführungszeit und Energieverbrauch der Programmvarianten werden in diesem Abschnitt der Einfluss der gewählten Matrixgröße sowie die Anzahl der ausgeführten AVX-Instruktionen betrachtet. Diese Betrachtung zeigt die Eigenschaften der erstellten Programmvarianten im Hinblick auf deren Vergleichbarkeit.

4.3.2.1. Einfluss der gewählten Matrixgröße auf die Ausführungzeit

Die Größe der Matrix A wird bestimmt durch die Anzahl n der Gleichungen des linearen Gleichungssystems, beziehungsweise durch die Anzahl der Unbekannten im Vektor x. Je nach gewählter Matrixgröße kann das Programm ein unterschiedliches

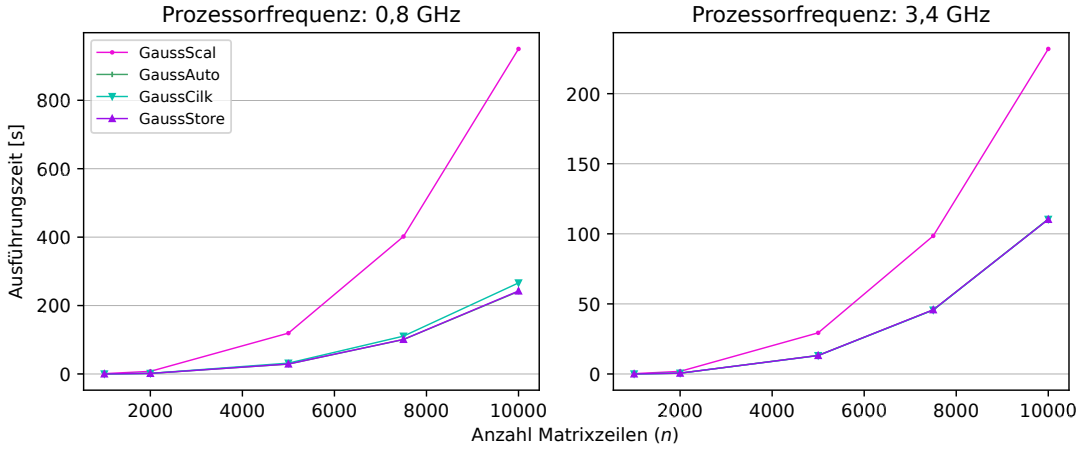

Abbildung 4.7.: Ausführungszeit der Programmvarianten mit unterschiedlichen Programmiertechniken in Abhängigkeit zur Zeilengröße von Matrix A auf der Skylake Architektur bei 0,8 GHz (links) und 3,4 GHz (rechts). Die Messergebnisse der Sandy Bridge und Haswell Architektur finden sich im Anhang Abb. C.1 und Abb. C.2.

Verhalten zeigen, da beispielsweise Cache-Speicher unterschiedlich ausgenutzt werden. Zusätzlich muss zur Messung des Energieverbrauchs mit Hilfe des RAPL Interface, die Ausführungszeit des gemessenen Programms eine gewisse minimale Ausführungszeit aufweisen, damit die Messwerte aussagekräftig sind [27]. Eine empirisch ermittelte Matrixgröße von $1\,000 \times 1\,000$ erfüllt die Voraussetzungen für die Energiemessungen. Bei einer gewählten Matrixgröße von $10\,000 \times 10\,000$ übersteigt der benötigte Speicherbedarf der genutzten Datenstrukturen die Größe des Caches nach Tab. 4.2, wodurch bei diese Größe Effekte durch unterschiedliche Cache-Ausnutzung ausgeschlossen werden.

In Abb. 4.7 wird die Ausführungszeit der Programmvarianten *GaussScal*, *GaussAuto*, *GaussCilk* und *GaussStore* in Abhängigkeit zur Anzahl der Matrixzeilen n, mit $A \in \mathbb{R}^{n \times n}$ dargestellt. Ein linearer Anstieg von n führt zu einem quadratischen Wachstum der Eingabematrix A, deren Elemente durch die Gauss-Elimination ($O(n^3)$[63]) genutzt werden. In Abb. 4.7 zeigt sich ein entsprechender kubischer Anstieg der Ausführungszeit. Eine unterschiedliche Cache-Ausnutzung würde diesen Anstieg verändern. Die Abwesenheit von Sprüngen und das kubische Wachstum der Ausführungszeit weisen darauf hin, dass es im Bereich der untersuchten Matrixgröße zu keinen Cache-Effekten kommt. Abbildung 4.7 zeigt die Ausführungszeiten für die Skylake Architektur, die gleichen Beobachtungen können für die Sandy Bridge und Haswell Architektur gemacht werden (siehe Anhang Abb. C.1 und Abb. C.2).

4.3.2.2. Ausgeführte AVX-Instruktionen der AVX-Programmvarianten

Bei der Ausführung der Programmvarianten auf den betrachteten Prozessorarchitekturen wird die Anzahl der tatsächlich ausgeführten AVX-Instruktionen gemessen. Die Anzahl der tatsächlich ausgeführten Instruktionen kann einen Hinweis auf unterschiedliches Verhalten bei der Programmausführung geben, sowie bei der automatischen

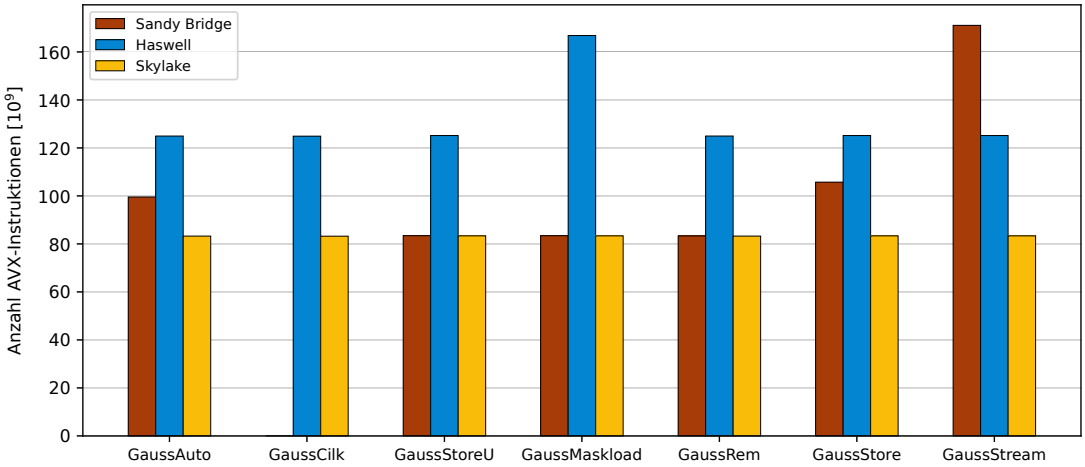

Abbildung 4.8.: Anzahl der auf dem Prozessor ausgeführten AVX 256-Bit Instruktionen der untersuchten vektorisierten Programmvarianten. Gezeigt wird jeweils die gemessene Anzahl AVX-Instruktionen auf den drei genutzten Prozessorarchitekturen pro Programmvariante mit $n = 10\,000$ bei der jeweils maximalen Prozessorfrequenz.

Vektorisierung als Indikator für die Güte der Vektorisierung genutzt werden. Zu erwarten sind bei der Messung der ausgeführten AVX-Instruktionen annähernd gleiche Werte für die Ausführung der unterschiedlichen Programmvarianten, da die Anzahl der ausgeführten Instruktionen im wesentlichen durch die Anzahl der AVX-Instruktionen im Assembler-Code der Programmvarianten bestimmt wird.

Die Anzahl der Instruktionen wird für jede der verwendeten Architekturen mit dem jeweils verfügbaren Event der PAPI-Bibliothek gemessen. Die genutzten Events der Architekturen unterschieden sich auf den Architekturen und sind wie folgt definiert:

- Sandy Bridge Architektur: SIMD_FP_256:PACKED_SINGLE zählt 256-Bit SIMD-Instruktionen auf `float` (single precision) Werten.
- Haswell Architektur: AVX:ALL zählt alle 256-Bit Instruktionen inklusive Lade- und Speicheroperationen.
- Skylake Architektur: FP_ARITH:256B_PACKED_SINGLE zählt die Anzahl der arithmetischen 256-Bit SIMD-Instruktionen auf `float` (packed single precision) Daten.

Die gemessenen Werte der AVX-Instruktionen werden in Abb. 4.8 für die SIMD Programmvarianten dargestellt. Bei der Ausführung auf der Haswell Architektur werden grundsätzlich ca. 50% mehr Operationen ausgeführt als auf den anderen beiden Architekturen. Dies begründet sich vor allem durch das unterschiedliche Messverhalten des PAPI-Events auf der Architektur, da hierbei ebenfalls die Lade- und Speicheroperationen enthalten sind, die bei den anderen beiden Architekturen nicht mitgezählt werden. Dieser Effekt ist noch deutlicher für die *GaussMaskload* Programmvariante, bei der im Vergleich zu den anderen Programmvarianten mehr Integer-Vektorvariablen für die Masken der Ladeinstruktionen geladen werden müssen.

Weitere Unterschiede der ausgeführten AVX-Instruktionen ergeben sich bei der automatisch vektorisierten *GaussAuto* und der *GaussStore* Programmvariante auf der Sandy Bridge Architektur. Beide Programmvarianten führen Lade- und Speicheroperationen auf Matrixelementen durch, welche die Alignment-Voraussetzungen erfüllen. Dazu werden in der *GaussStore* Programmvariante mehr Matrixelemente für die Berechnung genutzt, was eine Erhöhung der ausgeführten Instruktionen zur Folge haben könnte. Im Gegensatz dazu werden bei der automatischen Vektorisierung in der *GaussAuto* Programmvariante weniger Elemente in Vektorblöcken berechnet, da die ersten Elemente der Schleife seriell berechnet werden. Da bei beiden Programmvarianten eine höhere Anzahl ausgeführter AVX-Instruktionen auftritt, muss hier eine Abhängigkeit zur genutzten ausgerichteten Lade- und/oder Speicheroperation bestehen.

Bei der *GaussStream* Programmvariante tritt dieser Effekt stärker ausgeprägt auf. Dies könnte durch die Nutzung der non-temporal Schreiboperation hervorgerufen werden. Die genaue Ursache liegt jedoch im Microcode des Prozessors und kann nicht näher untersucht werden.

Bei der Ausführung der *GaussCilk* Programmvariante auf der Sandy Bridge Architektur werden keine ausgeführten AVX-Instruktionen gemessen. Der Compiler übersetzt die Intel Cilk Programmteile auf der Sandy Bridge Architektur in 128-Bit AVX-Instruktionen, die in der Zählung der 256-Bit AVX-Instruktionen nicht beachtet werden.

4.3.3. Ausführungszeit und Energieverbrauch in Abhängigkeit zur Prozessorfrequenz bei automatisch vektorisierten Programmvarianten

In diesem Abschnitt werden Programmvarianten besprochen, die durch automatische Vektorisierung erstellt werden. Die automatische Vektorisierung stellt unterschiedliche Anforderungen an den Umfang der nötigen Programmcodeänderungen, und damit auch an die Fähigkeiten des Programmierers. Ein Vergleich der erzielten Ergebnisse bei der Vektorisierung erlaubt dabei Rückschlüsse auf Güte der automatischen Vektorisierung.

Für die Untersuchung in diesem Abschnitt werden die folgenden Programmvarianten genutzt:

- *GaussScal* als serielle Basisvariante zur Einschätzung der Wirksamkeit der Vektorisierung.
- *GaussAuto* bei der die Vektorisierung vollständig durch den Compiler erfolgt.
- *GaussCilk* als Beispielimplementierung der Vektorisierung mit Array-Notation, die dem Compiler die automatische Vektorisierung erleichtern soll.
- *GaussStore* als Beispielimplementierung mit AVX-Intrinsics. Von den vorgestellten AVX-Implementierungen wird die *GaussStore* Programmvariante gewählt, da diese in weiten Teilen den Vektorisierungsansätzen des Compilers entspricht.

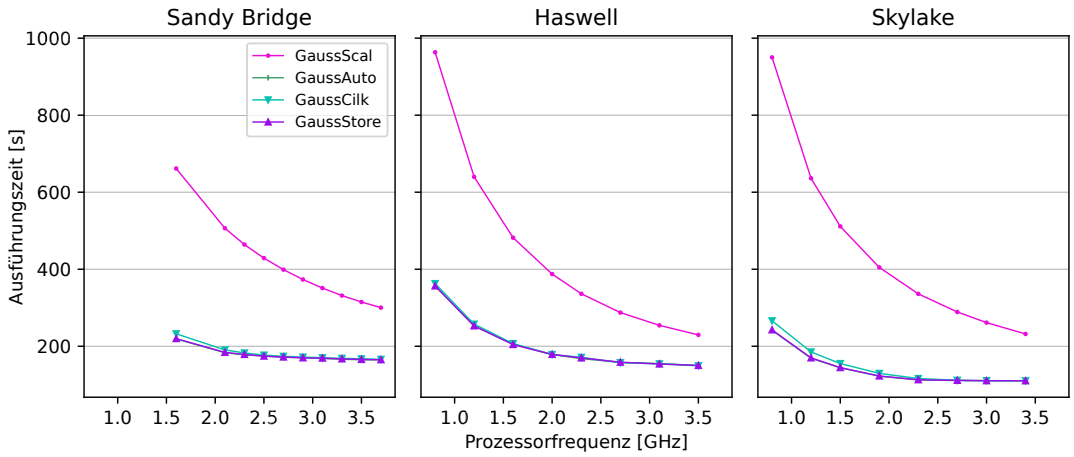

Abbildung 4.9.: Ausführungszeit der Programmvarianten zur Untersuchung der automatischen Vektorisierung, in Abhängigkeit zur eingestellten Prozessorfrequenz. Gezeigt werden die Werte für die drei genutzten Prozessorarchitekturen: Sandy Bridge (links), Haswell (mitte) und Skylake (links) bei einer Matrixgröße von $10\,000 \times 10\,000$.

4.3.3.1. Ausführungszeit der vektorisierten Gauss-Elimination

Eine Reduzierung der Ausführungszeit von Programmen ist meist das ausschlaggebende Ziel für die Vektorisierung von Programmen. Die Ausführungszeiten der betrachteten Programmvarianten werden in Abb. 4.9 in Abhängigkeit zur eingestellten Prozessorfrequenz gezeigt. Die Ausführungszeit aller Programmvarianten sinkt erwartungsgemäß mit steigender Prozessorfrequenz, da durch die höhere Taktrate mehr Operationen in der gleichen Zeit ausgeführt werden können. Bei den drei betrachteten Prozessorarchitekturen reduziert sich jeweils die Ausführungszeit mit der Verbesserung der Architekturfamilie.

Die vektorisierten Programmvarianten zeigen für alle Frequenzen geringere Ausführungszeiten als die serielle Programmvariante *GaussScal*. Die *GaussAuto* und *GaussStore* Programmvarianten zeigen nahezu identische Ausführungszeiten. Im Vergleich dazu, hat die *GaussCilk* Programmvariante bei den drei betrachteten Prozessorarchitekturen eine leicht höhere Ausführungszeit. Dies erklärt sich durch den vom Compiler erzeugten Assembler-Code. Für die *GaussAuto* und *GaussStore* Programmvarianten erzeugt der Compiler ähnlichen Assembler-Code.

Bei der *GaussCilk* Programmvariante werden durch den Compiler weitere Transformationen vorgenommen. Beispielsweise werden auf allen Prozessorarchitekturen Schleifen-Transformationen wie *loop peeling* und die Berechnung der letzten Elemente der Zeile durch 128-Bit AVX-Instruktionen eingebaut. Zusätzlich verändert der Compiler den Schleifenkörper durch *loop unrolling*. Diese Transformationen können bei vielen Anwendungen die Ausführungszeit verkürzen, führen jedoch zu zusätzlichen Abfragen und Verzweigungen.

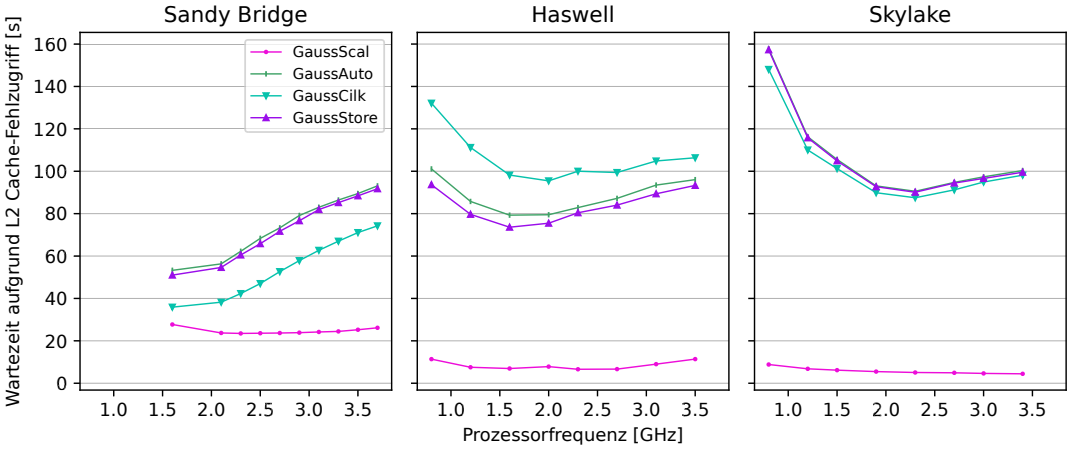

Abbildung 4.10.: Ausführungszeit während der keine Berechnungen durchgeführt werden, da auf eine Daten Lese- oder Schreiboperationen gewartet wird, die durch einen Level 2 Cache-Fehlzugriff (*engl. L2-Cache Miss*) ausgelöst wurde. Die Wartezeit berechnet sich durch die Anzahl der blockierten Prozessorzyklen (*engl. stalled cycles*) und die Prozessorfrequenz. Die Wartezeiten werden für die drei genutzten Prozessorarchitekturen Sandy Bridge (links), Haswell (mitte) und Skylake (links) bei einer Matrixgröße von $10\,000 \times 10\,000$ in Abhängigkeit zur eingestellten Prozessorfrequenz gezeigt (vgl. Ausführungszeiten in Abb. 4.9).

Wartezeiten durch blockierte Zyklen

Die Ausführungszeiten der vektorisierten Programmvarianten werden durch eine steigende Frequenz weniger reduziert, als dies für die serielle Programmvariante *Gauss-Scal* der Fall ist. Eine der Ursachen dafür findet sich in Zeiten, in denen der Prozessor keine Instruktionen ausführt. Mit der PAPI-Bibliothek können die Prozessorzyklen, in denen der Prozessor die Ausführung unterbrechen muss (*engl. stalled cycles*, blockierte Zyklen), gemessen werden.

In Abb. 4.10 wird die Wartezeit des Prozessors aufgrund eines Level 2 Cache-Fehlzugriffs (*L2 cache Miss*) dargestellt. Diese Wartezeit berechnet sich aus der Anzahl der blockierten Zyklen, die durch die PAPI-Bibliothek gemessen werden, und der eingestellten Prozessorfrequenz wie folgt:

$$\text{Wartezeit} = \text{blockierte Zyklen} \cdot \text{Länge eines Zyklus} = \frac{\text{blockierte Zyklen}}{\text{Prozessorfrequenz}}$$

In Abb. 4.10 zeigt sich für jeden der drei Prozessoren eine erhöhte Wartezeit in den vektorisierten Programmvarianten. Bei der Sandy Bridge Architektur steigt die Wartezeit mit steigender Prozessorfrequenz weiter an. Bei der Haswell und Skylake Architektur sinken die Werte für kleinere Prozessorfrequenzen und steigen ab einem bestimmten Wert wieder an. Die Prozessorfrequenz, für welche die Wartezeit dabei minimal wird, liegt in etwa bei der Speichertaktrate (Sandy Bridge: 1,3 GHz; Haswell 1,3 GHz; Skylake: 2,1 GHz).

Die Verbesserungen der Haswell und vor allem der Skylake Architektur zeigen vor allem bei der seriellen Programmvariante eine Reduzierung der Ausführungszeit. Bei-

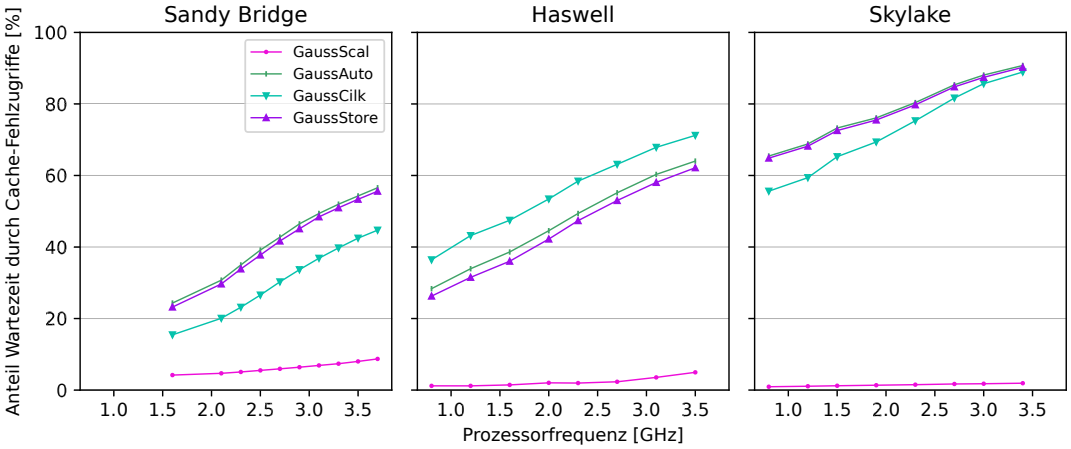

Abbildung 4.11.: Anteil der Wartezeit durch Cache-Fehlzugriffe aus Abb. 4.10 an der Gesamtausführungszeit der jeweiligen Programmvariante in Prozent. Gezeigt werden die Werte für die drei genutzten Prozessorarchitekturen: Sandy Bridge (links), Haswell (mitte) und Skylake (links) bei einer Matrixgröße von $10\,000 \times 10\,000$ (vgl. Ausführungszeiten in Abb. 4.9).

spiele für solche Verbesserungen sind Verbesserungen im Bereich des Speichervorgriffs (*engl. pre-fetching*) und eine erhöhte Pipelinetiefe [58]. Bei den vektorisierten Programmvarianten haben diese Prozessorverbesserungen zwar eine Auswirkung auf die Ausführungszeit, erhöhen dabei aber auch die Zeit, in der Prozessorzyklen blockiert sind. Dies zeigt die Begrenzung der vektorisierten Programmausführungen durch die Speicheranbindung im Vergleich zu Begrenzungen durch Eigenschaften in der Prozessorarchitektur.

Anteil der Wartezeit durch blockierte Zyklen an der Ausführungszeit

Abbildung 4.11 zeigt den Anteil der Ausführungszeit, der bei der Ausführung der Programmvarianten in blockierten Zyklen verbracht wird. Die serielle Programmvariante hat die höchste Ausführungszeit und die geringste Wartezeit, sodass der Großteil der Ausführungszeit für Berechnungen genutzt wird. Bei den vektorisierten Programmvarianten wird ein Großteil der Ausführungszeit auf die benötigten Speicheroperationen gewartet. Das Maximum wird dabei auf der Skylake Architektur bei der höchsten Prozessorfrequenz erreicht: Hier warten die vektorisierten Programmvarianten bis zu 90% ihrer Ausführungszeit auf Daten. Die geringere Ausführungszeit der vektorisierten Programmvarianten zeigt dabei, dass die Berechnungen um ein vielfaches schneller ausgeführt werden können als in der seriellen Programmvariante.

Bei allen Programmvarianten steigt der Anteil der Wartezeit mit steigender Frequenz an. Vor allem bei der seriellen Programmvariante, deren Ausführungszeit und absolute Wartezeit mit steigender Frequenz sinken, zeigt dies den Einfluss der steigenden Prozessorfrequenz auf die Berechnungsgeschwindigkeit, während die Datentransferrate unverändert bleibt.

Der Anstieg des Anteils der Wartezeit an der Ausführungszeit der vektorisierten

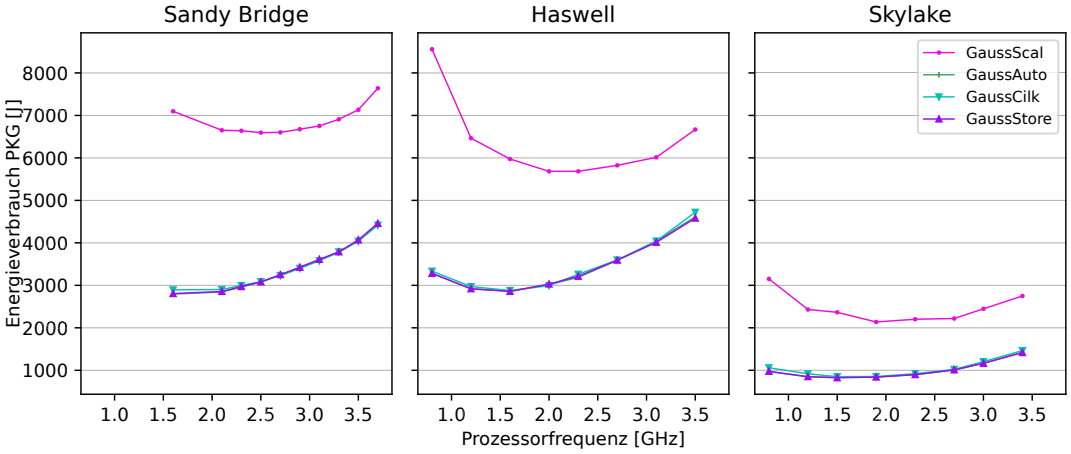

Abbildung 4.12.: Energieverbrauch des Prozessors (PKG) bei der Ausführung der Programmvarianten mit unterschiedlichen Vektorisierungstechniken in Abhängigkeit zur eingestellten Prozessorfrequenz auf den drei genutzten Prozessorarchitekturen: Sandy Bridge (links), Haswell (mitte) und Skylake (links) bei einer Matrixgröße von $10\,000 \times 10\,000$. Der Energieverbrauch wird durch das Auslesen der Prozessorregister des RAPL Interfaces gemessen.

Programmvarianten spiegelt sich in der geringeren Reduzierung der Ausführungszeit in Abb. 4.9 wider.

4.3.3.2. Energieverbrauch der vektorisierten Programmvarianten

Ein niedriger Energieverbrauch von Programmausführungen ergänzt kurze Ausführungszeiten um ein weiteres Optimierungsziel in der Softwareentwicklung. In diesem Abschnitt wird der Energieverbrauch und die Leistungsaufnahme der vektorisierten Programmvarianten diskutiert.

Der Energieverbrauch der Programmausführung wird mit Hilfe des RAPL Interface aus Prozessorregistern ausgelesen. In Abb. 4.12 werden die Energieverbrauchswerte des Prozessors (Package, PKG) dargestellt. Das Verhältnis des Energieverbrauchs der Programmvarianten zueinander ist dabei ähnlich zum Verhältnis der Ausführungszeiten.

Der Energieverbrauch der seriellen Programmvariante *GaussScal* zeigt ein typisches Verhalten, bei dem der minimale Energieverbrauch bei einer mittleren Prozessorfrequenz (ca. 2 GHz) erreicht wird. In [65] wird dieses Verhalten mit der Abhängigkeit des Energieverbrauchs zu einem Skalierungsfaktor für die Prozessorfrequenz gezeigt.

Die Ausführung der vektorisierten Programmvarianten zeigt einen wesentlich geringeren Energieverbrauch. Das Minimum des Energieverbrauchs ist dabei, im Vergleich zur seriellen Programmvariante, zu einer geringeren Prozessorfrequenz verschoben. Bei der Haswell und der Skylake Architektur liegt dieses Minimum bei ca. 1,5 GHz. In [65] wird diese Verschiebung des Energieverbrauchs durch eine Erhöhung des dynamischen Energieverbrauchs erklärt, der zum Beispiel durch eine höhere Anzahl aktiver Transistoren hervorgerufen wird. Bei der Sandy Bridge Architektur lässt sich ein ähnli-

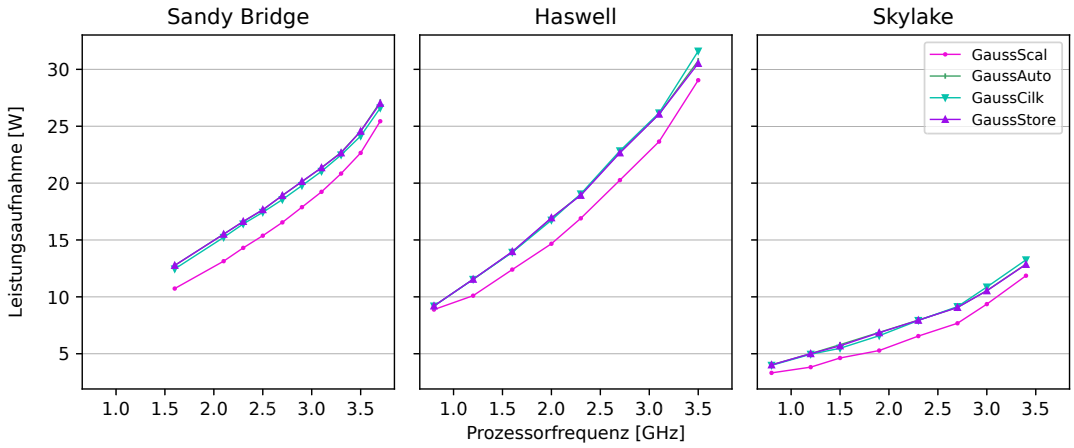

Abbildung 4.13.: Leistungsaufnahme der verwendeten Prozessoren in Abhängigkeit zur eingestellten Prozessorfrequenz bei der Ausführung der Programmvarianten mit unterschiedlichen Vektorisierungstechniken. Die Leistungsaufnahme wird aus den gemessenen Werten für Ausführungszeit und Energieverbrauch berechnet. Gezeigt werden die Werte für die drei genutzten Prozessorarchitekturen: Sandy Bridge (links), Haswell (mitte) und Skylake (links) bei einer Matrixgröße von $10\,000 \times 10\,000$.

cher Verlauf des Energieverbrauchs erwarten, die Prozessorfrequenz kann dafür jedoch nicht niedrig genug eingestellt werden.

4.3.3.3. Leistungsaufnahme der vektorisierten Programmvarianten

Energieverbrauch und Ausführungszeit der Programmvarianten sind miteinander verbunden, da eine längere Aktivierung des Prozessors einen höheren Grundenergieverbrauch (statischer Energieverbrauch) des Prozessors zur Folge hat. Zur Verdeutlichung der unterschiedlichen Veränderung zwischen Ausführungszeit und Energieverbrauch wird die Leistungsaufnahme betrachtet. Die Leistungsaufnahme wird durch die folgende Formel berechnet:

$$\text{Leistungsaufnahme}[W] = \frac{\text{Energieverbrauch}[J]}{\text{Ausführungszeit}[s]}$$

Die Leistungsaufnahme wird in Abb. 4.13 für die drei genutzten Prozessorarchitekturen gezeigt. Mit steigender Prozessorfrequenz steigt auch die Leistungsaufnahme der einzelnen Programmvarianten. Die Beziehung der Programmvarianten zueinander verhält sich dabei invers zum Verhältnis von Ausführungszeit beziehungsweise Energieverbrauch. Eine höhere Leistungsaufnahme bei der Programmausführung deutet auf eine Erhöhung des dynamischen Anteils des Energieverbrauchs hin. In der Modellierung aus [65] wird dies durch eine Erhöhung der Schaltwahrscheinlichkeit von Transistoren, die aus einer Erhöhung der Anzahl genutzter Transistoren abgeleitet werden kann, widergespiegelt.

Bei der Betrachtung der Leistungsaufnahme der Prozessoren fällt besonders die niedrige Leistungsaufnahme des Skylake Prozessors auf. Diese beruht im wesentlichen

auf der starken Reduzierung des Energieverbrauchs, bei nur leichter Reduzierung der Ausführungszeit.

Die Leistungsaufnahme der Programmvarianten wird hier als Durchschnittswert über die Ausführungszeit der jeweiligen Programmvariante angeben. Eine verkürzte Ausführungszeit, nach welcher der Prozessor nach der Bearbeitung in einen Schlafzustand versetzt wird, kann also eine höhere Leistungsaufnahme aufweisen als eine Programmausführung mit höherer Ausführungszeit. Eine Reduzierung der Leistungsaufnahme der Programmausführung kann somit auch dadurch erreicht werden, dass die Messung der Programmvarianten in einem festgelegten Zeitintervall stattfindet. Nach Beendigung der Berechnungen wird für die Prozessoren nur noch der Energieverbrauch des Prozessors im Schlafzustand bis zum Ende des Messintervalls gemessen. Für die vektorisierten Programmvarianten bedeutet dies, dass zwar eine höhere Momentan-Leistungsaufnahme gemessen wird, die durchschnittliche Leistungsaufnahme jedoch durch eine Verzögerung des Folgeprogramms rechnerisch reduziert werden kann.

Ähnliches gilt für die Wahl einer reduzierten Prozessorfrequenz zur Ausführung der vektorisierten Programmvarianten. Durch die niedrigere Prozessorfrequenz wird eine geringere Leistungsaufnahme gegenüber der seriellen Programmvariante auf der ursprünglichen Prozessorfrequenz erreicht. Die Reduzierung der Prozessorfrequenz erzeugt dabei eine höhere Ausführungszeit, die jedoch durch die Einsparung bei der Vektorisierung ausgeglichen werden kann. Auf diese Weise kann ein Programm eine Reduktion von Ausführungszeit, Energieverbrauch und Leistungsaufnahme durch die Nutzung von Vektorisierungstechniken erreichen.

4.3.4. Ausführungszeit und Energieverbrauch der AVX-Intrinsic-Programmvarianten der Gauss-Elimination

Bei der Programmierung mit AVX-Intrinsics hat der Programmierer verschiedene Möglichkeiten zur Implementierung eines Algorithmus. Die verschiedenen AVX-Intrinsic-Programmvarianten der Gauss-Elimination zeigen eine Auswahl von Implementierungsvarianten in Bezug auf die verwendeten Lade- und Speicheroperationen. Das generelle Verhalten von vektorisierten Programmvarianten ist bereits im vorherigen Abschnitt beschrieben, weshalb in diesem Abschnitt ausschließlich die Unterschiede der AVX-Implementierungsvarianten aus Abschn. 4.2.2 besprochen werden.

