Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Analyse gemischter Genauigkeit und der Nutzung von spezialisierten GPU-Matrixeinheiten anhand der PLSSVM Anwendung

Benjamin Kurz

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Dirk Pflüger
Betreuer/in:	Dr.-Ing. Martin Bernreuther, Marcel Breyer, M.Sc, Alexander Van Craen, M.Sc.
Beginn am:	5. April 2022
Beendet am:	5. Oktober 2022

Kurzfassung

Um die volle Rechenleistung moderner Hardware wie Graphics Processing Units (Deutsch: Grafikprozessoren) (GPUs) vollständig auszunutzen, muss auf deren Besonderheiten Rücksicht genommen werden. Neueste GPUs sind mit speziellen Hardware-Einheiten ausgestattet, wodurch die Berechnung von Matrizen je nach Genauigkeit und Architektur um 2 (FP64) bis 8 (TF32) Mal bei den in dieser Arbeit genutzten Zahlenformaten auf einer NVIDIA A100 beschleunigt werden kann. In dieser Arbeit wird anhand des Conjugate Gradient (Deutsch: Konjugierte Gradienten) (CG)-Algorithmus innerhalb der Parallel Least Squares Support Vector Machine (PLSSVM) Bibliothek aufgezeigt, wie die Herausforderungen für das Nutzen der Matrixeinheiten gelöst werden können und so zusätzliche Performance abgerufen werden kann. PLSSVM ist eine hochmoderne, effiziente und auf Beschleunigerkarten spezialisierte Support-Vector Machine (SVM), die aus Trainingsdaten eine Hyperebene berechnet mit deren Hilfe Daten klassifiziert werden können. Die hier verwendeten Methoden von der Problemanalyse bis zur Implementierung und des Profilings sind dabei leicht auf andere Probleme zu übertragen. Durch Stabilitätsverbesserungen am CG-Algorithmus können Teile des Algorithmus in einfacher Genauigkeit berechnet werden, was die theoretische Rechenleistung auf einer NVIDIA A100 verdoppelt und auf einer RTX 3080 sogar um Faktor 64 erhöht. Zusammen ergeben sich Implementierungen mit mehreren gleichzeitig genutzten Genauigkeiten, mit denen eine Laufzeitreduktion von bis zu 28 auf einer RTX 3080 erreicht werden und von bis zu 7.4 auf einer NVIDIA A100. Eine RTX 3080 unterbietet mit den Änderungen in Messungen die Laufzeit von 4 NVIDIA A100 Grafikkarten mit der vorherigen Implementierung bei der Ausführung des CG-Algorithmus mit ebenso hochwertigen Ergebnissen, trotz niedriger Genauigkeit auf der GPU, wodurch eine moderne Desktop-GPU die Leistung eines Rechenzentrumsknotens mit 4 GPUs erreicht. Daraus ergibt sich nicht nur eine signifikante Zeitersparnis, auch der finanzielle Faktor, der über das 40-fache beträgt zwischen einer RTX 3080 und 4 NVIDIA A100, ist beachtenswert.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Abkürzungsverzeichnis

ADU Address Divergence Unit (Deutsch: Adressen-Divergenzeinheit). 80

ALU Arithemitc Logic Unit (Deutsch: Arithemitische Logikeinheit). 80

AMD Advanced Micro Devices (Deutsch: Erweiterte Mikrogeräte). 26

CG Conjugate Gradient (Deutsch: Konjugierte Gradienten). 3

CPU Central Processing Unit (Deutsch: Zentrale Verarbeitungseinheit). 15

CUDA Compute Unified Device Architecture (Deutsch: Einheitliche Gerätearchitektur für die Datenverarbeitung). 7, 15

D-S-FR-0 Doppelte Genauigkeit, Standard-Kernel, β nach F-R und keinem Korrekturschema. 75

DAXPY Double precision (Deutsch: Doppelte Genauigkeit) $\alpha\mathbf{x} + \mathbf{y}$. 44

DDOT Double Dot Product (Deutsch: Skalarprodukt in doppelter Genauigkeit). 43

F-R Fletcher-Reeves. 52

FMA Fused Multiply–Add (Deutsch: Gesichertes Multiplizieren-Addieren). 38

FP16 Floating Point Number (Deutsch: Gleitkommazahl) mit 16 Bits. 15

FP64 Floating Point Number (Deutsch: Gleitkommazahl) mit 64 Bits. 15

FPGA Field-Programmable Gate Array (Deutsch: Feldprogrammierbares Gate-Array). 27

GEMV Generalized Matrix-Vector Multiplication (Deutsch: Verallgemeinerte Matrix-Vektor-Multiplikation). 43

GPGPU General Purpose Graphics Processing Unit (Deutsch: Allzweck-Berechnung auf einer Grafikprozessoreinheit). 18

GPU Graphics Processing Unit (Deutsch: Grafikprozessor). 3

HIP Heterogeneous-Computing Interface for Portability (Deutsch: Heterogene Rechnerschnittstelle für Portabilität). 15

HPC High-performance computing (Deutsch: Hochleistungsrechnen). 30

ISA Instruction Set Architecture (Deutsch: Befehlssatzarchitektur). 40

KI Künstliche Intelligenz. 38

LGS Lineares-Gleichungssystem. 15

LS-SVM Least-Squares Support-Vector Machine (Deutsch: Methode der kleinsten Quadrate SVM). 15

LSU Load Store Unit (Deutsch: Laden-Speichern-Einheit). 80

NaN Not a Number (Deutsch: Keine Zahl). 29

OpenCL Open Computing Language (Deutsch: Offene Computersprache). 26

OpenMP Open Multi-Processing (Deutsch: Offene Mehrprozessorverarbeitung). 26

P-R Polak-Ribière. 52

PLSSVM Parallel Least Squares Support Vector Machine. 3

PTX Parallel Thread Execution (Deutsch: Parallele Thread-Ausführung). 40

REF Referenz-Implementierung. 69

SIMT Single Instruction, Multiple Threads (Deutsch: Einzelne Anweisung, mehreren Threads). 33

SM Streaming Multiprocessor. 9

SMO Sequential Minimal Optimization (Deutsch: Sequentielle Minimaloptimierung). 17

SP Streaming Processor. 32

SVM Support-Vector Machine. 3

TF32 Tensor Float Number (Deutsch: Tensor-Gleitkommazahl) mit 19 Bits. 30

TPU Tensor Processing Unit (Deutsch: Tensor-Verarbeitungseinheiten). 38

WMMA Warp Matrix Multiply Add (Deutsch: Warpweite Matrixmultiplikation und -Addition). 40

1 Einleitung

Um die immer komplexeren und größeren Probleme unserer Zeit effektiv und schnell lösen zu können, ist es erforderlich moderne Hardware optimal auszunutzen. Bei rechenintensiven Berechnungen wird heutzutage häufig auf GPUs gesetzt, die eine höhere Rechenleistung als Central Processing Units (Deutsch: Zentrale Verarbeitungseinheiten) (CPUs) aufweisen, was sich in den Systemen der TOP500 Liste [TOP22] widerspiegelt. Zusätzlich sind neueste GPUs mit extra Hardwareeinheiten ausgestattet, welche die Berechnung von Matrizen in Abhängigkeit der benutzten Zahlenformate um 2 (Floating Point Number (Deutsch: Gleitkommazahl) mit 64 Bits (FP64)) bis 16 (Floating Point Number (Deutsch: Gleitkommazahl) mit 16 Bits (FP16)) Mal beschleunigen [AMD21; NVI20]. Aber auch ohne Matrixeinheiten ist die Rechenleistung moderner GPUs in einfacher Genauigkeit abhängig von der Architektur 2 (AMD CDNA und NVIDIA Datencenter) bis 64 (NVIDIA Ampere Desktop) Mal höher als in doppelter Genauigkeit [AMD21; NVI20; NVI21]. Werden die Matrixeinheiten und niedrige Genauigkeiten gezielt ausgenutzt, sind GPUs nochmals um ein Vielfaches schneller und energiesparender als CPUs. Für die Stabilität eines Programms kann die Genauigkeit aber nicht einfach beliebig reduziert werden. Aus diesem Grund gilt es zu analysieren, für welche Rechenschritte eine hohe Genauigkeit benötigt wird und wo eine niedrigere Genauigkeit ausreichend ist. Durch dieses Vorgehen werden verschiedene Genauigkeiten für verschiedene Berechnungen erhalten. Wie im Allgemeinen ein möglichst hoher Geschwindigkeitszuwachs unter Ausnutzung gemischter Genauigkeiten und GPU-Matrixeinheiten gewonnen werden kann unter möglichst qualitativ gleichbleibenden Ergebnissen, wird in dieser Arbeit anhand eines Fallbeispieles analysiert. Dabei werden Laufzeitreduktionen von 4.2-4.5 in doppelter Genauigkeit und von 7.4 mit gemischten Genauigkeiten auf einer NVIDIA A100 erreicht und eine Laufzeitreduktion auf einer RTX 3080 mit gemischten Genauigkeiten von 27, wodurch eine RTX 3080 sogar schnellere Laufzeiten als 4 NVIDIA A100 GPUs mit der bisherigen Implementierung erreicht.

Im Bereich des maschinellen Lernens wird mit immer größeren Datensätzen gerechnet, die immer mehr Rechenleistung für das Training und die Verarbeitung benötigen. Beliebte Anwendungen sind die Klassifikation und Regression, wofür SVMs [Vap06] weit verbreitet sind. Eine darauf basierende Bibliothek, die das Problem auf möglichst parallele Art und Weise löst, ist PLSSVM ([VBP22b]): Durch eine Umformulierung der SVM, die nur schwer zu parallelisieren ist, in eine Least-Squares Support-Vector Machine (Deutsch: Methode der kleinsten Quadrate SVM) (LS-SVM) [SGB+02] entsteht ein Lineares-Gleichungssystem (LGS). Die dabei entstehende Matrix ist symmetrisch und positiv definit und dadurch kann das LGS mit dem CG-Algorithmus gelöst werden, der sich gut durch GPUs beschleunigen lässt. PLSSVM ist in modernem C++17 geschrieben und eignet sich auch für große und dicht besetzte Datensätze. Für die rechenintensiven Teile stellt PLSSVM verschiedene Backends zur Verfügung, die eine ganze Reihe an verschiedenen Beschleunigerkarten unterschiedlicher Hersteller unterstützen. Allerdings werden nur für Compute Unified Device Architecture (Deutsch: Einheitliche Gerätearchitektur für die Datenverarbeitung) (CUDA) und Heterogeneous-Computing Interface for Portability (Deutsch: Heterogene Rechnerschnittstelle für Portabilität) (HIP) die Matrixeinheiten vollständig unterstützt. Da kein aktuelles Testsystem

für HIP zur Verfügung steht, wird die Arbeit beziehungsweise neue Kernel mit Matrixeinheiten nur für das Backend in CUDA umgesetzt. Viele angewandte Techniken innerhalb der Kernel und Verbesserungen des CG-Algorithmus sind aber auch voll auf die anderen Backends übertragbar.

Als eine Herausforderung in dieser Arbeit stellt sich die potenziell große Matrix-Kondition im zu lösenden LGS und der Aufbau dieser Matrix. Die vorhandene Implementierung von PLSSVM in einfacher Genauigkeit hat aus diesen Gründen schnell Probleme mit der numerischen Stabilität und liefert keine ausreichend genauen Ergebnisse. Daher werden Techniken vorgestellt, wie die Implementierung, trotz in einigen Teilen deutlich weniger präzisen Zahlenformaten, weiterhin numerisch eine gewisse Stabilität erhält. Zusätzlich werden diese Techniken auch auf eine Variante angewandt, die ausschließlich in doppelter Genauigkeit rechnet, um die Stabilität des CG-Algorithmus weiter zu erhöhen und dadurch auch seine Konvergenzeigenschaften. Diese wiederum verbessern die Laufzeit. Ebenfalls werden die Auswirkungen von Korrekturschemata, ein bereits in PLSSVM vorhandenes und die zuverlässigen Updates [CBB+10], und der Berechnungsvorschrift von β innerhalb des CG-Algorithmus analysiert.

Die zweite große Herausforderung liegt bei der Implementierung der Kernel mit Matrixeinheiten. Unter Ausnutzung der Matrixeinheiten erreichen GPUs zwar eine höhere Rechenleistung, allerdings sind die PLSSVM Kernel nicht durch diese limitiert, sondern durch Speichertransaktionen, worauf die Matrixeinheiten keine Auswirkung haben. Die Kernel werden dadurch von Grund aus neugeschrieben mit Fokus auf die Nutzung von Matrixeinheiten, einem hocheffizienten Speicherlayout mit asynchronen Speichertransaktionen und, soweit möglich, einer Erhöhung der numerischen Stabilität.

Da der CG-Algorithmus und hohe Matrixkonditionen nicht PLSSVM spezifisch ist, sondern typische Algorithmen oder Probleme darstellen, können die in dieser Arbeit gezeigten Vorgehensweisen für viele Probleme hilfreich sein. Auch bei der Implementierung der CUDA-Kernel, welche die Matrixeinheiten nutzen, wird versucht die genaue Funktionsweise, die Gedankengänge in der Implementierung und das Optimieren unter Hilfe von Profiling so darzustellen, dass die Konzepte auch auf andere Kernel übertragbar sind.

Nach dieser Einführung werden in Kapitel 2 verwandte Arbeiten zu verschiedenen SVMs Implementierungen und Erweiterungen erläutert, ebenso Arbeiten zu gemischten Genauigkeiten und Matrixeinheiten. In Kapitel 3 werden zuerst die theoretischen Grundlagen zu PLSSVM und der aktuelle Aufbau der Bibliothek zusammengefasst, darauf folgen die theoretischen Grundlagen zu Zahlenformaten und gemischten Genauigkeiten, bevor zuletzt ein Überblick über CUDA gegeben wird mit besonderem Fokus auf den Matrixeinheiten. Kapitel 4 befasst sich mit der Implementierung: Dabei liegt der Fokus auf Veränderungen am CG-Algorithmus und dem genauen Aufbau der CUDA-Kernel. Die sich daraus ergebende Veränderungen auf Konvergenzverhalten und Laufzeit werden in Kapitel 5 zusammengefasst. Zum Abschluss gibt es in Kapitel 6 eine Zusammenfassung und einen Ausblick auf zukünftige Arbeiten an PLSSVM.

2 Verwandte Arbeiten

Dieses Kapitel gibt einen Überblick über die aktuelle Forschung zu den Themen, die in dieser Arbeit behandelt werden. Zuerst werden Arbeiten zur SVM, besonders verschiedene Bibliotheken und Ansätze, betrachtet. Im nächsten Schritt werden Veröffentlichungen zum Thema gemischte Genauigkeiten betrachtet und nach möglichen Überschneidungen zu SVM gesucht. Zuletzt geht es um Nutzung von Matrixeinheiten und möglichen Überschneidungen zu gemischten Genauigkeiten oder SVM.

2.1 Support-Vector Maschine

Die Grundidee der SVM, das Prinzip der strukturellen Risikominimierung, wurde 1979 von Vapnik [Vap06] behandelt und führte zu dem 1992 von Boser et al. [BGV92] entwickelten Trainingsalgorithmus für optimale Margin-Klassifikatoren, bis dann 1995 von Cortes und Vapnik [CV95] die SVM in ihrer heutigen Grundform mit verschiedenen Kernelfunktionen vorgestellt worden ist.

In den folgenden Jahren folgten mehrere Erweiterungen und Lösungsansätze, von denen einige von Cristianini und Shawe-Taylor [CS00] zusammengefasst worden sind. Ein Algorithmus davon ist der von Platt [Pla98] vorgestellte Sequential Minimal Optimization (Deutsch: Sequentielle Minimaloptimierung) (SMO)-Algorithmus, welcher die Speicheranforderungen reduziert und teure Matrixoperationen vermeidet. Auf dieser Methode basiert auch die derzeit weit verbreitete SVM-Implementierung LIBSVM von Chang und Lin [CL11]. Diese ist inzwischen für viele Programmiersprachen wie C++, Java, Python, oder Matlab verfügbar.

Die Implementierung dieser Arbeit beruht auf dem LS-SVM Ansatz von Suykens und Vandewalle [SV99a], bei dem gegenüber des klassischen SVM-Ansatzes Gleichheitsbeschränkungen anstatt Ungleichungen betrachtet werden. Die durch diese Bedingung entstehenden Fehler werden durch eine Art Verlustfunktion mit kleinsten Quadraten ausgeglichen. Dadurch lässt sich ein LGS anstatt eines quadratischen Programmierungs-Problemes herleiten.

Der LS-SVM Ansatz wurde in folgenden Arbeiten vorangetrieben: Zum Beispiel wurde in Suykens und Vandewalle [SV99b] die LS-SVM um multiple Klassen erweitert; in Suykens et al. [SLV] wird untersucht wie die dünne Besetzung durch das Beschneiden von Stützwerten erhalten bleiben kann; in Suykens et al. [SBLV02] wird eine Gewichtung eingeführt, die für mehr Stabilität sorgt; in Huang et al. [HMHS17] wird ein indefiniter Kernel für die LS-SVM vorgeschlagen; während in Suykens et al. [SGB+02] die theoretischen Überlegungen vorangetrieben worden sind.

Die PLSSVM von Van Craen et al. [VBP22a] basiert auf dem LS-SVM Ansatz und der Arbeit von Van Craen [Van18], hierbei werden in erster Linie die Kernel durch optimale Ausnutzung moderner Hardware beschleunigt und können gegenüber vielen anderen Implementierungen bei

hoher Datenbesetzung die gleiche Geschwindigkeit gewähren. Das dabei entstehende LGS wird durch verschiedene Hardware beschleunigte Backends mit dem CG-Algorithmus gelöst. In weiteren Veröffentlichungen von Van Craen et al. [VBP22b] wurde die Implementierung mit anderen GPU-gestützten SVM Implementierungen verglichen und in Breyer et al. [BCP22] die verschiedenen Backends von PLSSVM auf verschiedener Hardware evaluiert.

2.2 Gemischte Genauigkeiten

Den vermutlich größten Einfluss hatte der Ansatz gemischter Genauigkeiten im Bereich der neuronalen Netze. In einer Arbeit von Micikevicius et al. [MNA+17] wurde zum Beispiel vorgestellt, dass viele Operationen bei neuronalen Netze in *Half-Precision* (Deutsch: Halbe Genauigkeit) berechnet werden können und dadurch der benötigte Speicher gegenüber *Single-Precision* (Deutsch: Einfache Genauigkeit) eingespart werden kann. Die Entwicklung in der künstlichen Intelligenz führte dazu, dass Grafikkarten mit Matrixeinheiten ausgestattet und neue Beschleuniger-Karten für diese Operationen entwickelt worden sind.

Überlegungen zu gemischten Genauigkeiten reichen weit in die Anfänge der Computergeschichte zurück. So wurde 1963 von Wilkinson [Wil94] eine Analyse über Fließkommazahlen, ihre Genauigkeiten, Konditionen und iterative Verfeinerungsverfahren geschrieben. In einer Arbeit von Baboulin et al. [BBD+09] wurden iterative Verfeinerungsalgorithmen für verschiedene CPUs anhand eines Beispieles, bei dem ein LGS gelöst worden ist, evaluiert.

Mit dem Aufkommen der General Purpose Graphics Processing Unit (Deutsch: Allzweck-Berechnung auf einer Grafikprozessoreinheit) (GPGPU), und Grafikkartenarchitekturen, bei denen die Geschwindigkeit bei geringeren Genauigkeiten deutlich höher ausfiel, nahm das Thema wieder an Relevanz zu. In einer Arbeit von Götz et al. [GWX+12] wird ein gemischter Genauigkeiten-Ansatz auf ein Molekulardynamik-Programm angewendet und die Auswirkungen der unterschiedlichen Genauigkeiten auf verschiedene Rechenoperationen hin untersucht. Bei einer Veröffentlichung von Göddeke et al. [GST05] wird eine Technik vorgestellt, wie bei einer Finite-Elemente-Simulation eine gemischte Präzisions-Fehlerkorrektur vorgenommen werden kann.

Veröffentlichungen zu SVM mit gemischten Genauigkeiten gibt es nach bestem Wissen nicht, der Sache am nächsten kommt eine Veröffentlichung von Wang et al. [WSH+21], bei der es, unter Ausnutzung von gemischten Genauigkeiten, um die Verarbeitung einer Schnittstellen zwischen Gehirn und Maschine mit anschließender Evaluation mithilfe einer SVM geht. Bei PLSSVM wird allerdings das LGS durch den CG-Algorithmus gelöst und für diesen Algorithmus gibt es verschiedene Veröffentlichungen zu gemischten Genauigkeiten. In dieser Veröffentlichung von Göddeke et al. [GST07] wird die iterative Verfeinerung erfolgreich auf den CG-Algorithmus angewandt. In einer anderen Arbeit von Clark et al. [CBB+10] wird die Technik der zuverlässigen Updates benutzt, um die sich fortführenden numerischen Fehler zu korrigieren.

2.2.1 GPU-Matrixeinheiten

Matrixeinheiten auf GPUs sind noch nicht so lange verfügbar und erweitern diese mit neuen Instruktionen. Die meisten Veröffentlichungen, wie die von Martineau et al. [MAM18], untersuchen entweder die praktischen Geschwindigkeitsvorteile der Matrixeinheiten oder, wie in der Arbeit von

Markidis et al. [MCL+18], in welchen Fällen Matrixeinheiten angewandt werden können und wie sich das auf die Genauigkeit auswirkt. Dabei ist relevant, dass die ersten Generationen von GPU Matrixeinheiten noch keine Unterstützung für FP64 boten, sondern nur für eine Mischung aus FP32 mit FP16 und dadurch zwangsweise viele Anwendungen gemischte Genauigkeiten benutzten. Die Überschneidung von Arbeiten über GPU Matrixeinheiten und gemischte Genauigkeiten ist folglich groß.

Bei neuronalen Netzwerken werden die Genauigkeiten unterdessen mit Ausnutzung der Matrixeinheiten noch geringer, und so stellt Li und Su [LS20] ein binäres neuronales Netzwerk vor, das durch die Hardware Unterstützung noch einmal schneller wird, auch wenn für Datenformate unter 4 Bit Genauigkeit die Unterstützung nur experimentell ist [NVI22a].

Eine weitere Arbeit von Haidar et al. [HBT+20] untersucht die Auswirkungen von Matrixeinheiten beim Lösen von LGS mit verschiedenen Techniken, wie der LU-Zerlegung, oder dem vorkonditionierten verallgemeinerten Algorithmus für minimale Residuen.

Eine Veröffentlichung von Tu et al. [TCJM21] befasst sich mit dem Lösen eines vorkonditionierten CG-Algorithmus mittels Matrixeinheiten, was einerseits sehr nahe an der Aufgabenstellung dieser Arbeit ist, anderseits aber bei PLSSVM der Aufbau der Matrix der rechenintensive Part ist.

Die Herausforderung dieser Arbeit unterscheidet sich einerseits von den anderen SVM Veröffentlichungen, da bei diesen nach bestem Wissen und Gewissen keine gemischten Genauigkeiten verwenden werden, anderseits unterscheidet sie sich ebenfalls von den Veröffentlichungen zum CG-Algorithmus, da bei dieser Arbeit der Fokus weniger auf der effizienten Lösung des CG liegt, sondern auf dem Aufbau der LS-SVM Matrix und gegenüber anderen Veröffentlichungen [Cla20] die Kondition dieser Matrix beliebig groß werden kann.

3 Grundlagen

In diesem Kapitel werden die theoretischen Grundlagen, welche für die weitere Arbeit notwendig sind, und die bisherige praktische Umsetzung in PLSSVM behandelt. Zuerst wird auf die theoretischen Grundlagen von PLSSVM eingegangen, danach auf die praktische Umsetzung in der Implementierung. Im weiteren Verlauf werden die grundlegenden Techniken und Funktionsweisen der zu analysierende Themen, gemischte Genauigkeiten und GPU-Matrixeinheiten, erläutert.

3.1 Parallel Least Squares Support Vector Machine

Die PLSSVM ist ein Programm, welches in seiner jetzigen Form ein binärer Klassifikator für Supervised Machine Learning (Deutsch: überwachtes maschinelles Lernen) ist. Die grundlegende Idee dahinter ist die SVM. Der Fokus des Programms liegt auf Performance durch das Ausnutzen der hohen Parallelität moderner Hardware wie GPUs. Damit dies möglich ist, wird eine spezielle Formulierung der SVM genutzt, die LS-SVM.

3.1.1 Klassische Support Vector Machine

In diesem Kapitel wird ein Überblick über die von Cortes und Vapnik [CV95] vorgestellte SVM gegeben. Für weitere Informationen empfiehlt sich dessen Ausarbeitung oder die Arbeit von Van Craen [Van18].

Für eine gegebene Menge an Objekten, die in zwei Klassen eingeteilt werden können, wie in Abbildung 3.1 zu sehen, besteht die grundlegende Idee darin, eine Hyperebene zu finden, welche die Klassen voneinander trennt und gleichzeitig den Abstand von den Objekten der Klasse maximiert, sodass ein möglichst breiter Bereich um die Hyperebene frei von Objekten ist. Durch diese Eigenschaft gehört die SVM zu den Large Margin Classifiers (Deutsch: Breiter-Rand-Klassifikatoren). Die Idee hinter dieser Eigenschaft ist es durch die Maximierung des Bereichs eine möglichst stabile und sichere Klassifikation zu ermöglichen. Würde eine beliebige Hyperebene gewählt, welche die Klassen trennen würde, wäre die Chance höher, dass für nicht gelernte neue Datenpunkten die Hyperebene nicht mehr optimal die Daten trennen würde. Die Hyperebene mit der größten Breite bietet dadurch die wahrscheinlich größte Generalisierbarkeit. Seinen Namen hat die SVM durch die Datenpunkte beziehungsweise Vektoren, welche die Hyperebene begrenzen und dadurch definieren, sogenannte Support Vektoren.