4.3.4.1. Ausführungszeit der AVX-Gauss-Elimination

Die Ausführungszeiten der AVX-Intrinsic-Programmvarianten sind in Abb. 4.14 dargestellt. Auf den drei genutzten Prozessoren zeigt die *GaussStream* Programmvariante die höchste Ausführungszeit. Die Speicherzugriffe der *GaussStream* Programmvariante werden auf ausgerichteten Speicherzellen ausgeführt, was ein vergleichbares Verhalten zur *GaussStore* Programmvariante erzeugen müsste. Der Unterschied der beiden Programmvarianten liegt in der Art der Schreiboperation: Die *GaussStream* Programmvariante nutzt non-temporal Schreiboperationen. Bei normalen (Write-Back) Speiche-

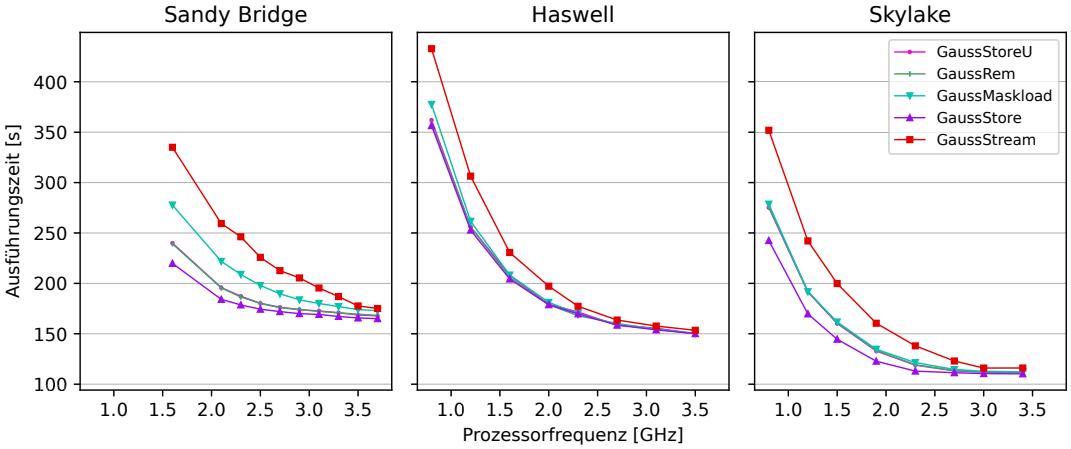

Abbildung 4.14.: Ausführungszeit der AVX-Intrinsic-Programmvarianten auf den verwendeten Prozessorarchitekturen Sandy Bridge (links), Haswell (mitte) und Skylake(rechts) bei einer Matrixgröße von $10\,000 \times 10\,000$ in Abhängigkeit zur eingestellten Prozessorfrequenz. Die Unterschiede der AVX-Intrinsic-Programmvarianten werden in Abschn. 4.2.2 beschrieben.

roperationen wird das Ergebnis der Berechnung in den Cache geschrieben und erst zu einem späteren Zeitpunkt in den Hauptspeicher zurückgeschrieben. Bei non-temporal Schreiboperationen wird das Ergebnis direkt in den Hauptspeicher geschrieben, sodass der Cache „umgangen“ wird und alle Kopien in den Caches als ungültig markiert werden. Dies kann dazu führen, dass auf die Beendigung einer Schreiboperation gewartet werden muss, bis die nächste Schreiboperation ausgeführt wird. Im Falle der Gauss-Elimination, die viele Lade- und Speicheroperationen und vergleichbar wenige Berechnungsoperationen hat, kann dies zu erhöhten Wartezeiten führen, die bei rechen-intensiveren Algorithmen nicht auftreten.

Mit der Sandy Bridge Architektur wurden erstmalig AVX-Instruktionen verfügbar, die in den darauffolgenden Prozessorarchitekturen fortlaufend verbessert wurden. Ein Beispiel für diese Verbesserungen lässt sich mit der *GaussMaskload* Programmvariante zeigen. Auf der Sandy Bridge Architektur zeigt die Programmvariante mit einer erhöhten Anzahl an maskierten Ladeoperationen eine höhere Ausführungszeit als beispielsweise die *GaussStoreU* Programmvariante. Mit der Haswell Architektur verringert sich die höhere Ausführungszeit und auf der Skylake Architektur ist die Ausführungszeit fast identisch zur *GaussStoreU* Programmvariante. Auf der Skylake Architektur gibt es damit keinen messbaren Unterschied in der Ausführungszeit zwischen einer maskierten Ladeoperation (mit vollbesetzter Maske) und einer `loadu` Instruktion.

Die *GaussRem* Programmvariante nutzt zur Berechnung der letzten Elemente der Matrixzeile eine Schleife mit der diese Elemente berechnet werden. Außer der Berechnungsweise dieser Elemente hat die *GaussRem* Programmvariante keine Unterschiede zur *GaussStoreU* Programmvariante. Der Unterschied zwischen serieller und vektorisierter Berechnung der letzten Elemente der Zeile zeigt sich jedoch nicht in der Ausführungszeit der beiden Programmvarianten in Abb. 4.14. Die beiden Programmvarianten

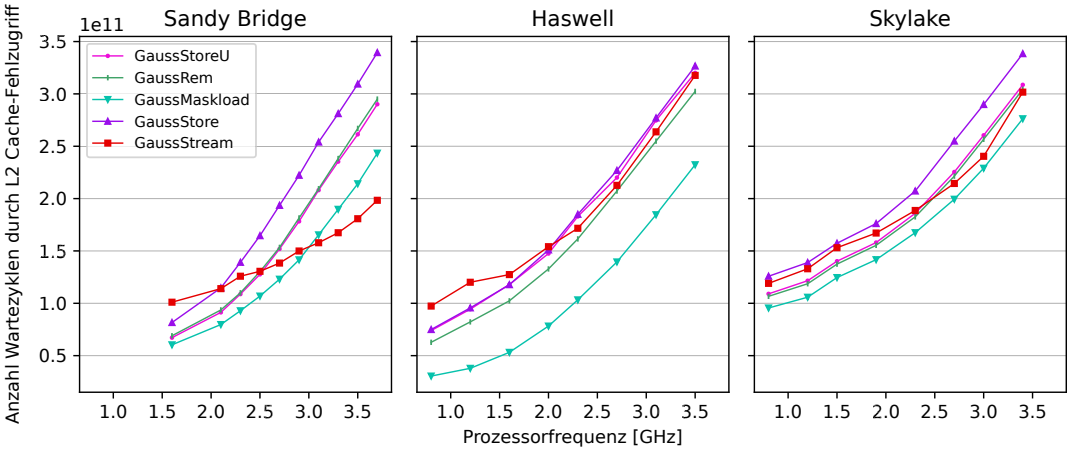

Abbildung 4.15.: Anzahl der Zyklen, in denen die Ausführung der AVX-Intrinsic-Programmvarianten blockiert ist, da auf eine Lade- oder Speicheroperationen gewartet wird, die durch einen Level 2 Cache-Fehlzugriff ausgelöst wurde. Die Werte werden in Abhängigkeit zur eingestellten Prozessorfrequenz auf den drei genutzten Architekturen Sandy Bridge (links), Haswell (mitte) und Skylake (rechts) bei einer Matrixgröße von $10\,000 \times 10\,000$ gezeigt (vgl. Ausführungszeit in Abb. 4.14).

haben auf den drei betrachteten Architekturen nahezu identische Ausführungszeiten, mit maximalen Abweichungen von weniger als 0,5% zueinander. Dies zeigt, dass sich der Einsatz von maskierten Operationen zur Berechnung der letzten Elemente der Zeile gegenüber seriellen Schleifen rechnet, obwohl der Einsatz von maskierten Operationen (vgl. *GaussMaskload*) gegenüber anderen AVX-Instruktionen die Ausführungszeit verlängern kann. Vor allem bei neueren Architekturen oder Algorithmen mit einer höheren Anzahl Berechnungen pro Lade-/Speicheroperation kann dies zu einer weiteren Reduzierung der Ausführungszeit gegenüber der seriellen Programmvariante führen.

Auf den drei genutzten Prozessorarchitekturen hat die *GaussStore* Programmvariante die geringste Ausführungszeit. Die geringere Ausführungszeit lässt sich durch die ausgerichteten Speicherzugriffe erklären, bei denen jeweils eine komplette Cachezeile geladen werden kann. Auf der Sandy Bridge und Skylake Architektur ist der Unterschied zu den anderen Programmvarianten deutlich erkennbar, während dieser bei der Haswell Architektur nur sehr gering ist. Vermutlich wurde bei der Weiterentwicklung der Haswell Architektur die Ausführung von unausgerichteten Lade- und Speicheroperationen verbessert und bei der Entwicklung der Skylake Architektur die Ausführung von ausgerichteten Lade- und Speicheroperationen.

Abbildung 4.15 zeigt die Anzahl der Prozessorzyklen in denen auf Speicheroperationen aufgrund eines Level 2 Cache-Fehlzugriffs (L2 Cache Miss) gewartet wird. Das Verhältnis der Kurven zu den meisten Programmvarianten ist invers zu deren Verhältnis bei der Ausführungszeit. Dies begründet sich darin, dass die begrenzte Speicherbandbreite des Prozessors bei Verkürzung der Zeit für Berechnungen deutlicher hervortritt.

Bei der *GaussStream* Programmvariante ist ein schwächerer Anstieg der Anzahl blo-

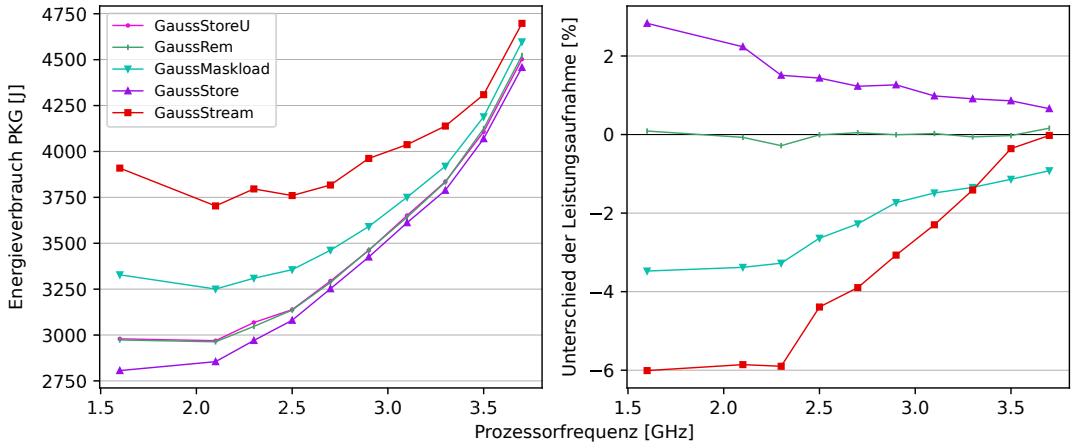

Abbildung 4.16.: Energieverbrauch (links) und Leistungsaufnahme (rechts) der AVX-Intrinsic-Programmvarianten bei der Ausführung auf der **Sandy Bridge** Prozessorarchitektur in Abhängigkeit zur eingestellten Prozessorfrequenz bei einer Matrixgröße von $10\,000 \times 10\,000$. Die Leistungsaufnahme ist als Unterschied zur *GaussStoreU* Programmvariante in Prozent dargestellt.

ckierter Zyklen bei steigender Prozessorfrequenz zu beobachten. Diese Beobachtung widerspricht der aufgrund der höheren Ausführungszeit erwarteten höheren Wartezeit der Programmvariante. In der *GaussStream* Programmvariante schreibt die stream Instruktion die Daten nicht in den Cache, sondern direkt in den Hauptspeicher zurück. Dies bedeutet, dass für die Ausführung der Instruktion kein blockierter Prozessorzyklus auf Basis eines Cache-Fehlzugriffs entstehen kann, es können nur blockierte Zyklen aufgrund von Ladeoperationen gezählt werden. Betrachtet man die blockierten Zyklen bezüglich der Wartezeit auf allgemeine Ressourcen (vgl. Anhang Abb. C.3), ist dort die Anzahl der blockierten Zyklen für die *GaussStream* Programmvariante wesentlich höher als für die anderen Programmvarianten.

Die drei genutzten Prozessorarchitekturen zeigen im allgemeinen das gleiche Verhalten für die *GaussStream* Programmvariante, bei der Sandy Bridge Architektur ist die flachere Kurve jedoch am deutlichsten erkennbar. Bei den blockierten Zyklen bezüglich allgemeiner Ressourcen ist dieser Unterschied nicht erkennbar.

4.3.4.2. Energieverbrauch und Leistungsaufnahme der AVX-Gauss-Elimination

Neben der Ausführungszeit ist der Energieverbrauch von Programmen ein Optimierungsziel bei der Programmausführung. Das Verhalten von vektorisierten Programmen wurde bereits in Abschn. 4.3.3 untersucht. In diesem Abschnitt wird daher untersucht, wie die Auswahl verschiedener AVX-Instruktionen dieses Verhalten beeinflusst.

Abbildung 4.16 zeigt den Energieverbrauch und die Leistungsaufnahme der AVX-Intrinsic-Programmvarianten für die Sandy Bridge Architektur und Abb. 4.17 für die Skylake Architektur. Die Werte für die Haswell Architektur finden sich im Anhang in Abb. C.4.

Der Energieverbrauch eines Programms hängt von der Ausführungszeit dieses Programmes ab. Diese Abhängigkeit begründet sich durch die Aktivierungszeit des Prozessors (vgl. [65]). In den Abb. 4.16 und 4.17 zeigt sich diese Abhängigkeit für den Energieverbrauch der AVX-Intrinsic-Programmvarianten durch die gleiche Relation der Programmvarianten zueinander.

Für die AVX-Intrinsic-Programmvarianten zeigt sich ebenfalls der niedrigste Energieverbrauch für mittlere Prozessorfrequenzen (vgl. Abschn. 4.3.3). Bei den AVX-Intrinsic-Programmvarianten treten jedoch zwei Effekte auf, die vom erwarteten Verhalten abweichen:

- Für höhere Frequenzen verringert sich der Unterschied im Energieverbrauch der Programmvarianten.
- Bei einigen Programmvarianten ist der Verlauf der Energieverbrauchskurve flacher, was vor allem im Bereich des erwarteten Minimums deutlich wird.

Bei höheren Frequenzen hängt ein Großteil der Ausführungszeit von der Speicheranbindung ab (vgl. Abb. 4.11). Da hierdurch der Einfluss der ausgeführten Instruktionen auf den Energieverbrauch sinkt, verringert sich auch der Einfluss durch die Auswahl von unterschiedlichen AVX-Instruktionen. Dieser Effekt kann bereits bei der Ausführungszeit der Programmvarianten (vgl. Abb. 4.14) beobachtet werden und zeigt sich noch deutlicher beim Energieverbrauch.

Die *GaussStream* Programmvariante erzeugt eine ähnliche Kurve des Energieverbrauchs wie die anderen Programmvarianten. Im Bereich des erwarteten Minimums auf der Haswell und Skylake Architektur ist die Kurve des Energieverbrauchs jedoch flacher. Ein ähnliches Verhalten zeigt die *GaussMaskload* Programmvariante. Eine Begründung für den flacheren Verlauf findet sich in der Abhängigkeit zur Speichertaktrate: Die beiden Programmvarianten mit dem flacheren Kurvenverlauf nutzen zum Laden und Speichern der Daten Operationen, bei denen mehr Instruktionen (*GaussMaskload*) bzw. kompliziertere Instruktionen (*GaussStream*) genutzt werden. Die dafür benötigte Ausführungszeit lässt sich teilweise durch die Wartezeiten auf Daten verdecken, weniger jedoch beim Energieverbrauch. Besonders deutlich tritt dies dabei im Bereich des erwarteten Minimums auf.

Leistungsausfnahme der AVX-Intrinsic-Programmvarianten

Die Leistungsaufnahme wird in den Abb. 4.16 und 4.17 als Unterschied zur *GaussStoreU* Programmvariante in Prozent dargestellt. Die serielle Berechnung der letzten Elemente der Matrixzeile in der *GaussRem* Programmvariante zeigt auch hier kaum ($< 0,5\%$) einen Unterschied zur *GaussStoreU* Programmvariante.

Die *GaussMaskload* Programmvariante zeigt auf der Sandy Bridge Architektur eine geringere Leistungsaufnahme als die Vergleichsvariante. Auf der Haswell und Skylake Architektur hat diese hingegen eine höhere Leistungsaufnahme als die Vergleichsvariante. Dieses Verhalten erklärt sich durch den erhöhten Energieverbrauch im Vergleich zu gleicher Ausführungszeit. Für die Sandy Bridge Architektur erhöhen sich sowohl

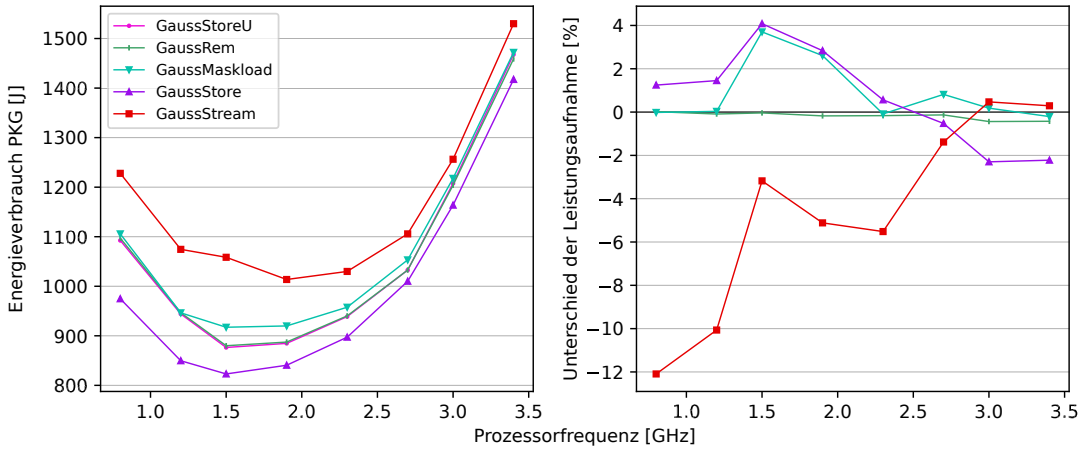

Abbildung 4.17.: Energieverbrauch (links) und Leistungsaufnahme (rechts) der AVX-Intrinsic-Programmvarianten bei der Ausführung auf der **Skylake** Prozessorarchitektur in Abhängigkeit zur eingestellten Prozessorfrequenz bei einer Matrixgröße von $10\,000 \times 10\,000$. Die Leistungsaufnahme ist als Unterschied zur *GaussStoreU* Programmvariante in Prozent dargestellt.

Energieverbrauch (+12% bei 1,6 GHz) als auch Ausführungszeit (+16% bei 1,6 GHz). Der unterschiedliche Anstieg von Energieverbrauch und Ausführungszeit führt zur reduzierten berechneten Leistungsaufnahme.

Bei der *GaussStream* Programmvariante wird auch bei geringeren Prozessorfrequenzen auf die Fertigstellung von Speicheroperationen gewartet. Während diesen Wartezeiten werden weniger Transistoren des Prozessors genutzt, wodurch sich das Verhältnis von Ausführungszeit und Energieverbrauch verschiebt (vgl. [65]). Dadurch entsteht bei geringeren Prozessorfrequenzen bei der *GaussStream* Programmvariante eine reduzierte Leistungsaufnahme.

Die Nutzung von ausgerichteten Lade- und Speicheroperationen führt zum geringsten Energieverbrauch auf allen Architekturen. Für die Haswell Architektur ist die Ausführungszeit jedoch nahezu identisch zur *GaussStoreU* Programmvariante, wodurch hier eine reduzierte Leistungsaufnahme erreicht wird. Auf der Skylake Architektur ist für die *GaussStore* Programmvariante ebenfalls ein flacherer Verlauf des Energieverbrauchs in Abhängigkeit zur Prozessorfrequenz zu beobachten. Für größere Frequenzen nähert sich der Energieverbrauch jedoch einem festen Abstand zur *GaussStoreU* Programmvariante an. Der Abstand wird für den Energieverbrauch ab einer Prozessorfrequenz von 2,0 GHz konstant, für die Ausführungszeit jedoch erst ab 2,7 GHz, sodass ab 2,7 GHz die *GaussStore* Programmvariante einen reduzierten Energieverbrauch und eine reduzierte Leistungsaufnahme zeigt.

Energy Delay Product

Das Energy Delay Product (EDP) [25] wird aus der Ausführungszeit t und dem Energieverbrauch E der Programmausführung nach der Formel $EDP = E \cdot t$ berechnet. Abbildung 4.18 zeigt das Energy Delay Product der AVX-Intrinsic-Programm-

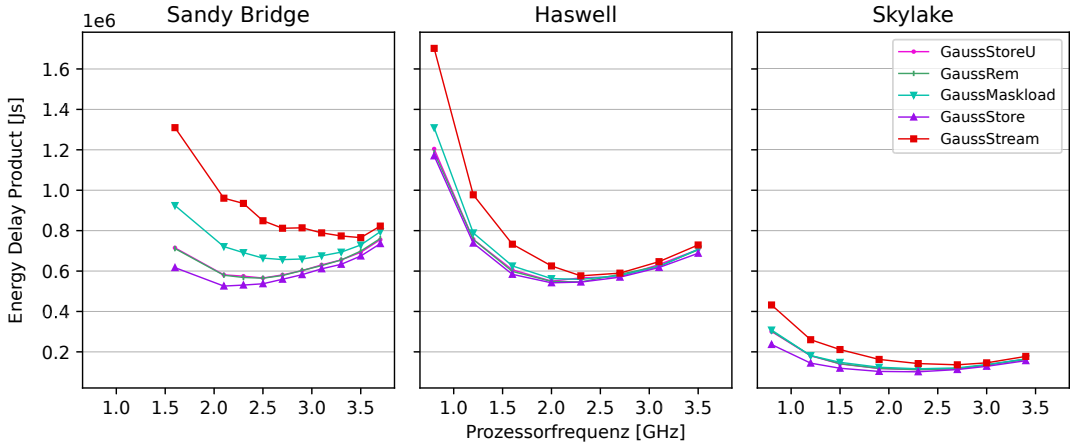

Abbildung 4.18.: Energy Delay Product der AVX-Intrinsic-Programmvarianten. Die Werte sind berechnet aus den gemessenen Werten der Ausführungszeit und des Energieverbrauchs der jeweiligen Programmvariante. Die Werte werden in Abhängigkeit zur eingestellten Prozessorfrequenz auf den drei genutzten Architekturen Sandy Bridge (links), Haswell (mitte) und Skylake (rechts) bei einer Matrixgröße von $10\,000 \times 10\,000$ gezeigt. Kleinere Werte sind besser.

varianten für die drei genutzten Prozessorarchitekturen. Die Programmvarianten mit geringeren Werten für Energieverbrauch und Ausführungszeit haben auch ein geringeres EDP, womit die Relation der Programmvarianten zueinander auf der jeweiligen Prozessorarchitektur gleich bleibt. Im Gegensatz zur Betrachtung der Ausführungszeit und des Energieverbrauchs bei denen sich die *GaussStoreU* und die *GaussRem* Programmvariante kaum unterscheiden, hat die *GaussRem* Programmvariante leicht geringere Werte für das EDP. Durch die Berechnung des EDP werden eine längere Ausführungszeit und eine Reduktion des Energieverbrauchs gegeneinander abgewogen. Daher werden die geringsten Werte des EDP bei einer höheren Frequenz erreicht, als das Minimum des Energieverbrauchs.

Bei der Ausführung der Programmvarianten auf der Skylake Architektur werden die geringsten Werte für das EDP berechnet, was für die drei untersuchten Prozessorarchitekturen zur besten Energie-Effizienz führt. Für die Haswell Architektur werden die höchsten EDP-Werte erreicht, was vor allem durch die geringe Reduktion der Ausführungszeit bei nahezu gleichem Energieverbrauch (gegenüber der Sandy Bridge Architektur) begründet ist.

XY-Diagramm von Ausführungszeit und Energieverbrauch

Eine alternative Möglichkeit zur gleichzeitigen Betrachtung von Ausführungszeit und Energieverbrauch ist die Darstellung beider Messwerte in einem XY-Diagramm. Dazu werden die Werte der Ausführungszeit auf der Y-Achse und die Werte des Energieverbrauchs auf der X-Achse des Diagramms angetragen. Bei dieser Darstellung werden die Kombinationen aus den folgenden Varianten eingetragen:

- 5 AVX-Intrinsic-Programmvarianten,

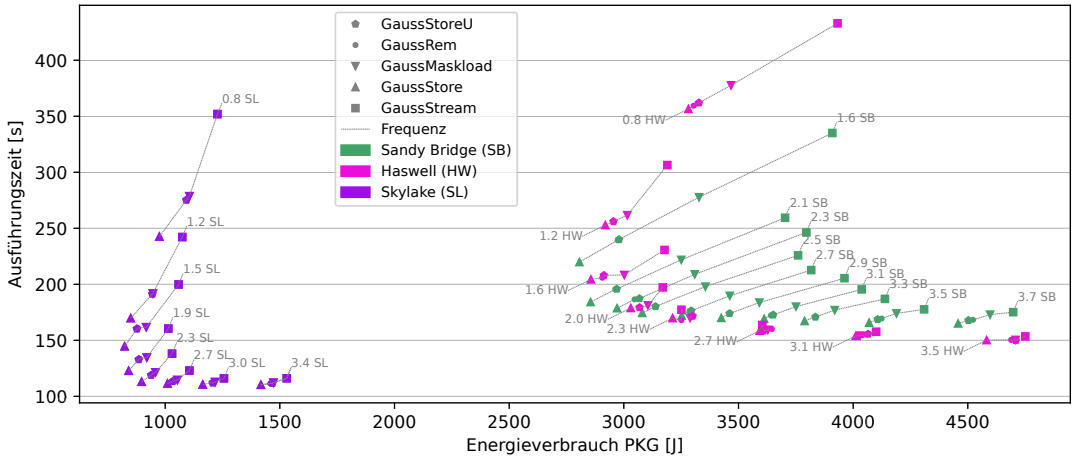

Abbildung 4.19.: Ausführungszeit und Energieverbrauch der AVX-Intrinsic-Programmvarianten für verschiedene Prozessorfrequenzen und Prozessorarchitekturen als XY-Diagramm. Dargestellt sind die Werte bei einer Matrixgröße von $10\,000 \times 10\,000$. Die eingestellte Prozessorfrequenz der dargestellten Messwerte wird als annotation in GHz in Verbindung mit der jeweils genutzten Prozessorarchitektur (Sandy Bridge [SB], Haswell [HW], Skylake [SL]) angegeben. Kleinere Werte sind besser.

- 3 Prozessorarchitekturen und
- 8 Prozessorfrequenzen (10 für Sandy Bridge).

Die insgesamt 390 Messpunkte werden entsprechend ihrer Ausführungszeit und dem Energieverbrauch in Abb. 4.19 eingetragen.

In Abb. 4.19 zeigt die Ausführung auf der Skylake Prozessorarchitektur die geringsten Werte für Ausführungszeit und Energieverbrauch. Durch die Darstellung im XY-Diagramm lässt sich die Programmvariante, Prozessorfrequenz und Prozessorarchitektur mit der minimalen Ausführungszeit, bzw. dem minimalen Energieverbrauch, in der linken unteren Ecke ablesen. Die minimalen Werte werden durch die *GaussStore* Programmvariante bei Ausführung auf der Skylake Architektur erhalten. Zusätzlich zeigt sich bei dieser Darstellung, dass die Wahl bestimmter Prozessorfrequenzen für die Ausführung einer Programmvariante keine Vorteile bringt. Ein Beispiel hierfür ist die Ausführung der *GaussStore* Programmvariante auf der Skylake Architektur, die bei einer Prozessorfrequenz von 1,2 GHz eine höhere Ausführungszeit und einen höheren Energieverbrauch hat, als bei einer Ausführung mit 1,5 GHz.

Abbildung 4.20 zeigt einen Ausschnitt der Messpunkte aus Abb. 4.19, bei dem Werte für die Sandy Bridge und Haswell Architektur genauer dargestellt werden. In der Abbildung wird deutlich inwieweit ein Kompromiss zwischen Ausführungszeit und Energieverbrauch als Optimierungskriterien erfolgen kann. Eine Reduzierung eines der Optimierungskriterien erzeugt eine Erhöhung des Wertes für das andere Optimierungskriterium. Die Reduzierung und Erhöhung stehen dabei nicht im gleichen Verhältnis, sodass eine Auswahl an Messpunkten entsteht, die jeweils ein Teiloptimum darstellen. Die Auswahl der besten Programmvariante, Prozessorfrequenz und Pro-

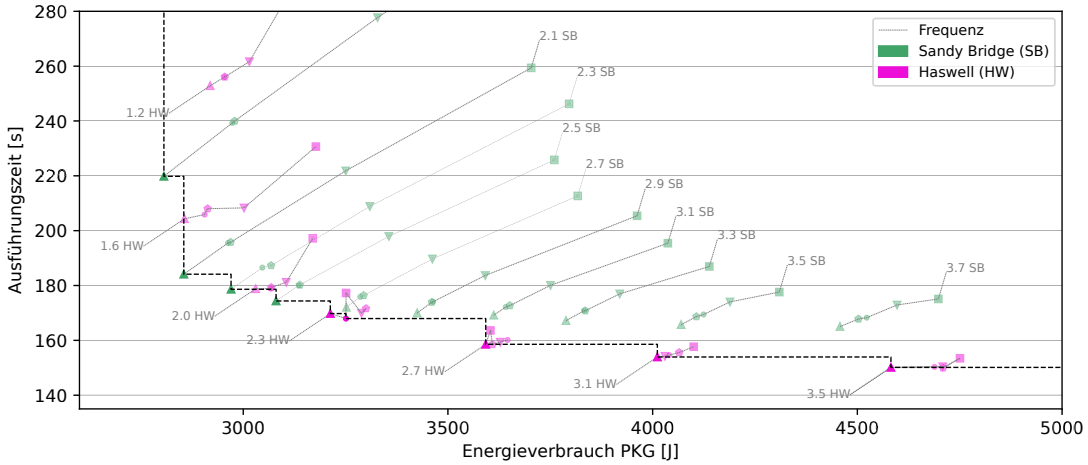

Abbildung 4.20.: Ausschnitt der Ausführungszeit und des Energieverbrauchs der AVX-Intrinsic-Programmvarianten aus Abb. 4.19. Hervorgehoben sind die Messpunkte, die ein Teiloptimum der Ausführungszeit und des Energieverbrauchs sind.

zessorarchitektur erfolgt somit mit anwendungsspezifischen Kriterien und stellt dafür einen Kompromiss zwischen geringer Ausführungszeit und geringem Energieverbrauch dar.

In Abb. 4.20 sind die teiloptimalen Messwerte hervorgehoben und verbunden. Bei der Auswahl eines passenden Kompromisses für Ausführungszeit und Energieverbrauch spielen solche Messwerte, die nicht auf dieser Verbindung liegen, keine Rolle, da sich hier immer beide Optimierungskriterien zu einem anderen Messpunkt reduzieren lassen. Im Gegensatz dazu gilt für die Messwerte auf dieser Verbindung: Eine Reduktion eines der Optimierungskriterien lässt sich nur durch die Erhöhung des anderen Optimierungskriteriums erreichen.

Abbildung 4.20 zeigt, dass sich für einen geringeren Energieverbrauch die *Gauss-Store* Programmvariante auf der Sandy Bridge Architektur am besten geeignet ist. Eine Reduktion der Ausführungszeit kann dabei nur erreicht werden, wenn eine Erhöhung des Energieverbrauchs in Kauf genommen wird. Ab einem gewissen Punkt (ca. 170 Sekunden) bietet dabei die Haswell Architektur die geringeren Messwerte für die Ausführungszeit. Die Prozessorfrequenz kann dabei für die Haswell Architektur sogar niedriger gewählt werden als bei der Sandy Bridge Architektur, und dennoch eine Reduktion der Ausführungszeit erreichen.

4.4. Zusammenfassung der Vektorisierung der Gauss-Elimination

Bei der Gauss-Elimination zeigen sowohl die automatische Vektorisierung durch den Compiler als auch die Nutzung der Cilk Array-Notation eine deutliche Reduzierung der Ausführungszeit und des Energieverbrauchs. Die Nutzung der Cilk Array-Notation

zeigt dabei eine geringere Reduzierung als die automatisch vektorisierte Programmvariante. Die automatische Vektorisierung erzielt bei der Gauss-Elimination die gleichen Werte, wie eine manuelle Vektorisierung durch AVX-Intrinsics. Die Abhängigkeit der Ausführungszeit von der Speicheranbindung ist ein Faktor, der vor allem durch die Vektorisierung, zu einer Limitierung der erreichten Performance führt. Dies wird deutlich durch den hohen Anteil an blockierten Zyklen, in denen der Prozessor auf Daten wartet und aus diesem Grund keine Berechnungen durchführen kann.

Der Energieverbrauch der vektorisierten Programmvarianten reduziert sich bei der Gauss-Elimination in ähnlichem Maße wie die Ausführungszeit. Betrachtet man dabei die Leistungsaufnahme des Prozessors wird jedoch deutlich, dass die Einsparungen der Ausführungszeit größer sind als die des Energieverbrauchs. Generell lässt sich jedoch in den meisten Fällen durch die Wahl einer geringeren Prozessorfrequenz eine Reduktion aller drei Kennwerte gegenüber der seriellen Ausführung erreichen.

Von den drei untersuchten Prozessorarchitekturen sind die Messwerte der vektorisierten Gauss-Elimination für Ausführungszeit und Energieverbrauch auf der Skylake Architektur am geringsten. Zwischen den Messwerten der Sandy Bridge und Haswell Architektur besteht keine klare Abgrenzung, sodass die beste Kombination aus Prozessorarchitektur und Prozessorfrequenz jeweils im Einzelfall bestimmt werden muss.

Bei der Vektorisierung mit AVX-Intrinsics kann die Auswahl der genutzten AVX-Instruktionen einen Einfluss auf die Laufzeitparameter der Programmausführung haben. Für die Gauss-Elimination zeigt die Verwendung der stream Instruktion zur Ausführung von non-temporal Schreiboperationen die schlechtesten Werte. Dies begründet sich vor allem durch die hohe Abhängigkeit zur Speicheranbindung und die daraus resultierenden Wartezeiten.

Die besten Werte für Ausführungszeit und Energieverbrauch bei der Wahl der AVX-Instruktionen erzielt die Nutzung von ausgerichteten Lade- und Speicheroperationen. Obwohl hier zusätzliche Matrixelemente für die Berechnung genutzt werden, zeigt die entsprechende Programmvariante die geringste Ausführungszeit, den geringsten Energieverbrauch und in einigen Fällen die geringste Leistungsaufnahme der vektorisierten Programmvarianten.

Bei der Vektorisierung der Gauss-Elimination wird das unterschiedliche Verhalten von verschiednenen AVX-Instruktionen gezeigt. Besonders deutlich wird dies anhand der *GaussMaskload* Programmvariante deren Unterschiede sich zur *GaussStore* Programmvariante bei jeder Architektur unterscheiden. Bei der Sandy Brdige Architektur tritt bei der Nutzung der maskierten Ladeoperation eine höhere Ausführungszeit und ein höherer Energieverbrauch auf, der bei der Skylake Architektur nicht auftritt.

5

SIMD Gram-Schmidt Prozess zur QR-Faktorisierung einer Matrix

Algorithmen zur QR-Zerlegung werden im wissenschaftlichen Rechnen für die Zerlegung einer Matrix in eine Orthogonalmatrix Q und eine obere Dreiecksmatrix R verwendet. Die QR-Zerlegung wird in verschiedenen Verfahren der linearen Algebra genutzt, wie z.B. bei der Annäherung der Eigenwerte einer Matrix [23] oder als Bestandteil der Methode der kleinsten Quadrate [8]. Um solche Verfahren effizient zu lösen ist eine effiziente Berechnung der QR-Zerlegung eine wichtige Voraussetzung.

Beim Gram-Schmidt Prozess zur Vektororthogonalisierung werden die einzelnen Spalten der Matrizen Q und R durch iterative Methoden berechnet [24]. Im wissenschaftlichen Rechnen ist die Nutzung einer abgeänderten Variante der Gram-Schmidt Methode verbreitet: Der modifizierte Gram-Schmidt Prozess zur Vektororthogonalisierung (MGS). Die darin implementierte Modifikation bietet den Vorteil einer numerisch stabileren Lösung der QR-Zerlegung, da sich Rundungsfehler im Vergleich zur ursprünglichen Variante weniger stark fortpflanzen (vgl. [24]). In [70] werden verschiedene Versionen des modifizierten Gram-Schmidt Prozesses untersucht und [53] gibt einen Überblick über verschiedene Artikel zum Gram-Schmidt Prozess und dessen Modifikationen.

In diesem Kapitel wird untersucht, welchen Einfluss verschiedene Programmtransformationen in Verbindung mit Vektorisierung auf die Ausführungszeit und den Energieverbrauch des modifizierten Gram-Schmidt Prozesses haben. Teilaspekte der Untersuchungen dieses Kapitels wurden bereits in [44] veröffentlicht.

5.1. Modifizierter Gram-Schmidt Prozess zur Vektororthogonalisierung

Für die QR-Zerlegung wird eine Matrix A transformiert, sodass die Spalten einer Matrix Q eine Menge von orthogonalen Vektoren darstellt und eine obere Dreiecksmatrix R entsteht. Sind alle Spalten der Matrix A linear unabhängig voneinander [24], so ergibt sich die QR-Zerlegung:

$$A = Q \cdot R \quad \text{mit} \quad m \leq n \text{ und } A \in \mathbb{R}^{m \times n}, Q \in \mathbb{R}^{m \times m} \text{ und } R \in \mathbb{R}^{m \times n} \tag{5.1}$$

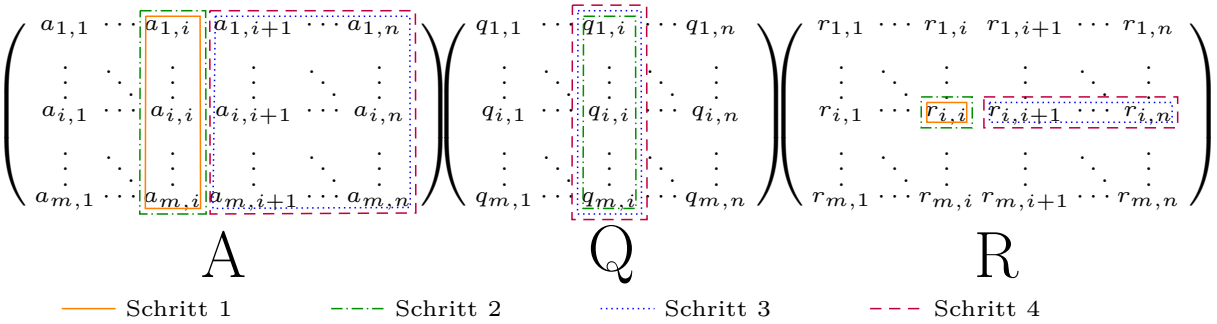

Abbildung 5.1.: Veranschaulichung des Datenzugriffs durch die vier Schritte des modifizierten Gram-Schmidt Prozesses (MGS) für die Berechnung von Spalte i der Matrix A. Hervorgehoben sind jeweils die Elemente der Matrizen A, Q und R die im entsprechenden Schritt gelesen oder geschrieben werden.

Die Berechnung der QR-Zerlegung durch den klassischen Gram-Schmidt Prozess führt bei der numerischen Berechnung durch Rundungsfehler zu Ergebnissen, bei denen die Spaltenvektoren der Matrix Q nicht orthogonal zueinander sind [8]. Der modifizierte Gram-Schmidt Prozess erzeugt eine durch geringere Rundungsfehler wesentlich bessere Orthogonalisierung [24] der Matrix Q.

5.1.1. Algorithmus des modifizierten Gram-Schmidt Prozesses

Die Berechnung der QR-Zerlegung mit dem modifizierten Gram-Schmidt Prozess (MGS) wird entlang der Spalten $a_{\_,i}$ der Ursprungsmatrix A durchgeführt. Für jede Spalte i mit $1 \leq i \leq n$ der Matrix A werden jeweils die folgenden vier Schritte durchgeführt [30]:

Schritt 1:

$$r_{i,i} = \|a_{\_,i}\|_2 \tag{5.2}$$

Die Berechnung in Gleichung 5.2 bestimmt die Vektornorm des Spaltenvektors $a_{\_,i}$ und legt diese an der Stelle $r_{i,i}$ in Matrix R ab. Die Vektornorm wird nach der folgenden Formel berechnet:

$$\|a_{\_,i}\|_2 = (a_{1,i}^2 + a_{2,i}^2 + \cdots + a_{m,i}^2)^{\frac{1}{2}} \tag{5.3}$$

Schritt 2:

$$q_{\_,i} = a_{\_,i}/r_{i,i} \tag{5.4}$$

In Gleichung 5.4 wird die Spalte $a_{\_,i}$ der Matrix A mit $r_{i,i}$ normiert und in der Matrix Q als i-te Spalte $q_{\_,i}$ abgespeichert.

Schritt 3:

$$[r_{i,i+1}, \ldots, r_{i,n}] = q^T_{\_,i} \cdot [a_{\_,i+1}, ..., a_{\_,n}] \tag{5.5}$$

Gleichung 5.5 berechnet die Elemente der i-ten Zeile der Matrix R. Die jeweiligen Elemente $r_{i,j}$, mit $i < j \leq n$, der Matrix R werden dabei durch das folgende Skalarprodukt aus den Spaltenvektoren $q_{\_,i}$ und $a_{\_,i}$ berechnet:

$$\begin{aligned} r_{i,j} &= q^T_{\_,i} \cdot a_{\_,j} \text{ mit } i+1 \leq j \leq n \\ &= \sum_{k=1}^{m} q_{k,i} \cdot a_{k,j} \end{aligned} \tag{5.6}$$

Schritt 4:

$$[a_{\_,i+1}, ..., a_{\_,n}] = [a_{\_,i+1}, ..., a_{\_,n}] - q_{\_,i} \cdot [r_{i,i+1}, \ldots, r_{i,n}] \tag{5.7}$$

In Gleichung 5.7 werden die Werte der Matrix A angepasst. Dazu wird von jedem Element $a_{k,j}$ der Matrix A das Produkt der Elemente $q_{k,i}$ und $r_{i,j}$ wie folgt abgezogen:

$$a_{k,j} = a_{k,j} - q_{k,i} \cdot r_{i,j} \quad \text{mit} \quad \begin{aligned} &i+1 \leq j \leq n \text{ und} \\ &1 \leq k \leq m \end{aligned} \tag{5.8}$$

Abbildung 5.1 zeigt die gelesenen und geschriebenen Matrixelemente der Matrizen A, Q und R durch die vier Schritte der Berechnung i-ten Spalte $a_{\_,i}$ der Matrix A. Die Spalte $a_{\_,i}$ wird jeweils in den Schritten 1 und 2 gelesen. Das Element $r_{i,i}$ wird in Schritt 1 geschrieben und in Schritt 2 gelesen. Die Teilmatrix $[a_{\_,i+1}, ..., a_{\_,n}]$ rechts der Spalte $a_{\_,i}$ der Matrix A wird in den Schritten 3 und 4 gelesen und in Schritt 4 neu berechnet. Die Teilzeile $[r_{i,i+1}, \ldots, r_{i,n}]$ rechts des Diagonalelements $_{i,i}$ der Matrix R wird in Schritt 3 berechnet und in Schritt 4 für die Berechnungen genutzt. Die i-te Spalte $q_{\_,i}$ der Matrix Q wird in Schritt 2 berechnet und in den Schritten 3 und 4 gelesen. Die Reihenfolge der Lese- und Schreiboperationen erzeugt eine Reihe von Datenabhängigkeiten, die eine sequentielle Berechnung der einzelenen Schritte erfordern.

5.1.2. Implementierung des modifizierten Gram-Schmidt Prozesses

Für die Implementierung des MGS werden die einzelnen Schritte innerhalb einer Schleife ausgeführt, die über die Spalten $a_{\_,i}$ der Matrix A iteriert. Die einzelnen Schritte werden nacheinander innerhalb dieser Schleife ausgeführt.

Speicherformat der Matrizen

Für die Speicherung der Matrizen werden eindimensionale Arrays genutzt. Der Indexzugriff auf die Elemente von zweidimensionalen Matrizen wird dabei auf einen eindimensionalen Index abgebildet. Dazu werden die Elemente von zweidimensionalen

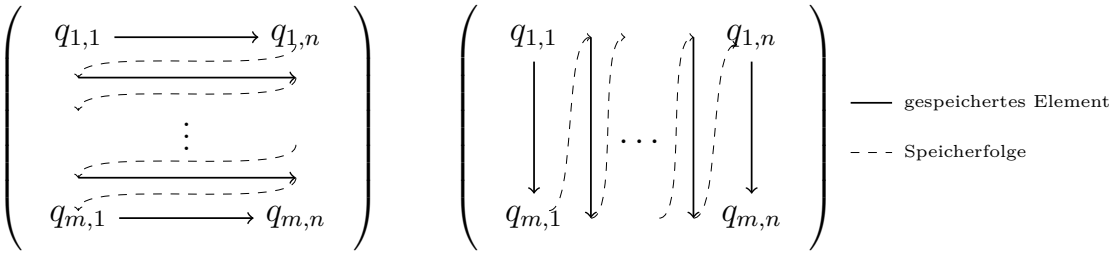

Abbildung 5.2.: Speicherformat und Indexzugriff von zweidimensionalen Matrizen in eindimensionalen Arrays. Die Werte der Matrizen können zeilenweise (links) oder spaltenweise (rechts) im Speicher abgelegt werden. Die Indexzugriffe müssen entsprechend des Speicherformates angepasst werden und berechnen sich aus Zeilen- und Spaltenindex, sowie Zeilen- bzw. Spaltenlänge.

Arrays zeilenweise (*engl. row-major*) im Speicher abgelegt. Die Reihenfolge der Elemente ist in Abb. 5.2 (links) dargestellt. Dabei werden Indexzugriffe auf die Elemente $q_{i,k}$ der Matrix Q mit Hilfe der Zeilenlänge n durch die folgende Formel berechnet:

$$q_{i,k} = Q[i * n + k] \tag{5.9}$$

In einigen Programmvarianten dieses Kapitels werden eine oder mehrere der Matrizen in einer spaltenweisen (*engl. column-major*) Ordnung abgespeichert. Hierbei werden die Elemente der Matrizen entlang der Spalten (vgl. Abb. 5.2 (rechts)) im Speicher abgelegt. Die Formel zur Berechnung der Indexzugriffe auf die Elemente $q_{i,k}$ der Matrix Q aus Gleichung 5.9 muss demnach mit Hilfe der Spaltenlänge m wie folgt angepasst werden:

$$q_{i,k} = Q[k * m + i] \tag{5.10}$$

Implementierung des Algorithmus

Algorithmus 5.1 zeigt die Implementierung der **MgsScal** Programmvariante zur seriellen Berechnung des MGS. Die *i-Schleife* (Zeile 2) iteriert dabei über die Spalten $a_{\_,i}$ der Matrix A und führt jeweils die Schritte 1 bis 4 aus.

Schritt 1: Innerhalb der *k-Schleife* (Zeile 6) wird die Summe der Quadrate $a^2_{k,i}$ berechnet (Zeile 7). Um die Vektornorm der Spalte $a_{\_,i}$ zu erhalten, wird aus dieser Summe in Zeile 8 die Wurzel gezogen und an die Stelle des Elements $r_{i,i}$ der Matrix R geschrieben.

Schritt 2: Die Elemente $q_{k,i}$ der Spalte $q_{\_,i}$ der Matrix Q werden berechnet, indem jedes Element $a_{k,i}$ der Spalte $a_{\_,i}$ durch die Vektornorm von $a_{\_,i}$ (gespeichert in $r_{i,i}$) dividiert wird.

Schritt 3: Im dritten Schritt werden die Elemente $[r_{i,i+1}, \ldots, r_{i,n}]$ rechts des Diagonalelements $r_{i,i}$ berechnet. Die *j-Schleife* in Zeile 17 iteriert dazu über die entsprechenden Elemente der Zeile $r_{i,\_}$. Für jedes Element $r_{i,j}$ iteriert die *k-Schleife* (Zeile 19)

```
1  // Schleife über die Spalten a_{_,i}:
2  for i = 0 < n
3
4  // Schritt 1:
5    // Schleife über die Elemente a_{_,i} der i-ten Spalte:
6    for k = 0 < m
7      norm += A[k*n+i] * A[k*n+i]; // sum_{k=1}^{n} a_{k,i}^2
8    R[i*n+i] = sqrt(norm);   // r_{i,i} = ||a_{_,i}||_2
9
10 // Schritt 2:
11   // Schleife über die Elemente a_{_,i} und q_{_,i} der i-ten Spalte:
12   for k = 0 < m
13     Q[k*n+i] = A[k*n+i] / R[i*n+i]; // q_{_,i} = a_{_,i} / r_{i,i}
14
15 // Schritt 3:
16   // Schleife über die Spalten a_{_,i+1} ... a_{_,n} und Elemente r_{i,i+1} ... r_{i,n}:
17   for j = i+1 < n
18     // Schleife über die Elemente a_{_,j} der j-ten Spalte und q_{_,i} der i-ten Spalte:
19     for k = 0 < m
20       R[i*n+j] += Q[k*n+i] * A[k*n+j]; // r_{i,j} = q_{_,i}^T * a_{_,j}
21
22 // Schritt 4:
23   // Schleife über die Spalten a_{_,i+1} ... a_{_,n} und Elemente r_{i,i+1} ... r_{i,n}:
24   for j = i+1 < n
25     // Schleife über die Elemente a_{_,j} der j-ten Spalte und q_{_,i} der i-ten Spalte:
26     for k = 0 < m
27       A[k*n+j] -= Q[k*n+i] * R[i*n+j]; // a_{_,j} = a_{_,j} - q_{_,i} * r_{i,j}
```

Algorithmus 5.1: MgsScal: Algorithmus des modifizierten Gram-Schmidt Prozesses zur Vektororthogonalisierung in C-Pseudocode. Die Matrix A wird in zwei Matrizen Q und R zerlegt, wobei die Spalten der Matrix Q orthogonale Vektoren darstellen und die Matrix R eine obere Dreiecksmatrix ist. Algorithmus nach [24], Notation angepasst.

über die Elemente $q_{k,i}$ und $a_{k,j}$ um in Zeile 20 das Skalarprodukt aus $q^T_{\_,i}$ und $a_{\_,i}$ nach Gleichung 5.6 zu berechnen.

Schritt 4: In Zeile 27 wird von jedem Element $a_{k,j}$ das Produkt aus $q_{k,i}$ und $r_{i,j}$ subtrahiert. Durch die Verschachtelung der beiden Schleifen (Zeilen 24 und 26) werden alle Elemente $a_{k,j}$ rechts der Spalte $a_{\_,i}$ der Matrix A nach Gleichung 5.8 neu berechnet.

5.1.3. Eigenschaften des modifizierten Gram-Schmidt Prozesses im Hinblick auf eine Vektorisierung

Beim modifizierten Gram-Schmidt Prozess (MGS) hat die Berechnung der einzelnen Spalten $a_{\_,i+1}$ der Matrix A jeweils eine Abhängigkeit zur vorherigen Spalte $a_{\_,i}$. Ebenso haben die einzelnen Schritte Abhängigkeiten zum jeweils vorherigen Schritt.

Eine Vektorisierung, oder Parallelisierung, kann demnach jeweils nur für jeden Schritt einzeln erfolgen.

Die einzelnen Schritte unterscheiden sich in der Anzahl der zugegriffenen Matrixelemente und deren Speicherstruktur (vgl. Abb. 5.1) sowie der Anzahl der ausgeführten Schleifen (vgl. Alg. 5.1) und den pro Schritt durchgeführten Operationen (vgl. Gleichung 5.2 bis 5.7). Dies ermöglicht eine Reihe von Transformationen und Implementierungsvarianten, die die verschiedenen Berechnungen der Schritte ersetzen oder modifizieren können. Im folgenden sind einige der Eigenschaften des modifizierten Gram-Schmidt Prozesses im Hinblick auf die Implementierungsvarianten aufgeführt:

- Die Schleifennester in den Schritten 3 und 4 ermöglichen die Untersuchung von Schleifentransformationen, wie *loop tiling* oder *loop interchange*. Die Anwendung dieser Schleifentransformationen kann sowohl auf serielle als auch auf vektorisierte Programme erfolgen.

- Der spaltenweise Zugriff auf die Matrixelemente erfordert bei der zeilenweisen Speicherung der Matrizen eine gesonderte Behandlung der Ladeoperationen bei der Vektorisierung. Da die einzelnen Matrixelemente dabei nicht aufeinander folgend im Speicher liegen, müssen diese zum Laden in einem Vektorregister gesammelt werden. Für dieses Sammeln der Matrixelemente bestehen bei der Vektorisierung verschiedene Möglichkeiten.

- Die Datenzugriffe auf die Matrizen A und Q werden im MGS durch eine spaltenweise Abarbeitung der Matrixelemente realisiert. Bei einer spaltenweisen Abspeicherung der Matrizen erfolgt der Datenzugriff dadurch auf Matrixelemente, die im Speicher aufeinanderfolgend abgelegt sind. Durch diese Transformation wird bei der Vektorisierung eine veränderte Behandlung der Lade- und Speicheroperationen möglich.

- Der MGS berechnet in den Schritten 1 und 3 jeweils eine Summe aus Matrixelementen. Die Summenberechnung lässt dich durch die Nutzung von Teilsummen parallelisieren. Diese Teilsummen werden dann mit Hilfe einer Reduktionsoperation (vgl. Akkumulation in [63]) zu einem Skalar kombiniert. Da bei der Vektorisierung die Teilsummen im selben Vektorregister vorliegen, kann die Akkumulation auf verschiedene Weisen durchgeführt werden.

5.1.4. Programmvarianten durch Transformationen des modifizierten Gram-Schmidt Prozesses

Programmtransformationen zur Modifikation der Datenzugriffsreihenfolge können auf die serielle Programmvariante *MgsScal* aus Alg. 5.1 angewendet werden. Tabelle 5.1 zeigt eine Liste der verschiedenen Programmvarianten für den seriellen Algorithmus, sowie eine Übersicht der entsprechenden Veränderungen in diesen Programmvarianten.

Ansatz	Programmvariante	Programmtransformation	Referenz
—	**MgsScal**	keine	S. 87; Alg. 5.1
Datenzugriffsreihenfolge	**MgsScalLi**	Schleifentausch (*loop interchange*) in den Schritten 3 und 4	S. 91
	ColMgsScal	Spaltenweise Speicherung der Matrizen A und Q	S. 91
	ColMgsScalLi	Spaltenweise Speicherung der Matrizen A und Q, sowie Schleifentausch (*loop interchange*) in den Schritten 3 und 4	S. 92
	MgsScalTile	Schleifen-Blockzerlegung (*loop tiling*) in den Schritten 3 und 4	S. 92; Alg. 5.2
	ColMgsScalTile	Spaltenweise Speicherung der Matrizen A und Q, sowie Schleifen-Blockzerlegung (*loop tiling*) in den Schritten 3 und 4	S. 93

Tabelle 5.1.: Übersicht über die Programmvarianten des modifizierten Gram-Schmidt Prozesses basierend auf der seriellen Implementierung in Alg. 5.1. Die angewendeten Programmtransformationen werden kurz zusammengefasst, die detaillierte Beschreibung der Programmvariante ist referenziert.

5.1.4.1. Die *MgsScalLi* Programmvariante mit Schleifentausch

Bei der **MgsScalLi** Programmvariante werden die Schleifennester in den Schritten 3 und 4 transformiert. Hierzu werden die *j-Schleife* und die *k-Schleife* in den Zeilen 17 und 19 in Schritt 3 vertauscht. Auf gleiche Weise werden die *j-Schleife* und die *k-Schleife* in den Zeilen 24 und 26 vertauscht. Der Schleifentausch (*engl. loop interchange*) ist hierbei möglich, da die Abhängigkeiten der Iterationen beim Schleifentausch erhalten bleiben [3]. Durch den Schleifentausch wird in beiden Schritten die Reihenfolge der Elementzugriffe auf die Matrix A geändert: Über die Matrixelemente wird nun zeilenweise iteriert. Aufgrund der zeilenweisen Speicherung (vgl. Abb. 5.2(links)) der Matrizen erfolgt der zeilenweise Elementzugriff mit einer höheren räumlichen Lokalität (vgl. [63]), wodurch eine bessere Cache-Ausnutzung zu erwarten ist.

5.1.4.2. Spaltenweise Matrixspeicherung in der *ColMgsScal* Programmvariante

Mit der **ColMgsScal** Programmvariante soll ebenfalls die räumliche Lokalität der Speicherzugriffe erhöht werden. Hierzu werden die Matrizen A und Q spaltenweise im Speicher abgelegt (vgl. Abb. 5.2 (rechts)). Die spaltenweise Speicherung der Matrizen, unter Beibehaltung der Schleifen aus Alg. 5.1, führt dabei zu einem Zugriff auf aufeinanderfolgende Speicherelemente für die beiden Matrizen.

Zur Umsetzung der Programmvarianten werden in Alg. 5.1 die jeweiligen Index-

```
1  // Schritt 3:
2    // Schleife über die Blöcke (tiles ) für die Spalten von Matrix A:
3    for j = i+1 < n; j += tile_size
4      // Schleife über die Blöcke der Elemente von A:
5      for k = 0 < m; k += tile_size
6        // Schleife über die Spalten innerhalb des Blockes:
7        for jt = j < j + tile_size && jt < n
8          // Schleife über die Elemente innerhalb des Blockes:
9          for kt = k < k + tile_size && kt < m
10           R[i*n+jt] += Q[kt*n+i] * A[kt*n+jt]; // r_{i,j} = q^T_{_,i} · a_{_,j}
```

Algorithmus 5.2: MgsScalTile: Schleifen-Blockzerlegung der Schleifennester am Beispiel von Schritt 3. Die Schrittweite der Schleifen wird um die Größe tile_size erweitert. Die übersprungenen Elemente werden in zusätzlichen Schleifen innerhalb des Schleifennestes berechnet. In den inneren Schleifen wird zusätzlich die Schleifenobergrenze abgefangen um unvollständige Blöcke am Ende des Iterationsraums korrekt zu beenden. Veränderte Zeilen im Vergleich zur *MgsScal* Programmvariante aus Alg. 5.1 sind hervorgehoben. Die Änderungen von Schritt 4 erfolgen analog und finden sich im Anhang in Alg. D.1.

zugriffe auf die Matrizen A und Q verändert. Die Indexzugriffe werden dazu wie in Gleichung 5.10 (bzw. Abb. 5.2 (rechts)) angepasst. Zusätzlich wird die Matrix A vor der Ausführung in das spaltenweise Speicherungsformat überführt.

5.1.4.3. Die *ColMgsScalLi* Programmvariante als Kombination von Schleifentausch und spaltenweiser Speicherung

Die **ColMgsScalLi** Programmvariante ist eine Kombination der beiden zuvor gezeigten Programmvarianten und damit eine Kombination von Schleifentausch und spaltenweiser Speicherung. Hierbei werden die beiden Matrizen A und Q spaltenweise im Speicher abgelegt, wodurch die Indexberechnung wie in der *ColMgsScal* Programmvariante angepasst wird. Zusätzlich werden die beiden Schleifen in den Schritten 3 und 4 wie in der *MgsScalLi* Programmvariante vertauscht.

Die zuvor beschriebenen erwarteten Auswirkungen der einzelnen Programmtransformationen auf die Programmausführung sollten sich dabei gegenseitig aufheben. Das erwartete Ausführungsverhalten der **ColMgsScalLi** Programmvariante sollte damit ähnlich zur *MgsScal* Programmvariante sein. Für eine Vektorisierung dieser Programmvariante kann diese Annahme jedoch nicht getroffen werden.

5.1.4.4. Schleifen-Blockzerlegung in der *MgsScalTile* Programmvariante

In der **MgsScalTile** Programmvariante werden die Schleifennester der Schritte 3 und 4 aus Alg. 5.1 mittels Schleifen-Blockzerlegung (*engl. loop tiling*[42] oder *loop*

blocking[3]) transformiert. Bei der Schleifen-Blockzerlegung wird der Iterationsraum in Blöcke oder Kacheln (*engl. tiles*) aufgeteilt, um die Lokalität der Datenzugriffe zu erhöhen. Dazu werden die Iterationsvariablen der Schleifen des Schleifennestes jeweils um eine Blockgröße erhöht. Die dadurch übersprungenen Iterationen werden innerhalb des Schleifennestes durch zusätzliche Schleifen ausgeführt. Betrachtet man dabei die Speicherzugriffe auf die genutzten Elemente des Blockes, liegen diese näher beieinander, was zu einer besseren Cache-Ausnutzung (räumliche Lokalität) führen kann. Zusätzlich erlaubt die geringere Iterationszahl der inneren Schleifen, dass beim nochmaligen Zugriff Variablen noch im Cache gespeichert sind (zeitliche Lokalität [63]).

In Alg. 5.2 wird die Anwendung der Schleifen-Blockzerlegung auf Schritt 3 des MGS dargestellt. Die Schrittweite der *j-Schleife* (Zeile 3) und der *k-Schleife* (Zeile 5) wird dazu jeweils um den Faktor tile_size erweitert. In Zeile 7 wird eine zusätzliche Schleife (*jt-Schleife*) eingefügt. Die *jt-Schleife* iteriert über die vormaligen Iterationen der *j-Schleife*, die durch die erhöhte Schrittweite übersprungen werden: $j, \ldots, j+$ tile_size -1. Analog dazu wird in Zeile 9 eine *kt-Schleife* hinzugefügt. Bei der Berechnung der Elemente in Zeile 10 werden die Iterationsvariablen j und k durch die Iterationsvariablen jt und kt der neuen Schleifen ersetzt. Die Anzahl der Iterationen pro Schleife ist nicht in jeder Iteration der *i-Schleife* durch die Blockgröße tile_size teilbar. Dazu wird bei der *jt-Schleife* (und *kt-Schleife*) die Abbruchbedingung der *j-Schleife* (und *k-Schleife*) hinzugefügt. Diese Bedingung führt zu einem korrekten Abbruch der Schleife im letzten Block der *j-Schleife* (bzw. *k-Schleife*). Die Implementierung für Schritt 4 wird auf die gleiche Weise verändert und ist im Anhang in Alg. D.1 gezeigt.

5.1.4.5. Die *ColMgsScalTile* Programmvariante mit Schleifen-Blockzerlegung und spaltenweiser Speicherung

In der **ColMgsScalTile** Programmvariante werden die Transformationen der *ColMgsScal* Programmvariante und der *MgsScalTile* Programmvariante kombiniert. Dazu werden die Matrizen A und Q spaltenweise gespeichert und die Indexzugriffe entsprechend angepasst (vgl. *ColMgsScal*). Zusätzlich wird die Schleifen-Blockzerlegung aus der *MgsScalTile* Programmvariante auf die Schleifennester der Schritte 3 und 4 angewendet. Mit dieser Programmvariante wird der Einfluss der Schleifen-Blockzerlegung bei spaltenweiser Speicherung der Matrix untersucht.

5.2. AVX-Programmvarianten des modifizerten Gram-Schmidt Prozesses

Die Vektorisierung des modifizierten Gram-Schmidt Prozesses (MGS) erfolgt mit Hilfe von AVX-Intrinsics auf Basis der im vorherigen Abschnitt vorgestellten seriellen Programmvarianten. Zur besseren Lesbarkeit der Algorithmen wird eine Behandlung der verbleibenden Iterationen einer Schleife (vgl. Loop Splitting in Abschn. A.3) nicht dargestellt.

Für die Untersuchungen in diesem Kapitel werden zusätzliche Programmvarianten für die Ausführung mit AVX512 erstellt. Die Umwandlung der Programme aus den gezeigten 256-Bit Intrinsics lässt sich in den meisten Fällen durch den Austausch der Registergröße im Funktionsnamen des Intrinsics von 256 auf 512 realisieren. Eine Ausnahme stellt hier beispielsweise die broadcast Instruktion dar, die für AVX512 Intrinsics als set1 bezeichnet ist (vgl. Anhang Abschn. A.5). Zusätzlich muss der Schleifenzähler entsprechend um den Wert 16 (statt 8) erhöht werden, da in den 512-Bit Registern des AVX512 Befehlssatzes insgesamt 16 `float` Werte enthalten sind.

Tabelle 5.2 zeigt eine Übersicht über die vektorisierten Programmvarianten sowie eine kurze Beschreibung der Modifikationen dieser Programmvarianten.

Ansatz	Programmvariante	Programmtransformation	Referenz
—	**MgsAvx**	keine	S. 95; Alg. 5.3; Alg. 5.4
Datenzugriffsreihenfolge	**MgsAvxLi**	Schleifentausch (*loop interchange*) in den Schritten 3 und 4	S. 97; Alg. 5.5
	ColMgsAvx	Spaltenweise Speicherung der Matrizen A und Q	S. 98
	ColMgsAvxLi	Spaltenweise Speicherung der Matrizen A und Q, sowie Schleifentausch (*loop interchange*) in den Schritten 3 und 4	S. 99
	MgsAvxTile	Schleifen-Blockzerlegung (*loop tiling*) in den Schritten 3 und 4	S. 99
	ColMgsAvxTile	Spaltenweise Speicherung der Matrizen A und Q, sowie Schleifen-Blockzerlegung (*loop tiling*) in den Schritten 3 und 4	S. 99
Speicherzugriff	**MgsAvxAload**	Laden der verteilt im Speicher liegenden Elemente durch Zwischenspeicherung in einem temporären Array	S. 99; Alg. 5.6
	MgsAvxAstore	Speichern der verteilt im Speicher liegenden Elemente durch Zwischenspeicherung in einem temporären Array	S. 99
	MgsAvxGatherIdx	Laden der verteilt gespeicherten Matrixelemente durch die gather Instruktion und einen Index-Vektor	S. 100; Alg. 5.7
	MgsAvx-GatherBase	Laden der verteilt gespeicherten Matrixelemente durch die gather Instruktion und einen Offset-Vektor	S. 100; Alg. 5.7

Tabelle 5.2: Übersicht über die vektorisierten Programmvarianten des MGS. Die angewendeten Programmtransformationen werden kurz zusammengefasst, die detaillierte Beschreibung der Programmvariante ist referenziert. **Fortsetzung der Tabelle auf der nächsten Seite.**

Ansatz	Programmvariante	Programmtransformation	Referenz
	ColMgsAvxPeel	Loop Splitting zum Laden an ausgerichteten Speicherstellen	S. 101
	ColMgsAvxAlign	Loop Splitting zum Laden an ausgerichteten Speicherstellen und Austausch der Ladeinstruktion für ausgerichteten Speicher	S. 101
Veränderte Reduktion	**ColMgsAvx-Vred[1,2,4,8,16,i]**	Teilweise serielle Berechnung der Reduktionsoperation	S. 102
	MgsAvxMred	Verschieben der Reduktion in Schritt 3 aus der inneren Schleife in der *MgsAvx* Programmvariante	S. 102; Alg. 5.8
	ColMgsAvxMred	Verschieben der Reduktion in Schritt 3 aus der inneren Schleife in der *ColMgsAvx* Programmvariante	S. 102
	ColMgsAvx-MredAlign	Verschieben der Reduktion in Schritt 3 aus der inneren Schleife in der *ColMgsAvxAlign* Programmvariante	S. 102

Tabelle 5.2.: Übersicht über die vektorisierten Programmvarianten des MGS. Die angewendeten Programmtransformationen werden kurz zusammengefasst, die detaillierte Beschreibung der Programmvariante ist referenziert.

5.2.1. *MgsAvx*: Vektorisierung des modifizierten Gram-Schmidt Prozesses

Für die Erstellung der **MgsAvx** Programmvariante wird der in Alg. 5.1 gezeigte Algorithmus ohne weitere Programmtransformation mittels intrinsischer Funktionen vektorisiert. Die Datenabhängigkeiten der einzelnen Schritte zueinander erlauben keine Kombination der einzelnen Schritte, sodass jeder Schritt des Algorithmus einzeln vektorisiert wird.

Algorithmus 5.3 zeigt die Schritte 1 und 2 der **MgsAvx** Programmvariante. Für die Berechnung von Schritt 1 wird die Schleife in Blöcke zerlegt, sodass jeder Block genau acht Iterationen enthält. Die Berechnungen dieser Blöcke werden durch intrinsische Funktionen umgesetzt. Dazu wird ein temporärer Vektor mit Null initialisiert (Zeile 6), auf den im folgenden die Quadrate der Werte des Spaltenvektors $a_{\_,i}$ (geladen in Zeile 10) addiert (Zeile 12) werden. Die Teilsummen des Vektorregisters werden in Zeile 13 zur Gesamtsumme aufaddiert (vgl. Abschn. A.4 für vektorisierte Reduktion) und deren Quadratwurzel im Element $r_{i,i}$ abgelegt (Zeile 14).