Für ein gegebenes Datenset

$$\{(\mathbf{x}_i, y_i) | \mathbf{x}_i \in \mathbb{R}^d, y_i \in \{1, -1\}\}_{i=0}^{m} \tag{3.1}$$

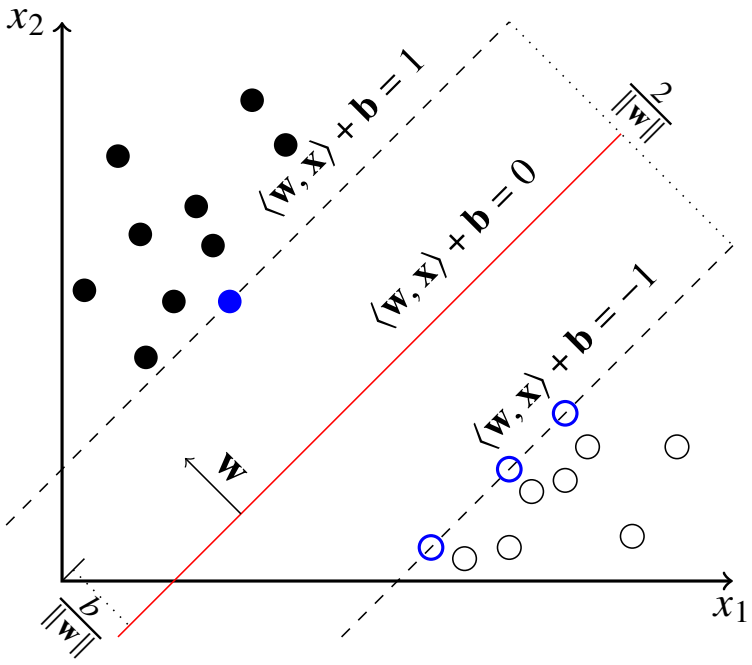


Abbildung 3.1: SVM: Durch ausgefüllte oder umrandete Kreise wird ein 2 dimensionaler Datensatz in einem Koordinatensystem dargestellt. Diese werden durch eine in rot dargestellte Hyperebene, oder im 2D Fall einer Geraden, getrennt. Dabei ist der Abstand von den Datenpunkten zur Hyperebene maximal. Die blauen Kreise sind dabei die Datenpunkte, welche die Hyperebene definieren.

mit m-Datenpunkten, die jeweils d-Merkmale besitzen ($\mathbf{x}_i \in \mathbb{R}^d$) und einer der zwei mit -1 und 1 gekennzeichneten Klassen ($y_i = -1$ oder $y_i = 1$) gehören, wird eine Hyperebene gesucht. Sollten die vorliegenden Daten anders gekennzeichnet sein, muss zuvor eine Anpassung bzw. Transformation durchgeführt werden. Es wird zum jetzigen Zeitpunkt davon ausgegangen, dass die Klassen linear separierbar sind, also eine Hyperebene existiert, mit der die Klassen perfekt voneinander getrennt werden können.

Eine Hyperebene

$$(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) = 0 \tag{3.2}$$

lässt sich durch einen Normalenvektor $\mathbf{w} \in \mathbb{R}^d$und eine Konstante b definieren. $\mathbf{w}$ wird dabei so skaliert, dass das Trainingsdaten-freie Band um die Hyperebene, die sogenannte Bandbreite, $\frac{2}{\|\mathbf{w}\|_2}$ ist. Der Abstand der Hyperebene zum Ursprung beträgt dann $\frac{b}{\|\mathbf{w}\|_2}$ und wird Bias genannt. Daraus ergibt sich, dass für alle Trainingsdaten

$$y_i \cdot (\langle \mathbf{w}, \mathbf{x}_i \rangle + b) \geq 1 \ \forall i \tag{3.3}$$

gelten muss.

Aus Abschnitt 3.1.1 folgt, dass das Finden einer Hyperebene mit der maximalen Bandbreite äquivalent ist zur Minimierung der quadratischen Norm $\|\mathbf{w}\|_2^2$ unter der Nebenbedingung Gleichung (3.3).

Für eine gegebene Hyperebene, welche die Daten richtig separiert, kann nun für gegebenes $\mathbf{x}$ auf die Klassenzugehörigkeit getestet werden, indem berechnet wird auf welcher Seite der Hyperebene der Datenpunkt liegt

$$y = \operatorname{sgn}(\langle \mathbf{w}, \mathbf{x} \rangle - b), \tag{3.4}$$

wobei die Signumfunktion ein positives Ergebnis auf die *1*-Klasse projiziert und ein negatives Ergebnis auf die *-1*-Klasse. Ist die Hyperebene einmal gefunden worden, kann mit nur $O(d)$ Operationen ein unbekannter Datenpunkt klassifiziert werden.

Allerdings sind viele Datensätze nicht perfekt linear separierbar, das heißt, dass eine wie oben beschriebene Hyperebene gar nicht existiert. Ein möglicher Grund kann ein Rauschen oder (Mess-) Fehler in den Daten sein, in vielen Fällen gibt es aber einfach für die Daten keine 100% perfekte Hyperebene. Für diesen Fall werden Verletzungen der Nebenbedingungen aus Gleichung (3.3) erlaubt, die aber möglichst gering ausfallen sollen. Umgesetzt wird dies mit sogenannten Schlupfvariablen ξ_i, was zu folgenden Gleichungen führt

$$\min_{\mathbf{w},\mathbf{b},\xi} \left(\frac{1}{2} \|\mathbf{w}\|_2^2 + C \sum_{\forall i} \xi_i \right) \tag{3.5}$$
$$\text{unter den Bedingungen } y_i \cdot (\langle \mathbf{w}, \mathbf{x}_i \rangle) \geq 1 - \xi_i \ \wedge \ \xi_i \geq 0 \qquad \forall i,$$

wobei ähnlich zu einer *Loss-function* die Fehler aufsummiert, mit einer Konstanten C multipliziert und ebenso minimiert werden. Dies ergibt ein konvexes quadratisches Problem.

Um das Problem in der Form von Gleichung (3.5) zu lösen, wird häufig der SMO-Algorithmus benutzt [Pla98]. Dieser analysiert zwei Datenpunkte und bestimmt die Hyperebene zwischen diesen, danach werden weitere Punkte darauf untersucht, ob sie gegen diese Lösung verstoßen und, falls ja, wird die Hyperebene angepasst. Dies wird so lange wiederholt, bis zu einer gewissen Genauigkeit keine Verstöße mehr vorliegen. Dieser Algorithmus konvergiert garantiert, ist aber in seiner Grundform streng sequentiell und nutzt daher nicht das volle Potenzial moderner Hardware und Beschleunigerkarten aus. Weitere Informationen dazu gibt es der Veröffentlichung von Platt [Pla98].

3.1.2 Least Squares Support Vector Machine

Die LS-SVM von Suykens und Vandewalle [SV99a] verändert das ausgehende Optimierungsproblem minimal. Die Fehlerterme aus Gleichung (3.5) werden zuerst quadriert, aufsummiert und dann durch zwei geteilt und die Ungleichungen aus Gleichung (3.5) werden zu einer Gleichung, sodass folgendes Gleichungssystem erhalten wird:

$$\min_{\mathbf{w},\mathbf{b},\xi} \left(\frac{1}{2} \|\mathbf{w}\|_2^2 + \frac{C}{2} \sum_{\forall i} \xi_i^2 \right) \tag{3.6}$$
$$\text{unter der Bedingung } y_i \cdot (\langle \mathbf{w}, \mathbf{x}_i \rangle) = 1 - \xi_i \ \wedge \ \xi_i \geq 0 \qquad \forall i.$$

Dadurch wird die Hyperebene nicht nur durch wenige Supportvektoren bestimmt, sondern jeder Datenpunkt wird proportional zum Abstand zur Hyperebene gewichtet. Gewichtungen können nun negativ als auch positiv sein, weshalb diese quadriert werden.

Dieses Problem lässt sich mithilfe seines dualen Problems [BT69] und den Karush-Kuhn-Tucker Bedingungen [KT13] in ein LGS umformen,

$$\begin{bmatrix} \mathbf{Q} & \mathbf{1}_m \\ \mathbf{1}_m^T & 0 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{a} \\ b \end{bmatrix} = \begin{bmatrix} \mathbf{y} \\ 0 \end{bmatrix} \tag{3.7}$$

bei dem die Matrix $\mathbf{Q} \in \mathbb{R}^{m \times m}$ ist, die Matrixeinträge

$$Q_{ij} = \langle \mathbf{x}_i, \mathbf{x}_j \rangle + \frac{1}{C}\delta_{ij} \tag{3.8}$$

sind, wobei δ_{ij} das Kronecker Delta ist

$$\delta_{ij} = \begin{cases} 0, & \text{if } i \neq j \\ 1, & \text{if } i = j \end{cases} . \tag{3.9}$$

Durch die Definition der Q_{ij} ist die Matrix $\mathbf{Q}$ symmetrisch positiv definit. Dies hat den großen Vorteil, dass es für positiv definite lineare Gleichungssysteme eine ganze Reihe an etablierten und auch parallelisierbaren Algorithmen zur Auswahl gibt.

PLSSVM nutzt hierbei das CG-Verfahren, welches in Abschnitt 3.1.4 genauer behandelt wird.

3.1.3 Kernel Trick

Datensätze können, wie in Abschnitt 3.1.1 erwähnt, nicht linear separierbar sein. Im Bereich des maschinellen Lernens im Allgemeinen wird in solchen Fällen gerne der Kernel-Trick [MR16] angewandt. Dabei wird das Problem in eine höhere Dimension transformiert und in diesem Raum eine Hyperebene gesucht.

Ein Kernel ist eine beliebige Abbildung $K : X \times X \to \mathbb{R}$, für die ein Skalarproduktraum $(F, \langle \cdot, \cdot \rangle)$ und eine Abbildung $\phi : X \to F$ definiert ist, sodass $K(x, y) = \langle \phi(x), \phi(y) \rangle \wedge \forall x, y \in X$. Im Falle der SVM werden die Datenpunkte $\mathbf{x}_i \in \mathbb{R}^d(X)$ durch eine Funktion ϕ in einen sogenannten potentiell höher dimensionalen Merkmalsraum transformiert und dort die Hyperebene gesucht. Aus der Nebenbedingung aus 3.6 wird dadurch

$$y_i \cdot k(\mathbf{w}, \mathbf{x}_i) = 1 - \xi_i \ \wedge \ \xi_i \geq 0 \ \forall i \tag{3.10}$$

und die Matrixeinträge aus 3.8 ändern sich ebenfalls ab zu

$$Q_{ij} = k(\mathbf{x}_i, \mathbf{x}_j) + \frac{1}{C}\delta_{ij}. \tag{3.11}$$

Hierbei fällt auf, dass das zuvor genutzte Skalarprodukt ebenfalls ein möglicher Kernel ist. In PLSSVM sind derzeit drei weit verbreitete Kernel implementiert:

- Der lineare Kernel (Skalarprodukt): $k(\mathbf{x}_i, \mathbf{x}_j) = \langle \mathbf{x}_i, \mathbf{x}_j \rangle$
- Der polynomiale Kernel: $k(\mathbf{x}_i, \mathbf{x}_j) = (\gamma \langle \mathbf{x}_i, \mathbf{x}_j \rangle + r)^d, \gamma > 0, d \in \mathbb{N}$
- Der Radiale-Basisfunktion-Kernel (RBF): $k(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\gamma \|\mathbf{x}_i - \mathbf{x}_j\|^2), \gamma > 0$

Weitere Informationen können zum Beispiel in der Arbeit von Murty und Raghava [MR16] nachgeschlagen werden.

Eine Multi-GPU-Implementierung liegt allerdings nur für den linearen Kernel vor, weshalb dieser in der weiteren Ausarbeitung auch im Fokus liegt.

3.1.4 CG-Algorithmus

Das CG-Verfahren wurde 1952 von Hestenes und Stiefel [HS52] vorgeschlagen und gehört zu den *Krylow-Unterraum-Verfahren* und den *Lanczos-type product methods*. Es ist ein numerisches Verfahren um lineare Gleichungssysteme der Form $\mathbf{Ax} = \mathbf{b}$ zu lösen unter der Bedingung, dass die Matrix $\mathbf{A} \in \mathbb{R}^{m \times m}$ symmetrisch und positiv definit ist. Letzteres bedeutet, dass für ein beliebiges $\mathbf{v} \in \mathbb{R}^m, \mathbf{v} \neq 0$ gilt, dass $\mathbf{v}^\intercal \mathbf{Av} > 0$, dies ist gleichbedeutend dazu, dass alle Eigenwerte positiv sein müssen.

Die Idee hinter dem CG-Verfahren ist es, anstatt $\mathbf{Ax} = \mathbf{b}$ direkt zu lösen, die quadratische Form

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\intercal \mathbf{Ax} - \mathbf{b}^\intercal \mathbf{x} \tag{3.12}$$

zu minimieren, wobei $f : \mathbb{R}^m \to \mathbb{R}$ ist. Die Funktion $f(\mathbf{x})$ hat dabei die geometrisch Form eines m-dimensionalen Paraboloides. Der Gradient $\nabla f(\mathbf{x}) = \mathbf{Ax} - \mathbf{b}$ ist genau im Minimum 0, und ist dadurch identisch zur gesuchten Gleichung $\mathbf{Ax} = \mathbf{b}$.

Bei einem verwandten Algorithmus, dem Gradientenverfahren, wird nun nach einem Startwert für $\mathbf{x}$ der Gradient gebildet und in Richtung des steilsten Abstieges ($-\nabla f(\mathbf{x}) = \mathbf{b} - \mathbf{Ax} = \mathbf{r}$, dabei ist $\mathbf{r}$ das Residuum und gleichzeitig die Richtung des Abstieges) auf dem Paraboliden in Richtung des Minimums iteriert. Ein Iterationsschritt ist dann von der Form

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \alpha_k \mathbf{r}^k \tag{3.13}$$

mit

$$\mathbf{r}^k = \mathbf{b} - \mathbf{Ax}^k \text{ und } \alpha_k = \frac{\mathbf{r}^{k\intercal}\mathbf{r}^k}{\mathbf{r}^{k\intercal}\mathbf{Ar}^k}. \tag{3.14}$$

Dies hat aber den Nachteil, dass der Abstieg besonders bei unterschiedlich großen Eigenwerten eine langsame Konvergenz haben und einen *Zick-Zack-Pfad* annehmen kann, da die Richtung des lokalen Abstieges nicht global gesehen optimal ist und dadurch immer wieder in den gleichen Dimensionen iteriert wird.

Genau diesen Punkt versucht das CG-Verfahren anders zu machen, indem die Suchrichtungen nun nicht mehr das Residuum ist, sondern eine Projektion des Gradienten, die orthogonal zu allen vorherigen Suchrichtungen ist:

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \alpha_k \mathbf{d}^k \tag{3.15}$$

mit

$$\begin{aligned} \mathbf{d}^{k\intercal}\mathbf{A}\mathbf{d}^j &= 0 && \text{für } k \neq j \\ \mathbf{d}^k &\perp \mathbf{A}\mathbf{d}^i && \text{für } i < k \\ \mathbf{d}^k &\perp_{\mathbf{A}} \mathbf{d}^i. \end{aligned}$$

Der sich daraus ergebende Algorithmus wird in Abschnitt 4.1 genauer analysiert und ist dort in Listing 4.1 zu sehen. Genauere Information und Herleitungen können zum Beispiel in Shewchuk [She94] nachgelesen werden.