Für die Vektorisierung von Schritt 2 wird in Zeile 17 der Wert $r_{i,i}$ an alle Stellen eines Vektors kopiert. Durch diesen Wert $r_{i,i}$ werden die Elemente des Spaltenvektors (geladen in Zeile 21) in Zeile 22 geteilt. Das Speichern der Elemente (Zeile 25) erfolgt

```
1  // Schleife über die Spalten a_{_,i}:
2  for i = 0 < n
3
4  // Schritt 1:
5    // Initialisiere v_norm mit Null-Elementen:
6    __m256 v_norm = _mm256_setzero_ps();
7    // Schleife über die Elemente a_{_,i} der i-ten Spalte:
8    for k = 0 < m; k += 8
9      // Laden der Elemente a_{k,i}, a_{k+1,i}, ..., a_{k+7,i}:
10     __m256 v_a = _mm256_setr_ps(&A[k*n+i], ..., &A[(k+7)*n+i]);
11     // Teilsummen der Werte von a^2_{_,i}:
12     v_norm = _mm256_fmadd_ps(v_a, v_a, v_norm);
13   norm = addreduce(v_norm); // Reduktion der Teilsummen zu sum_{k=1}^{n} a^2_{k,i}
14   R[i*n+i] = sqrt(norm); // r_{i,i} = ||a_{_,i}||_2
15
16 // Schritt 2:
17   __m256 v_Rii = _mm256_broadcast_ss(&R[i*n+i]); // Laden von r_{i,i}
18   // Schleife über die Elemente a_{_,i} und q_{_,i} der i-ten Spalte:
19   for k = 0 < m; k += 8
20     // Laden der Elemente a_{k,i}, a_{k+1,i}, ..., a_{k+7,i}:
21     __m256 v_a = _mm256_setr_ps(&A[k*n+i], ..., &A[(k+7)*n+i]);
22     __m256 v_Qki = _mm256_div_ps(v_a, v_Rii); // q_{_,i} = a_{_,i} / r_{i,i}
23     for pos = 0 < 8; pos++
24       // Speichern der Elemente q_{k,i}, q_{k+1,i}, ..., q_{k+7,i}:
25       Q[(k+pos)*n+i] = ((float*)&v_Qki)[pos];
```

Algorithmus 5.3: MgsAvx Programmvariante: Schritte 1 und 2 der AVX-Implementierung des MGS. Die Schleifen der einzelnen Schritte werden vektorisiert und dazu in Blöcke der Größe 8 zerlegt. Bei der Vektorisierung werden diese Blöcke durch die entsprechenden Intrinsics ersetzt. Die Schritte 3 und 4 sind in Alg. 5.4 gezeigt.

dabei durch eine Schleife (Zeile 23), da die Matrixelemente nicht nacheinander im Speicher abgelegt sind.

Algorithmus 5.4 zeigt die Vektorisierung der Schritte 3 und 4 der **MgsAvx** Programmvariante. In beiden Schritten wird die innere Schleife des Schleifennestes vektorisiert. In Schritt 3 werden die Elemente der Spalte $q_{\_,i}$ in Zeile 32 und die Elemente der Spalte $a_{\_,i}$ in Zeile 34 geladen. In Zeile 36 werden die jeweiligen Elemente multipliziert und in Zeile 37 mit einer Reduktion auf den skalaren Wert $r_{i,j}$ addiert.

In Schritt 4 wird innerhalb der äußeren Schleife (Zeile 41) der jeweilige Wert $r_{i,j}$ an alle Stellen eines Vektors kopiert (Zeile 42). Innerhalb der *k-Schleife* (Zeile 44) werden dann die Elemente der Spalten $q_{\_,i}$ (Zeile 46) und $a_{\_,j}$ (Zeile 48) geladen. Die Berechnung in Zeile 49 multipliziert die Elemente der Spalte $q_{\_,i}$ jeweils mit $r_{i,j}$ und zieht das Zwischenergebnis von den Elemente der Spalte $a_{\_,i}$ ab. In Zeile 52 werden die so berechneten Werte der Elemente der Spalte $a_{\_,i}$ abgespeichert.

```
26 // Schritt 3:
27   // Schleife über die Spalten a_,i+1 ... a_,n und Elemente r_i,i+1 ... r_i,n:
28   for j = i+1 < n; j++
29     // Schleife über die Elemente a_,j der j-ten Spalte und q_,i der i-ten Spalte:
30     for k = 0 < m; k += 8
31       // Laden der Elemente q_k,i, q_k+1,i, ..., q_k+7,i:
32       __m256 v_Qki = _mm256_setr_ps(&Q[k*n+i], ..., &Q[(k+7)*n+i]);
33       // Laden der Elemente a_k,j, a_k+1,j, ..., a_k+7,j:
34       __m256 v_a = _mm256_setr_ps(&A[k*n+j], ..., &A[(k+7)*n+j]);
35       // Teilergebnisse des Skalarproduktes von q_i^T · a_k,j:
36       __m256 v_Rij = _mm256_mul_ps(v_Qki, v_a);
37       R[i*n+j] += addreduce(v_Rij); // r_i,j = q_,i^T · a_k,j
38
39 // Schritt 4:
40   // Schleife über die Spalten a_,i+1 ... a_,n und Elemente r_i,i+1 ... r_i,n:
41   for j = i+1 < n; j++
42     __m256 v_Rij = _mm256_broadcast_ss(&R[i*n+j]); // Laden von r_i,j
43     // Schleife über die Elemente a_,j der j-ten Spalte und q_,i der i-ten Spalte:
44     for k = 0 < m; k += 8
45       // Laden der Elemente q_k,i, q_k+1,i, ..., q_k+7,i:
46       __m256 v_Qki = _mm256_setr_ps(&Q[k*n+i], ..., &Q[(k+7)*n+i]);
47       // Laden der Elemente a_k,j, a_k+1,j, ..., a_k+7,j:
48       __m256 v_a = _mm256_setr_ps(&A[k*n+j], ..., &A[(k+7)*n+j]);
49       v_a = _mm256_fnmadd_ps(v_Qki, v_Rij, v_a); // a_,j - q_,i · r_i,j
50       for pos = 0 < 8; pos++
51         // Speichern der Elemente a_k,j, a_k+1,j, ..., a_k+7,j:
52         A[(k+pos)*n+j] = ((float*)&v_a)[pos];
```

Algorithmus 5.4: MgsAvx Programmvariante: Schritte 3 und 4 der AVX-Implementierung des MGS. Die jeweils inneren Schleifen der Schritte 3 und 4 werden dazu in Blöcke der Größe 8 zerlegt. Bei der Vektorisierung werden diese Blöcke durch die entsprechenden Intrinsics ersetzt. Die Schritte 1 und 2 sind in Alg. 5.3 gezeigt.

5.2.2. Modifikationen der Datenzugriffsreihenfolge

Die Modifikationen der Datenzugriffsreihenfolge, die für die seriellen Programmvarianten implementiert sind, werden ebenso für die vektorisierten Programmvarianten untersucht. Dazu werden auf Basis der *MgsAvx* Programmvariante die Transformationen Schleifentausch, spaltenweise Speicherung und Schleifen-Blockzerlegung implementiert.

5.2.2.1. Die MgsAvxLi Programmvariante mit Schleifentausch

In der **MgsAvxLi** Programmvariante werden die beiden Schleifen der Schleifennester in den Schritten 3 und 4 miteinander getauscht (vgl. *MgsScalLi* Programmvariante). Nach dem Schleifentausch wird ebenfalls die innere Schleife des Schleifennestes vektorisiert, was zu einer veränderten Berechnungsreihenfolge und zu einem veränderten Zugriff auf die Matrixelemente führt.

```
1  // Schritt 3:
2    // Schleife über die Zeilen a_{1,_} ... a_{m,_} und Elemente der Spalte q_{_,i}:
3    for k = 0 < m; k++
4      __m256 v_Qki = _mm256_broadcast_ss(&Q[k,i]); // Laden von q_{k,i}
5      // Schleife über die Elemente a_{k,_} der k-ten Zeile und Elemente r_{i,_} i-ten Zeile:
6      for j = i+1 < n; j += 8
7        // Laden der Elemente a_{k,j}, a_{k,j+1}, ..., a_{k,j+7}:
8        __m256 v_a = _mm256_loadu_ps(&A[k,j]);
9        // Laden der Elemente r_{i,j}, r_{i,j+1}, ..., r_{i,j+7}:
10       __m256 v_Rij = _mm256_loadu_ps(&R[i,j]);
11       // Teilergebnisse des Skalarproduktes von q_i^T · a_{k,j}:
12       v_Rij = _mm256_fmadd_ps(v_Qki, v_a, v_Rij);
13       // Speichern der Elemente r_{i,j}, r_{i,j+1}, ..., r_{i,j+7}:
14       _mm256_storeu_ps(&R[i,j], v_Rij);
```

Algorithmus 5.5: MgsAvxLi Programmvariante: Schritt 3 der AVX-Implementierung des MGS mit vertauschten Schleifen. Durch den Schleifentausch wird bei der Vektorisierung auf andere Elemente der Matrizen zugegriffen und damit die Lade- und Speicheroperation, sowie die Berechnungsreihenfolge verändert. Veränderte Zeilen im Vergleich zur *MgsAvx* Programmvariante in Alg. 5.4 sind hervorgehoben. Schritt 4 wird analog verändert und ist im Anhang in Alg. D.2 gezeigt.

Algorithmus 5.5 zeigt die Veränderungen in Schritt 3 der **MgsAvxLi** Programmvariante. Das Element $q_{k,i}$ der Matrix Q wird an alle Stellen einer Vektorvariablen kopiert (Zeile 4). Innerhalb der *j-Schleife* (Zeile 6) werden durch die veränderte Berechnungsreihenfolge mehrere Elemente der Matrixzeile $a_{k,\_}$ der Matrix A (Zeile 8) und der Matrixzeile $r_{i,\_}$ der Matrix R (Zeile 10) geladen. Bei der Berechnung in Zeile 12 wird mit diesen Werten ein neues Zwischenergebnis für die Elemente der Matrixzeile $r_{i,\_}$ berechnet, dass in Zeile 14 in den Speicher geschrieben wird. Nach der Ausführung der *k-Schleife* enthält die Matrixzeile $r_{i,\_}$ die Ergebniswerte von Schritt 3. In Schritt 4 werden die Veränderungen analog zu Schritt 3 durchgeführt (siehe Anhang Alg. D.2).

5.2.2.2. *ColMgsAvx* mit spaltenweiser Matrixspeicherung

Bei der spaltenweisen Speicherung der Matrizen A und Q in der **ColMgsAvx** Programmvariante werden die Zugriffe auf die Elemente der Spalten der beiden Matrizen verändert. Die Elemente einer Matrixspalte liegen nun aufeinanderfolgend im Speicher und können somit durch loadu Instruktionen geladen werden. Dazu werden die setr Intrinsics der *MgsAvx* Programmvariante aus Alg. 5.3 und Alg. 5.4 jeweils durch eine loadu Instruktion ersetzt. Analog dazu wird die Speicherung der Elemente durch eine storeu Instruktion durchgeführt.

5.2.2.3. *ColMgsAvxLi*: Kombination von spaltenweiser Speicherung und Schleifentausch

Die Kombination von spaltenweiser Speicherung der Matrizen A und Q mit dem vertauschen der Schleifen in den Schritten 3 und 4 wird in der Programmvariante **ColMgsAvxLi** implementiert. Für die Schritte 1 und 2 wird die Implementierung der *ColMgsAvx* Programmvariante übernommen. Bei den Schritten 3 und 4 wird die *MgsAvxLi* Programmvariante aus Alg. 5.5 als Ausgangspunkt genutzt. Dabei werden die Ladeinstruktionen in entsprechende setr Intrinsics umgewandelt (vgl. *MgsAvx* Programmvariante). Analog dazu werden die storeu Instruktionen in die elementweisen Speicherinstruktionen umgewandelt.

5.2.2.4. Schleifen-Blockzerlegung in der *MgsAvxTile* und der *ColMgsAvxTile* Programmvariante

Für die Schleifen-Blockzerlegung in der **MgsAvxTile** Programmvariante wird die *MgsAvx* Programmvariante aus Alg. 5.4 als Ausgangspunkt genutzt. Die Schleifen-Blockzerlegung erfolgt hierbei wie in der *MgsScalTile* Programmvariante aus Alg. 5.2. Die Blockgrößen der neuen inneren Schleifen werden so gewählt, dass deren Größe ein vielfaches der Vektorgröße von AVX ist.

Auf die gleiche Weise kann die **ColMgsAvxTile** Programmvariante aus der Übernahme der Schleifen aus der *ColMgsScalTile* Programmvariante erfolgen. Die Vektorisierung wird dafür entsprechend aus der *ColMgsAvx* Programmvariante übernommen.

5.2.3. Modifikationen des Speicherzugriffs

Die Vektorisierung der Programmvarianten bietet noch weitere Implementierungsvarianten, die sich aus unterschiedlichen Datenzugriffsmustern und Instruktionen zusammen setzen. Diese weiteren Möglichkeiten der Modifikation beziehen sich dabei auf die Auswahl unterschiedlicher Lade- und Speicherinstruktionen, die AVX zur Verfügung stellt.

5.2.3.1. Zwischenspeicherung in temporärem Array in der *MgsAvxAload* und *MgsAvxAstore* Programmvariante

Das in den bisher gezeigten Programmvarianten genutzte setr Intrinsic entspricht keiner Assemblerinstruktion, sondern wird durch eine Sequenz an verschiedenen Instruktionen implementiert [41]. Das setr Intrinsic bekommt dabei die Werte der zu ladenden Elemente übergeben und schreibt diese in ein Vektorregister. Ein vergleichbares Verhalten kann in AVX dadurch erzeugt werden, dass die zu ladenden Werte in ein temporäres Array kopiert werden, aus dem diese anschließend mit einer load Instruktion geladen werden.

In der **MgsAvxAload** in Alg. 5.6 werden die setr Intrinsics der *MgsAvx* Programmvariante auf die beschriebene Weise ersetzt. In Zeile 7 wird dazu ein temporäres Arrays erstellt, das mit den zu ladenden Werten initialisiert wird. Die Werte des temporären

```
1 // Schritt 1:
2   // Initialisiere   v_norm mit Null-Elementen:
3   __m256 v_norm = _mm256_setzero_ps();
4   // Schleife über die Elemente a_,i der i-ten Spalte:
5   for k = 0 < m; k += 8
6     // Laden der Elemente a_{k,i}, a_{k+1,i}, ..., a_{k+7,i} in temporäres Array:
7     float t[8] = {A[k*n+i],...,A[(k+7)*n+i]};
8     // Laden der Elemente in Vektorregister:
9     __m256 v_a = _mm256_load_ps(t);
```

Algorithmus 5.6: **MgsAvxAload** Programmvariante: setr Intrinsics werden durch Kopieren der Werte in ein temporäres Array und anschließendes Laden ersetzt. Gezeigt ist ein Ausschnitt von Schritt 1. Die Ersetzung findet für alle setr Instruktionen analog statt. Veränderte Zeilen im Vergleich zur *MgsAvx* Programmvariante in Alg. 5.3 sind hervorgehoben.

Arrays werden im Anschluss (Zeile 9) in ein Vektorregister geladen. Algorithmus 5.6 zeigt die Modifikation am Beispiel der Ladeoperation in Schritt 1, die verbleibenden setr Intrinsics der *MgsAvx* Programmvariante werden für die **MgsAvxAload** Programmvariante auf die gleiche Weise ersetzt.

Analog zum Laden der Elemente mit Hilfe eines temporären Arrays kann auch das Speichern über ein temporäres Array erfolgen. Dazu werden in der **MgsAvxAstore** Programmvariante die Speicheroperationen der *MgsAvx* Programmvariante modifiziert. Hierbei wird das Vektorregister zuerst in ein temporäres Array gespeichert und im Anschluss die einzelnen Werte des Arrays an ihre korrekte Position kopiert.

5.2.3.2. *MgsAvxGatherIdx* und *MgsAvxGatherBase*: Nutzung der AVX2-Gather-Instruktion zum Laden der verteilt abgespeicherten Daten

Mit der AVX2-Erweiterung sind spezielle gather Instruktionen verfügbar geworden. Diese erlauben es Datenelemente, die im Speicher verteilt abgelegt sind, durch Angabe eines Indexvektors in ein Vektorregister zu laden.

Algorithmus 5.7 zeigt den Aufbau der gather Instruktion und den dafür nötigen Index-Vektor für die **MgsAvxGatherIdx** Programmvariante in den Zeilen 1 bis 6. In Zeile 4 wird ein Integer-Vektor mit den Indizes der Array-Elemente erstellt. Dieser Vektor wird zusammen mit der Startadresse des Arrays an die gather Instruktion übergeben (Zeile 6). Auf diese Weise werden alle angegebenen Elemente in die Vektorvariable v_a geladen.

Für die **MgsAvxGatherBase** Programmvariante (Alg. 5.7 Zeile 9 bis 14) wird dieses Vorgehen modifiziert. Dazu wird statt den absoluten Index-Werten nur der Abstand (Offset) der Elemente in einen Index-Vektor geschrieben (Zeile 12 in Alg. 5.7). Dieser Offset-Vektor ist im Anschluss für alle ausgeführten Ladeoperationen identisch

```
1  // MgsAvxGatherIdx:
2  // Laden der Elemente a_{k,i}, a_{k+1,i}, ..., a_{k+7,i}:
3     // Setzen der Array-Indizes in ein Integer-Vektorregister
4     __m256i idx = _mm256_setr_epi32(k*n+i,(k+1)*n+i,...,(k+7)*n+i);
5     // Laden der verteilt abgelegten Werte durch Array-Index
6     __m256 v_a = _mm256_i32gather_ps(&A[0], idx, sizeof(float));
7
8
9  // MgsAvxGatherBase:
10 // Laden der Elemente a_{k,i}, a_{k+1,i}, ..., a_{k+7,i}:
11    // Setzen der Offset-Indizes in ein Integer-Vektorregister
12    __m256i idx = _mm256_setr_epi32(0,n,...,7*n);
13    // Laden der Werte mit Offset-Vektor und Adresse des ersten Elements:
14    __m256 v_a = _mm256_i32gather_ps(&A[k*n+i], idx, sizeof(float));
```

Algorithmus 5.7: **MgsAvxGatherIdx** (oben) und **MgsAvxGatherBase** (unten) Programmvariante: Die setr Intrinsics der *MgsAvx* Programmvariante aus Alg. 5.4 werden durch gather Intrinsics ersetzt.
In der **MgsAvxGatherIdx** Programmvariante (oben) werden die absoluten Indizes ab Beginn des Arrays genutzt. In der **MgsAvxGatherBase** Programmvariante (unten) wird ein Offset-Vektor zum ersten zu ladenden Element genutzt. Gezeigt ist ein Ausschnitt von Schritt 1. Veränderte Zeilen im Vergleich zur *MgsAvx* Programmvariante in Alg. 5.3 sind hervorgehoben.

und muss somit nur einmalig initialisiert werden. Die Startadresse, die an die gather Instruktion übergeben wird, referenziert hierbei das erste zu ladende Element des Arrays (Zeile 14).

In der **MgsAvxGatherIdx** und der **MgsAvxGatherBase** Programmvariante werden alle setr Intrinsics der *MgsAvx* Programmvariante durch das jeweilige gather Intrinsic und den dazugehörigen Index-Vektor ausgetauscht.

5.2.3.3. Ausgerichteter Speicherzugriff in der *ColMgsAvxPeel* und der *ColMgsAvxAlign* Programmvariante

Der Zugriff auf Daten an ausgerichteten Speicherstellen ermöglicht dem Prozessor ein vereinfachtes Laden der Daten in Vektorregister (vgl. Anhang Abschn. A.2). Zur Sicherstellung des ausgerichteten Speicherzugriffs wird das Loop Peeling (bzw. Loop Splitting) verwendet (vgl. Anhang Abschn. A.3).

In der **ColMgsAvxPeel** Programmvariante wird die *ColMgsAvx* Programmvariante als Ausgangspunkt zur Herstellung der ausgerichteten Speicherzugriffe genutzt. Durch Loop Peeling werden die ersten Iterationen der jeweiligen vektorisierten Schleife entfernt und seriell berechnet.

In der *ColMgsAvxPeel* Programmvariante wird nur der ausgerichtete Speicherzugriff hergestellt, nicht jedoch die Ladeinstruktion verändert. Die Ladeinstruktion wird

in der **ColMgsAvxAlign** Programmvariante angepasst. Dafür werden die Modifikationen der *ColMgsAvxPeel* Programmvariante übernommen und zusätzlich die entsprechenden Lade- und Speicherinstruktionen in die Instruktionen für ausgerichteten Speicher geändert.

5.2.4. Modifikationen der SIMD Reduktion

Die SIMD-Reduktion (vgl. Anhang Abschn. A.4) ist durch die mehreren Umsortierinstruktionen des Vektorregister eine sehr aufwändige Operation. Die erwarteten Vorteile einer parallelen Reduktion bei der Vektorisierung können sich dabei durch die aufwändigen Operationen reduzieren.

5.2.4.1. Teilweise serielle Berechnung in der vektorisierten Reduktion

Die Operationen der vektorisierten Reduktionsoperation können in mehrere Schritte eingeteilt werden: Die `float` Werte werden umsortiert und aufaddiert und es entstehen jeweils halb so viele Werte mit Zwischensummen wie vor dem jeweiligen Schritt (vgl. Anhang Abschn. A.4). Nach jedem dieser Schritte kann die vektorisierte Berechnung unterbrochen werden und stattdessen eine serielle Berechnung fortgeführt werden. Die genauen Stellen der Unterbrechung sind in Alg. A.1 im Anhang gezeigt.

Die **ColMgsAvxVred** Programmvarianten erlauben eine Untersuchung des optimalen Punktes für eine Unterbrechung der vektorisierten Reduktion. Dazu werden mehrere Programmvarianten **ColMgsAvxVred** aus der *ColMgsAvx* Programmvariante erstellt, welche die Reduktion jeweils an einer anderen Stelle unterbrechen und durch eine serielle Berechnung fortsetzen. An der Stelle der Unterbrechung wird das Vektorregister mit den Zwischenergebnissen in ein temporäres Array gespeichert und die Additionen im Anschluss innerhalb einer Schleife auf diesem Array durchgeführt. Die Programmvarianten erhalten dafür einen Suffix, der die Anzahl der verbleibenden Elemente des Vektorregisters nach der vektorisierten Reduktion angibt. Die Programmvariante **ColMgsAvxVred8** berechnet somit alle 8 Elemente des Vektorregisters seriell. Im Gegensatz dazu führt die **ColMgsAvxVred1** Programmvariante keine seriellen Additionen durch und entspricht damit der *ColMgsAvx* Programmvariante. Weitere Programmvarianten sind *ColMgsAvxVred2* und *ColMgsAvxVred4*.

Bei der Nutzung von AVX512 enthält ein Vektorregister bis zu 16 `float` Werte, wodurch hierfür die vollständig serielle Reduktion in der **ColMgsAvxVred16** Programmvariante implementiert ist. Zusätzlich steht bei der AVX512-Erweiterung ein Intrinsic zur Reduktion zur Verfügung (vgl. Anhang Abschn. A.5). Das reduce_add Intrinsic wird durch eine Serie von Shift-, Kopier- und Umsortierinstruktionen implementiert und in der Programmvariante **ColMgsAvxVredi** genutzt.

5.2.4.2. Modifikation der Position der Reduktionsoperation

Zusätzlich zur Modifikation der Durchführung der Reduktionsoperation kann auch die Position der Reduktionsoperation angepasst werden. Die Reduktionsoperation in

```
1  // Schritt 3:
2    // Schleife über die Spalten a_{_,i+1} ... a_{_,n} und Elemente r_{i,i+1} ... r_{i,n}:
3    for j = i+1 < n; j++
4      __m256 v_Rij = _mm256_setzero_ps();
5      // Schleife über die Elemente a_{_,j} der j-ten Spalte und q_{_,i} der i-ten Spalte:
6      for k = 0 < m; k += 8
7        // Laden der Elemente q_{k,i}, q_{k+1,i}, ..., q_{k+7,i}:
8        __m256 v_Qki = _mm256_setr_ps(&Q[k*n+i], ..., &Q[(k+7)*n+i]);
9        // Laden der Elemente a_{k,j}, a_{k+1,j}, ..., a_{k+7,j}:
10       __m256 v_a = _mm256_setr_ps(&A[k*n+j], ..., &A[(k+7)*n+j]);
11       // Teilergebnisse des Skalarproduktes von q_i^T · a_{k,j}:
12       v_Rij = _mm256_fmadd_ps(v_Qki, v_a, v_Rij);
13     R[i*n+j] += addreduce(v_Rij); // r_{i,j} = q_{_,i}^T · a_{k,j}
```

Algorithmus 5.8: MgsAvxMred Programmvariante: Die Reduktionsoperation in Schritt 3 wird aus der inneren Schleife in die äußere Schleife verschoben. Gezeigt ist ein Ausschnitt von Schritt 3. Veränderte Zeilen im Vergleich zur *MgsAvx* Programmvariante in Alg. 5.4 sind hervorgehoben.

Schritt 3 der *MgsAvx* Programmvariante in Alg. 5.4 wird innerhalb der inneren Schleife ausgeführt, was einen häufigen Aufruf dieser Funktion bedeutet. In der **MgsAvxMred** Programmvariante wird die Reduktionsoperation aus der inneren Schleife heraus gelöst und in die äußere Schleife von Schritt 3 verschoben.

Algorithmus 5.8 zeigt wie in der **MgsAvxMred** Programmvariante die Reduktion in Zeile 13 aus der inneren Schleife heraus gelöst wird. Dazu wird die Vektorvariable v_Rij in Zeile 4 mit Nullen initialisiert. In Zeile 12 wird die Multiplikation um eine Addition erweitert. Zum Erhalt des Wertes für $r_{i,j}$ müssen dann die Zwischenergebnisse der Vektorvariablen v_Rij in Zeile 13 reduziert werden.

Analog zur *MgsAvxMred* wird die **ColMgsAvxMred** Programmvariante aus der *ColMgsAvx* Programmvariante erstellt. Zusätzlich wird eine **ColMgsAvxMredAlign** Programmvariante aus der *ColMgsAvxAlign* Programmvariante erstellt.

5.3. Ausführungszeit und Energieverbrauch der vektorisierten Programmvarianten des Gram-Schmidt Prozesses

Die vorgestellten Programmvarianten des Gram-Schmidt Prozesses zur Vektororthogonalisierung werden in diesem Abschnitt auf ihre Ausführungszeit und den Energieverbrauch untersucht. Die Messergebnisse werden vorgestellt und deren Besonderheiten diskutiert.

Architektur	Jahr	Prozessor	AVX-Version	Frequenzbereich
Sandy Bridge	2011	Core i7-2600	AVX	1,5-3,7 GHz
Haswell	2013	Core i7-4770K	AVX2	0,8-3,5 GHz
Skylake	2015	Core i7-6700	AVX2	0,8-3,5 GHz
Knights Landing	2016	XEON Phi 7250	AVX512	1,0-1,4 GHz
Skylake	2017	XEON Gold 6130	AVX512	1,0-2,1 GHz

Tabelle 5.3.: Verwendete Prozessoren zur Ausführung der Programmvarianten der MGS mit der jeweiligen Prozessorarchitektur, dem Erscheinungsjahr, dem höchsten unterstützten AVX- Instruktionssatz und des einstellbaren Frequenzbereichs. Vergleiche [40].

5.3.1. Ausführungsumgebung für die Untersuchung der Programmvarianten des Gram-Schmidt Prozesses

Die Messungen diese Kapitels werden auf 3 Desktop und 2 Server Prozessoren aus verschiedenen Prozessorarchitekturen durchgeführt. Tabelle 5.3 zeigt eine Auflistung der verwendet Prozessoren. Die Prozessoren unterscheiden sich unter anderem in der Prozessorfamilie und dem unterstützten AVX-Instruktionssatz.

Die Programmvarianten werden mit der Intel Compiler Suite (icc Version 17) mit den Compilerflags −std=c++11,−O3 und −restrict erzeugt. Für die Auswahl der AVX-Version des Prozessors werden die folgenden Compilerflags genutzt: −mavx (Sandy Bridge,AVX), −march=core−avx2 (AVX2) oder −xcore−avx512 (AVX512). Die Prozessorfrequenz wird mit dem `cpu-freq` Tool eingestellt und der Energieverbrauch wird mit Hilfe des RAPL Interfaces bestimmt (vgl. Abschn. 4.3.1).

Vor jeder Messung wird die entsprechende Programmvariante im Sinne eines *warm-up* des Prozessors ausgeführt. Im Anschluss wird jede Programmvariante fünf mal ausgeführt und der Mittelwert der Messwerte gebildet. Bei Programmvarianten, die eine Umordnung der Datenstrukturen erfordern (*Col*-Präfix), wird die Umordnung der Datenstrukturen in die Messung eingeschlossen.

5.3.2. Ausführungszeit der vektorisierten MGS-Programmvarianten

In diesem Abschnitt werden die Ausführungszeiten der erstellten Programmvarianten des MGS vorgestellt und die Besonderheiten diskutiert. Zuerst werden dafür die seriellen und vektorisierten Programmvarianten mit modifizierter Datenzugriffsreihenfolge aus den Tabellen 5.1 und 5.2 untersucht. Im Anschluss werden die Modifikation der vektorisierten Programmvarianten, aufgeteilt nach Programmvarianten mit spaltenweisem und zeilenweisem Datenzugriff, sowie der teilweise seriellen Reduktionsoperation besprochen.

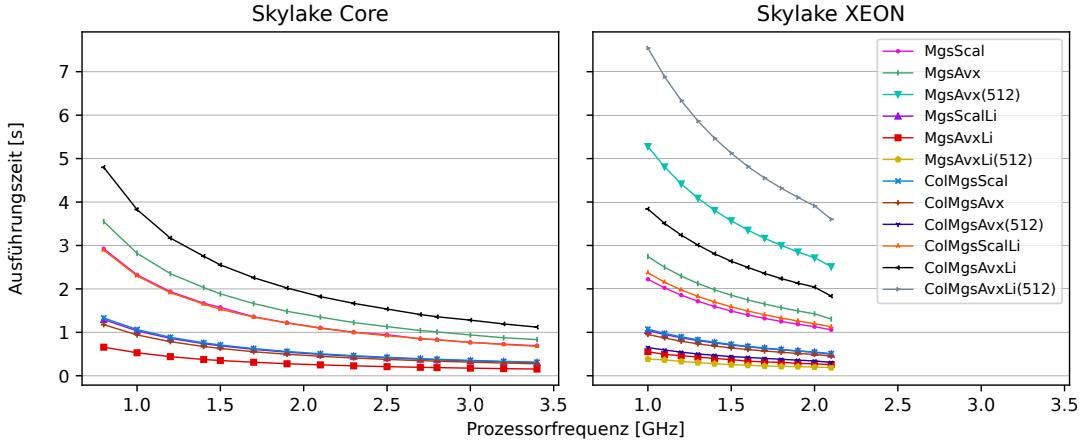

Abbildung 5.3.: Ausführungszeit der Programmvarianten mit unterschiedlicher Datenzugriffsreihenfolge in Abhängigkeit zur Prozessorfrequenz auf den beiden Skylake Prozessoren. Matrixgröße: 820×820. Die Ausführungszeiten aller Prozessoren sind im Anhang in den Abb. D.1 bis D.5 dargestellt.

5.3.2.1. Programmvarianten mit modifizierter Datenzugriffsreihenfolge

Die Veränderung der Datenzugriffsreihenfolge soll die Unterschiede der Zugriffe auf die Matrixelemente der seriellen und der vektorisierten Programmvarianten zeigen.

Zeilen- und spaltenweiser Elementzugriff

Abbildung 5.3 zeigt die Ausführungszeiten der seriellen und vektorisierten Programmvarianten mit Schleifentausch und/oder spaltenweiser Speicherung der Matrizen auf den Skylake Prozessoren. Bei den seriellen Programmvarianten haben die Programmvarianten mit spaltenweisem Elementzugriff (*MgsScal* und *ColMgsScalLi*) wie erwartet nahezu gleiche Ausführungszeiten. Die seriellen Programmvarianten mit zeilenweisem Elementzugriff (*MgsScalLi* und *ColMgsScal*) verhalten sich analog.

Die vektorisierten Programmvarianten mit spaltenweisem Elementzugriff haben eine höhere Laufzeit als deren serielles Gegenstück. Weiterhin zeigt sich, dass die spaltenweise Abspeicherung der Matrizen zusätzliche Laufzeit benötigt, weshalb die *ColMgsAvxLi* Programmvariante im Vergleich zur *MgsAvx* Programmvariante eine höhere Ausführungszeit aufweist.

Bei den vektorisierten Programmvarianten mit zeilenweisem Elementzugriff (*MgsAvxLi* und *ColMgsAvx*) zeigt sich durch die Vektorisierung eine Reduktion der Ausführungszeit. Auch hierbei wirkt sich der zusätzliche Aufwand zum Umsortieren der Matrizen in einer geringeren Reduktion bei der *ColMgsAvx* Programmvariante aus.

Auf dem Skylake XEON Prozessor (Abb. 5.3 (rechts)) wird die Reduktion und Erhöhung der Ausführungszeit der vektorisierten Programmvarianten bei der Verwendung von AVX512 noch deutlicher hervorgehoben. Durch die größeren Vektorregister werden beim spaltenweisen Elementzugriff noch mehr Datenelemente je Vektor gesammelt, was zu einer zusätzlichen Erhöhung der Ausführungszeit im Vergleich zu den

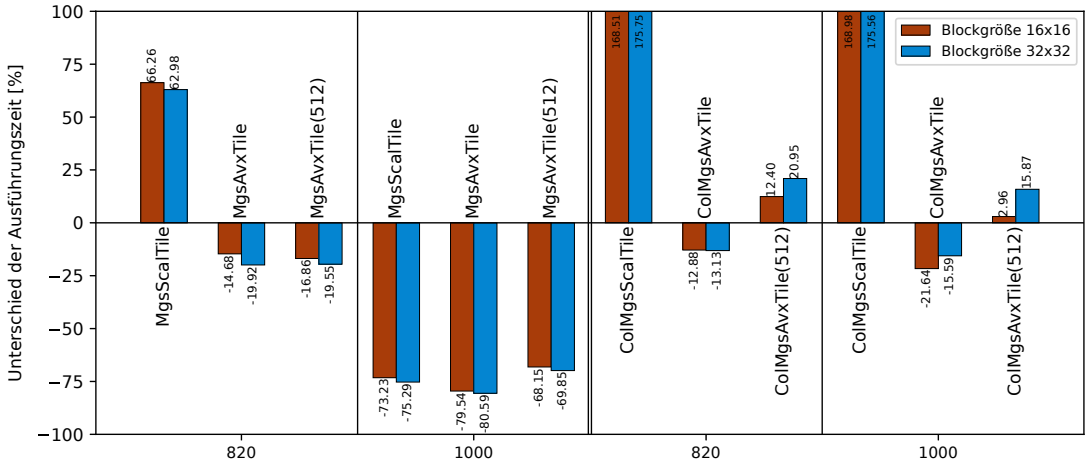

Abbildung 5.4.: Unterschied der Ausführungszeit in % der AVX-Programmvarianten mit *loop tiling* im Vergleich zur Programmvariante ohne *loop tiling*. Gezeigt sind die Werte des Skylake XEON Prozessors; Prozessorfrequenz: 2,0GHz; Matrixgröße: 820×820 bzw. 1000×1000. Negative Werte sind kürzere Ausführungszeiten. Die Messergebnisse der verbleibenden Prozessoren sind im Anhang in Abb. D.6 und D.6 dargestellt.