3.1.5 Programm Umsetzung

Der Fokus von PLSSVM liegt bei einer unter Ausnutzung moderner Hardware hoch parallelen und effizienten Ausführung. Um gleichzeitig eine hohe Bandbreite an Hardware von verschiedenen Herstellern zu unterstützen, stehen mehrere Backends in unterschiedlichen Sprachen für die rechenintensive Teile zur Verfügung: In Compute Unified Device Architecture (Deutsch: Einheitliche Gerätearchitektur für die Datenverarbeitung) (CUDA) [NVI22c], Heterogeneous-Computing Interface for Portability (Deutsch: Heterogene Rechnerschnittstelle für Portabilität) [AMD22], Open Computing Language (Deutsch: Offene Computersprache) (OpenCL) [Gro22a], Open Multi-Processing (Deutsch: Offene Mehrprozessorverarbeitung) (OpenMP) [Ope22] und SYCL [Gro22b]. Es kann beim Kompilieren angegeben werden, für welche Backends kompiliert werden soll und diese dann dynamisch als Kommandozeilenparameter beim Aufruf des Programms ausgewählt werden.

Die Ausführung des Lernprozesses der SVM lässt sich in 4 Teile gliedern:

1. Das Einlesen der Trainingsdaten: PLSSVM unterstützt Datensätze im weit verbreiteten LIBSVM- oder ARFF-Format. Die Datensätze werden hierbei von der Festplatte in ein zweidimensionales Feld im Hauptspeicher geladen.
2. Vorbereitungsphase: Sich aus den Trainingsdaten ergebende Hilfsvariablen werden berechnet, die Trainingsdaten für das entsprechende Backend vorbereitet und die notwendigen Daten auf die entsprechenden Devices geladen, dabei müssen die Trainingsdaten in den Speicher der Devices passen.
3. Gleichungssystem lösen: Das LGS wird mithilfe des CG-Algorithmus gelöst. Hierbei ist wichtig, dass dabei die Matrix nicht explizit gespeichert wird. Da in aller Regel deutlich mehr Datenpunkte als Features vorhanden sind, benötigt das Abspeichern der Matrix $\mathbf{Q} \in \mathbb{R}^{m \times m}$ deutlich mehr Ressourcen als die Trainingsdaten $\mathbf{X} := (\mathbf{x}_0, \mathbf{x}_1 \dots \mathbf{x}_m), \mathbf{x}_i \in \mathbb{R}^d \rightarrow \mathbf{X} \in \mathbb{R}^{d \times m}$ und damit potenziell mehr Speicher als Hauptspeicher oder Videospeicher zur Verfügung steht.
4. Abspeichern des Ergebnisses: Die gefundene Hyperebene beziehungsweise ihre Support-Vektoren und Gewichte werden auf die Festplatte geschrieben.

Schritt eins und vier sind unabhängig vom benutzten Backend und kein Gegenstand dieser Masterarbeit. Die Unterschiede in Schritt zwei beinhalten Initialisierungen auf Host und Device, so wie Umwandlungen und werden nicht genau aufgeführt. Die eigentlichen Unterschiede und auch der Fokus der Arbeit liegt daher auf Schritt drei, dem Lösen des CG-Algorithmus.

Wichtig für die weitere Arbeit ist dabei die CUDA Implementierung, weshalb CUDA in Abschnitt 3.3 gesondert vorgestellt wird. Matrixeinheiten werden derzeit nur von CUDA und HIP vollständig unterstützt, leider gab es Probleme ein ausreichend modernes Advanced Micro Devices (Deutsch: Erweiterte Mikrogeräte) (AMD) System mit Matrixeinheiten für Tests zur Verfügung gestellt zu bekommen, weshalb die HIP Implementierung nicht durchgeführt werden konnte. Weitere und ausführlichere Informationen zu den Backendplattformen können in Breyer et al. [BCP22] nachgeschlagen werden.

3.2 Gemischte Genauigkeiten

Unter Algorithmen mit gemischter Genauigkeit versteht man Algorithmen, die gezielt unterschiedlich genaue Datenformate nutzen um schneller, energieeffizienter und besser die Hardware ausnutzend das Problem in vergleichbarer Qualität zu lösen. Die grundlegende Idee mit verschieden genauen Zahlenformaten geht weit zurück: So wurde 1966 im Fortran 66 Standard [Gor14] ein Real- und Double-Precision Zahlenformat eingeführt. 1947 wurde von von Neumann und Goldstine [NG47] vorgeschlagen, bei Additionen eine höhere Genauigkeit auszunutzen und erst das Endergebnis auf die ursprüngliche Genauigkeit zu runden.

In der frühen Prozessor-Geschichte hatten viele CPUs ihre ganz eigenen Formate für Gleitkommazahlen und die dazugehörige Software oder Gleitkommaeinheit. Eine geringere Genauigkeit lässt sich mit viel weniger Transistoren und mit schnelleren Schaltzeiten verwirklichen, kann aber zu ungenauen beziehungsweise falschen Ergebnisse führen. Dadurch konnten auf verschiedenen CPUs ganz unterschiedliche Lösungen entstehen. Durch die Norm *IEEE 754*, welche zuerst 1985 verabschiedet worden ist, sind die Darstellungen für Gleitkommazahlen weitgehend festgelegt. Bekannte und gebräuchliche Formate sind zum Beispiel der Datentyp *float* oder *double*, welche auf dem binary32 (Umgangssprachlich *Einfache Genauigkeit* oder FP32) beziehungsweise binary64 (Umgangssprachlich *Doppelte Genauigkeit* oder FP64) Format der Spezifikation beruhen. Bei Integer beziehungsweise ganzen Zahlen wurde noch keine Norm verabschiedet. Die Unterschiede in der Qualität der Ergebnisse war hier weniger ein Problem als bei den Gleitkommazahlen, dafür ist die Anzahl an Bits und dafür der Wertebereich zum Teil Sprach- beziehungsweise Plattformabhängig. Gängige Formate sind zum Beispiel *int*, *unsigned* oder *long*. In Field-Programmable Gate Array (Deutsch: Feldprogrammierbares Gate-Array)s (FPGAs) finden sich aber noch immer sehr spezifische Zahlenformate, welche optimal auf den Zahlenbereich und die benötigte Genauigkeit angepasst werden können.

Um die Auswirkungen auf die Genauigkeit besser nachvollziehen zu können, gibt es im Folgenden einen kurzen Überblick über verschiedene Zahlentypen.

3.2.1 Ganze Zahlen

In diesem Abschnitt werden gebräuchlich verwendete Darstellungen von ganzen Zahlen auf Computern, auch Integer Zahlen genannt, kurz vorgestellt.

Die am Computer darstellbaren Integer Zahlen entsprechen den bekannten ganzen Zahlen $\mathbb{Z}$ auf Zweier ($\mathbb{Z}_2$) anstatt Zehner ($\mathbb{Z}_{10}$) Basis, in einem von der Anzahl an Bits abhängigen Wertebereich. Wie in Abbildung 3.2 zu sehen stellen in einem Integer mit 32 Bit die ersten 31 Bits ganz normal die Zahl in Zweier-Basis dar, speziell ist hierbei nur das 32. Bit. Dieses bestimmt das Vorzeichen und entspricht dem *Wert* -2^{32}. Die Schrittweise zwischen zwei möglichst nahen Zahlen ist immer 1. Die kleinste darstellbare Zahl ist −2.147.483.648, die Größte 2.147.483.647. Um aus einer negativen Zahl eine positive Zahl zu machen, wird jedes Bit gedreht und plus eins auf das Ergebnis addiert, dieses System nennt sich Zweierkomplement. Die Reihenfolge der gespeicherten Bits ist nicht vorgegeben, typische Formate sind *Big-Endian* (größte Zweierpotenz *links*) und *Little-Endian* (größte Zweierpotenz *rechts*). Bei rein positiven Zahlenformaten wie *unsigned* fällt beim 32. Bit das negative Vorzeichen weg, die darstellbaren Zahlen gehen dann von 0 bis $2^{32} - 1$. Im Allgemeinen lassen sich für n-Bits $2^n - 1$ Zahlen korrekt darstellen. Wird der Zahlenbereich unter-

beziehungsweise überschritten, spricht man von einem sogenannten under- und overflow. Wird auf ein 8-Bit Integer mit dem Wert 127 eins addiert, wird −128 erhalten, bei einem 8 Bit Unsigned Integer ergibt 255 + 1 = 0. Die genaue Umsetzung in den Programmiersprachen ist allerdings unterschiedlich. So führt bei C++ ein over- oder underflow eines Integers zu undefined behavior, während bei unsigned Integers das hier beschriebene Modulo Verhalten auftritt.

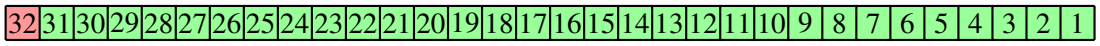


Abbildung 3.2: 32 Bit Integer in Big-Endian Darstellung.
Jedes Bit repräsentiert den Wert 2^{Stelle}, das Bit an Stelle 32 den Wert -2^{32} und bestimmt damit das Vorzeichen

3.2.2 Gleitkommazahlen

Im Gegensatz zu Integer Zahlen können Gleitkommazahlen einen deutlich größeren Zahlenbereich abdecken und auch mit Kommazahlen arbeiten, dafür sind die meisten Zahlen nur Näherungen des exakten Wertes. Normalisierte Gleitkommazahlen lassen sich wie folgt definieren [Gol91]

$$\mathbb{F}_{\beta,t} := \{(-1)^s \cdot m \cdot \beta^e \mid m = 0 \ \cup \beta^{t-1} \leq |m| < \beta^t, \quad s \in \{0,1\}, m, t \in \mathbb{N}, \beta \in \mathbb{N} \setminus \{0,1\}, e \in \mathbb{Z}\}, \tag{3.16}$$

dabei ist β die Basis, m die Mantisse, e der Exponent und s das Vorzeichen. Um eine eindeutige Darstellung zu gewährleisten wird zusätzlich die Zahl normalisiert, so dass keine führenden Nullen erlaubt sind. Für $\beta = 10$ und $|m| = 3$ wären ansonsten $(-1)^0 \cdot 002 \cdot 10^2 = (-1)^0 \cdot 200 \cdot 10^0$ beides korrekte Darstellungen der gleichen Zahl 200.

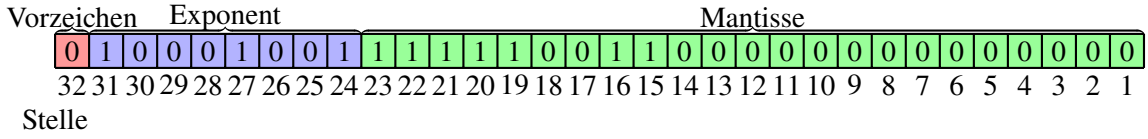


Abbildung 3.3: 32 Bit Gleitkommazahl nach *IEEE 754* Spezifikation.
Die ersten 23 Bit sind die Mantisse, die nächsten 8 der Exponent und das letzte Bit das Vorzeichen. Dargestellt wird die Zahl 2022.

Wie beschrieben wurde mit der Norm *IEEE 754* die Darstellung und das Verhalten von Gleitkommazahlen weitgehendst festgelegt. Für eine typische *single-* oder *binary32*-Gleitkommazahl ist, wie in Abbildung 3.3 zu sehen, die Basis $b = 2$, es gibt 1 Bit für das Vorzeichen, die Länge der Bits für den Exponenten e ist 8 und die Länge der Mantisse m ist 24, wobei nur 23 Bits gespeichert werden und das erste Bit ein sogenanntes *hidden bit* ist, das durch die Normalisierung immer 1 wäre und somit weggelassen wird. Um einen negativen Exponenten darstellen zu können, wird im Gegensatz zu den Integern kein Zweierkompliment, sondern ein Exponenten Bias genutzt. Dabei wird nicht der eigentliche Exponent e gespeichert, sondern $E = e + B$, dabei ist B der Biaswert und berechnet sich aus dem kleinstmöglichem Exponenten. Bei einem *single* ist der Bias $B = 2^{8-1} - 1 = 127$ und transferiert jeden möglichen biased Exponenten E in die positiven Zahlen. Der eigentliche Exponent ist dann $e = E - B$. Zusätzlich gibt es 2 Sonderfälle für den Exponenten, der Wert $E = 0$ wird freigehalten für den Gleitkommazahl-Wert 0, da durch die Normalisierung der Mantisse dieser Wert nicht auf normalem Weg erreicht werden kann. Der andere

Fall ist für $E = 255$, dieser ist für Sonderfälle wie Not a Number (Deutsch: Keine Zahl) (NaN) reserviert. Dieser Fall kann zum Beispiel bei einer Division mit 0 oder dem Ziehen der Wurzel einer negativen Zahl auftreten. Dadurch kann e Werte zwischen -126 bis 127 annehmen. Insgesamt können *binary-32*-Gleitkommazahlen betragsmäßige Werte zwischen $\approx 1.2 \cdot 10^{-38}$ bis $\approx 3.4 \cdot 10^{38}$ darstellen, was häufig ein deutlicher größerer und praktischerer Bereich für das Rechnen mit nicht weiter spezifizierten Zahlen darstellt.

Um die Zahl aus Abbildung 3.3 in eine Dezimalzahl umzustellen, werden das Vorzeichen, der Exponent und die Mantisse betrachtet. Diese sind $s = 0$, $E = 137$ und inklusive *hidden Bit* $m = 1,11111001100000000000000_2$. Mit dem Bias lässt sich $e = 137 - 127 = 10$ bestimmen, dadurch wird die Mantisse um 10 stellen nach links verschoben, beziehungsweise das Komma wandert 10 Stellen nach rechts, daraus ergibt sich $11111100110_2 = 2022_{10}$.

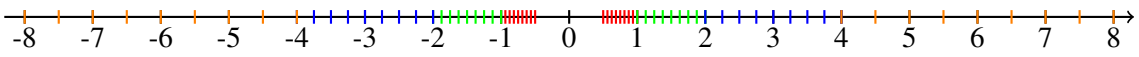


Abbildung 3.4: Darstellbare Zahlen für eine Gleitkommazahl mit Basis 2, $e \in \{-1, \ldots, 2\}$, 3 Bits für die Mantisse und ein Bit für das Vorzeichen. Dabei sind die möglichen Zahlen für $e = -1$ in Rot, $e = 0$ in Grün, $e = 1$ in Blau und $e = 2$ in Orange dargestellt.

Bei Integer Zahlen sind die Abstände zur nächstgelegenen Zahl gleichmäßig, bei den Gleitkommazahlen hängt der Abstand zwischen den Zahlen vom Exponenten ab. Für größere Exponenten wird der Abstand zwischen zwei Zahlen immer größer. Wie in Abbildung 3.4 zu sehen, gibt es aber auch einen größeren Bereich um 0, bei der eine große Ungenauigkeit vorherrscht. Um diese Zahlen besser darzustellen, gibt es die subnormalen Zahlen, auf die hier aber nicht genauer eingegangen wird. Aber auch Zahlen wie zum Beispiel 0.1 können nicht genau dargestellt werden, sondern nur mit einem Fehler in der Größenordnung von 10^{-9}, da sie kein Vielfaches der Basis sind.

Auch gelten viele Rechengesetze, die im Alltag als natürlich angesehen werden, nicht. So sind zum Beispiel Gleitkommazahlen nicht geschlossen, sie sind nicht assoziativ oder distributiv und es kommt zu Effekten wie der Auslöschung oder Absorption 3.2.3, welche im weiteren Verlauf des Kapitels vorgestellt werden.

Eine ausführlichere und lesenswerte Zusammenfassung zu diesem Thema gibt es von Goldberg [Gol91].

3.2.3 Umsetzung gemischte Genauigkeiten

Die Umsetzung der Nutzung von gemischten Genauigkeiten ist sehr unterschiedlich und problemabhängig. Bei neuronalen Netzen hat sich herausgestellt, dass viele Operationen, beziehungsweise einzelne *Layer* in niedriger Genauigkeit berechnet werden können. So wird bei den neuesten GPUs nicht nur die Leistung für *double*- oder *single*-Gleitkommaoperationen pro Sekunde angegeben, sondern auch für Datentypen wie *FP16/halfprecision*, *BF16/Brain Floating Point*, *FP8* oder *INT8* [NVI22d]. Dabei liegt die Anzahl der Operationen pro Sekunde für Matrixoperationen, auf die in Abschnitt 3.3 genauer eingegangen wird, auf der NVIDIA A100 Grafikkarte zwischen 19,5 TFLOPS für FP64 und 312 TFLOPS für FP16. Es gibt sogar neuronale Netze mit *Layern*, die eine noch niedrigere Genauigkeit nutzen wie INT4, also ein Integer mit nur 4 Bits. Hier gilt es für die Problemstellung bei gemischten Genauigkeiten herauszufinden, welche Operationen wie berechnet werden

müssen. Für neuronale Netze ist dieses Vorgehen recht verbreitet und funktioniert inzwischen sogar automatisiert während des Trainingsvorgangs ohne speziell erforderliches Wissen oder Eingabe des Nutzers.

Für andere Anwendungen im Bereich High-performance computing (Deutsch: Hochleistungsrechnen) (HPC), worunter zum Beispiel eine ganze Reihe unterschiedlicher Simulationen gehören, unterscheiden sich die Ansätze, der Nutzen und das mathematische Modell deutlich. Die Unterschiede liegen häufig an der Stabilität der einzelnen Algorithmen, der Kondition oder auch an Sicherheitsvorschriften, wie in der Luftfahrt.

Eine sehr einfache Möglichkeit bei iterativen Verfahren ist es, die Lösung bis zu einem gewissen Fehler in niedriger Genauigkeit zu approximieren und dann mit einer höheren Genauigkeit bis zur endgültig gewünschten Genauigkeit nachzuiterieren. Ein anderes Vorgehen ist, für einzelne Operationen abzuschätzen, wie sehr sich hier eine Verringerung der Genauigkeit auf den Fehler auswirken kann und dann demnach verschiedene Operationen in verschiedener Genauigkeit zu berechnen. Um das Problem schneller lösen zu können, liegt dabei der Fokus besonders auf den rechenintensiven Teilen des Algorithmus. Dabei hat sich gezeigt, dass sich unterschiedliche Rechenoperationen verschieden stark auswirken [Gol91].

Multiplikation

Bei der Multiplikation zweier Gleitkommazahlen wird das Vorzeichen Bit durch eine XOR-Operation der beiden Ausgangszahlen berechnet. Die Exponenten werden addiert und der Bias abgezogen. Die beiden Mantissen der Länge m werden multipliziert und normiert. Theoretisch würde dabei eine neue Mantisse der Länge $2 \cdot m + 1$ entstehen, aber es kann als Ergebnis nur wieder eine Mantisse der Länge m gespeichert werden. Das Ergebnis wird daher gerundet. Dadurch geht bei der Multiplikation generell ein Teil der Genauigkeit verloren.

Dieser Umstand wird auch von aktueller Hardware ausgenutzt, zum Beispiel bei den Matrixeinheiten der NVIDIA A100 (Abschnitt 3.3). Dort werden zum Beispiel in einem Modus FP32-Zahlen vor der Multiplikation konvertiert in den Datentyp Tensor Float Number (Deutsch: Tensor-Gleitkommazahl) mit 19 Bits (TF32). Dieser hat im Exponenten die gleiche Anzahl an Bits wie eine FP32 Gleitkommazahl und die Mantisse hat die gleiche Länge wie die Mantisse einer FP16 Gleitkommazahl. Die Mantisse ist also massiv kürzer, der große allgemeine Zahlenbereich bleibt aber erhalten und bei der Multiplikation geht nur ein geringer Teil an Informationen verloren, da die hier entstehende Mantisse nicht auf die halbe Länge gekürzt werden muss. Die einzelnen unterstützten Formate unterscheiden sich aber je nach Hersteller oder auch zwischen den Hardware-Generationen. So unterstützt AMD kein TF32 Format und NVIDIA kein ausschließlich FP32 nutzendes Format.

Addition

Bei der Addition verhält es sich anders. Hier gibt es speziell 2 bekannte Fälle, die Probleme bereiten können.

Da ist einerseits die Auslöschung. Diese tritt auf, wenn zwei ähnlich große Zahlen voneinander subtrahiert werden. Gegeben zwei FP16-Zahlen a und b, die ein bereits *verrauschtes* Zwischenergebnis oder eine nicht exakt darstellbare Gleitkommazahl darstellen. Seien der Exponent und

das Vorzeichen der beiden Zahlen identisch. Die Mantisse für FP16 hat eine Länge von 10. Sei $m_a = 0101010111$ die Mantisse von a und dementsprechend $m_b = 0101010100$ die Mantisse von b, dann ergibt sich

$$\begin{array}{r} 0101010111 \\ -0101010100 \\ \hline 0000000011 \end{array}$$

und dadurch eine Verschiebung des Exponenten. Es ergibt sich in diesem Falle eine verbleibende Genauigkeit von 2 Stellen der Mantisse beziehungsweise ein möglicher Fehler von 8 Stellen und somit von 12.5%. Im Extremfall können sich die beiden Zahlen komplett auslöschen und das Ergebnis wird 0, was sich in potenziell nachfolgenden Berechnungen stark auswirken kann.

Der andere Fall ist die Absorption. Diese tritt auf, wenn große und kleine Zahlen addiert oder subtrahiert werden. Sind die Exponenten zweier Gleitkommazahlen um mehr Stellen voneinander unterschieden als die Mantisse lang ist, dann hat eine Addition keine Auswirkung auf die größere Zahl. Das Ergebnis ist dann schlicht die größere Zahl. In weniger stark ausgeprägten Fällen, also wenn die Exponenten nicht zu weit voneinander entfernt sind, gehen nur einige Stellen an Informationen verloren. Kritisch kann es dann werden, wenn viele Zahlen aufsummiert werden. Um diesen Fehler zu minimieren, gibt es eigens entwickelte Algorithmen wie Kahans-Summierungsalgorithmus [Kah65] oder Ansätze wie die paarweise Addition, bei der die Zahlen wie in einem umgekehrten Binärbaum aufaddiert werden, und weitere, die in der Arbeit von Higham [Hig93] analysiert werden. Diese Algorithmen können zum Teil auch miteinander verbunden werden, oder die paarweise Addition wird aus Gründen der Performanz nicht in Zweierpaare unterteilt, sondern in größere Untergruppen, die dann direkt oder hierarchisch aufsummiert werden.

3.3 Grafikprozessoren und Matrixeinheiten

GPUs verschiedener Hersteller unterscheiden sich zwar im Detail, in der eigentlichen Funktionsweise sind sie aber sehr ähnlich. Auch die Programmierung mittels HIP oder CUDA unterscheidet sich kaum. Um Redundanzen zu vermeiden und auch konkrete Beispiele nennen zu können, basieren die Erklärungen auf der CUDA-Plattform und einer aktuellen NVIDIA A100 GPU [NVI20], welche auch Grundlage für einen Großteil der Implementierungsaufgaben waren.

Die Hauptaufgabe einer GPU ist, wie es der Name schon sagt, die Computergrafik anzuzeigen und zu berechnen. Seit Ende der 1990er Jahre, mit dem Aufkommen von 3D-Computerspielen, haben GPUs dafür eine spezielle Hardware-Pipeline um Schatten, Rasterisierung oder die einzelnen Farbwerte zu berechnen. Da einzelne Berechnungen für mehrere Tausend Pixel wiederholt werden mussten, arbeiteten schon damals Grafikkarten hoch parallel die Pixel ab, aber auch die Farben rot, grün, blau und ein alpha-Wert wurden gleichzeitig berechnet. Anfang der 2000er Jahre konnten aber einige Wissenschaftler GPUs für wissenschaftliche Berechnungen nutzen. Dies war relativ aufwändig und nicht optimal, da damalige GPUs nicht für solche Berechnungen ausgelegt waren. Im Jahr 2006 stellte NVIDIA eine neue Grafikkartengeneration vor, welche von dem bisherigen Modell abwich, etablierte die CUDA-Plattform [NVI22c] und die Unterstützung von FP64. GPUs waren nicht mehr ausschließlich für Grafikberechnungen ausgelegt, sondern waren über die mitgelieferte

CUDA-Plattform frei programmierbar und direkt ähnlich wie C zu programmieren. Der einfache Zugang, die große Verbreitung von GPUs und die theoretisch deutlich höhere Leistung gegenüber CPUs machte den Weg frei für GPGPU.

3.3.1 Funktionsweise

Ein typisches CUDA-Programm lässt sich aufteilen in Host und Device Code. Der Host-Code ist typischerweise für die Initialisierung, den Programmfluss und einfache, nicht parallel ausführbare Berechnungen zuständig und wird auf der CPU ausgeführt. Dieser Teil nimmt häufig einen Großteil des geschriebenen Codes an, allerdings nur einen geringen Anteil der eigentlichen Berechnungen. Diese rechenintensiven Teile werden dann auf die GPU beziehungsweise das Device ausgelagert. Dafür belegt die CPU Speicher auf der GPU, transferiert alle notwendigen Daten auf diese und startet dann einen CUDA-Kernel, der die Berechnungen auf der GPU steuert und ausführt. Dieser Kernel hat im Gegensatz zu einer normalen Funktion noch weitere durch <<<. . . >>> gekennzeichnete Parameter, welche unter anderem die Häufigkeit und die Einteilung der Ausführung des Kernels bestimmen.

Thread-Hierarchie

Dafür gibt es eine Hierarchie aus Threads, Blöcken und einem Grid: Mehrere Threads bilden dabei einen Block und mehrere Blöcke ein Grid, dabei können die Indizes für Blöcke und Threads jeweils, wie in Listing 3.1 zu sehen, 3 dimensional sein, was die Aufteilung der Gesamtrechnung in einzelne Blöcke und Threads bei mehrdimensionalen Problemen vereinfacht. Ein einzelner Block kann jeweils bis zu 2048 Threads enthalten, ein Grid bis zu 65535 Blöcke pro Dimension (NVIDIA A100 [NVI20]).

Listing 3.1 Einfaches CUDA-Kernel-Aufruf-Beispiel mit dim3 und skalaren Integer.

```
1   dim3 threadsPerBlock(1,2,4);
2   dim3 numBlocks(4,2,1);
3   MyKernelA<<<numBlocks, threadsPerBlock>>>();
4   MyKernelB<<<8, 100>>>();
```

Jeder Block wird lokal auf einer Einheit berechnet. Diese grundlegende Einheit auf Grafikkarten ist der SM. Dieser verfügt über 4 Streaming Processors (SPs), die sich 64 Integer-, 4 Matrix-, 64 FP32- und 32 FP64-Gleitkommazahlen-Recheneinheiten, mehrere Register und andere Dinge untereinander in gleichen Verhältnissen aufteilen. Zusätzlich befindet sich auf jedem SM sogenannter geteilter Speicher (Englisch: Shared Memory) und der Cache, die für alle SPs dienen. Ein einzelner SM entspricht dadurch in gewisser Weise einer simplen Mehrkern-CPU.