256-Bit Programmvarianten führt. Bei den *MgsAvxLi* und *ColMgsAvx* Programmvarianten können größere Speicherblöcke als bei den 256-Bit Programmvarianten geladen werden, wodurch sich die Ausführungszeiten weiter reduzieren.

Das besprochene Verhalten zeigt sich auch für die verbleibenden Prozessoren.

Schleifen-Blockzerlegung

Bei der Schleifen-Blockzerlegung (*loop tiling*) wird der Iterationsraum der Schleifennester in den Schritten 3 und 4 in Blöcke unterteilt. Durch die veränderte Bearbeitungsreihenfolge inerhalb dieser Blöcke soll die Ausnutzung der Caches erhöht werden (höhere zeitliche Lokalität [63]), wodurch Speicherzugriffszeiten reduziert werden.

Abbildung 5.4 zeigt den Unterschied der Ausführungszeiten bei der Anwendung der Schleifen-Blockzerlegung im Vergleich zu entsprechenden Programmvariante ohne Schleifen-Blockzerlegung. Bei der *MgsScalTile* Programmvariante erhöht sich die Ausführungszeit bei einer Matrixgröße von 820×820 Elementen und reduziert sich bei einer Matrixgröße von 1000×1000 Elementen. Dies erklärt sich durch die Größe der zu speichernden Matrizen, da bei einer Größe von 820×820 alle genutzten Matrixelemente im Level-3 Cache des Prozessors gespeichert werden können. Hierbei führen die zusätzlichen Operationen und Sprungbefehle durch die Schleifen-Blockzerlegung zu einer höheren Ausführungszeit, die kaum von der höheren zeitlichen Lokalität profitiert. Bei der Vergrößerung der Matrizen können diese nicht mehr vollständig im Cache abgelegt werden, wodurch die höhere zeitliche Lokalität zu einer geringeren Ausführungszeit führt. Bei den vektorisierten Programmvarianten (*MgsAvxTile*) reduziert sich die Ausführungszeit, da die Speicherzugriffe innerhalb eines Blockes auf aufeinanderfolgende Speicherstellen durchgeführt werden können und somit weniger

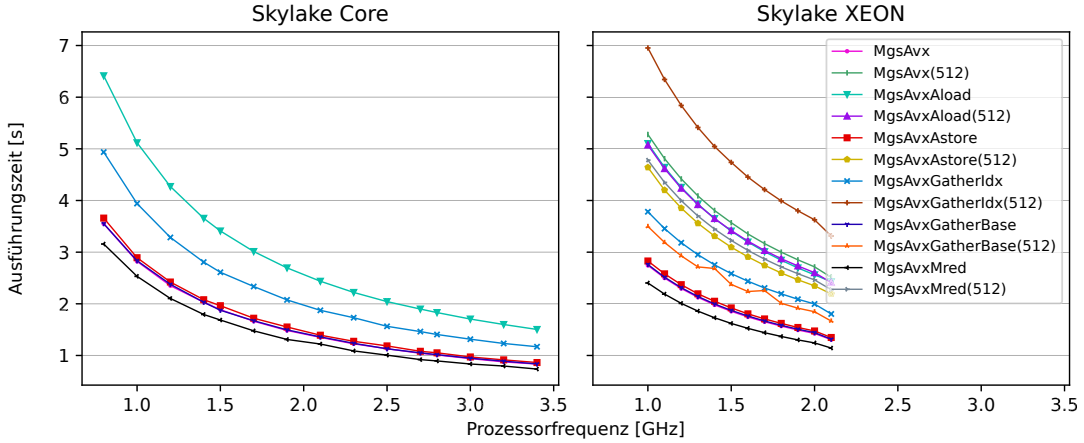

Abbildung 5.5.: Ausführungszeit der AVX-Programmvarianten mit spaltenweisem Datenzugriff in Abhängigkeit zur Prozessorfrequenz auf den beiden Skylake Prozessoren. Matrixgröße: 820×820. Die Ausführungszeiten aller Prozessoren sind im Anhang in den Abb. D.1 bis D.5 dargestellt.

verteilte Speicherzugriffe durchgeführt werden.

Programmvarianten mit spaltenweiser Matrixspeicherung (*Col*-Präfix) greifen auf aufeinanderfolgend abgelegte Speicherlemente zu. Die Anwendung der Schleifen-Blockzerlegung fügt dabei wieder Speicherzugriffe auf andere Speicherbereiche hinzu. Aus diesem Grund führt die Anwendung der Schleifen-Blockzerlegung bei solchen Programmvarianten zu einer Erhöhung der Ausführungszeit. Bei den vektorisierten Programmvarianten bleibt der zeilenweise Zugriff dabei erhalten. Die Erhöhung der zeitlichen Lokalität bewirkt bei der *ColMgsAvxTile* Programmvariante sogar eine leichte Reduzierung der Ausführungszeit.

Auf den verbleibenden Prozessoren (siehe Anhang Abb. D.6 und D.7) führt die Anwendung von Schleifen-Blockzerlegung bei spaltenweisem Elementzugriff stets zu einer reduzierten Ausführungszeit, was sich aus der geringeren Cache-Größe ergibt. Bei zeilenweisem Elementzugriff (*Col*-Präfix) führt die Schleifen-Blockzerlegung bei dem Sandy Bridge und dem XEON Phi Prozessor zu einer Verlängerung und bei dem Haswell und dem Skylake Core Prozessor zu einer Reduzierung der Ausführungszeit. Dies weist auf eine hardware-seitige Optimierung der Speicherstruktur oder des Prefetchers hin, bei der die höhere zeitliche Lokalität besser genutzt werden kann.

5.3.2.2. Ausführungszeit der AVX-Programmvarianten mit spaltenweisem Datenzugriff

Abbildung 5.5 zeigt die modifizierten AVX-Programmvarianten mit zeilenweisem Matrixelementzugriff. Die Zwischenspeicherung der zu sammelnden Matrixelemente (*MgsAvxAload*) bzw. der zu speichernden Matrixelemente (*MgsAvxAstore*) in einem temporären Array hat dabei für die 256-Bit Instruktionen eine höhere Ausführungszeit, als der direkte Zugriff auf das Vektorregister. Bei den 512-Bit Instruktion führt das Zwischenspeichern jedoch zu einer Reduzierung der Ausführungszeit. Dies deutet auf

prozessor-interne Optimierungen, wie z.B. beim Prefetching, hin, die beim Laden und Speichern verteilter Daten den Zugriff verbessern können. Übersteigt die Datenmenge dabei jedoch eine gewisse Größe kann der Zugriff durch die serielle Sammlung der Daten effizienter durchgeführt werden.

Die Nutzung der gather Instruktion mit einem Indexvektor (*MgsAvxGatherIdx*) erzeugt eine höhere Ausführungszeit im Vergleich zur *MgAvx* Programmvariante. Dies lässt sich darauf zurückführen, dass bei jeder Nutzung der gather Instruktion eine zusätzliche setr Instruktion zum Setzen der Indizes ausgeführt wird.

Wird die gather Instruktion mit einem konstanten Offset-Vektor verwendet (*MgsAxvGatherBase*) zeigt sich für die 256-Bit Instruktionen kein Unterschied in der Ausführungszeit. Für die 512-Bit Instruktion reduziert sich die Ausführungszeit im Vergleich zur *MgsAvx* Programmvariante. Dieses Verhalten ist dabei ähnlich zur *MgsAvxAload* Programmvariante und weist darauf hin, dass es bei der setr Instruktion eine Optimierung des Datenzugriffs gibt, die bei 512-Bit Instruktionen nicht mehr vorhanden ist.

In der *MgsAvxMred* Programmvariante wird die Reduktion in Schritt 3 der Berechnungen aus der innersten Schleife verschoben und dadurch weniger häufig aufgerufen. Der seltenere Aufruf der kostenintensiven Reduktion (vgl. Anhang Abschn. A.4) zeigt sich in einer Reduzierung der Ausführungszeit gegenüber der *MgsAvx* Programmvariante.

Die Untersuchungsergebnisse der Ausführungszeit der Programmvarianten mit spaltenweisem Datenzugriff können auf die Messungen auf dem XEON Phi Prozessor (siehe Anhang Abb. D.5) übertragen werden, wobei die Unterschiede in den Ausführungszeiten geringer sind. Für den Sandy Bridge (Anhang Abb. D.1) und den Haswell (Anhang Abb. D.2) Prozessor sind keine Unterschiede für die hier diskutierten Programmvarianten erkennbar. Für diese beiden älteren Prozessoren erklärt sich dies durch den geringen Einfluss von ausgetauschten Instruktionen, der durch die Abhängigkeit von der Speicheranbindung verdeckt wird. Dies wird auch bei den anderen Prozessoren deutlich, wenn die Größe der verwendeten Matrizen erhöht wird und sich somit die Cache-Ausnutzung verringert.

5.3.2.3. Ausführungszeit der AVX-Programmvarianten mit zeilenweisem Datenzugriff

Der zeilenweise Datenzugriff auf die Matrixelemente in Verbindung mit der Vektorisierung hat sich bereits im Vergleich mit den seriellen Programmvarianten als eine geeignete Methode zur Reduzierung der Ausführungszeit gezeigt. In diesem Abschnitt stehen deshalb die Modifikation der AVX-Programmvarianten mit zeilenweisem Datenzugriff im Vordergrund.

Abbildung 5.6 zeigt die Ausführungszeiten für diese Programmvarianten auf dem Sandy Bridge (links) und dem Skylake XEON (rechts) Prozessor. Auf dem Sandy Bridge Prozessor reduziert der ausgerichtete Datenzugriff die Ausführungszeit der *ColMgsAvxPeel* Programmvariante leicht. Die zusätzliche Nutzung einer load Instruktion zum ausgerichteten Laden reduziert die Ausführungszeit zusätzlich. Auf dem Sky-

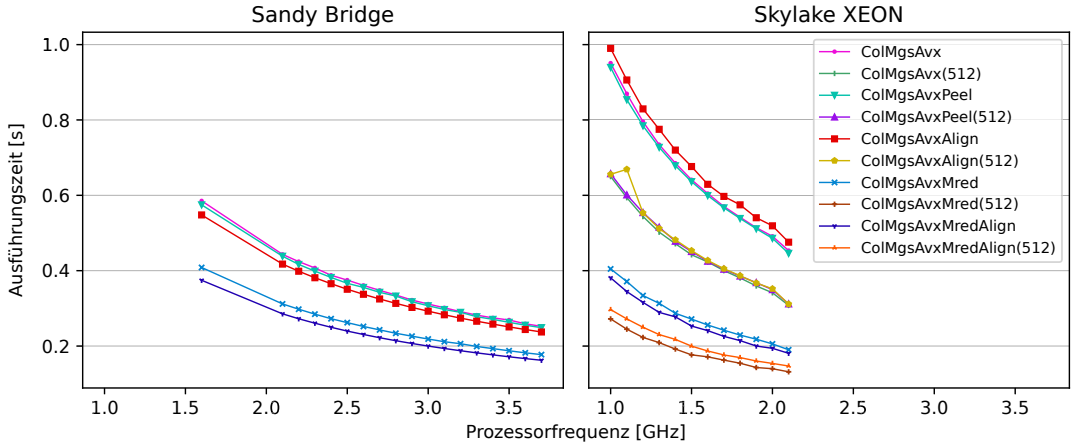

Abbildung 5.6.: Ausführungszeit der AVX-Programmvarianten mit zeilenweisem Datenzugriff in Abhängigkeit zur Prozessorfrequenz auf dem Sandy Bridge und dem Skylake XEON Prozessor. Matrixgröße: 820×820. Die Ausführungszeiten aller Prozessoren sind im Anhang in den Abb. D.1 bis D.5 dargestellt.

lake XEON Prozessor bringt der ausgerichtete Speicherzugriff in der *ColMgsAvxPeel* Programmvariante nur eine geringe Reduzierung der Ausführungszeit und die Nutzung der load Instruktion verlängert die Ausführungszeit sogar noch. Auf dem Haswell, Skylake Core und XEON Phi Prozessor zeigen sich nahezu keine Unterschiede zwischen ausgerichteten und unausgerichteten Speicherzugriffen (siehe Anhang Abb. D.2, D.3 und D.5).

Die Verschiebung der Reduktionsoperation aus der inneren Schleife in Schritt 3 in der *ColMgsAvxMred* Programmvariante reduziert die Ausführungszeit auf allen Prozessoren deutlich. Vor allem im Vergleich zur gleichen Modifikation bei spaltenweiser Speicherung zeigt sich hier die geringere Abhängigkeit zur Speicheranbindung, wodurch der Effekt von effizienteren Berechnungen stärker hervortritt. Die zusätzliche Verwendung von ausgerichteten Speicherinstruktionen in der *ColMgsAvxMredAlign* Programmvariante verändert die Laufzeit der *ColMgsAvxMred* Programmvariante entsprechend der vorherigen Beobachtung.

5.3.2.4. Ausführungszeiten mit teilweise serieller Berechnung der vektorisierten Reduktion

Zum Vergleich der Kosten einer vektorisierten Reduktion (vgl. Anhang Abschn. A.4) mit einer seriellen oder teilweise seriellen Reduktion werden die *ColMgsAvxVred** Programmvarianten untersucht.

Abbildung 5.7 zeigt die Ausführungszeiten der *Vred** Programmvarianten auf den verschiedenen Prozessoren. Die Zahl im Namen der Programmvariante gibt dabei die Anzahl der seriell reduzierten Elemente an. Für alle Prozessoren zeigt sich eine geringere Ausführungszeit je mehr Elemente durch die vektorisierte Reduktion berechnet werden. Für die 256-Bit Instruktionen verringert sich die Ausführungszeit mit der

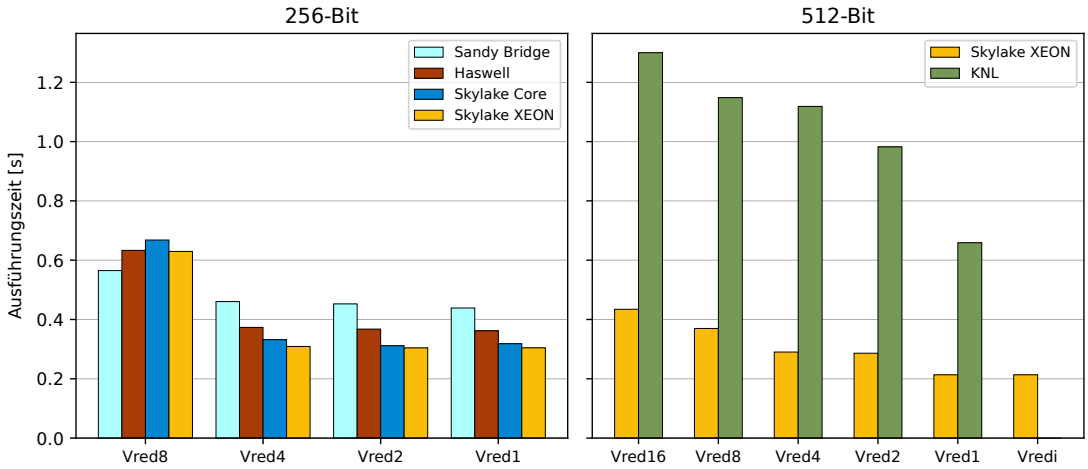

Abbildung 5.7.: Ausführungszeit der *ColMgsAvxVred** Programmvarianten mit teilweise serieller Reduktionsoperation. Die Zahl im Namen der Programmvariante gibt die Anzahl seriell berechneter Elemente an. Eingestellte Prozessorfrequenz: Sandy Bridge 1,6GHz, Haswell 1,4GHz, Skylake Core 1,4GHz, Skylake XEON 1,5GHz und XEON Phi 1,4GHz. Matrixgröße: 820×820. Niedrigere Werte sind besser.

Vred4 Variante am meisten. Die weitere Vektorisierung der Reduktion in den Varianten *Vred2* und *Vred1* zeigt nur noch geringe Veränderungen der Ausführungszeit. Dies zeigt, dass die geringere Ausführungszeit durch eine vektorisierte Reduktion nur bis zu einem gewissen Punkt von der parallelen Berechnung der Elemente profitiert, da die Ausführung der kostenintensiven Umsortierinstruktionen diesen Vorteil aufwiegt. Zusätzlich wird für die erste Umsortierinstruktion ein Tausch der 128-Bit Blöcke des Vektorregisters genutzt, der effizienter ist als die darauf folgenden Umsortierinstruktionen innerhalb der 128-Bit Blöcke.

Für die Ausführung mit 512-Bit Instruktionen zeigen die Programmvarianten ein unterschiedliches Verhalten. Je mehr Schritte der Reduktion vektorisiert werden, desto geringer wird die Ausführungszeit der Programmvariante. Dabei entsteht bei dem Skylake XEON Prozessor zwischen der *Vred4* und *Vred2* Programmvariante, sowie bei dem XEON Phi Prozessor zwischen der *Vred8* und *Vred4* Programmvariante, nur eine geringe Verringerung der Ausführungszeit. Diese erklärt sich durch die unterschiedliche Implementierung der Umsortierinstruktionen auf den verschiedenen Prozessoren, die sich von den 256-Bit Umsortierinstruktionen unterscheiden.

Auf dem Skylake XEON Prozessor wird zusätzlich ein reduce_add Intrinsic angeboten, das durch eine Serie von Shift-, Kopier- und Umsortierinstruktionen implementiert ist. Die Verwendung (*Vredi* in Abb. 5.7) dieses speziellen Intrinsics verringert die Ausführungszeit im Vergleich zur *Vred1* Programmvariante weiter.

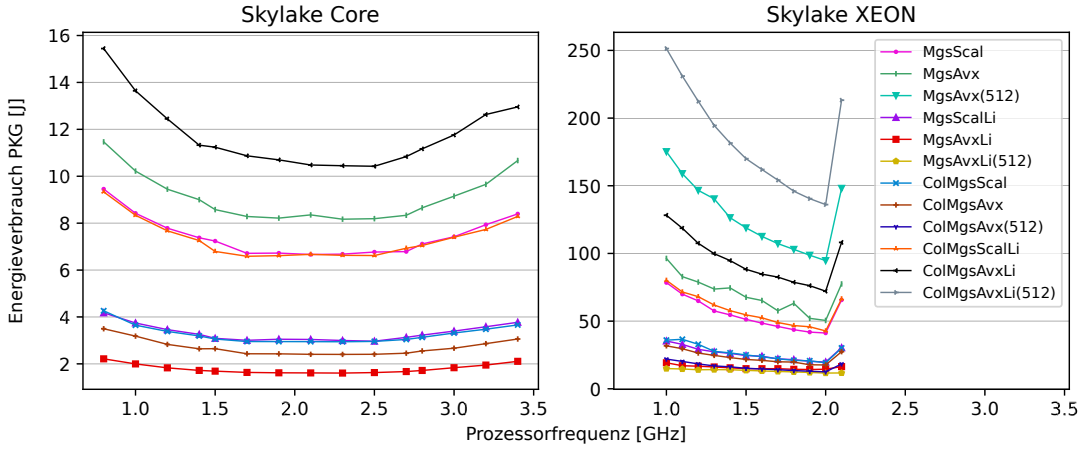

Abbildung 5.8.: Energieverbrauch der Programmvarianten mit unterschiedlicher Datenzugriffsreihenfolge in Abhängigkeit zur Prozessorfrequenz auf den beiden Skylake Prozessoren. Matrixgröße: 820×820. Der Energieverbrauch aller Prozessoren ist im Anhang in den Abb. D.8 bis D.12 dargestellt.

5.3.3. Energieverbrauch der vektorisierten MGS-Programmvarianten

In diesem Abschnitt werden die Besonderheiten des Energieverbrauchs und der Leistungsaufnahme der verschiedenen Programmvarianten diskutiert. Die Korrelation von Energieverbrauch und Ausführungszeit ist bereits in Kapitel 4 gezeigt und wird hier nicht besprochen. In diesem Abschnitt werden jedoch Effekte diskutiert, die von dieser direkten Korrelation abweichen. Die Messergebnisse des Energieverbrauchs der verwendeten Prozessoren findet sich im Anhang in den Abb. D.8 bis D.12.

Abbildung 5.8 zeigt den Energieverbrauch der Programmvarianten mit unterschiedlicher Datenzugriffsreihenfolge auf den Skylake Prozessoren. Bei den Messwerten zeigt sich wie erwartet der geringste Energieverbrauch bei einer mittleren Prozessorfrequenz. Das Verhältnis der Energieverbrauchswerte der Programmvarianten ist identisch zum Verhältnis der Ausführungszeiten zueinander.

Der absolute Energieverbrauch der Ausführungen auf dem Skylake XEON Prozessor ist fast 10-mal höher als die Energieverbrauchswerte der gleichen Programmvariante auf dem Skylake Core Prozessor. Dieser Unterschied begründet sich durch die unterschiedlichen Produktspezifikationen der beiden Prozessoren. Der Skylake XEON Prozessor hat 4-mal so viele physische Prozessorkerne wie der Skylake Core Prozessor [40]. Der Skylake XEON Prozessor hat dadurch auch im Leerlauf einen wesentlich höheren Energieverbrauch. Die Programmvarianten dieser Arbeit untersuchen jedoch nur den Einfluss der Vektorisierung, wodurch die höhere Anzahl an Prozessorkernen keinen Einfluss auf die Ausführungszeit hat. Somit entfällt ein Großteil des Energieverbrauchs des Skylake XEON Prozessors auf einen statischen Anteil der durch den Prozessor im Leerlauf konsumiert wird. Eine Ausführung bei Abschaltung einzelner Prozessorkerne, wie in [52] modelliert, würde den statischen Energieverbrauch redu-

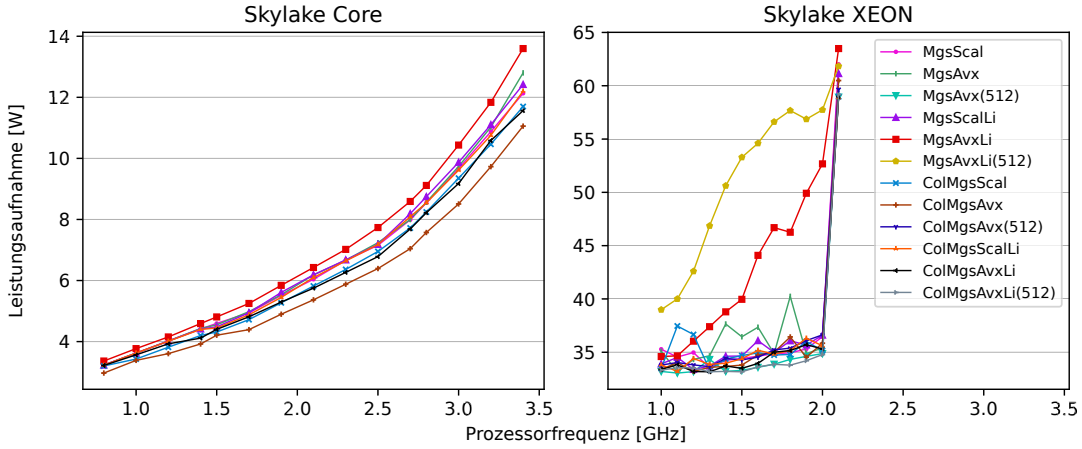

Abbildung 5.9.: Leistungsaufnahme der Programmvarianten mit unterschiedlicher Datenzugriffsreihenfolge in Abhängigkeit zur Prozessorfrequenz auf den beiden Skylake Prozessoren. Matrixgröße: 820×820. Die Leistungsaufnahme aller Prozessoren ist im Anhang in den Abb. D.13 bis D.17 dargestellt.

zieren, kann jedoch aufgrund der Prozessorarchitektur nicht durchgeführt werden.

Bei dem Skylake XEON Prozessor zeigt sich in Abb. 5.8(rechts) eine Besonderheit bei der maximal einstellbaren Prozessorfrequenz von 2,1GHz. Zu erwarten wäre für die Prozessorfrequenz ein vergleichbarer Wert zur Ausführung mit einer Prozessorfrequenz von 2,0GHz. Stattdessen zeigen fast alle Programmvarianten hier einen sprunghaften Anstieg des Energieverbrauchs. Dieser Anstieg bei der maximalen Prozessorfrequenz könnte dabei ein Hinweis auf einen aktivierten Turbo-Boost Modus des Prozessors sein. Betrachtet man dazu jedoch die Leistungsaufnahme der Programmvarianten in Abb. 5.9(rechts) widerspricht die stetig steigende Leistungsaufnahme der *MgsAvxLi* Programmvariante dieser Annahme. Verschiebt man die Werte der Leistungsaufnahme von 2,1GHz auf die Tubro-Boost-Frequenz von 3,7GHz ergibt sich für den Energieverbrauch in etwa das erwartete Verhalten, was wiederum für einen aktiven Turbo-Boost Modus spricht.

Zum Überprüfen der Verfügbarkeit des Turbo-Boost Modus bietet der Prozessor ein Register an (vgl. [36, Vol.4 2-134]). Der Wert des Registers gibt jedoch einen deaktivierten Turbo-Boost Modus an. Als mögliche Begründung für dieses Verhalten bleibt ein Fehler innerhalb des Prozessors oder beim Tool zum Einstellen der Prozessorfrequenz, durch den die Turbo-Boost Funktion oder eine vergleichbare Funktionalität nicht korrekt eingestellt wird.

Für die Leistungsaufnahme in Abb. 5.9(links) zeigen sich auch Unterschiede im Verhalten der *MgsAvx*, *ColMgsAvx* und *ColMgsScal* Programmvarianten bei einem Vergleich mit deren Energieverbrauch in Abb. 5.8(links). Die *MgsAvx* Programmvariante hat im Vergleich zu den restlichen Programmvarianten einen hohen Energieverbrauch, zeigt aber auch eine hohe Leistungsaufnahme. Zusätzlich zeigen die *ColMgsScal* und die *ColMgsAvx* Programmvariante einen niedrigen Energieverbrauch, aber auch eine niedrige Leistungsaufnahme. Eine Erklärung für dieses Verhalten findet sich in der

Kombination aus Umspeicherung der Matrizen und dem verbesserten Datenzugriff: Bei der Umsortierung werden primär Speicheroperationen durchgeführt, während denen der Prozessor weniger Energie verbraucht. Im Anschluss müssen für Datenzugriffe weniger Prozessorinstruktionen ausgeführt werden, da die Daten nicht aus mehren Speicherblöcken gesammelt werden.

Die verbleibenden Programmvarianten zeigen dass erwartete Verhalten, dass bei einer Verringerung der Ausführungszeit der Energieverbrauch ebenfalls sinkt. Dabei erhöht sich die Leistungsaufnahme für diese Programmvarianten um 10-20%.

5.4. SIMD-Programmierung für prozessor-integrierte Grafikprozessoren

Moderne Prozessoren bieten mit prozessor-integrierten Grafikprozessoren (iGPUs) neben AVX-Einheiten eine weitere Form der SIMD-Ausführung von Programmen an. Die iGPU besitzt dabei keinen eigenen Speicher sondern nutzt den Hauptspeicher sowie Teile der Cache-Struktur des Prozessors [35]. Die Ausführung von SIMD Programmen auf der iGPU kann somit im Vergleich zur Ausführung auf einer Grafikkarte auf Kopieroperationen der genutzten Daten verzichten und im Optimalfall auf bereits in den Cache geladene Daten zugreifen.

In diesem Abschnitt wird die Nutzung der iGPU zur Berechnung einer QR-Zerlegung mit dem modifizierten Gram-Schmidt Prozess kurz vorgestellt. Die entsprechenden Programmvarianten sowie eine Diskussion der Messergebnisse wurde bereits in [45] veröffentlicht.

Für die Nutzung der iGPU zur Ausführung von SIMD-Programmen werden Programmvarianten mit OpenCL [48] erstellt und für die integrierte GPU angepasst. Die *Zero Copy* Programmvariante vermeidet dabei Kopieroperationen zwischen CPU und iGPU und zeigt die geringste Ausführungszeit der in [45] vorgestellten Programmvarianten.

Abbildung 5.10 zeigt die Ausführungszeit und den Energieverbrauch der CPU-Programmvariante und der *Zero Copy* Programmvariante auf den in diesem Kapitel verwendeten Haswell und Skylake Core Prozessoren. Bei der Erstellung der CPU-Programmvariante wird die automatische Vektorisierung des Compilers nicht explizit unterbunden, sodass eine automatische Vektorisierung stattfindet. Beim Vergleich der CPU und iGPU Programmvarianten zeigt sich, dass die Ausführung des MGS auf der iGPU ab einer bestimmten Matrixgröße (Übergangspunkt) mit geringerer Ausführungszeit und geringerem Energieverbrauch durchgeführt werden kann. Zu beachten ist dabei, dass dieser Übergangspunkt für die Ausführungszeit und den Energieverbrauch bei unterschiedlichen Matrixgrößen liegt, da die deaktivierte iGPU in den CPU-Programmvarianten keine Energie verbraucht. Die iGPU verfügt über eine große Anzahl parallel genutzter Recheneinheiten (168 in der Skylake-Gen9 iGPU [35]), sodass im Vergleich zur Berechnung auf der CPU mehr Elemente in kürzerer Zeit verarbeitet werden können. Für kleinere Matrizen reicht die Anzahl der parallel

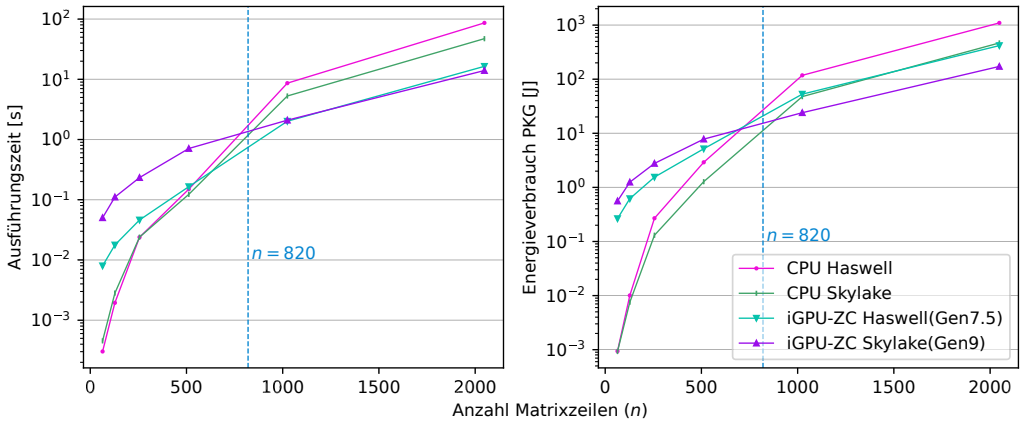

Abbildung 5.10.: Ausführungszeit (links) und Energieverbrauch (rechts) der CPU-Programmvariante (CPU) und der *Zero Copy* Programmvariante (iGPU-ZC) in Abhängigkeit zur Größe der verwendeten Matrizen des modifizierten Gram-Schmidt Prozesses. Prozessor und iGPU werden jeweils mit der höchsten einstellbaren Frequenz betrieben. Die im Rest des Kapitels genutzte Matrixgröße von 820×820 ist markiert.

zu verarbeitenden Elemente nicht aus, um den Aufwand zur Aktivierung der iGPU auszugleichen.

Die zuvor vorgestellten vektorisierten Programmvarianten der MGS können an Stelle der CPU-Programmvariante eingesetzt werden, um die jeweils kürzeste Ausführungszeit bei einer gegebenen Matrixgröße zu ermitteln. Verkürzt sich die Ausführungszeit der CPU-Programmvariante, verschiebt sich der *Übergangspunkt* zu einer größeren Matrixgröße. Wird die Ausführungszeit der iGPU-Programmvarianten reduziert, verschiebt sich der *Übergangspunkt* zu einer kleineren Matrixgröße. Die Nutzung der iGPU bietet somit eine zusätzliche Möglichkeit zur Reduzierung von Ausführungszeit und Energieverbrauch bei der Ausführung von SIMD-Programmen.

5.5. Zusammenfassung der Vektorisierung des Gram-Schmidt Prozesses

Die vorgestellten vektorisierten Programmvarianten des Gram-Schmidt Prozesses zur Vektororthogonalisierung unterscheiden sich in der Datenzugriffsreihenfolge, dem Speicherzugriff und der Umsetzung der vektorisierten Reduktion. Die kürzeste Ausführungszeit und der geringste Enegieverbrauch kann dabei mit einer Komibination aus zeilenweisem Elementzugriff und vollständig vektorisierter Reduktion erreicht werden.

Die Untersuchungen zeigen, dass die Anpassung der Datenzugriffreihenfolge auf eine vektorisierte Berechnung der QR-Zerlegung den größten Einfluss haben. Eine passende Reihenfolge kann dabei entweder durch einen Schleifentausch oder durch Veränderung des Speicherformats der genutzten Matrizen erreicht werden. Es zeigt

sich, dass solche Anpassungen bei der Nutzung von AVX512 noch größere Bedeutung haben, als bei der Nutzung von 256-Bit Instruktionen. Die Nutzung von ausgerichteten Speicherzugriffen lohnt sich dabei vor allem auf älteren Prozessoren für die AVX und AVX2-Erweiterungen.

Die Erhöhung der zeitlichen Lokalität von Speicherzugriffen durch Schleifen-Blockzerlegung bietet vor allem dann Vorteile, wenn die verwendeten Daten die Cache-Größe übersteigen. Zusätzlich kann die Schleifen-Blockzerlegung die Datenzugriffsreihenfolge von vektorisierten Programmen verbessern.

Kann ein verteilter Datenzugriff in vektorisierten Programmen nicht verhindert werden, stellt die Nutzung der gather Instruktion mit einem konstanten Offset-Vektor die beste Alternative zur Reduzierung der Ausführungszeit dar.

Der Energieverbrauch und die Leistungsaufnahme der vektorisierten Programmvarianten verhält sich bei den vorgestellten Programmvarianten wie erwartet: Eine geringere Ausführungszeit führt zu einem geringeren Energieverbrauch aber zu einer erhöhten Leistungsaufnahme.

6

Zusammenfassung zur energie-effizienten Vektorisierung der Algorithmen

In dieser Arbeit wurden verschiedene Algorithmen der linearen Algebra für die SIMD Ausführung auf Prozessoren vektorisiert und in Bezug auf ihre Ausführungszeit und Energie-Effizienz hin untersucht. Die untersuchten Algorithmen der linearen Algebra wurden im Hinblick auf verschiedene Eigenschaften bei der Vektorisierung ausgewählt, sodass die Ergebnisse der Untersuchungen auf ein breites Spektrum an Algorithmen übertragen werden können. Die Vektorisierung der Algorithmen erfolgte durch automatische Vektorisierung und durch die Vektorisierung mittels AVX-Intrinsics. Die vorgestellten Programmvarianten wurden dabei unter anderem mit verschiedenen Programmtransformationen, wie Schleifen-Blockzerlegung, und einer unterschiedlichen Auswahl an AVX-Instruktionen implementiert.