Bei der aktuellen Nvidia A100 GPU werden weiterhin 2 SMs zu einem *Texture Processor Cluster* (TPC) zusammengefasst und 8 TPCs zu einem *GPU processing clusters* (GPC), wovon die GPU im Ganzen 7 Aktive enthält. Insgesamt hat diese GPU somit 108 SMs und 3456 FP64 Gleitkommaeinheiten [NVI20]. Die einzelnen FP32 oder FP64 Recheneinheiten werden auch *CUDA-Cores* genannt. Die Thread-Blöcke werden bei einem Kernelaufruf gleichmäßig auf die SMs der GPU aufgeteilt

Der SM bündelt jeweils 32 aufeinanderfolgende Threads zu einem Warp. Die Threads eines Warps durchlaufen den Programmfluss als eine Einheit. Gibt es Divergenzen, zum Beispiel durch ein *if else*-Konstrukt, durchläuft jeder Thread alle Pfade, die zumindest 1 Thread des Warps begeht (Single Instruction, Multiple Threads (Deutsch: Einzelne Anweisung, mehreren Threads) (SIMT)), was die Geschwindigkeit deutlich reduzieren kann. Dabei ist jeder Thread nur dann aktiv, wenn die Bedingung für diesen Pfad erfüllt wird. Folglich gilt es innerhalb eines Warps die Divergenzen zu minimieren.

Auf einem SM können, wenn genügend Ressourcen vorhanden sind, bis zu 32 Blöcke parallel berechnet werden. Ansonsten, falls es mehr Blöcke als Ressourcen auf den SMs gibt, was die Norm ist, werden diese nacheinander abgearbeitet. Ein Vorteil bei den so gestalteten Kernelaufrufen ist, dass sich der Programmierer bezüglich der Skalierbarkeit auf einer GPU nur bedingt beschäftigen muss. Es genügt das Programm in voneinander unabhängige Blöcke aufzuteilen und diese für die SMs zu optimieren, die Skalierung und Verteilung auf die einzelnen SMs übernimmt dann der Compiler. Es ist bereits angekündigt, dass für die kommende Grafikkartengeneration eine weitere Hierarchiestufe, der Thread-Block-Cluster [NVI22d], hinzukommt, um noch besser die Lokalitäten des Speichers und Caches ausnutzen zu können.

Auf einem SM können zwar nur 4 Warps simultan aktiv sein, trotzdem wird versucht weitere Warps gleichzeitig auf einem SM laufen zu lassen. Im Gegensatz zu einer CPU sind die Threads sehr leichtgewichtig und ein Wechsel von einem Warp zum anderen geht ohne großen Overhead. Diese Eigenschaft wird ausgenutzt um Latenzen, die zum Beispiel beim Warten auf Daten anfallen, zu verbergen. Da die Reihenfolge, welche Warps wann aktiv sind, nicht deterministisch ist, ist es häufig bei Operationen auf dem geteilten Speicher wichtig, den Block zu synchronisieren. Dabei wird jeder Warp eines Blockes an einem Synchronisationspunkt angehalten, bis jeder Warp an dieser Stelle ist. Dies produziert zwar Overhead, ist aber unumgänglich, wenn zum Beispiel von jedem Thread Daten in den geteilten Speicher geladen werden, die nachher von potenziell anderen Threads genutzt werden. Der Befehl für die Synchronisation lautet `__syncthreads();` .

Speicher-Hierarchie

Ebenso wichtig wie die Thread-Hierarchie ist die Speicher-Hierarchie. Der Speicher unterscheidet sich dabei nicht nur durch seine Latenz, sondern auch durch seine Zugehörigkeit. Die wichtigsten Speicherarten sind dabei folgende [NVI22b]:

- **Register:** Die Register liegen direkt auf dem SP und werden gleichmäßig auf alle aktiven Threads verteilt. Jeder Thread hat dabei seine eigenen privaten Register und für einen Zugriff wird kein extra Taktzyklus benötigt. Insgesamt stehen pro SM 65536 an 32 Bit-Registern zur Verfügung, wobei ein Thread maximal 255 Register belegen kann.
  Innerhalb eines Kernels kann eine Variable wie gewohnt zum Beispiel mit `int var = 1;` initialisiert werden.
- **Geteilter Speicher:** Der geteilte Speicher sitzt auf dem SM. Jeder Thread eines Blockes hat auf den geteilten Speicher Zugriff. Die Zugriffszeiten sind gering und liegen im niedrigen einstelligen Taktbereich. Die Nvidia A100 GPU hat 48KB statischen geteilten Speicher, es kann allerdings auf Kosten der Größe des L1 Caches der geteilte Speicher auf bis zu 163 kB

erhöht werden.
Eine Variable im geteilten Speicher wird durch das Schlüsselwort __shared__, also zum Beispiel mit `__shared__ int var_s = 1;` im Kernel initialisiert.

- **Globaler Speicher:** Der globale Speicher liegt außerhalb des GPU Chips, vergleichbar mit dem Hauptspeicher der CPU. Es ist der größte Speicher, der allerdings auch eine sehr hohe Latenz aufweist, die bis zu Faktor 100 größer sein kann als beim geteilten Speicher, wobei einige professionelle Karten wie die A100 mit ihrem High Bandwith Memory der zweiten Generation (HBM2) die Lücke verkleinern. Auf den globalen Speicher hat jeder Thread Zugriff und er wird auch benutzt um Daten zwischen CPU und GPU zu transferieren.
  Globaler Speicher wird durch das Schlüsselwort `__device__` initialisiert.
- **Lokale Speicher:** Im Gegensatz zum Namen ist der lokale Speicher nicht lokal, sondern befindet sich wie der globale Speicher außerhalb des Chips beziehungsweise ist er ein Teil des globalen Speichers. Falls der Compiler davon ausgeht, dass die Register nicht ausreichen, können zum Beispiel Speicherintensive Arrays automatisch ausgelagert werden. Vorteil am Lokalen Speicher ist, dass dieser gecached werden kann.
  Primär werden zum Beispiel große auf den Registern gespeicherte Arrays wie `int var[128];` automatisch und ohne großen Einfluss des Programmierers zu lokalem Speicher gemacht.
- **Constant und Texture Speicher:** Diese Speicher sitzen ebenfalls außerhalb des Chips, können gecached werden, aber sind nur *read only*. Diese sind genau dann effektiv, wenn viele Threads auf ein oder dieselbe Variable beziehungsweise Konstante zugreifen. In diesem Fall sind sie ähnlich effektiv wie Register, sparen aber diesen Platz. Eine weitere Besonderheit ist, dass zum Beispiel eine Konstante auf alle Threads gleichzeitig geladen werden kann.

CUDA-Minimalbeispiel

In Beispiel Listing 3.2 sollen zwei Arrays oder Vektoren addiert werden. In der Main-Methode werden dafür zuerst 3 Arrays initialisiert und mit Daten gefüllt. In line 20 werden 3 Pointer definiert. Mit dem Befehl `cudaMalloc(void** devPtr, size_t size)` wird diesen in line 23 Speicher zugewiesen. Die Pointer zeigen durch diesen Befehl auf den Beginn ihres Speichers. In line 26 wird mit dem Befehl `cudaMemcpy (void* dst, const void* src, size_t count, cudaMemcpyKind kind )` V1 vom Host auf das Device kopiert. Dabei wird das Ziel, die Quelle, die Größe und die Art des Transfers, ob von HostToDevice oder von DeviceToHost. Im nächsten Schritt wird das Problem in Blöcke mit jeweils 64 Threads, also 2 Warps pro Block, unterteilt. Damit die Threads auf den richtigen Daten rechnen, besitzt jeder Thread eine jeweils 3 dimensionale eigene ThreadId und BlockId. Für eindimensionale Daten wird in diesem Fall eine typische Aufteilung vorgenommen: Die Threads pro Block werden so festgelegt, dass eine effiziente Berechnung auf einem SM stattfinden kann. Die Anzahl an Blöcken ergibt sich dann aus der Größe des Gesamtproblems geteilt durch die Threads pro Block: Indem der Index im Kernel in line 4 aus `blockIdx.x * blockDim.x + threadIdx.x;` berechnet wird, wobei `blockDim.x` die Anzahl an Threads in der x-Koordinate und damit 64 entspricht, wird jedes der N Elemente des Vektors einmal bearbeitet. Nach Ausführung des Kernels wird das Ergebnis wieder zurück auf den Host kopiert.

3.3.2 Programmier-Herausforderungen

Die theoretische Rechenleistung einer GPU ist sehr hoch. Allerdings eignen sich viele Probleme oder Algorithmen gar nicht für den parallelen Aufbau der GPU oder es müssen einige Änderungen vorgenommen werden. Um eine möglichst hohe Leistung zu erreichen, sollten einige Dinge bekannt sein und beachtet werden.

- **Parallelisierbarkeit:** Das Problem muss sich parallel lösen lassen. Dafür muss es in kleine Unterprobleme aufgeteilt werden können, die relativ unabhängig voneinander gelöst werden können. In vielen Fällen ist es dafür nötig vom optimalen seriellen Algorithmus abzuweichen. Solange sich der Mehraufwand in Grenzen hält, überwiegt der Geschwindigkeitsvorteil der GPU aber deutlich. Dabei verhält es sich häufig ähnlich zur Parallelisierung auf CPUs.

Listing 3.2 CUDA Minimalbeispiel anhand einer Vektoraddition. Dabei werden Arrays auf der CPU und GPU angelegt, die Daten auf die GPU kopiert, auf dieser berechnet, und das Ergebnis zurück auf die CPU kopiert.

```
// GPU Code, markiert durch __global__
__global__ void VecAdd(float* V1, float* V2){
   // Index-Berechnung
   int index = blockIdx.x * blockDim.x + threadIdx.x;
   // Eigentliche Rechenoperation
   V1[index] = V1[index] + V2[index];
}
// Host/CPU Code
int main(){
   // N is length of the vectors and a multiple of 64
   size_t N = 192;
   // Initialisierung 3er Vektoren für die CPU
   float* V1_host = (float*)malloc(N * sizeof(float));
   float* V2_host = (float*)malloc(N * sizeof(float));
   for(size_t i = 0; i < N; ++i){
      V1_host[i] = i;
      V2_host[i] = N - i;
   }
   // Anlegen von Pointer auf der GPU
   float* V1_device;
   float* V2_device;
   // Speicherreservierung auf der GPU
   cudaMalloc(&V1_device, N * sizeof(float));
   cudaMalloc(&V2_device, N * sizeof(float));
   // Kopieren von V1 und V2 auf die GPU
   cudaMemcpy(V1_device, V1_host, N * sizeof(float), cudaMemcpyHostToDevice);
   cudaMemcpy(V2_device, V2_host, N * sizeof(float), cudaMemcpyHostToDevice);
   // Aufteilung in Blöcke der Länge 64
   int threadsPerBlock = 64;
   int numBlocks = N / threadsPerBlock;
   // Start des CUDA-Kernels
   VecAdd<<<numBlocks, threadsPerBlock>>>(V1_device, V2_device);
   // Zurückkopieren des Ergebnisses auf das Host System
   cudaMemcpy(V1_host, V1_device, N * sizeof(float), cudaMemcpyDeviceToHost);
}
```

- **Rechenintensität:** Eine Nvidia A100 hat eine für CPUs unerreichte Speicherbandbreite von 1555 GB/Sekunde und dazu eine Rechenleistung von 9.7 FP64 TFLOPS.

$$\frac{8\,\mathrm{B} \cdot 9{,}7\,\mathrm{TFLOPS}}{1555\,\mathrm{GB/sec}} \approx 50\,\mathrm{FLOPS\,double}^{-1} \tag{3.17}$$

  Es muss folglich jede FP64-Zahl, die aus dem globalen Speicher geladen wird, im Durchschnitt in 50 Operationen verwendet werden, dass die theoretische Rechenleistung erreicht werden kann.

- **Aufteilung:** Die Aufteilung in Blöcke sollte möglichst einfach und wenig rechenintensiv sein. Im Optimalfall reichen berechnete Indizes, die sich relativ direkt aus den jedem Thread innewohnenden bis zu 3 Dimensionen großen Variablen ThreadIdx und BlockIdx ergeben. Ein beispielsweise 2 dimensionales Gitter mit 1024 Einträgen pro Dimension lässt sich gut unterteilen in 1024 Blöcke von 32×32 Gitterpunkten, die direkt über ihre Thread- und Block-Id identifiziert werden können. Ein weiteres typisches Konstrukt für Algorithmen auf Gittern, die abhängig sind von ihren Nachbarzellen, ist ein *Halo*. Hierbei werden in jeden Block zusätzlich noch die benötigten Nachbarzellen der Ränder in den *Halo* geladen und für die Berechnung der Ränder des Blockes genutzt, die Datenpunkte im *Halo* werden allerdings nicht weiter berechnet, sondern werden aus den Nachbarblöcken für die nächste Iteration geladen. Diese *Halos* sind ebenso ein Beispiel für zusätzlichen vertretbaren Mehraufwand um Algorithmen parallelisieren zu können.

- **Register:** Bei mehreren Hilfsvariablen oder längeren Schleifen-Entrollen werden viele Register benötigt. Dies kann dazu führen, dass Variablen in den lokalen Speicher ausgelagert werden, weniger Blöcke gleichzeitig auf einer SM aktiv sein können oder der Kernel gar nicht erst startet. Eine genaue Abschätzung für den Registerbedarf ist dabei gar nicht trivial, so können kleine Codeänderungen zu einem ganz anderen Registerverhalten führen. Für genaue Informationen sollte daher ein Profiling Tool verwendet werden.

- **Warp Divergenzen:** Es gilt Warp Divergenzen möglichst zu vermeiden, da alle Pfade, die mindestens 1 Thread bestreitet, vom ganzen Warp bestritten werden. Um Divergenzen zu vermeiden, bietet es sich manchmal an, einen kompletten Warp einen Teil einer If-Else-Anweisung ausführen zu lassen und den anderen Teil von einem anderen Warp.

- **Speichermanagement:** Der vermutlich wichtigste Part ist das Speichermanagement. Dabei gilt es effektiv die für den Block notwendigen Daten aus dem globalen Speicher zu laden. Diese Daten sollten möglichst ausgerichtet und hintereinander im Speicher vorliegen. Im Falle der Nvidia A100 ist laut Nvidia hierbei eine Ausrichtung und Transaktionsgröße von 128 Bit pro Thread optimal, was einer Cacheline pro 8 Threads entspricht [NVI22b]. Wird ein 2 dimensionales Feld allerdings in einem Array gespeichert, liegen die Daten entweder nur in X- oder Y-Dimension fortlaufend vor. In diesem Fall kann es sinnvoll sein, die Daten blockweise in den geteilten Speicher zu laden, von wo aus dann jeder Thread die von ihm benötigten Daten schnell und effizient laden kann. Aber auch beim geteilten Speicher gibt es mehrere Dinge zu beachten. So hat dieser pro SM 32 Speicherbänke, wobei eine Zelle jeweils 32 Bit umfasst. Wird ein Array in den Speicher geladen, so landen die ersten 32 Bit in der ersten Speicherbank, die zweiten 32 Bit in der zweiten Speicherbank und so weiter, bis nach 32 Speicherbänken wieder in die erste Bank geladen wird. Bei jedem Laden und Speichern aus oder in den geteilten Speicher kann jeweils jede Bank nur eine Operation ausführen. Liegen folglich alle benötigten Daten eines Warps in einer Bank, da der Block zum Beispiel 32×32

groß ist und ein 2 dimensionales Feld geladen wurde, kommt es zum Bank-Konflikt und die Aufrufe werden hintereinander abgearbeitet. Da durch die Größe des Warps die Dimension der Blöcke häufig ein Vielfaches von 32 sind, kommt dieses Problem häufig vor und lässt sich durch einen Offset abmildern oder komplett beheben. Soll also weiterhin ein 32×32 großes Feld in den geteilten Speicher geladen werden, so kann dieser Array als [32][33] großes Feld angelegt werden und das 33. Feld würde auf die gleiche Bank wie der erste Eintrag gehen, wird aber leer gelassen. Dies führt pro Zeile um eine Verschiebung um eins, lädt ein Warp jetzt die Elemte mit Index `array[ThreadIdx.x][0]` aus dem geteilten Speicher, greift jeder Thread auf eine einzelne Bank zu, genauso wie bei einem Aufruf der Form `array[0][ThreadIdx.x]`. Seit CUDA 11 gibt es zudem noch asynchrone Kopieroperationen `cuda::memcpy_async()` in den geteilten Speicher, diese werden im Kernel gestartet, der Thread läuft aber weiter und macht die Speicheroperationen im Hintergrund, bis er auf eine Barriere `bar.arrive_and_wait();` trifft, bei der alle Threads des Blocks synchronisiert werden und auf die Fertigstellung der asynchronen Kopiervorgänge gewartet wird. Ein weiterer Vorteil an diesen ist, dass die Daten nicht erst über den L2-Cache, L1-Cache und Register in den geteilten Speicher geladen werden, sondern direkt über den L2-Cache in den geteilten Speicher gehen.

- **Latenzen verbergen:** Bei einem typischen Kernel werden zuerst Daten aus dem globalen Speicher geladen, diese dann verarbeitet und am Ende zurück in den globalen Speicher geschrieben. Um zwischendurch größere Latenzen zu vermeiden, werden viele Daten blockweise, zum Beispiel zu Beginn oder pro Iteration geladen. Damit die SM in dieser Zeit nicht nur auf die Daten wartet, gibt es 2 größere Strategien: Entweder wird versucht die Daten asynchron zu laden, was allerdings weder zu Kernel Beginn noch Ende funktioniert, oder, wenn genügend Register und geteilter Speicher zur Verfügung stehen, dass mehrere Blöcke gleichzeitig auf einem SM aktiv sind. Während ein Block also in der *Rechenphase* ist, wartet der andere auf seine Daten und umgekehrt. In der Rechenphase werden dann Zugriffe auf den globalen Speicher vermieden und nur auf Register oder den geteilten Speicher zugegriffen. Gleichzeitig bietet sich bei For-Schleifen an, diese zu entrollen, sodass der Compiler früh genug die Daten aus dem geteilten Speicher in die Register laden kann und somit kaum Verzögerungen durch Latenzen auftreten.

- **Fehlersuche und Debugging:** Einerseits werden hauptsächlich große und rechenintensive Probleme auf GPUs portiert, anderseits ist die Reihenfolge der Ausführung eines Kernels nicht deterministisch. So ist unklar, welcher Block zu welchem Zeitpunkt beginnt und endet oder ob Indexberechnungen global oder nur für den überprüften Testfall korrekt sind. Dies macht es schwierig numerische Fehler von Programmierfehlern zu unterscheiden oder diese überhaupt zu finden. Hinzu kommt: Wird nicht explizit eine Fehlerabfrage durchgeführt, läuft das Programm nach einem Fehler einfach fehlerhaft weiter. Es wird daher eindringlich empfohlen Macros zu benutzen, welche bei jeder CUDA Host-Funktion diese auf Fehler überprüfen. Ein Beispiel für solch ein Macro wird unter anderem im CUDA-Leitfaden [NVI22a; NVI22b] vorgestellt.

- **Profiling:** Um all diese Ziele zu vereinbaren, haben die GPU-Hersteller Profiling Werkzeuge zur Verfügung gestellt. Bei Nvidia ist eines der Werkzeuge *Nsight Compute*. Dieses analysiert Kernel auf einzelne Fehler, Laufzeit, erstellt eine Roofline-Analyse, welches die theoretischen Limitierungen über Speicherbandbreite oder Rechenintensität angibt. Hinzu kommen detaillierte Speicheranalysen, Speicherbank-Konflikte, Angaben zu theoretisch möglichen Warps oder Blöcken, Register- und geteilter Speicher-Auslastung, sowie die ausgeführten

Instruktionen in ihrer Häufigkeit, um nur einige wichtige Metriken zu nennen. Das Profiling lässt sich entweder interaktiv wie ein Debugger steuern oder es wird nach dem Aufruf ein Bericht erstellt, der gleichzeitig Verbesserungsvorschläge gibt, beziehungsweise Hinweise, wie solche Probleme oder Limitierungen entstehen und wie sie eventuell behoben werden können.

3.3.3 Matrixeinheiten

Im Jahre 2009 veröffentlichte Raina et al. [RMN09] das erste Paper über ein neuronales Netz, das auf Grafikkarten trainiert worden ist. In den folgenden Jahren wurden mehrere Künstliche Intelligenz (KI)-Wettbewerbe, wie 2012 der Bilderklassifizierungswettbewerb ImageNet [KSH17], durch ein neuronales Netz gewonnen und neuronale Netze erlebten eine große Renaissance mit vielen neuen Durchbrüchen. Der rechenintensive Trainingsteil dieser Netze eignet sich sehr gut für GPUs und besteht im Großen und Ganzen aus vielen Matrixmultiplikationen.

Um diese Operationen weiter zu beschleunigen, entwickelte unter anderem Google eine eigene Tensor Processing Unit (Deutsch: Tensor-Verarbeitungseinheiten) (TPU), welche seit 2015 von Google intern genutzt wird, weiterentwickelt worden ist und seit 2018 auch für die Öffentlichkeit zugänglich ist. Diese TPUs waren spezialisiert auf diese Matrizenmultiplikationen. Aber auch die Grafikkartenhersteller entwickelten extra Hardware-Einheiten für diese Matrixoperationen und so veröffentlichte Nvidia 2017 mit der Nvidia V100 die erste Grafikkarte mit sogenannten *Tensor-Cores*, die zusätzlich auf den SMs untergebracht sind. Mit diesen Tensor-Cores kann die NVIDIA V100 bis zu achtmal schneller FP16×FP16+FP32 Matrixoperationen ausführen als über die normalen CUDA-Cores mit Fused Multiply–Add (Deutsch: Gesichertes Multiplizieren-Addieren) (FMA) Einheiten. Durch die bereits hohe Verbreitung von GPUs und darauf trainierten Netzwerken inklusive Software, ist die GPU die wichtigste Hardware-Einheit für KI und neuronale Netze im Speziellen. Inzwischen bietet Nvidia eine dritte Generation von Tensor-Cores an, welche eine deutlich höhere Möglichkeit an verschiedenen Dateiformaten bietet, und auch AMD hat nun GPUs mit in diesem Falle sogenannten *Matrix-Cores* in zweiter Generation im Sortiment (Stand September 2022). Dabei gilt: Je niedriger die Genauigkeit, desto schneller sind die Matrixeinheiten.

Funktionsweise

Gegeben Matrizen beziehungsweise Submatrizen $\mathbf{A}^{m,k}$, $\mathbf{B}^{k,n}$, $\mathbf{C}^{m,n}$ und $\mathbf{D}^{m,n}$, dann berechnet eine Matrixeinheit für bestimmte Kombinationen aus m, n, k, die primär abhängig vom Datentyp sind, in einem Schritt, der nur wenige Zyklen dauert,

$$\mathbf{D} = \mathbf{A} \cdot \mathbf{B} + \mathbf{C}, \tag{3.18}$$

wobei $\mathbf{C} = \mathbf{D}$ zulässig ist. Diese Operation wird Warp-Synchron durchgeführt, dies bedeutet, dass jeder Thread beteiligt wird und jeder Thread ein Bruchteil der Daten aus den Matrizen **A**, **B**, **C** und **D** in seinen Registern hält. Für das FP64 Format gibt es auf der Nvidia A100 nur eine valide Größe für m, n und k : 8, 8 und 4. Jeder Thread hält hierbei eine Zahl aus **A** und **B** und 2 Zahlen aus **C** und **D**.

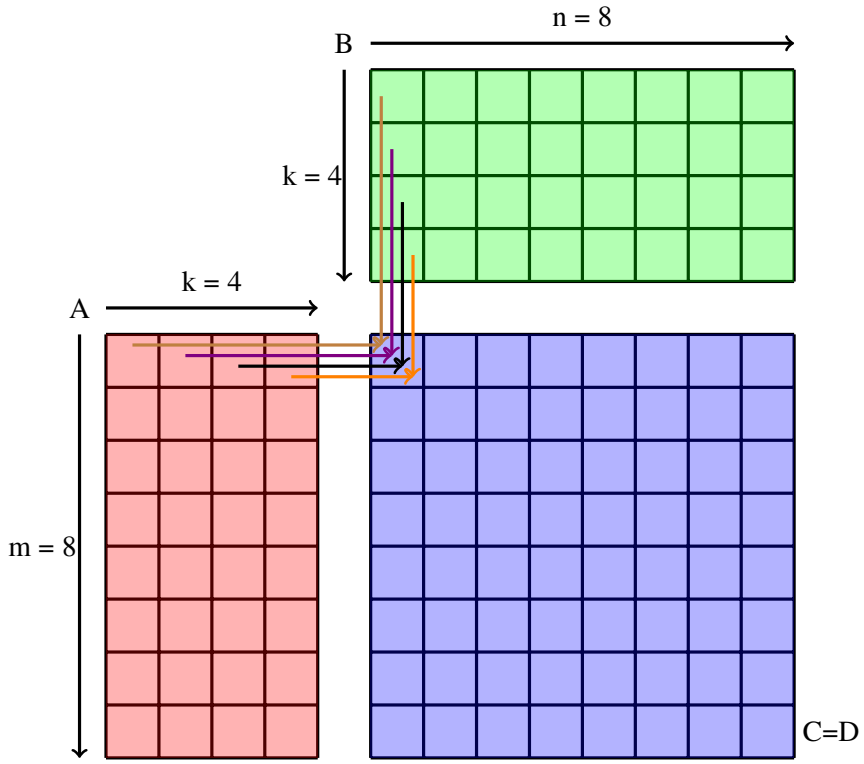


Abbildung 3.5: Graphisch dargestelle inplace Tensor Core Operation für FP64: $\mathbf{C} = \mathbf{A} \cdot \mathbf{B} + \mathbf{C}$, mit den Dimensionen $\mathbf{A}^{8,4}$, $\mathbf{B}^{4,8}$ und $\mathbf{C}^{8,8}$. Auf jeden Eintrag $C_{i,j}$ werden die 4 Produkte aus $\mathbf{A}_{i,0\ldots3} \cdot \mathbf{B}_{0\ldots3,j}$ addiert.

Bei einem Aufruf wird, wie in Abbildung 3.5 zu sehen, für jeden Eintrag der Matrix $\mathbf{C_{i,j}}$ 4 Multiplikationen und 4 Additionen vorgenommen und folglich $8 \cdot 8 \cdot (4+4) = 512$ Rechenoperationen. Für FP64 ist die theoretische Rechenleistung für Tensor Cores doppelt so hoch wie mit CUDA Cores [NVI20], dadurch lässt sich die Zyklendauer einer Operation theoretisch bestimmen: Eine SM hat 4 Tensor Cores und 32 FP64 FMA Einheiten, folglich muss gelten

$$4(\text{TC}) \cdot 512 \text{ Operationen} := 2(\times \text{ as fast}) \cdot x \cdot 32(\text{FP64}) \cdot 2 \text{ Operationen(FMA)} \rightarrow x = 16, \quad (3.19)$$

womit eine Operation 16 Zyklen braucht. Ein Nachteil an den Tensor-Cores ist, dass damit zwar die theoretische Rechenleistung steigt, die Bandbreite aber gleich bleibt.