Die Matrix-Multiplikation eignet sich durch die gleichmäßige Struktur und den hohen Grad der Parallelität für die Untersuchung von Schleifentransformationen und unterschiedlichen Datenzugriffsmustern. Die Vektorisierung mit AVX-Intrinsics bei der Anwendung der Schleifen-Blockzerlegung zur Herstellung eines zeilenweisen Zugriffs auf die Matrizen konnte hierbei die geringste Ausführungszeit und den geringsten Energieverbrauch erreichen. Die Schleifen-Blockzerlegung sollte dabei mit einer Blockgröße durchgeführt werden, die auf die Anzahl der verfügbaren AVX-Register abgestimmt ist, um die besten Ergebnisse zu erhalten.

Die Gauss-Elimination wird aufgrund der unregelmäßigen Struktur durch die unterschiedliche Anzahl der Datenelemente pro Schleifendurchlauf, und die damit verbundene Speicherstelle des ersten Elements eines Vektors, untersucht. Die Nutzung von Lade- und Speicheroperationen auf ausgerichtete Speicherstellen hat dabei die geringste Ausführungszeit und den geringsten Energieverbrauch gezeigt. Die automatische Vektorisierung durch den Compiler und die Nutzung der Intel Cilk Array-Notation erreichen dabei die gleiche Ausführungszeit und den gleichen Energieverbrauch, wie die Programmvarianten mit AVX-Intrinsics.

Der modifizierte Gram-Schmidt Prozess zur Vektororthogonalisierung (MGS) bietet durch die Berechnung mehrerer voneinander abhängiger Schritte die Möglichkeit zur Untersuchung verschiedener Programmtransformationen bei der Vektorisierung mit AVX-Intrinsics. Die spaltenweise Speicherung der Matrizen zeigt dabei die größte

Reduzierung von Ausführungszeit und Energieverbrauch. Besonders wichtig ist diese Transformation bei der Vektorisierung, da sich bei Programmvarianten mit spaltenweisem Zugriff die Ausführungszeit und der Energieverbrauch durch Vektorisierung erhöhen können. Die Nutzung der Schleifen-Blockzerlegung reduziert Ausführungszeit und Energieverbrauch vor allem für Programmvarianten mit spaltenweisem Matrixzugriff.

Der Energieverbrauch von Programmen wird durch die Vektorisierung in ähnlichem Maße reduziert wie die Ausführungszeit. Das Verhältnis von Ausführungszeit und Energieverbrauch, die Leistungsaufnahme des Prozessors, erhöht sich durch die Nutzung der größeren Register und Recheneinheiten um bis zu 20%. Ausführungszeit, Energieverbrauch und Leistungsaufnahme von vektorisierten Programmvarianten zeigen die gleichen Abhängigkeiten zur Prozessorfrequenz wie serielle Programmvarianten. Im Detail sinkt die Ausführungszeit mit steigender Prozessorfrequenz, wobei die Leistungsaufnahme steigt. Der Energieverbrauch ist dadurch bei einer mittleren Prozessorfrequenz minimal. Die minimalen Werte von Energieverbrauch und Leistungsaufnahme verschieben sich dabei durch die höhere Prozessorausnutzung hin zu einer niedrigeren Prozessorfrequenz.

Die Kombination von Vektorisierung und Schleifen-Blockzerlegung ist dann besonders effektiv, wenn die Speicherzugriffe der begrenzende Faktor für die Ausführungsführungszeit sind. Die Schleifen-Blockzerlegung kann dabei, wie in seriellen Programmen, die Caches aufgrund der Annahme der zeitlichen Lokalität besser ausnutzen. Zusätzlich kann die Schleifen-Blockzerlegung den Datenzugriff auf Matrizen von spaltenweisen in zeilenweise Zugriffsmuster verändern, was bei der Vektorisierung die Nutzung schnellerer Lade- und Speicheroperationen erlaubt.

Die Auswahl der verschiedenen Lade- und Speicherinstruktionen bei der Implementierung mit AVX-Intrinsics hat einen Einfluss auf die Ausführungszeit des Programms. Die Nutzung von ausgerichteten Lade- und Speicherinstruktionen führt dabei vor allem bei den ersten AVX-fähigen Prozessoren bei speicherintensiven Anwendungen zu einer Reduktion von Ausführungszeit und Energieverbrauch. Für die Speicheroperationen mit non-temporal Flag (streaming stores) konnte keine Reduktion von Ausführungszeit oder Energieverbrauch nachgewiesen werden. Stattdessen steigen die beiden Messwerte bei der Ausführung von speicherintensiven Anwendungen sogar an.

Die Nutzung des höchsten verfügbaren AVX-Instruktionssatzes hat die geringste Ausführungszeit und den geringsten Energieverbrauch gezeigt. Zum Einen bietet dieser eine größere Auswahl an AVX-Instruktionen an, wodurch beispielsweise gather Instruktionen genutzt werden können. Zum Anderen bietet die größere Breite und höhere Anzahl der Vektorregister bei der Nutzung von AVX512-Instruktionen einen zusätzlichen Vorteil gegenüber den AVX2-Instruktionen.

Literatur

[1] D. Aberdeen und J. Baxter. „Emmerald: A Fast Matrix–Matrix Multiply Using Intel's SSE Instructions“. In: *Concurrency and Computation: Practice and Experience* 13.2 (2001), S. 103–119. ISSN: 1532-0634. DOI: 10.1002/cpe.549.

[2] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek und S. Tomov. „Numerical Linear Algebra on Emerging Architectures: The PLASMA and MAGMA Projects“. In: *Journal of Physics: Conference Series* 180 (Juli 2009), S. 012037. ISSN: 1742-6596. DOI: 10.1088/1742-6596/180/1/012037.

[3] R. Allen und K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. 1st ed. San Francisco: Morgan Kaufmann Publishers, 2001. 790 S. ISBN: 978-1-55860-286-1.

[4] F. Almeida, J. Arteaga, V. Blanco und A. Cabrera. „Energy Measurement Tools for Ultrascale Computing: A Survey“. In: *Supercomputing Frontiers and Innovations* 2.2 (24. Aug. 2015), S. 64–76. ISSN: 2313-8734. DOI: 10.14529/jsfi150204.

[5] H. Amiri und A. Shahbahrami. „SIMD Programming Using Intel Vector Extensions“. In: *Journal of Parallel and Distributed Computing* 135 (1. Jan. 2020), S. 83–100. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2019.09.012.

[6] J. R. Ball, R. C. Bollinger, T. A. Jeeves, R. C. McReynolds und D. H. Shaffer. „On the Use of the SOLOMON Parallel-Processing Computer“. In: *Proceedings of the December 4-6, 1962, Fall Joint Computer Conference*. AFIPS '62 (Fall). New York, NY, USA: Association for Computing Machinery, 4. Dez. 1962, S. 137–146. ISBN: 978-1-4503-7879-6. DOI: 10.1145/1461518.1461532.

[7] C. Bekas und A. Curioni. „A New Energy Aware Performance Metric“. In: *Computer Science - Research and Development* 25.3-4 (1. Sep. 2010), S. 187–195. ISSN: 1865-2034, 1865-2042. DOI: 10.1007/s00450-010-0119-z.

[8] Å. Björck. „Solving Linear Least Squares Problems by Gram-Schmidt Orthogonalization“. In: *BIT Numerical Mathematics* 7.1 (1. März 1967), S. 1–21. ISSN: 1572-9125. DOI: 10.1007/BF01934122.

[9] BLAST Forum. *Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard*. Knoxville, TN: University of Tennessee, Aug. 2001. URL: http://www.netlib.org/blas/blast-forum/blas-report.pdf.

[10] J. Brenner, J. Keller und C. Kessler. „Executing PRAM Programs on GPUs“. In: *Procedia Computer Science* 9 (2012), S. 1799–1806. ISSN: 18770509. DOI: 10.1016/j.procs.2012.04.198.

[11] D. Buono, M. Danelutto, T. D. Matteis, G. Mencagli und M. Torquati. „A Lightweight Run-Time Support for Fast Dense Linear Algebra on Multi-Core“. In: Software Engineering / 811: Parallel and Distributed Computing and Networks / 816: Artificial Intelligence and Applications. ACTA Press, 18. März 2014. DOI: 10.2316/P.2014.811-029.

[12] J. Carretero, S. Distefano, D. Petcu, D. Pop, T. Rauber, G. Rünger und D. E. Singh. „Energy-Efficient Algorithms for Ultrascale Systems“. In: *Supercomputing frontiers and innovations* 2.2 (24. Aug. 2015), S. 77–104. ISSN: 2313-8734. DOI: 10.14529/jsfi150205.

[13] J. M. Cebrian, M. Jahre und L. Natvig. „ParVec: Vectorizing the PARSEC Benchmark Suite“. In: *Computing* 97.11 (1. Nov. 2015), S. 1077–1100. ISSN: 0010-485X, 1436-5057. DOI: 10.1007/s00607-015-0444-y.

[14] J. M. Cebrián, L. Natvig und J. C. Meyer. „Performance and Energy Impact of Parallelization and Vectorization Techniques in Modern Microprocessors“. In: *Computing* 96.12 (1. Dez. 2014), S. 1179–1193. ISSN: 0010-485X, 1436-5057. DOI: 10.1007/s00607-013-0366-5.

[15] C. Chen, J. Fang, T. Tang und C. Yang. „LU Factorization on Heterogeneous Systems: An Energy-Efficient Approach towards High Performance“. In: *Computing* 99.8 (1. Aug. 2017), S. 791–811. ISSN: 1436-5057. DOI: 10.1007/s00607-016-0537-2.

[16] R. Clint Whaley, A. Petitet und J. J. Dongarra. „Automated Empirical Optimizations of Software and the ATLAS Project“. In: *Parallel Computing*. New Trends in High Performance Computing 27.1 (1. Jan. 2001), S. 3–35. ISSN: 0167-8191. DOI: 10.1016/S0167-8191(00)00087-9.

[17] Cray Research Inc. *The Cray-1 Computer System*. 2240008 B. 1977, S. 14. URL: http://archive.computerhistory.org/resources/text/Cray/Cray.Cray1.1977.102638650.pdf.

[18] K. Czechowski, V. W. Lee, E. Grochowski, R. Ronen, R. Singhal, R. Vuduc und P. Dubey. „Improving the Energy Efficiency of Big Cores“. In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, S. 493–504. ISBN: 978-1-4799-4394-4. DOI: 10.1145/2678373.2665743.

[19] J. Demmel. „LAPACK: A Portable Linear Algebra Library for Supercomputers“. In: *IEEE Control Systems Society Workshop on Computer-Aided Control System Design*. IEEE Control Systems Society Workshop on Computer-Aided Control System Design. Dez. 1989, S. 1–7. DOI: 10.1109/CACSD.1989.69824.

[20] J. J. Dongarra, F. G. Gustavson und A. Karp. „Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine“. In: *SIAM Review* 26.1 (1. Jan. 1984), S. 91–112. ISSN: 0036-1445. DOI: 10.1137/1026003.

[21] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam und D. Burger. „Dark Silicon and the End of Multicore Scaling“. In: *Proceeding of the 38th Annual International Symposium on Computer Architecture - ISCA '11*. Proceeding of the 38th Annual International Symposium. San Jose, California, USA: ACM Press, 2011, S. 365. ISBN: 978-1-4503-0472-6. DOI: 10.1145/2000064.2000108.

[22] M. Flynn. „Very High-Speed Computing Systems“. In: *Proceedings of the IEEE* 54.12 (Dez. 1966), S. 1901–1909. ISSN: 1558-2256. DOI: 10.1109/PROC.1966.5273.

[23] J. G. F. Francis. „The QR Transformation—Part 2“. In: *The Computer Journal* 4.4 (1. Jan. 1962), S. 332–345. ISSN: 0010-4620. DOI: 10.1093/comjnl/4.4.332.

[24] G. H. Golub und C. F. Van Loan. *Matrix Computations*. 4. ed. Baltimore, Md.: Johns Hopkins University Pr., 2013. ISBN: 978-1-4214-0794-4.

[25] R. Gonzalez und M. Horowitz. „Energy Dissipation in General Purpose Microprocessors“. In: *IEEE Journal of Solid-State Circuits* 31.9 (Sep. 1996), S. 1277–1284. ISSN: 0018-9200. DOI: 10.1109/4.535411.

[26] *Green500 | TOP500 Supercomputer Sites*. URL: https://www.top500.org/green500/.

[27] M. Hähnel, B. Döbel, M. Völp und H. Härtig. „Measuring Energy Consumption for Short Code Paths Using RAPL“. In: *SIGMETRICS Perform. Eval. Rev.* 40.3 (Jan. 2012), S. 13–17. ISSN: 0163-5999. DOI: 10.1145/2425248.2425252.

[28] S. A. Hassan, M. M. Mahmoud, A. Hemeida und M. A. Saber. „Effective Implementation of Matrix–Vector Multiplication on Intel's AVX Multicore Processor“. In: *Computer Languages, Systems & Structures* 51 (1. Jan. 2018), S. 158–175. ISSN: 1477-8424. DOI: 10.1016/j.cl.2017.06.003.

[29] J. L. Hennessy, D. A. Patterson und K. Asanović. *Computer Architecture: A Quantitative Approach*. 5th ed. Waltham, MA: Morgan Kaufmann/Elsevier, 2012. 493 S. ISBN: 978-0-12-383872-8.

[30] W. Hoffmann. „Iterative Algorithms for Gram-Schmidt Orthogonalization“. In: *Computing* 41.4 (1. Dez. 1989), S. 335–348. ISSN: 0010-485X, 1436-5057. DOI: 10.1007/BF02241222.

[31] J. Hofmann, J. Treibig, G. Hager und G. Wellein. „Comparing the Performance of Different X86 SIMD Instruction Sets for a Medical Imaging Application on Modern Multi- and Manycore Chips“. In: *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*. WPMVP '14. New York, NY, USA: ACM, 2014, S. 57–64. ISBN: 978-1-4503-2653-7. DOI: 10.1145/2568058.2568068.

[32] C.-H. Hsu, W.-c. Feng und J. Archuleta. „Towards Efficient Supercomputing: A Quest for the Right Metric“. In: *19th IEEE International Parallel and Distributed Processing Symposium*. 19th IEEE International Parallel and Distributed Processing Symposium. Denver, CO, USA: IEEE, 2005. ISBN: 978-0-7695-2312-5. DOI: 10.1109/IPDPS.2005.440.

[33] M. E. A. Ibrahim, M. Rupp und H. A. H. Fahmy. „Code Transformations and SIMD Impact on Embedded Software Energy/Power Consumption“. In: *2009 International Conference on Computer Engineering Systems*. 2009 International Conference on Computer Engineering Systems. Dez. 2009, S. 27–32. DOI: 10.1109/ICCES.2009.5383317.

[34] *Intel Cilk Plus Is Being Deprecated*. URL: https://software.intel.com/en-us/forums/intel-cilk-plus/topic/745556.

[35] Intel Corporation. *The Compute Architecture of Intel Processor Graphics Gen9*. Aug. 2015. URL: https://software.intel.com/content/dam/develop/external/us/en/documents/the-compute-architecture-of-intel-processor-graphics-gen9-v1d0-166010.pdf.

[36] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C,2D, 3A, 3B, 3C, 3D and 4*. Intel Corporation, Mai 2018. URL: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[37] Intel Corporation. *Thermal Design Power (TDP) in Intel® Processors*. 000055611. 21. Nov. 2019. URL: https://www.intel.com/content/www/us/en/support/articles/000055611/processors.html.

[38] Intel Corporation. *Intel® C++ Compiler 19.1 Developer Guide and Reference*. Apr. 2021. URL: https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top.html.

[39] Intel Corporation. *Accelerate Fast Math with Intel® oneAPI Math Kernel Library*. URL: https://www.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl.html.

[40] Intel Corporation. *Intel Product Specifications*. URL: https://ark.intel.com/content/www/us/en/ark.html.

[41] Intel Corporation. *Intel® Intrinsics Guide*. URL: https://software.intel.com/sites/landingpage/IntrinsicsGuide/#.

[42] F. Irigoin. „Tiling“. In: *Encyclopedia of Parallel Computing*. Hrsg. von D. Padua. Boston, MA: Springer US, 2011, S. 2040–2049. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_511.

[43] T. Jakobs, M. Hofmann und G. Rünger. „Reducing the Power Consumption of Matrix Multiplications by Vectorization“. In: *2016 IEEE Intl Conference on Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC) and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering (DCABES)*. 2016 IEEE Intl Conference on Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC) and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering (DCABES). Aug. 2016, S. 213–220. DOI: 10.1109/CSE-EUC-DCABES.2016.187.

[44] T. Jakobs, B. Naumann und G. Rünger. „Performance and Energy Consumption of the SIMD Gram–Schmidt Process for Vector Orthogonalization“. In: *The Journal of Supercomputing* (5. Apr. 2019). ISSN: 1573-0484. DOI: 10.1007/s11227-019-02839-0.

[45] T. Jakobs, L. Reinhardt und G. Rünger. „Performance and Energy Consumption of a Gram–Schmidt Process for Vector Orthogonalization on a Processor Integrated GPU“. In: *Sustainable Computing: Informatics and Systems* 29 (1. März 2021), S. 100456. ISSN: 2210-5379. DOI: 10.1016/j.suscom.2020.100456.

[46] T. Jakobs und G. Rünger. „Examining Energy Efficiency of Vectorization Techniques Using a Gaussian Elimination“. In: *2018 International Conference on High Performance Computing Simulation (HPCS)*. 2018 International Conference on High Performance Computing Simulation (HPCS). IEEE, Juli 2018, S. 268–275. ISBN: 978-1-5386-7879-4. DOI: 10.1109/HPCS.2018.00054.

[47] T. Jakobs und G. Rünger. „On the Energy Consumption of Load/Store AVX Instructions“. In: *Annals of Computer Science and Information Systems*. Proceedings of the 2018 Federated Conference on Computer Science and Information Systems. Bd. 15. 2018, S. 319–327. ISBN: 978-83-949419-5-6. DOI: 10.15439/2018F28.

[48] Khronos Group. *OpenCL*. URL: https://www.khronos.org/opencl/.

[49] C. Kim, N. Satish, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar und P. Dubey. *Closing the Ninja Performance Gap through Traditional Programming and Compiler Technology*. Intel® Corporation, Okt. 2013. URL: https://software.intel.com/sites/default/files/article/478267/intel-labs-closing-ninja-gap-paper.pdf.

[50] J. Kurzak, W. Alvaro und J. Dongarra. „Optimizing Matrix Multiplication for a Short-Vector SIMD Architecture – CELL Processor“. In: *Parallel Computing*. Revolutionary Technologies for Acceleration of Emerging Petascale Applications 35.3 (1. März 2009), S. 138–150. ISSN: 0167-8191. DOI: 10.1016/j.parco.2008.12.010.

[51] R. Lee. „Subword Parallelism with MAX-2“. In: *IEEE Micro* 16.4 (Aug. 1996), S. 51–59. ISSN: 1937-4143. DOI: 10.1109/40.526925.

[52] J. Lenhardt, W. Schiffmann und J. Keller. „Interplay of Power Management at Core and Server Level“. In: *International Journal of Computer and Information Engineering* 10.1 (6. Jan. 2016), S. 136–141. DOI: doi.org/10.5281/zenodo.1338686.

[53] S. J. Leon, Å. Björck und W. Gander. „Gram-Schmidt Orthogonalization: 100 Years and More“. In: *Numerical Linear Algebra with Applications* 20.3 (2013), S. 492–532. ISSN: 1099-1506. DOI: 10.1002/nla.1839.

[54] H. Lien, L. Natvig, A. A. Hasib und J. C. Meyer. „Case Studies of Multi-Core Energy Efficiency in Task Based Programs“. In: *ICT as Key Technology against Global Warming*. International Conference on Information and Communication on Technology. Springer, Berlin, Heidelberg, 6. Sep. 2012, S. 44–54. DOI: 10.1007/978-3-642-32606-6_4.

[55] E. Lindholm, J. Nickolls, S. Oberman und J. Montrym. „NVIDIA Tesla: A Unified Graphics and Computing Architecture“. In: *IEEE Micro* 28.2 (März 2008), S. 39–55. ISSN: 1937-4143. DOI: 10.1109/MM.2008.31.

[56] C. Lomont. *Introduction to Intel Advanced Vector Extensions*. Introduction to Intel advanced vector extensions. Mai 2011. URL: https://software.intel.com/sites/default/files/m/d/4/1/d/8/Intro_to_Intel_AVX.pdf.

[57] M. Lorenz, L. Wehmeyer und T. Dräger. „Energy Aware Compilation for DSPs with SIMD Instructions“. In: *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems*. LCTES/SCOPES '02. New York, NY, USA: ACM, 2002, S. 94–101. ISBN: 978-1-58113-527-5. DOI: 10.1145/513829.513847.

[58] R. Meghana. *An Overview of the 6th Generation Intel® Core™ Processor (Code-Named Skylake)*. 23. März 2016. URL: https://software.intel.com/en-us/articles/an-overview-of-the-6th-generation-intel-core-processor-code-named-skylake.

[59] D. A. Orbits und D. A. Calahan. *Data Flow Considerations in Implementing A Full Matrix Solver with Backing Store on the Cray-1.* MICHIGAN UNIV ANN ARBOR DEPT OF ELECTRICAL AND COMPUTER ENGINEERING, 1. Sep. 1976. URL: https://apps.dtic.mil/sti/citations/ADA033378.

[60] K. Pingali. „Locality of Reference and Parallel Processing“. In: *Encyclopedia of Parallel Computing*. Hrsg. von D. Padua. Boston, MA: Springer US, 2011, S. 1051–1056. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_206.

[61] A. Pohl, B. Cosenza, M. A. Mesa, C. C. Chi und B. Juurlink. „An Evaluation of Current SIMD Programming Models for C++“. In: *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing*. WPMVP '16. Barcelona, Spain: Association for Computing Machinery, 13. März 2016, S. 1–8. ISBN: 978-1-4503-4060-1. DOI: 10.1145/2870650.2870653.

[62] G. Rapaport, A. Zaks und Y. Ben-Asher. „Streamlining Whole Function Vectorization in C Using Higher Order Vector Semantics“. In: *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. 2015 IEEE International Parallel and Distributed Processing Symposium Workshop. Mai 2015, S. 718–727. DOI: 10.1109/IPDPSW.2015.37.

[63] T. Rauber und G. Rünger. *Parallele Programmierung, 3. Auflage*. Springer Verlag, eXamen.press, 2012. ISBN: 978-3-642-13604-7.

[64] T. Rauber und G. Rünger. „Optimal Data Distributions for LU Decomposition“. In: *EURO-PAR '95 Parallel Processing*. European Conference on Parallel Processing. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 29. Aug. 1995, S. 391–402. ISBN: 978-3-540-44769-6. DOI: 10.1007/BFb0020480.

[65] T. Rauber und G. Rünger. „Modeling the Energy Consumption for Concurrent Executions of Parallel Tasks“. In: *Proceedings of the 14th Communications and Networking Symposium*. CNS '11. San Diego, CA, USA: Society for Computer Simulation International, 2011, S. 11–18. URL: http://dl.acm.org/citation.cfm?id=2048416.2048418.

[66] T. Rauber, G. Rünger, M. Schwind, H. Xu und S. Melzner. „Energy Measurement, Modeling, and Prediction for Processors with Frequency Scaling“. In: *The Journal of Supercomputing* 70.3 (2014), S. 1451–1476. ISSN: 0920-8542.

[67] T. Rauber, G. Rünger und M. Stachowski. „Performance and Energy Metrics for Multi-Threaded Applications on DVFS Processors“. In: *Sustainable Computing: Informatics and Systems* 17 (1. März 2018), S. 55–68. ISSN: 2210-5379. DOI: 10.1016/j.suscom.2017.10.015.

[68] S. Rivoire, M. A. Shah, P. Ranganathan, K. Kozyrakis und J. Meza. „Models and Metrics to Enable Energy-Efficiency Optimizations“. In: *Computer* 40.12 (Dez. 2007), S. 39–48. ISSN: 0018-9162. DOI: 10.1109/MC.2007.436.

[69] S. I. Roberts, S. A. Wright, S. A. Fahmy und S. A. Jarvis. „Metrics for Energy-Aware Software Optimisation“. In: *High Performance Computing*. International Supercomputing Conference. Lecture Notes in Computer Science. Springer, Cham, 18. Juni 2017, S. 413–430. ISBN: 978-3-319-58667-0. DOI: 10.1007/978-3-319-58667-0_22.

[70] G. Rünger und M. Schwind. „Comparison of Different Parallel Modified Gram-Schmidt Algorithms“. In: *Euro-Par 2005 Parallel Processing*. Hrsg. von J. C. Cunha und P. D. Medeiros. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, S. 826–836. ISBN: 978-3-540-31925-2. DOI: 10.1109/IPDPS.2008.4536474.

[71] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar und P. Dubey. „Can Traditional Programming Bridge the Ninja Performance Gap for Parallel Computing Applications?“ In: *Proceedings of the 39th Annual International Symposium on Computer Architecture*. ISCA '12.

Washington, DC, USA: IEEE Computer Society, 2012, S. 440–451. ISBN: 978-1-4503-1642-2. DOI: 10.1145/2366231.2337210.

[72] Y. Sazeides, R. Kumar, D. M. Tullsen und T. Constantinou. „The Danger of Interval-Based Power Efficiency Metrics: When Worst Is Best“. In: *IEEE Computer Architecture Letters* 4.1 (Jan. 2005), S. 1–1. ISSN: 1556-6056. DOI: 10.1109/L-CA.2005.2.

[73] G. L. Steele und W. D. Hillis. „Connection Machine Lisp: Fine-Grained Parallel Symbolic Processing“. In: *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*. LFP '86. Cambridge, Massachusetts, USA: Association for Computing Machinery, 8. Aug. 1986, S. 279–297. ISBN: 978-0-89791-200-6. DOI: 10.1145/319838.319870.

[74] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico und P. Walker. „The ARM Scalable Vector Extension“. In: *IEEE Micro* 37.2 (März 2017), S. 26–39. DOI: 10.1109/MM.2017.35.

[75] V. Strassen. „Gaussian Elimination Is Not Optimal“. In: *Numerische Mathematik* 13.4 (1. Aug. 1969), S. 354–356. ISSN: 0945-3245. DOI: 10.1007/BF02165411.

[76] A. Takahashi, M. Soliman und S. Sedukhin. „Parallel LU-Decomposition on Pentium Streaming SIMD Extensions“. In: *High Performance Computing*. Hrsg. von A. Veidenbaum, K. Joe, H. Amano und H. Aiso. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2003, S. 423–430. ISBN: 978-3-540-39707-6. DOI: 10.1007/978-3-540-39707-6_37.

[77] L. Tan, S. Kothapalli, L. Chen, O. Hussaini, R. Bissiri und Z. Chen. „A Survey of Power and Energy Efficient Techniques for High Performance Numerical Linear Algebra Operations“. In: *Parallel Computing* 40.10 (1. Dez. 2014), S. 559–573. ISSN: 0167-8191. DOI: 10.1016/j.parco.2014.09.001.

[78] Thinking Machines Corporation. *C* Programming Guide*. Mai 1993.

[79] P. R. Turner. *Gauss Elimination: Workhorse of Linear Algebra.* ADA313547. NAVAL AIR WARFARE CENTER AIRCRAFT DIV PATUXENT RIVER MD, 5. Aug. 1995. URL: https://apps.dtic.mil/sti/citations/ADA313547.

[80] S. Winograd. „A New Algorithm for Inner Product“. In: *IEEE Transactions on Computers* C-17.7 (Juli 1968), S. 693–694. ISSN: 1557-9956. DOI: 10.1109/TC.1968.227420.

[81] J. Zhuo und C. Chakrabarti. „Energy-Efficient Dynamic Task Scheduling Algorithms for DVS Systems“. In: *ACM Trans. Embed. Comput. Syst.* 7.2 (Jan. 2008), 17:1–17:25. ISSN: 1539-9087. DOI: 10.1145/1331331.1331341.

A

Anhang: Grundlagen zur AVX-Programmierung

A.1. AVX-Datentypen

Durch AVX werden Vektordatentypen zur Nutzung mit intrinsischen Funktionen bereit gestellt (siehe auch [36] und [41]). Die maximale Größe der verfügbaren Datentypen richtet sich dabei nach der verfügbaren Registergröße der AVX-Erweiterung, für AVX und AVX2 Prozessoren können 128-Bit oder 256-Bit Datentypen genutzt werden, mit der AVX512-Erweiterung sind zusätzlich 512-Bit Datentypen verfügbar. Die Anzahl der skalaren Daten, die in dem jeweiligen Vektordatentyp genutzt werden können, richtet sich dabei nach der Größe des Vektordatentyps und der Größe des skalaren Datentyps. Die folgenden Datentypen und deren Entsprechung werden zur Nutzung mit intrinsischen Funktionen bereit gestellt:

Datentyp	Größe	Entsprechung
__m128	128-Bit	vier 32-Bit `float` Werte
__m128d	128-Bit	zwei 64-Bit `double` Werte
__m128i	128-Bit	16 8-Bit, acht 16-Bit, vier 32-Bit oder zwei 64-Bit Ganzzahlwerte
__m256	256-Bit	acht 32-Bit `float` Werte
__m256d	256-Bit	vier 64-Bit `double` Werte
__m256i	256-Bit	32 8-Bit, 16 16-Bit, acht 32-Bit oder vier 64-Bit Ganzzahlwerte
__m512	512-Bit	16 32-Bit `float` Werte
__m512d	512-Bit	acht 64-Bit `double` Werte
__m512i	512-Bit	64 8-Bit, 32 16-Bit, 16 32-Bit oder acht 64-Bit Ganzzahlwerte

Tabelle A.1.: Vektordatentypen in C/C++ für die Nutzung mit intrinsischen Funktionen und deren Entsprechung gegenüber skalaren Datentypen.

A.2. Alignment: Ausgerichtete Speicherzugriffe mit Vektordatentypen

Einige Lade- und Speicherinstruktionen bei der Vektorisierung setzten einen ausgerichteten Speicherzugriff voraus. Beispiele für solche Instruktionen sind die load oder store Instruktionen. Die Besonderheit von ausgerichteten (*engl. aligned*) Speicherpositionen besteht in den Zugriffsmöglichkeiten des Prozessors auf die Daten im Cache: Bei einem ausgerichteten Speicherzugriff befinden sich alle benötigten Daten immer in der selben Cachezeile. Auf diese Weise können die Daten für ein Vektorregister durch den Prozessor mit einer Instruktion geladen/gespeichert werden. Befinden sich die Daten nicht in der selben Cachezeile, muss der Speicherzugriff durch den Prozessor in zwei Speicherzugriffe auf unterschiedliche Cachezeilen aufgeteilt werden, wodurch mehrere Zugriffsoperationen durchgeführt werden. Abbildung A.1 veranschaulicht die Datenzugriffe auf die Cachezeilen bei ausgerichteten und unausgerichteten Speicherzugriffen.

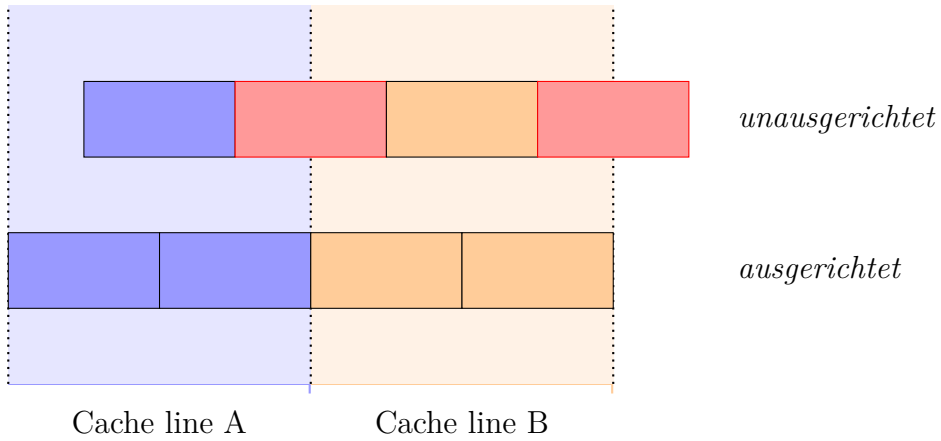

Abbildung A.1.: Darstellung der zugegriffenen Speicherbereiche beim Zugriff mit Vektordatentypen auf ausgerichtete und unausgerichtete Speicherbereiche. Bei unausgerichteten Speicherbereichen fallen unter Umständen zusätzliche Ladeoperationen im Prozessor an, um ein Vektorregister zu füllen. Darstellung nach [44].

Ausgerichtete Speicherbereiche können mit der Allokationsfunktion _mm_maloc allokiert werden, sodass die erste Position des allokierten Speicherbereichs das angegebene Alignment erfüllt. Weitere ausgerichtete Speicherstellen können durch die Größe der genutzten Datentypen und den Abstand zur ersten Speicherstelle berechnet werden. Bei der Vektorisierung kann diese Berechnung oft dadurch vereinfacht werden, dass die genutzten Vektordatentypen die Größe eines ausgerichteten Blockes haben. Für AVX wird der Begriff ausgerichtet genutzt, wenn eine Speicheradresse durch 32 Byte (256 Bit) teilbar ist [36, Vol. 1, Kap. 14.9], bei AVX512 ist das nötige Alignment 64 Byte (512 Bit).

A.3. Anpassung des Iterationsraums auf die Nutzung von Vektoren

Bei der Nutzung fester Vektorgrößen, die durch die AVX-Datentypen vorgegeben werden, muss der Iterationsraum einer Schleife mitunter auf diese Anzahl angepasst werden. Dies tritt auf, wenn ein ausgerichteter Datenzugriff zu Beginn des Iterationsraums hergestellt wird, oder wenn am Ende des Iterationsraums nicht ausreichend Elemente für einen Vektor verbleiben.

Loop Splitting zum Auslösen von Iterationen

Mit Hilfe von Loop Splitting [3] werden einzelne oder mehrere Iterationen des Iterationsraums aus einer Schleife heraus genommen und vor bzw. nach der Schleife gesondert berechnet. Das Entfernen der ersten Iterationen wird auch als Loop Peeling bezeichnet [3] und ist somit ein Spezialfall des Loop Splitting.

Beim Loop Splitting werden die Iterationen, die aus der Schleife entfernt werden sollen, vor der Original-Schleife ausgeführt und der Iterationsraum der Original-Schleife entsprechend gekürzt. Analog können die letzten Iterationen einer Schleife mittels Loop Splitting gesondert behandelt werden. Eine Schleife mit der Iterationsvariable `i`, mit $0 \leq i < N$, kann nach dem abtrennen der jeweils ersten und letzten 2 Iterationen mittels Loop Splitting wie folgt dargestellt werden:

```
for(i = 0; i < 2; i++) //Loop Splitting der ersten 2 Iterationen
    //Schleifenkörper
for(i = 2; i < N - 2; i++) //Originalschleife mit angepassten Schleifengrenzen
    //Schleifenkörper
for(i = N - 3; i < N; i++) //Loop Splitting der letzten 2 Iterationen
    //Schleifenkörper
```

Bei der Vektorisierung wird das Loop Splitting der ersten Iterationen einer Schleife zur Herstellung eines ausgerichteten Speicherzugriffs genutzt. Die Anzahl der abgetrennten Iterationen richtet sich somit nach dem gewünschten Startindex eines im Schleifenkörper genutzten Arrays. Das Loop Splitting der letzten Iterationen wird bei der Vektorisierung eingesetzt, um die Anzahl der Iterationen der Originalschleife auf eine Iterationsanzahl zu kürzen, die durch die Anzahl der Elemente in einem genutzten Vektordatentyp teilbar ist.

Vektorisierung nach Loop Splitting

Durch die Anwendung des Loop Splitting lässt sich die Originalschleife damit besser Vektorisieren, die abgetrennten Schleifen benötigen jedoch eine gesonderte Behandlung.

Die Anzahl der Iterationen der einzelnen Schleifen des Loop Splitting ist bei der Anwendung in der Vektorisierung kleiner als die Anzahl der Elemente eines Vektordatentyps. Die so erstellten Schleifen können dabei entweder seriell (nicht vektorisiert)

ausgeführt werden, oder mit Hilfe von speziellen Instruktionen vektorisiert werden, die nur Teile der Vektordatentypen nutzen. Solche Instruktionen werden als maskierte Instruktionen [3] bezeichnet, da sie als zusätzlichen Parameter eine Maske erhalten, in der die zu nutzenden Elemente entsprechend markiert sind.

AVX bietet maskierte Instruktionen zum Laden und Speichern von Vektordatentypen an. Beim Laden von Daten in einen Vektordatentypen werden mit der maskload Instruktion nur solche Elemente geladen, deren Markierung in der Maske ungleich 0 ist. Die nicht geladenen Elemente werden durch die Ladeinstruktion auf den Wert 0 gesetzt. Vergleichbar dazu bietet die maskstore Instruktion die Möglichkeit nur solche Elemente zu Speichern, deren Position in der Maske mit einem Wert ungleich 0 gekennzeichnet ist. Die unmarkierten Elemente des Vektors werden dabei nicht gespeichert/geschrieben.

A.4. Vektorisierte Reduktion

Bei der vektorisierten Reduktion wird eine Summe aus den einzelnen Elementen des selben Vektorregisters gebildet. Algorithmus A.1 zeigt den Programmcode der vektorisierten Reduktion für acht `float` Werte. Die Reduktion in Alg. A.1 wird in drei Schritten durchgeführt:

1. In Zeile 8 werden die zwei 128-Bit Blöcke des Vektorregister miteinander getauscht. Im Anschluss wird in Zeile 10 der getauschte Vektor und der Originalvektor miteinander addiert. Somit enthält jeder 128-Bit Block die Teilsumme aus beiden 128-Bit Blöcken.

2. In Zeile 17 werden die Elemente des Ergebnisvektors aus Schritt 1 innerhalb des 128-Bit Blockes getauscht, sodass der neue Vektor die folgende Reihenfolge der Ursprungselemente enthält: 2,3,0,1. In Zeile 19 wird dieser Vektor auf den Ergebnisvektor aus Schritt 1 addiert.

3. In Zeile 26 werden die Elemente des Ergebnisvektors aus Schritt 2 so getauscht, dass die Elemente des 128-Bit Blockes die Reihenfolge 1,0,3,2 darstellen und in Zeile 28 mit dem Ergebnisvektor von Schritt 2 addiert.

Durch die Umsortierung und Addition der Elemente wird die Summe aller Elemente des Vektors in Element 0 berechnet. Dieses Element wird in Zeile 32 in eine skalare Variable gespeichert.

Die Umsortierinstruktionen für Vektoren haben im Vergleich zu arithmetischen Instruktionen höhere Ausführungskosten [41]. Aus diesem Grund ist die Reduktion mit den drei enthaltenen Umsortierinstruktionen eine sehr kostenintensive Funktion.

```
1  float addreduce(const __m256 vec)
2  {
3    float ret = 0.0;
4    __m256 _t1;
5    __m256 _t2;
6
7    // Vertausche die 128-Bit Blöcke:
8    _t1 = _mm256_permute2f128_ps(vec, vec, 1);
9    // Addiere die Register
10   _t1 = _mm256_add_ps(vec, _t1);
11   // [0] - [3] enthalten die Teilsummen [0]+[4], [1]+[5], [2]+[6], [3]+[7]
12
13 // Unterbrechung für 4 skalare Additionen
14
15   // Vertausche 64-Bit Blöcke:
16   // neue Reihenfolge: [2],[3],[0],[1]
17   _t2 = _mm256_permute_ps(_t1, 0x4E);
18   // Addiere die Register
19   _t1 = _mm256_add_ps(_t1, _t2);
20   // [0] - [1] enthalten die Teilsummen [0]+[2]+[4]+[6], [1]+[3]+[5]+[7]
21
22 // Unterbrechung für 2 skalare Additionen
23
24   // Vertausche 32-Bit Blöcke:
25   // neue Reihenfolge: [1],[0],[3],[2]
26   _t2 = _mm256_permute_ps(_t1, 0xB1);
27   // Addiere die Register
28   _t1 = _mm256_add_ps(_t1, _t2);
29   // [0] enthält die Summe von [0] - [7]
30
31   // Abspeichern des Ergebnislements
32   ret = ((float*)&_t1)[0];
33   return ret;
34 }
```

Algorithmus A.1: 256-Bit AVX Implementierung einer Reduktionsoperation mit der die einzelnen `float` Werte einer Vektorvariablen aufaddiert werden. Die vektorisierte Ausführung kann an den gezeigten Stellen unterbrochen und die restliche Reduktion durch eine Schleife ersetzt werden.

A.5. Liste der verwendeten Intrinsics in dieser Arbeit

Intrinsic	Ausgeführte Operation
a = _mm<size>_loadu_ps(*mem)	Lädt $\frac{size}{32}$ `float` Werte beginnend von einer unausgerichteten Speicherstelle mem in eine Vektorvariable a.

Tabelle A.2: Übersicht und Erklärung der AVX-Intrinsics, die in dieser Arbeit verwendet werden. Beschreibung angepasst aus [41]. **Fortsetzung der Tabelle auf der nächsten Seite.**

Intrinsic	Ausgeführte Operation
a = _mm<size>_load_ps(∗mem)	Lädt $\frac{size}{32}$ `float` Werte beginnend von einer ausgerichteten Speicherstelle mem in eine Vektorvariable a.
a = _mm256_loadu_si256(∗mem)	Lädt 256 Bit in Ganzzahldatentypen von einer unausgerichteten Speicherstelle mem in eine Vektorvariable a. Der Ganzzahldatentyp ist hierbei nicht festgelegt.
a = _mm<size>_setr_ps(f0,f1,...)	Lädt die $\frac{size}{32}$ explizit angegebenen `float` Werte (übergeben als Kopie der Werte f0 , f1 ,...) in eine Vektorvariable a. Dieses Intrinsic entspricht keiner einzelnen AVX-Instruktion sondern einer Sequenz an Instruktionen.
a = _mm<size>_setr_epi32(f0,f1,...)	Lädt die $\frac{size}{32}$ explizit angegebenen Integer-Werte (übergeben als Kopie der Werte f0 , f1 ,...) in eine Vektorvariable a. Dieses Intrinsic entspricht keiner einzelnen AVX-Instruktion sondern einer Sequenz an Instruktionen.
a = _mm256_maskload_ps(∗mem,mask)	Lädt eine Auswahl (markiert durch nicht Null-Bits in mask) von `float` Werten von einer Speicherposition mem in eine Vektorvariable a. Ungeladene Vektorelemente werden mit 0 initialisiert.
a = _mm<size>_i32gather_ps(∗mem, ind, scale)	Lädt $\frac{size}{32}$ `float` Werte von einer angegebenen Position nach der Speicherstelle mem. Die Position der geladenen Speicherelemente wird durch die in der Vektorvariablen ind angegeben Indizes und die Datengröße in Bytes in scale bestimmt.
a = _mm256_broadcast_ss(∗mem)	Lädt einen einzelnen `float` Wert (von Speicherstelle mem) in alle 8 Elemente einer 256-Bit Vektorvariablen a.
a = _mm512_set1_ps(f)	Schreibt einen `float` Wert f an alle 16 Stellen einer 512-Bit Vektorvariablen a. Dieses Intrinsic entspricht keiner einzelnen AVX-Instruktion sondern einer Sequenz an Instruktionen.
a = _mm<size>_set1_epi32(f)	Schreibt einen Integer-Wert f an alle $\frac{size}{32}$ Stellen einer Vektorvariablen a. Dieses Intrinsic entspricht keiner einzelnen AVX-Instruktion sondern einer Sequenz an Instruktionen.

Tabelle A.2: Übersicht und Erklärung der AVX-Intrinsics, die in dieser Arbeit verwendet werden. Beschreibung angepasst aus [41]. **Fortsetzung der Tabelle auf der nächsten Seite.**

Intrinsic	Ausgeführte Operation
a = _mm<size>_setzero_ps()	Setzt alle Elemente einer Vektrovariablen a auf den Wert 0.
_mm<size>_store_ps(∗mem, a)	Speichert die $\frac{size}{32}$ `float` Elemente einer Vektorvariaben a an eine ausgerichtete Speicherstelle beginnend mit mem.
_mm<size>_storeu_ps(∗mem, a)	Speichert die $\frac{size}{32}$ `float` Elemente einer Vektorvariaben a an eine unausgerichtete Speicherstelle beginnend mit mem.
_mm256_maskstore_ps(∗mem,mask,a)	Speichert eine Auswahl (markiert durch nicht Null-Bits in mask) von `float` Werten an eine Speicherstelle beginnend mit mem. Ungespeicherte Elemente (0-Bits) werden nicht verändert.
_mm256_stream_ps(∗mem,a)	Speichert die 8 `float` Elemente einer Vektorvariablen a an eine ausgerichtete Speicherstelle beginnend mit mem. Bei der Speicheroperation wird ein *non-temporal hint* genutzt, der eine Write-Through Operation auslöst.
c = _mm<size>_add_ps(a,b)	Addiert die `float` Elemente von zwei Vektorvariablen a und b jeweils elementweise. Das Ergebnis wird in Vektorvariable c geschrieben.
c = _mm<size>_add_epi32(a,b)	Addiert die Integer-Elemente von zwei Vektorvariablen a und b jeweils elementweise. Das Ergebnis wird in Vektorvariable c geschrieben.
c = _mm256_sub_ps(a,b)	Subtrahiert die 8 `float` Elemente von zwei Vektorvariablen a und b jeweils elementweise mit $a - b$. Das Ergebnis wird in Vektorvariable c geschrieben.
c = _mm<size>_mul_ps(a,b)	Multipliziert die `float` Elemente von zwei Vektorvariablen a und b jeweils elementweise. Das Ergebnis wird in Vektorvariable c geschrieben.
c = _mm<size>_mul_epi32(a,b)	Multipliziert die Integer-Elemente von zwei Vektorvariablen a und b jeweils elementweise. Das Ergebnis wird in Vektorvariable c geschrieben.

Tabelle A.2: Übersicht und Erklärung der AVX-Intrinsics, die in dieser Arbeit verwendet werden. Beschreibung angepasst aus [41]. **Fortsetzung der Tabelle auf der nächsten Seite.**

Intrinsic	Ausgeführte Operation
c = _mm<size>_div_ps(a,b)	Dividiert die `float` Elemente von zwei Vektorvariablen a und b jeweils elementweise mit a/b. Das Ergebnis wird in Vektorvariable c geschrieben.
x = _mm<size>_fmadd_ps(a,b,c)	Berechnet die `float` Elemente von drei Vektorvariablen a, b und c jeweils elementweise nach der Formel $x = (a \cdot b) + c$. Bei der Berechnung wird eine der drei Vektorvariablen überschrieben, der Compiler bestimmt welche dies ist.
x = _mm<size>_fnmadd_ps(a,b,c)	Berechnet die `float` Elemente von drei Vektorvariablen a, b und c jeweils elementweise nach der Formel $x = -(a \cdot b) + c$. Bei der Berechnung wird eine der drei Vektorvariablen überschrieben, der Compiler bestimmt welche dies ist.
f = _mm512_reduce_add_ps(a)	Reduziert die 16 `float` Werte einer 512-Bit Vektorvariablen a durch aufsummieren zu einem einzelnen `float` Wert f. Dieses Intrinsic entspricht keiner einzelnen AVX-Instruktion sondern einer Sequenz an Instruktionen.
c = _mm256_permute2f128_ps(a,b,key)	Setzt die 128-Bit Blöcke einer 256-Bit Vektorvariable c, sodass die Blöcke aus den Vektorvariablen a und b anhand von key ausgewählt werden. Die Reihenfolge wird in den kleinsten 8-Bit von key durch die Reihenfolge der Binärzahlreprasentation von 0 bis 4 angegeben, wobei 0 und 1 für die Blöcke von Vektrovariable a und 2 und 3 für die Blöcke von Vektorvariable b stehen.
b = _mm256_permute_ps(a,key)	Vertauscht die `float` Elemente der Vektorvariablen a innerhalb der 128-Bit Blöcke anhand von key. Die Reihenfolge wird in den kleinsten 8-Bit von key durch die Reihenfolge der Binärzahlreprasentation von 0 bis 4 angegeben. Das Ergebnis wird in Vektorvariable b geschrieben.

Tabelle A.2.: Übersicht und Erklärung der AVX-Intrinsics, die in dieser Arbeit verwendet werden. Beschreibung angepasst aus [41].

B

Anhang zur Matrix-Multiplikation

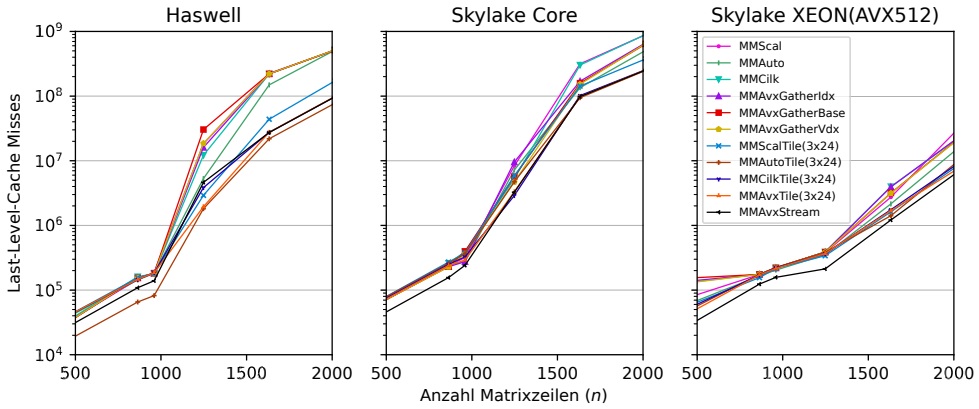

Abbildung B.1.: Anzahl der Last-Level-Cache Misses der Programmvarianten der Matrix-Multiplikation in Abhängigkeit zur Matrixgröße auf der Haswell (3,5GHz), Skylake Core (3,4GHz) und Skylake XEON (2,0GHz) Architektur.

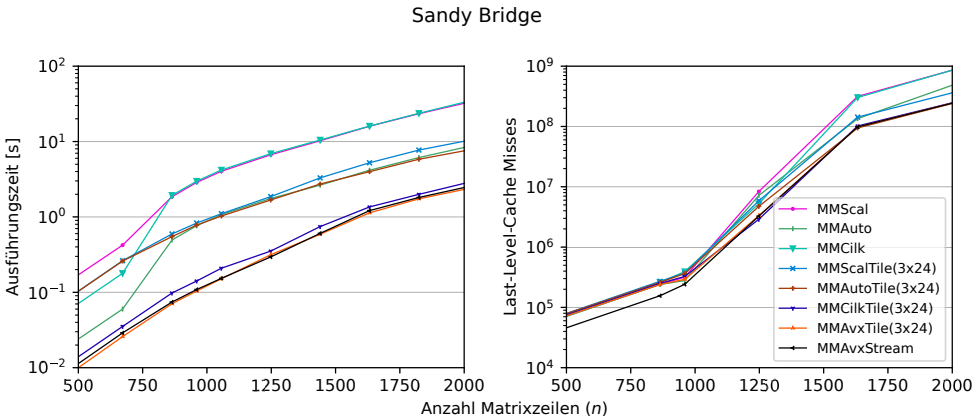

Abbildung B.2.: Ausführungszeit(links), Anzahl der Last-Level-Cache Misses(mitte) und Energieverbrauch(rechts) der Programmvarianten der Matrix-Multiplikation in Abhängigkeit zur Matrixgröße auf der Sandy Bridge (3,4GHz) Architektur.

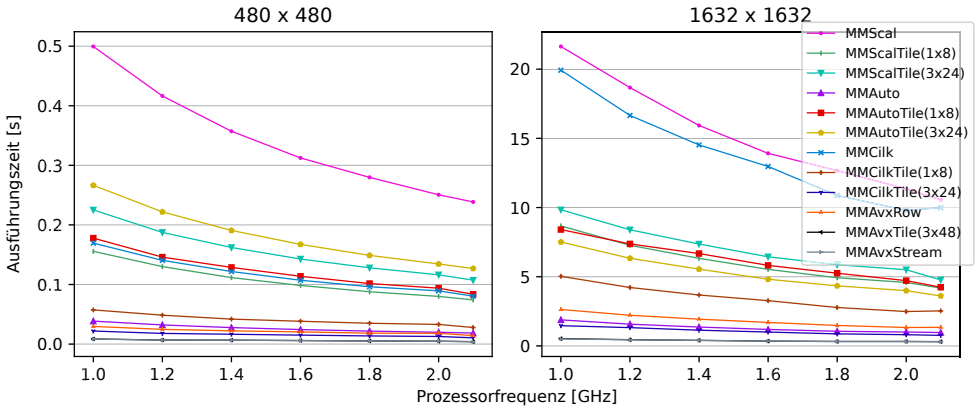

Abbildung B.3.: Ausführungszeit der Programmvarianten mit zeilenweisem Zugriff auf Matrix B und Schleifen-Blockzerlegung mit unterschiedlichen Blockgrößen auf dem Skylake XEON Prozessor. Die Blockgröße $(a \times b)$ entspricht der Anzahl der Matrixelemente in jede Dimension. Matrixgröße: 480×480(links) und 1632×1632(rechts). Die Programmvarianten werden mit AVX512-Instruktionen ausgeführt.

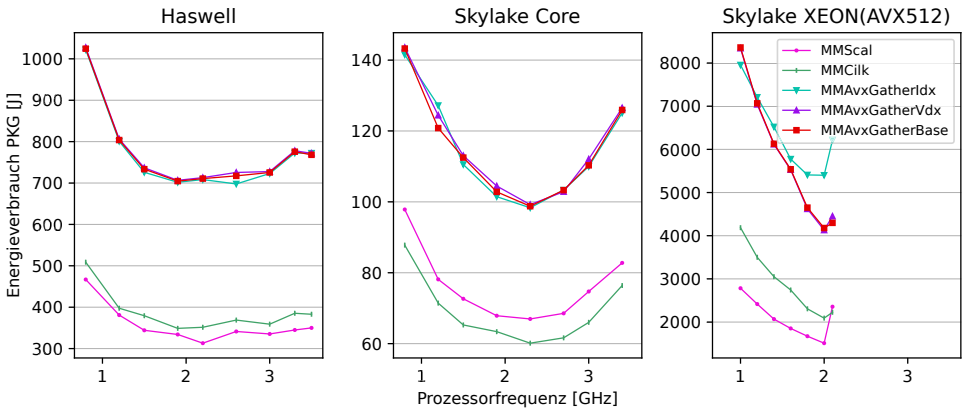

Abbildung B.4.: Energieverbrauch der Programmvarianten mit spaltenweisem Zugriff auf Matrix B in Abhängigkeit zur Prozessorfrequenz auf den Haswell und Skylake Prozessoren. Matrixgröße: 1632×1632.

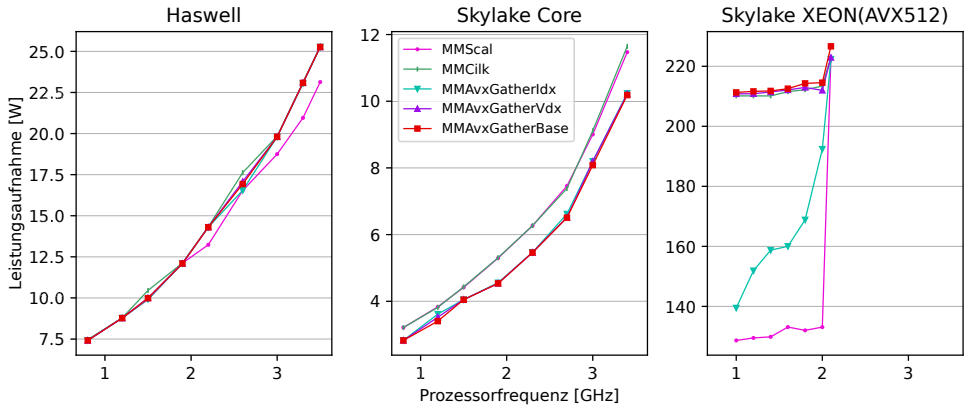

Abbildung B.5.: Leistungsaufnahme der Programmvarianten mit spaltenweisem Zugriff auf Matrix B in Abhängigkeit zur Prozessorfrequenz auf den Haswell und Skylake Prozessoren. Matrixgröße: 1632×1632.

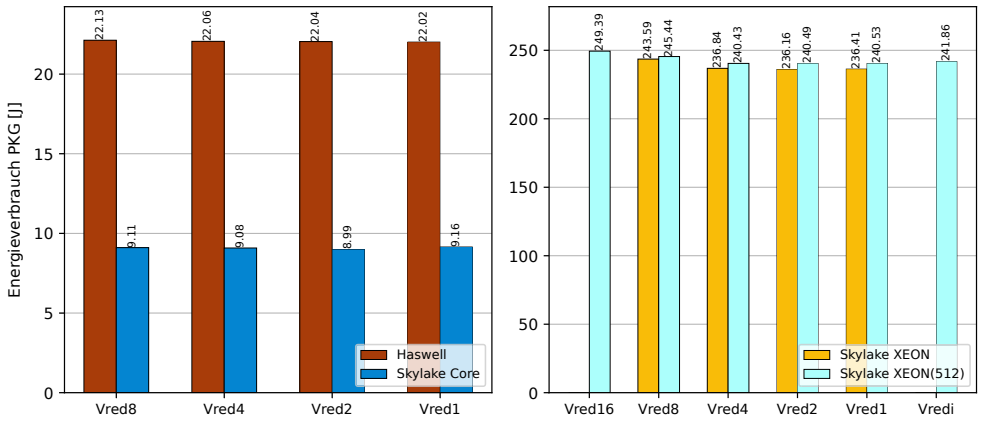

Abbildung B.6.: Energieverbrauch der *MMAvxVred** Programmvarianten der Matrix-Multiplikation auf der Haswell (3,5GHz), Skylake Core (3,4GHz) und Skylake XEON (2,0GHz) Architektur.

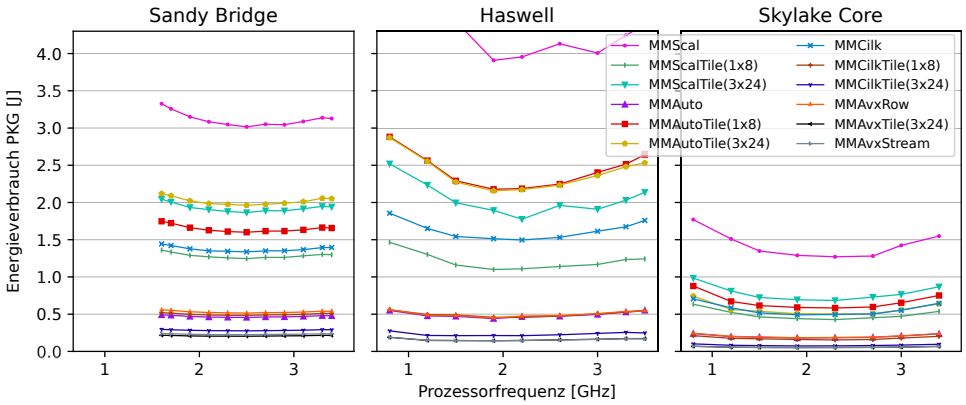

Abbildung B.7.: Energieverbrauch der Programmvarianten mit zeilenweisem Zugriff auf Matrix B und Schleifen-Blockzerlegung mit unterschiedlichen Blockgrößen auf dem Sandy Bridge, Haswell und Skylake Core Prozessor. Die Blockgröße $(a \times b)$ entspricht der Anzahl der Matrixelemente in jede Dimension. Matrixgröße: 480×480.

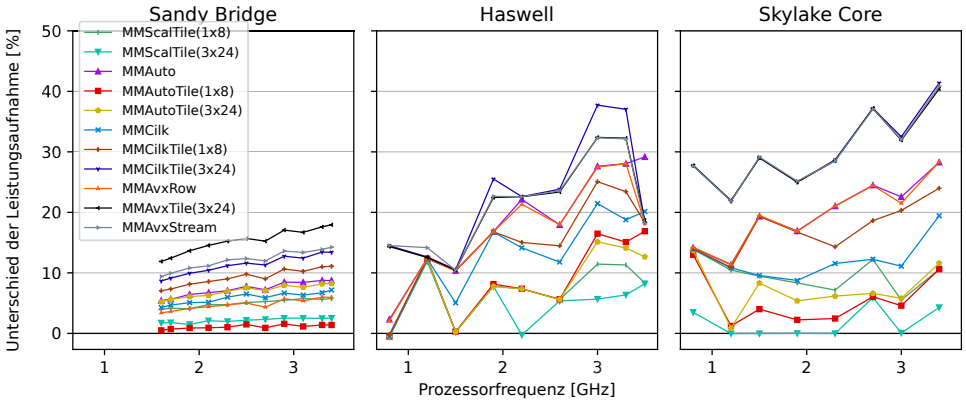

Abbildung B.8.: Leistungsaufnahme der Programmvarianten mit zeilenweisem Zugriff auf Matrix B und Schleifen-Blockzerlegung mit unterschiedlichen Blockgrößen auf dem Sandy Bridge, Haswell und Skylake Core Prozessor. Die Blockgröße $(a \times b)$ entspricht der Anzahl der Matrixelemente in jede Dimension. Matrixgröße: 480×480.

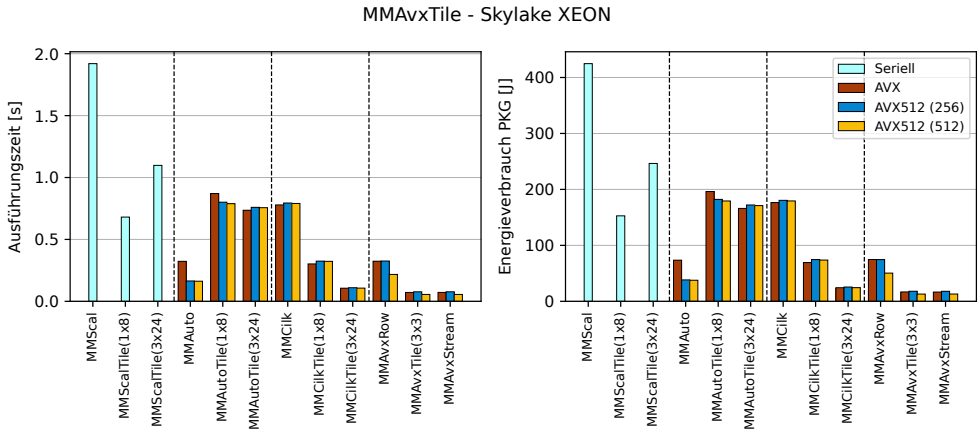

Abbildung B.9.: Ausführungszeit(links) und Energieverbrauch(rechts) der Programmvarianten mit zeilenweisem Zugriff auf Matrix B und Schleifen-Blockzerlegung mit unterschiedlichen Blockgrößen auf dem Skylake XEON Prozessor. Die Blockgröße entspricht der Anzahl der Matrixelemente, bzw. bei den *MMAvx** Programmvarianten der Anzahl der AVX-Vektoren, in jede Dimension. Matrixgröße: 960×960; Prozessorfrequenz 2,0GHz.

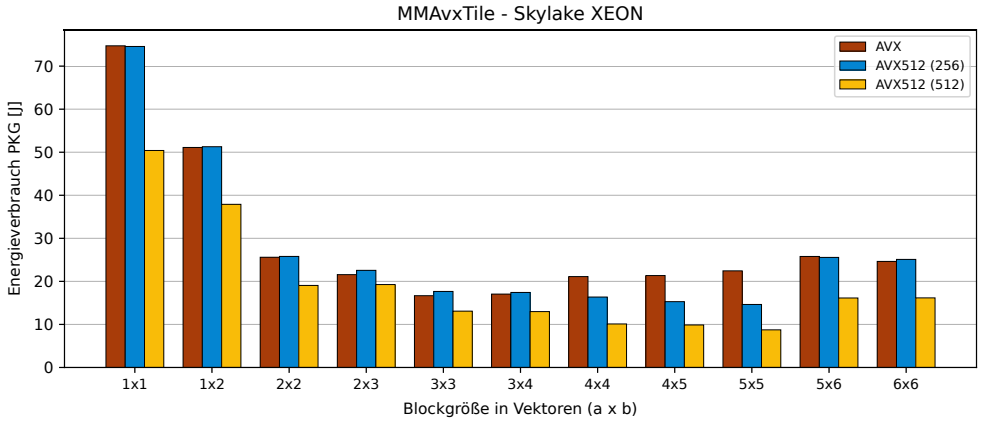

Abbildung B.10.: Energieverbrauch der Programmvarianten mit Schleifen-Blockzerlegung mit unterschiedlichen Blockgrößen auf dem Skylake XEON Prozessor. Die Blockgröße entspricht der Anzahl der AVX-Vektoren (vgl. Abb. 3.2) in jede Dimension. Matrixgröße: 960×960; Prozessorfrequenz 2,0GHz.

C

Anhang zur Gauss-Elimination

```
1 // Schleife über die Zeilen der Matrix A
2 for k = n-1 >= 0; k--=1
3   sum = 0.0
4   // Schleife über die Elemente der Zeilen von A
5   for j = k+1 < n
6     // Summe der bereits bekannten Werte für x_j und deren Koeffizienten a_{k,j}
7     sum += a[k * n + j] * x[j]
8   // Berechnung der Unbekannten nach Gleichung 4.5
9   x[k] = (b[k] - sum) / a[k * n + k]
```

Algorithmus C.1: Implementierung der Rückwärtssubstitution für die skalare Programmvariante *GaussScal* aus Alg. 4.1.

```
1   // b_ = b_ − l_,k · b_k
2   for i = k + 1 < n − 8; i+=8
3   {
4     //b[j] = b[j] − b[i] ∗ l[j];
5     lv = _mm256_loadu_ps(&lt[k ∗ n + i]); // Laden von l_i,k ... l_i,k+7
6     bk = _mm256_broadcast_ss(&b[k]);     // Laden von b_k
7     bi = _mm256_loadu_ps(&b[i]);         // Laden von b_i ··· b_i+7
8     bk = _mm256_mul_ps(bk, lv);          // b_k · l_i,k
9     bi = _mm256_sub_ps(bi, bk);          // b_i − (b_k · l_i,k)
10    _mm256_storeu_ps(&b[i], bi);         // Speichern von b_i ··· b_i+7
11  }
12  if(j < length)
13  {
14    // Hilfsvariable mit 256 0−Bits gefolgt von 256 1−Bits:
15    int loadmask = {0,0,0,0,0,0,0,0,−1,−1,−1,−1,−1,−1,−1,−1};
16    // Laden der Maske, sodass n − (k + 1) % 8 Elemente mit 1−Bits markiert werden:
17    __m256i mask = _mm256_loadu_si256((__m256i∗)&loadmask[n−j]);
18    j = length − 8; // Index für die letzen 8 Elemente des Vektors setzen
19    lv = _mm256_loadu_ps(&lt[k ∗ n + i]); // Laden von l_i,k ... l_i,k+7
20    bk = _mm256_broadcast_ss(&b[k]);     // Laden von b_k
21    bi = _mm256_loadu_ps(&b[i]);         // Laden von b_i ··· b_i+7
22    bk = _mm256_mul_ps(bk, lv);          // b_k · l_i,k
23    bi = _mm256_sub_ps(bi, bk);          // b_i − (b_k · l_i,k)
24    _mm256_maskstore_ps(&b[j], mask, bj); // Speichern von b_i ··· b_n
25  }
```

Algorithmus C.2: AVX-Implementierung der Berechnung des Vektors b in der *GaussStoreU* Programmvariante in Alg. 4.4.

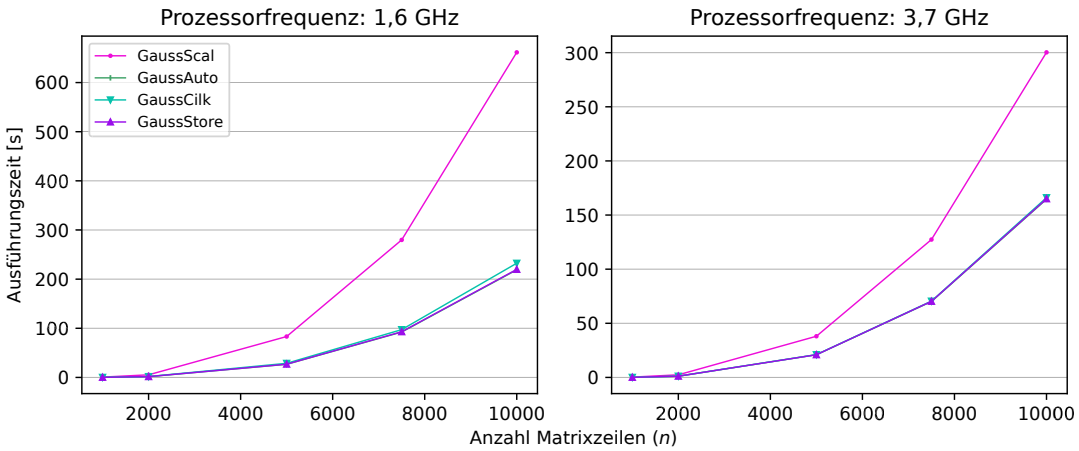

Abbildung C.1.: Ausführungszeit der Programmvarianten mit unterschiedlichen Programmiertechniken in Abhängigkeit zur Zeilengröße von Matrix A auf der Sandy Bridge Architektur bei 1,6 GHz (links) und 3,7 GHz (rechts).

Abbildung C.2.: Ausführungszeit der Programmvarianten mit unterschiedlichen Programmiertechniken in Abhängigkeit zur Zeilengröße von Matrix A auf der Haswell Architektur bei 0,8 GHz (links) und 3,5 GHz (rechts).

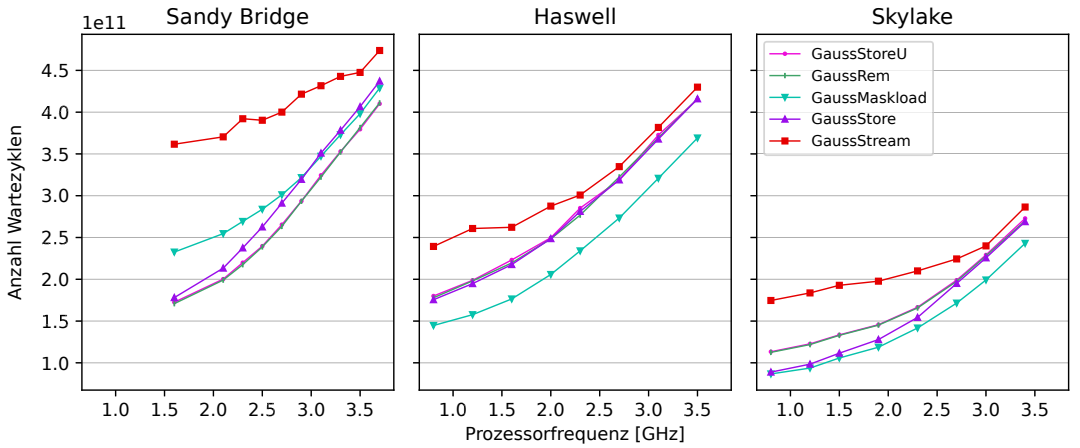

Abbildung C.3.: Anzahl der Zyklen, in denen die Ausführung der AVX-Programmvarianten blockiert ist, da auf andere Ressourcen gewartet wird. Die Werte werden in Abhängigkeit zur eingestellten Prozessorfrequenz auf den drei genutzten Architekturen Sandy Bridge (links), Haswell (mitte) und Skylake (rechts) gezeigt.

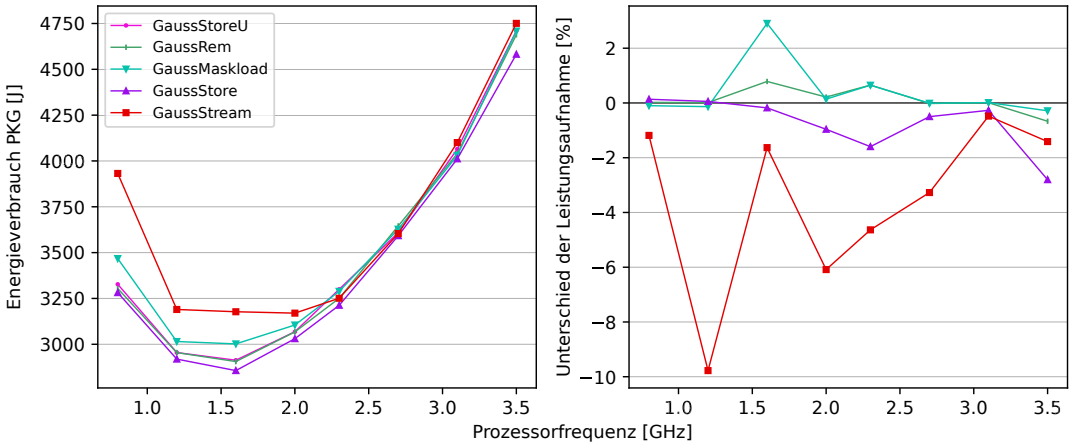

Abbildung C.4.: Energieverbrauch (links) und Leistungsaufnahme (rechts) der AVX-Programmvarianten bei der Ausführung auf der Haswell Prozessorarchitektur in Abhängigkeit zur eingestellten Prozessorfrequenz. Die Leistungsaufnahme ist als Unterschied zur *GaussStoreU* Programmvariante in Prozent dargestellt.

D

Anhang zum Gram-Schmidt Prozess für Vektororthogonalisierung

```
1  // Schritt 3:
2    // Schleife über die Blöcke (tiles ) für die Spalten von Matrix A:
3    for j = i+1 < n; j += tile_size
4      // Schleife über die Blöcke der Elemente von A:
5      for k = 0 < m; k += tile_size
6        // Schleife über die Spalten innerhalb des Blockes:
7        for jt = j < j + tile_size && jt < n
8          // Schleife über die Elemente innerhalb des Blockes:
9          for kt = k < k + tile_size && kt < m
10           R[i*n+jt] += Q[kt*n+i] * A[kt*n+jt]; // r_{i,j} = q^T_{_,i} · a_{_,j}
11
12
13 // Schritt 4:
14   // Schleife über die Blöcke (tiles ) für die Spalten von Matrix A:
15   for j = i+1 < n; j += tile_size
16     // Schleife über die Blöcke der Elemente von A:
17     for k = 0 < m; k += tile_size
18       // Schleife über die Spalten innerhalb des Blockes:
19       for jt = j < j + tile_size && jt < n
20         // Schleife über die Elemente innerhalb des Blockes:
21         for kt = k < k + tile_size && kt < m
22           A[kt*n+jt] -= Q[kt*n+i] * R[i*n+jt]; // a_{_,j} = a_{_,j} − q_{_,i} · r_{i,j}
```

Algorithmus D.1: MgsScalTile: Schleifen-Blockzerlegung der Schleifennester in der *MgsScalTile* Programmvariante. Die Schrittweite der Schleifen wird um die Größe tile_size erweitert. Die übersprungenen Elemente werden in zusätzlichen Schleifen innerhalb des Schleifennestes berechnet. In den inneren Schleifen wird zusätzlich die Schleifenobergrenze abgefangen, um unvollständige Blöcke am Ende des Iterationsraums korrekt zu beenden.

```
// Schritt 3:
  // Schleife über die Zeilen a_{1,_} ... a_{m,_} und Elemente der Spalte q_{_,i}:
  for k = 0 < m; k++
    __m256 v_Qki = _mm256_broadcast_ss(&Q[k,i]); // Laden von q_{k,i}
    // Schleife über die Elemente a_{k,_} der k-ten Zeile und Elemente r_{i,_} i-ten Zeile:
    for j = i+1 < n; j += 8
      // Laden der Elemente a_{k,j}, a_{k,j+1}, ..., a_{k,j+7}:
      __m256 v_a = _mm256_loadu_ps(&A[k,j]);
      // Laden der Elemente r_{i,j}, r_{i,j+1}, ..., r_{i,j+7}:
      __m256 v_Rij = _mm256_loadu_ps(&R[i,j]);
      // Teilergebnisse des Skalarproduktes von q_i^T · a_{k,j}:
      v_Rij = _mm256_fmadd_ps(v_Qki, v_a, v_Rij);
      // Speichern der Elemente r_{i,j}, r_{i,j+1}, ..., r_{i,j+7}:
      _mm256_storeu_ps(&R[i,j], v_Rij);

// Schritt 4:
  // Schleife über die Zeilen a_{1,_} ... a_{m,_} und Elemente der Spalte q_{_,i}:
  for k = 0 < m; k++
    __m256 v_Qki = _mm256_broadcast_ss(&Q[k,i]); // Laden von q_{k,i}
    // Schleife über die Elemente a_{k,_} der k-ten Zeile und Elemente r_{i,_} i-ten Zeile:
    for j = i+1 < n; j += 8
      // Laden der Elemente r_{i,j}, r_{i,j+1}, ..., r_{i,j+7}:
      __m256 v_Rij = _mm256_loadu_ps(&R[i,j]);
      // Laden der Elemente a_{k,j}, a_{k,j+1}, ..., a_{k,j+7}:
      __m256 v_a = _mm256_loadu_ps(&A[k,j]);
      // Teilergebnisse von a_{_,j} − q_{_,i} · r_{i,j}
      v_a = _mm256_fnmadd_ps(v_Qki, v_Rij, v_a);
      // Speichern der Elemente a_{k,j}, a_{k,j+1}, ..., a_{k,j+7}:
      _mm256_storeu_ps(&A[k,j], v_a);
```

Algorithmus D.2: MgsAvxLi Programmvariante: Schritt 3 und 4 der AVX-Implementierung des MGS mit vertauschten Schleifen. Durch den Schleifentausch wird bei der Vektorisierung auf andere Elemente der Matrizen zugegriffen und damit die Lade- und Speicheroperation sowie die Berechnungsreihenfolge verändert.

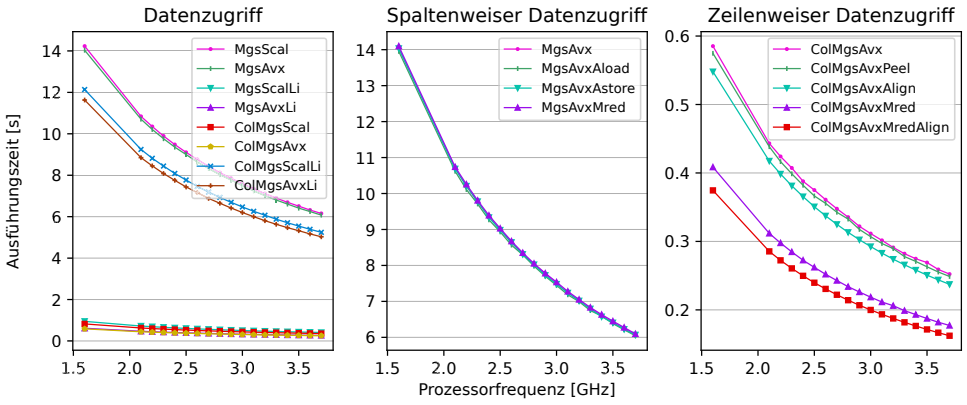

Abbildung D.1.: Ausführungszeit der Programmvarianten der MGS in Abhängigkeit zur Prozessorfrequenz auf der **Sandy Bridge** Architektur. Matrixgröße: 820 × 820 Elemente.

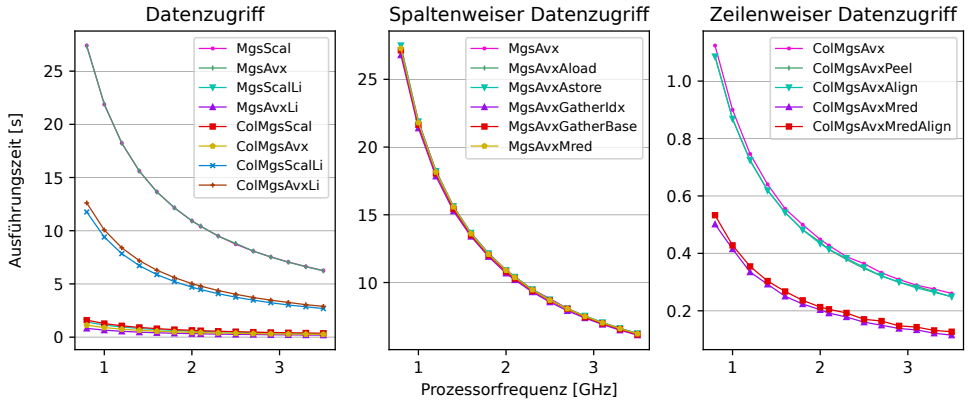

Abbildung D.2.: Ausführungszeit der Programmvarianten der MGS in Abhängigkeit zur Prozessorfrequenz auf der **Haswell** Architektur. Matrixgröße: 820× 820 Elemente.

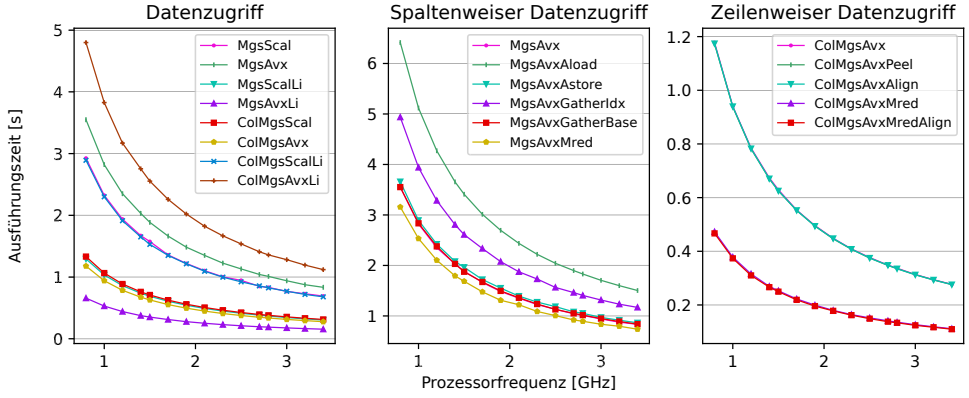

Abbildung D.3.: Ausführungszeit der Programmvarianten der MGS in Abhängigkeit zur Prozessorfrequenz auf der **Skylake Core** Architektur. Matrixgröße: 820 × 820 Elemente.

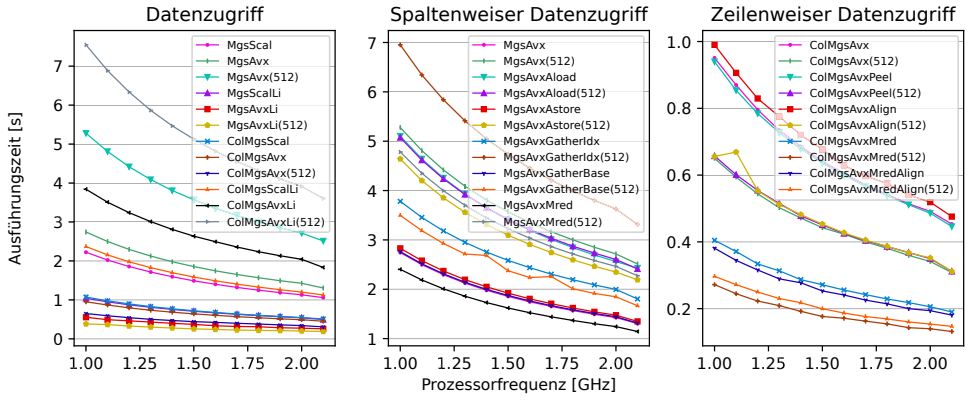

Abbildung D.4.: Ausführungszeit der Programmvarianten der MGS in Abhängigkeit zur Prozessorfrequenz auf der **Skylake XEON** Architektur. Matrixgröße: 820 × 820 Elemente.

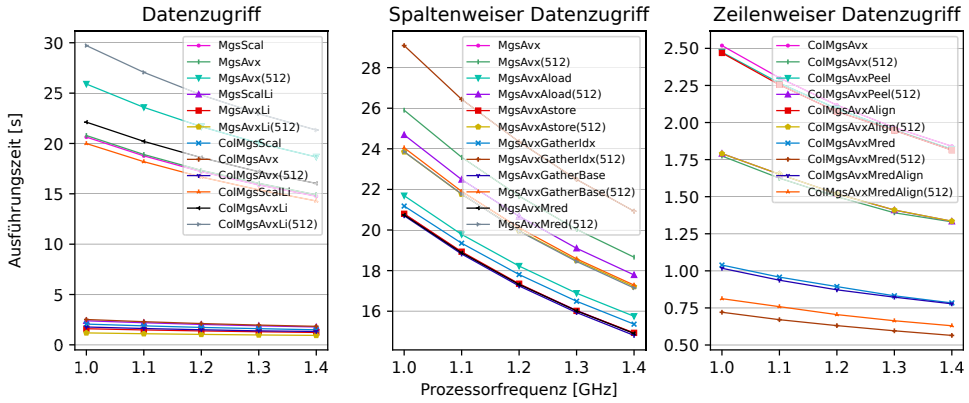

Abbildung D.5.: Ausführungszeit der Programmvarianten der MGS in Abhängigkeit zur Prozessorfrequenz auf der **XEON Phi** Architektur. Matrixgröße: 820×820 Elemente.

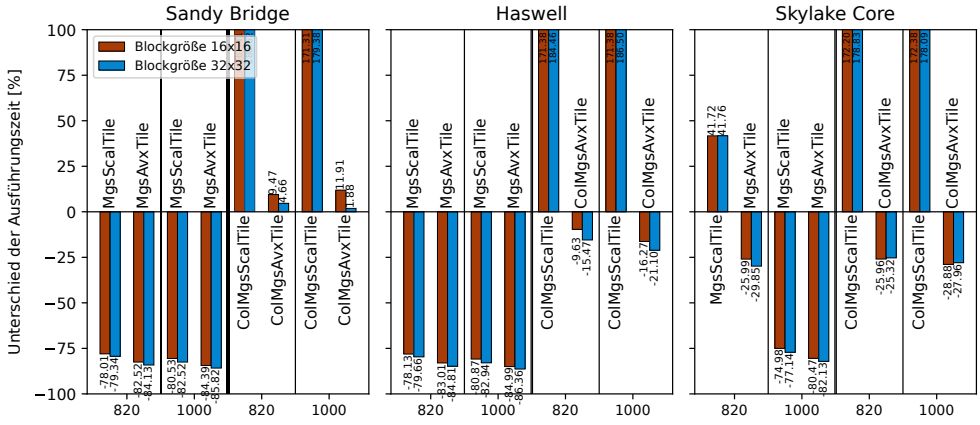

Abbildung D.6.: Unterschied der Ausführungszeit in % der AVX-Programmvarianten mit *loop tiling* im Vergleich zur Programmvariante ohne *loop tiling*. Gezeigt sind die Werte des Sandy Bridge, Haswell und Skylake Core Prozessors. Prozessorfrequenz: 2,1GHz; Matrixgröße: 820×820 bzw. 1000×1000. Negative Werte sind kürzere Ausführungszeiten.

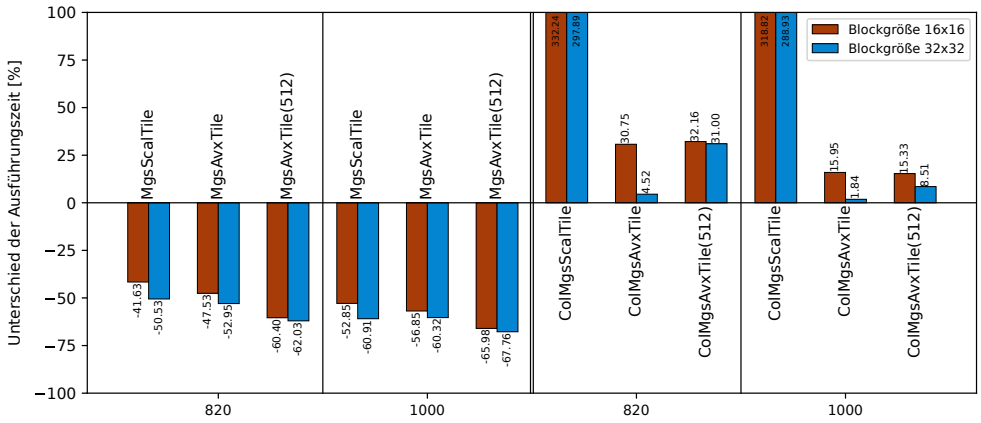

Abbildung D.7.: Unterschied der Ausführungszeit in % der AVX-Programmvarianten mit *loop tiling* im Vergleich zur Programmvariante ohne *loop tiling*. Gezeigt sind die Werte des XEON Phi Prozessors. Prozessorfrequenz: 1,4GHz; Matrixgröße: 820×820 bzw. 1000×1000. Negative Werte sind kürzere Ausführungszeiten.

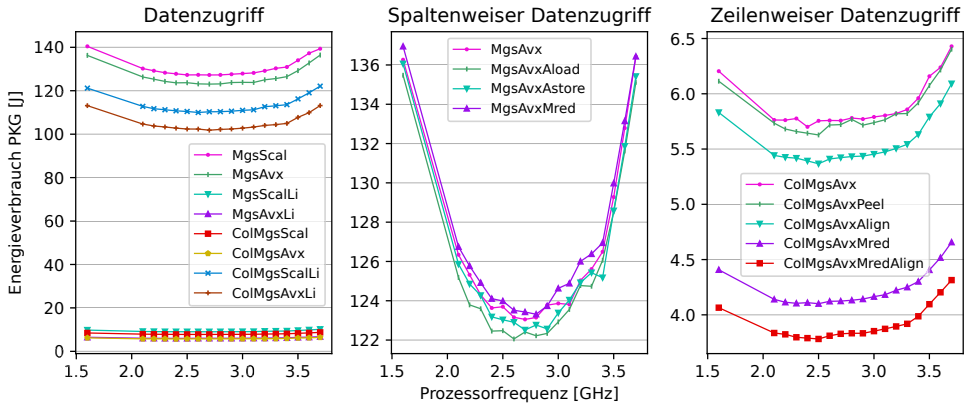

Abbildung D.8.: **Energieverbrauch** der Programmvarianten der MGS in Abhängigkeit zur Prozessorfrequenz auf der **Sandy Bridge** Architektur. Matrixgröße: 820×820 Elemente.

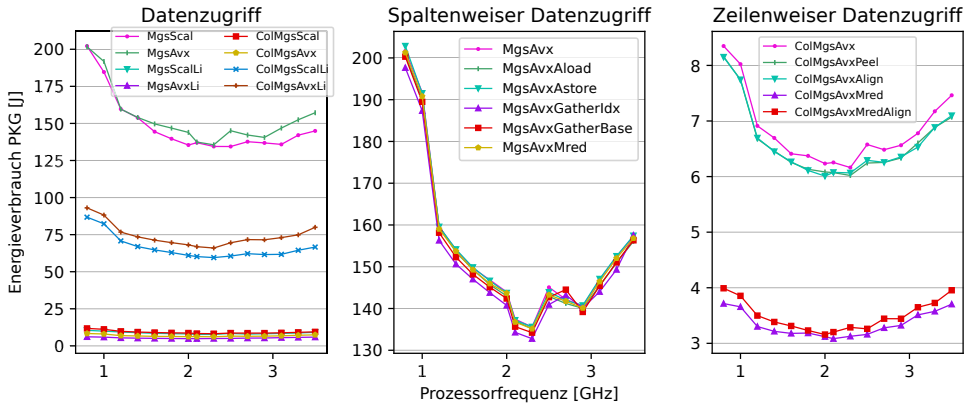

Abbildung D.9.: Energieverbrauch der Programmvarianten der MGS in Abhängigkeit zur Prozessorfrequenz auf der **Haswell** Architektur. Matrixgröße: 820× 820 Elemente.

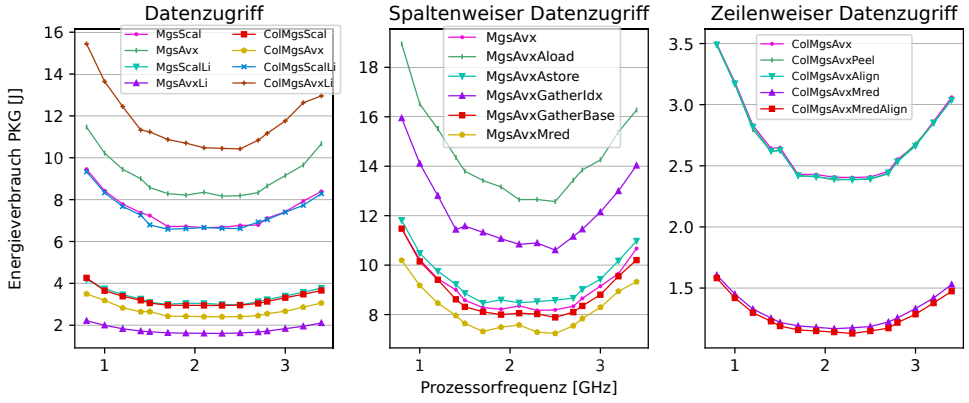

Abbildung D.10.: Energieverbrauch der Programmvarianten der MGS in Abhängigkeit zur Prozessorfrequenz auf der **Skylake Core** Architektur. Matrixgröße: 820 × 820 Elemente.

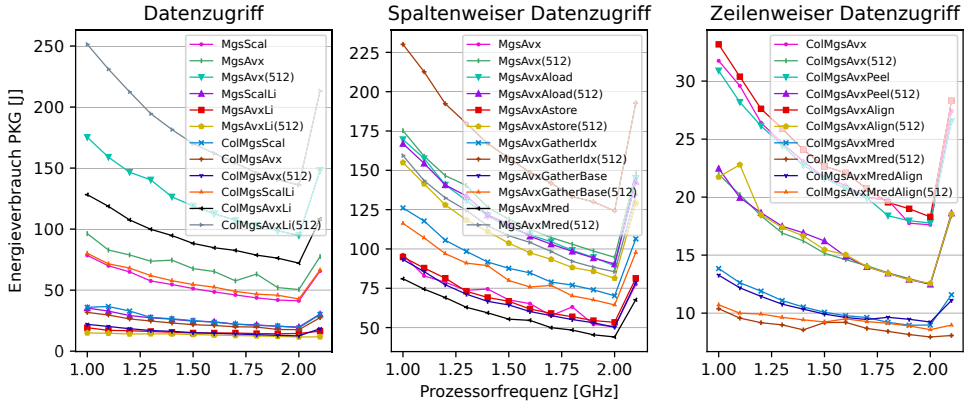

Abbildung D.11.: Energieverbrauch der Programmvarianten der MGS in Abhängigkeit zur Prozessorfrequenz auf der **Skylake XEON** Architektur. Matrixgröße: 820 × 820 Elemente.

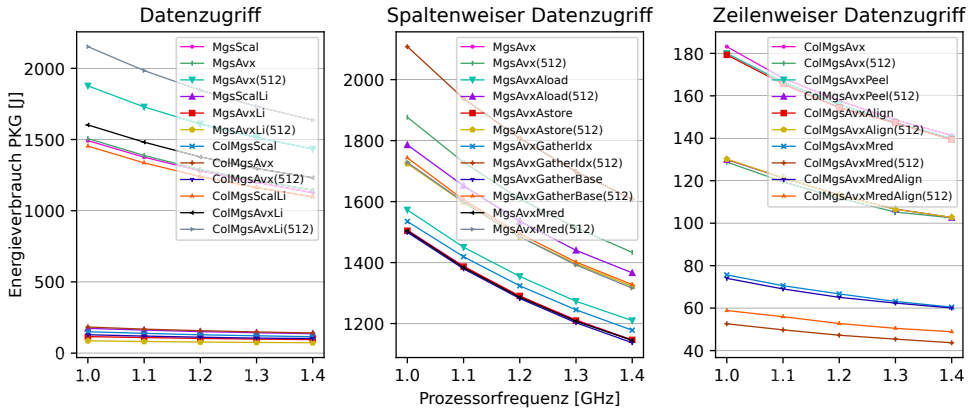

Abbildung D.12.: Energieverbrauch der Programmvarianten der MGS in Abhängigkeit zur Prozessorfrequenz auf der **XEON Phi** Architektur. Matrixgröße: 820 × 820 Elemente.

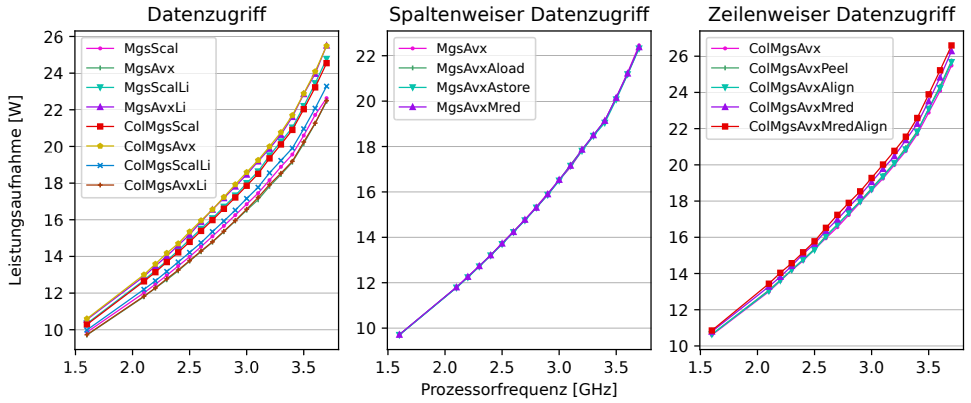

Abbildung D.13.: Leistungsaufnahme der Programmvarianten der MGS in Abhängigkeit zur Prozessorfrequenz auf der **Sandy Bridge** Architektur. Matrixgröße: 820 × 820 Elemente.

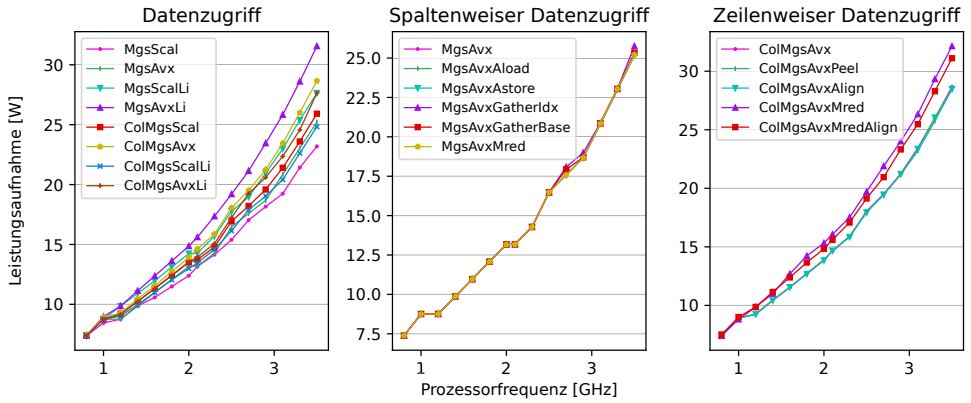

Abbildung D.14.: Leistungsaufnahme der Programmvarianten der MGS in Abhängigkeit zur Prozessorfrequenz auf der **Haswell** Architektur. Matrixgröße: 820 × 820 Elemente.

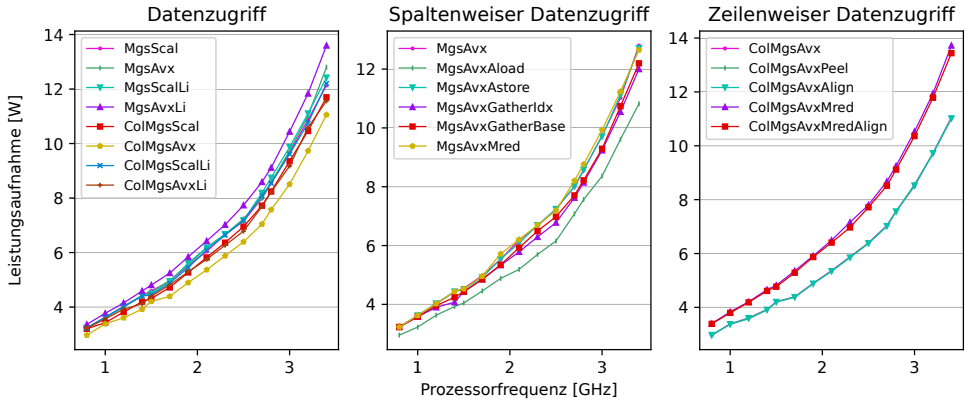

Abbildung D.15.: Leistungsaufnahme der Programmvarianten der MGS in Abhängigkeit zur Prozessorfrequenz auf der **Skylake Core** Architektur. Matrixgröße: 820 × 820 Elemente.

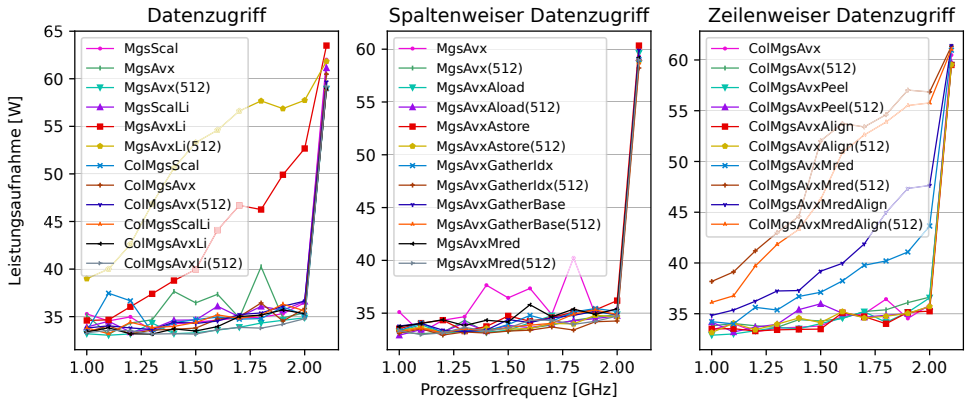

Abbildung D.16.: Leistungsaufnahme der Programmvarianten der MGS in Abhängigkeit zur Prozessorfrequenz auf der **Skylake XEON** Architektur. Matrixgröße: 820 × 820 Elemente.

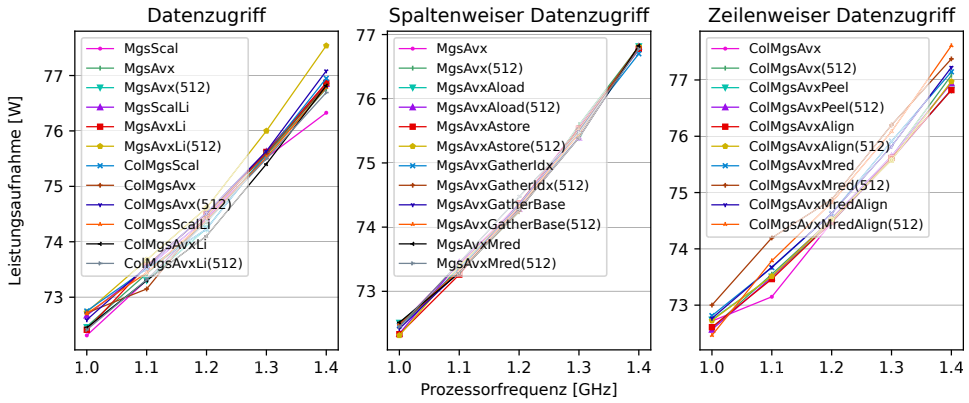

Abbildung D.17.: Leistungsaufnahme der Programmvarianten der MGS in Abhängigkeit zur Prozessorfrequenz auf der **XEON Phi** Architektur. Matrixgröße: 820 × 820 Elemente.

E
Thesen

These 1

Die Vektorisierung von Algorithmen senkt den Energieverbrauch des ausgeführten Programms.

These 2

Die höhere Prozessorauslastung durch vektorisierte Programme führt zu einer erhöhten Leistungsaufnahme bei gleicher Prozessorfrequenz.

These 3

Die Nutzung des höchsten verfügbaren AVX-Instruktionssatzes erzeugt die energie-effizientesten Vektorisierungen.

These 4

Der Zugriff auf aufeinanderfolgende Speicherstellen ist essentiell für energie-effiziente AVX-Programme.

These 5

Die Auswahl der verwendeten AVX-Lade- und Speicheroperationen hat einen Einfluss auf die Ausführungszeit und den Energieverbrauch des Programms.