Listing 3.3 Eine einzelne FP64 Tensor-Core Operation geschrieben als skalare Matrixoperation.

```
1       for(int i = 0; i < 8; ++i){
2          for(int j = 0; j < 8; ++j){
3             D[i][j] = A[i][0] * B[j][0] + A[i][1] * B[j][1] +
4             A[i][2] * B[j][2] + A[i][3] * B[j][3] + C[i][j];
5          }
6       }
```

Programmierung

Über CUDA gibt es 3 Möglichkeiten die Tensor-Cores anzusprechen:

- Über Bibliotheken, Frameworks und Templates. Besonders bei neuronalen Netzen erkennen und nutzen die meisten Bibliotheken eine Tensor-Core fähige GPU. Der Nutzer braucht dafür kein besonderes Wissen über die internen Vorgänge.
- Über Parallel Thread Execution (Deutsch: Parallele Thread-Ausführung) (PTX) und Instruction Set Architecture (Deutsch: Befehlssatzarchitektur) (ISA), eine mit Assembler-Code vergleichbare *low-level* GPU Sprache. So übersetzt der CUDA-Compiler den CUDA-Code als Zwischenschritt in PTX Code, der dann optimiert und für die entsprechenden Ziel-GPU-Architekturen übersetzt wird. Inline PTX Code war die erste Möglichkeit die Tensor-Cores außerhalb der ersten Variante zu nutzen, inzwischen wird PTX-Code allerdings nur noch selten verwendet.
- In CUDA lassen sich die Tensor-Cores inzwischen normal programmieren. Die dafür benötigten Befehle sind zusammengefasst in einer eigenen Namensraumuntergruppe `nvcuda::wmma`, das steht für Warp Matrix Multiply Add (Deutsch: Warpweite Matrixmultiplikation und -Addition) (WMMA).

In Kapitel 4 werden die Matrixeinheiten über die WMMA-Befehle angesprochen. Die wichtigsten Befehle und Konzepte werden im Folgenden vorgestellt, ein unvollständiges Kurz-Beispiel wird in Listing 3.4 gegeben: Gerechnet wird auf Fragmenten `nvcuda::wmma::fragment<nvcuda::wmma::Typ, m, n, k, type, nvcuda::wmma::Layout> var;`, diese können vom Typ `matrix_a` (**A**), `matrix_b` (**B**)oder `accumulator` (**C**, **D**) sein und weisen dem Fragment einen Operanden aus Gleichung (3.18) zu. Für den type FP64 sind m, n und k fest mit den Werten 8, 8 und 4. Zusätzlich wird das Matrix-Layout spezifiziert, dabei kann zwischen mem_col_major (spaltenbasiert) und mem_row_major (zeilenbasiert) gewählt werden. Die Fragmente belegen für jeden Thread des Warps die Register, in denen die Daten gespeichert werden. Dabei ist zu beachten, dass die Abbildung, welche Daten in welchem Thread liegen laut NVIDIA [NVI22a] unspezifiziert ist. Soll eine identische Operation auf jedes Element eines Fragmentes angewandt werden, wie zum Beispiel eine Division durch 2, kann dies über eine Schleife der Form `for(int i=0; i<frag.num_elements; ++i) frag_name.x[i] /= 2.0;` realisiert werden. Da die Daten aber unspezifiziert abgespeichert werden, müssen für nicht Tensor-Core Operationen oder Operationen ohne einheitliche Konstanten die Daten in den geteilten oder globalen Speicher geladen werden, bevor sie weiterverwendet werden können.

Um Daten in die Fragmente zu laden oder aus ihnen zu speichern, gibt es die Befehle `nvcuda::wmma::load_matrix_sync(Fragment, matrix_ptr, offset, nvcuda::wmma::Layout)` und `nvcuda::wmma::store_matrix_sync(matrix_ptr , Fragment, offset, nvcuda::wmma::Layout)`: Für diese wird jeweils ein Fragment angegeben, ein 256 bit ausgerichteter Pointer zum Matrixinhalt im globalen oder geteilten Speicher und ein Offset. Dieses gibt an, wie weit vom Startpunkt des Pointers entfernt die nächste Zeile der Matrix im Speicher ist, von der Daten geladen werden sollen. Für eine Matrix $A^{32\times32}$, welche als Row-Major (Deutsch: Zeilen Major) Array, also Zeile für Zeile, abgespeichert worden ist, wäre der Offset folglich 32. Optional kann hier das Layout der Daten angegeben werden, also ob das Fragment Spalten oder Reihen basiert abgespeichert wird, für Akkumulatoren ist dieser Parameter allerdings Pflicht. Eine weitere Möglichkeit die Daten in Fragmente zu verändern ist über den Befehl `nvcuda::wmma::fill_fragment(Fragment, value)`,

dabei wird ein Fragment auf einen bestimmten Wert gesetzt. Ein Anwendungsbeispiel ist für eine einfach Matrix-Multiplikation ohne zusätzliche Addition das Akkumulator-Fragment, das aufaddiert werden soll, auf 0 zu setzen. Die eigentliche Operation wird dann durch den Befehl `nvcuda::wmma::mma_sync(solution_acc, matrix_a, matrix_b, addition_acc)` durchgeführt und entspricht Gleichung (3.18). Der erste Parameter ist der Lösungsakkumulator, darauf folgen die Fragmente `matrix_a` und `matrix_b` und am Ende der aufzuaddierende Akkumulator, der identisch mit dem Lösungsakkumulator sein kann. Eine vollständige Abfolge der Befehle kann in Listing 3.4 nachvollzogen werden.

Listing 3.4 Unvollständiges Minimalbeispiel einer FP64 Tensor-Core Operation: $\mathbf{D}_{frag} = \mathbf{A}_{frag} \cdot \mathbf{B}_{frag}$. Dabei werden 8×4-Fragmente der Matrizen **A** und **B** aus dem geteilten Speicher geladen und das Ergebnis $\mathbf{D}_{frag}$ in den geteilten Speicher zurückgeschrieben.

```
1       // Initialisierung eines matrix_a und b Fragmentes und eines Akkumulators
2       nvcuda::wmma::fragment<nvcuda::wmma::matrix_a, 8, 8, 4, double, nvcuda::wmma::col_major> a_frag;
3       nvcuda::wmma::fragment<nvcuda::wmma::matrix_b, 8, 8, 4, double, nvcuda::wmma::row_major> b_frag;
4       nvcuda::wmma::fragment<nvcuda::wmma::accumulator, 8, 8, 4, double>   d_frag;
5       // Aus dem geteilten Speicher werden Daten in die Register
6       // von a_frag und b_frag geladen.
7       nvcuda::wmma::load_matrix_sync(a_frag, shmem_A_ptr, Offset);
8       nvcuda::wmma::load_matrix_sync(b_frag, shmem_B_ptr, Offset);
9       // d_frag wird auf 0 gesetzt.
10      nvcuda::wmma::fill_fragment(d_frag, 0.0f);
11      // Tensoroperation: d_frag += a_frag * b_frag
12      nvcuda::wmma::mma_sync(d_frag, a_frag, b_frag, d_frag);
13      // Ergebnis aus d_frag wird aus den Registern in den geteilten Speicher transferiert.
14      nvcuda::wmma::store_matrix_sync(shmem_D_ptr , d_frag, Offset, nvcuda::wmma::mem_row_major);
```

4 Implementierung

In diesem Kapitel geht es um die Veränderungen am PLSSVM Programm. Dabei werden speziell die Herausforderungen und Änderungen bei der Implementierung untersucht. *Organisatorische* Anpassungen wie die zusätzliche Initialisierung von weiteren Genauigkeiten sowie das teilweise Aufbrechen der Templates um Berechnungen mit gemischten Genauigkeiten zu ermöglichen, werden nicht behandelt. Damit spielt sich die Arbeit in einem relativ kurzen Teil der Implementierung ab, genauer in der Lösung des LGS mittels CG-Algorithmus (3.1.4). Da bisher nur eine Multi-GPU-Implementierung für den linearen Kernel besteht, liegt der Fokus ganz klar auf diesem. Der polynomiale Kernel ist sehr ähnlich, allerdings kann der Multi-GPU Ansatz, bei dem die Matrizen anhand ihrer Features zerlegt und die Zwischenergebnisse nach der Operation aufsummiert werden, nicht angewandt werden, da der komplette Wert einzelner Matrixeinträge $Q_{i,j}$ verrechnet wird. Der radiale Kernel verhält sich in der Beziehung gleich wie der polynomiale Kernel, allerdings finden bei diesem zusätzlich Subtraktionen statt und der Kernel ist folglich weniger für Matrixeinheiten geeignet. Aus diesem Grund wurde kein zusätzlicher radialer Kernel für Matrixeinheiten implementiert. Zu Beginn sollen zuerst die theoretischen Aspekte des CG-Algorithmus erläutert werden, bevor genauer auf die eigentliche Umsetzung eingegangen wird. Danach wird die Implementierung mittels Matrixeinheiten für den linearen Kernel vorgestellt.

4.1 CG-Analyse

Der komplette CG-Algorithmus wird in Listing 4.1 dargestellt. Bei der ersten Iteration ist das Residuum ebenfalls die Suchrichtung $\mathbf{d}$ (line 7). α, β und δ werden dabei als Hilfsvariable genutzt, um Rechenzeit zu sparen oder den Code übersichtlicher zu gestalten. In der For-Schleife wird zuerst die Schrittweite α berechnet, dann mit dieser und der Richtung $\mathbf{d}$ werden $\mathbf{x}$ und $\mathbf{r}$ aktualisiert. Falls δ beziehungsweise $\mathbf{r}^\intercal\mathbf{r}$ größer als $\epsilon^2 \cdot \delta^{start}$ ist, was auch Abbruchkriterium genannt wird, wird die neue Suchrichtung bestimmt und weiter iteriert bis entweder das Abbruchkriterium erfüllt wird oder nach Anzahl der Dimensionen von $\mathbf{A}$ in alle Richtungen einmal korrigiert worden ist. Nach spätestens Dimension von $\mathbf{A}$ Schritten wird dabei, unter Annahme einer nicht existierenden optimalen Genauigkeit, immer das exakte Ergebnis erreicht. Für einen besseren Lesefluss wird bezüglich $\mathbf{r}^\intercal\mathbf{r}$ einfach vom Residuum gesprochen, was in diesem Falle ein Skalar und kein Vektor ist und dadurch zu unterscheiden.

Als Nächstes wird der CG-Algorithmus auf Komplexität und mathematische Operationen hin untersucht. Der Algorithmus besteht zur Erstberechnung des Residuum in line 5 aus einer Generalized Matrix-Vector Multiplication (Deutsch: Verallgemeinerte Matrix-Vektor-Multiplikation) (GEMV) und liegt dadurch in $O(\#datapoints^2)$. Die Berechnung von Delta (line 8 und line 21) ist ein Double Dot Product (Deutsch: Skalarprodukt in doppelter Genauigkeit) (DDOT), welches eine Komplexität von $O(\#datapoints)$ innehat. Danach beginnt die eigentliche Schleife, bis das Residuum r ausreichend genau berechnet worden ist und die Abbruchbedingung erfüllt. In jeder

Iteration kommen nun ein weiteres GEMV hinzu für die Berechnung von $\mathbf{q}$ (line 14), 2 weitere DDOTs für die Berechnungen von α und δ (line 16 und line 21), und 3 Operationen der Form Double precision (Deutsch: Doppelte Genauigkeit) $\alpha\mathbf{x} + \mathbf{y}$ (DAXPY) in line 18, line 19 und line 27, die ebenso wie DDOT in $\mathcal{O}(\#datapoints)$ liegen.

Listing 4.1 Kompletter CG-Algorithmus in C-ähnlichem Pseudocode wie er, bis auf kleine Abweichungen, in PLSSVM verwendet wird. Dabei werden GEMV-Operationen in Grün, DDOT-Operationen in Orange und DAXPY-Operationen in Gelb dargestellt.

```
1   solver_CG(vector b, matrix A, ϵ){
2     // geschätzte Lösung
3     vector x^0 = 1.0;
4     // Berechnung des Residuum
5     vector r^0 = b − Ax^0;
6     // In der ersten Iteration ist das Residuum die Suchrichtung d.
7     vector d^0 = r^0;
8     float  δ^0 = transposed(r^0) · r^0;
9     // Für das relative Abbruchkriterium wird das Startresiduum gespeichert.
10    float  δ^start = δ^0;
11    // Loop über die Anzahl an Dimensionen der Matrix
12    for(size_t k = 0; k < x.size(); ++k){
13      // Zwischenergebnis für die A-Konjugation
14      vector q^k = Ad^k;
15      // Schrittweite
16      float  α^k = δ^k/(transposed(d^k) · q^0);
17      // Aktualisierung von x und r
18      vector x^{k+1} = x^k − α^k · d^k;
19      vector r^{k+1} = r^k + α^k · q^k;
20      // Falls die Norm des Residuums genau genug ist, wird abgebrochen
21      float  δ^{k+1} = transposed(r^{k+1}) · r^{k+1};
22      if(δ^{k+1} < δ^start · ϵ · ϵ){
23        break;
24      }
25      float  β^k = δ^{k+1}/δ^k;
26      // Update der konjugierten Suchrichtung
27      vector d^{k+1} = r^{k+1} + β^k · d^k;
28    }
29    return x^{k+1}
30  }
```

Dies bedeutet, dass die mit Abstand meisten Rechenoperationen bei der Berechnung von $\mathbf{q}$ gemacht werden. Es ist also essenziell, dass dieser Part mit niedriger Genauigkeit gerechnet werden kann. Hinzu kommt, dass genau dieser Teil, also die beiden GEMV Operationen, bereits auf der GPU berechnet wird und moderne Desktop-GPUs eine 64× höhere FP32 Rechenleistung haben als FP64 im Falle von NVIDIAs Ampere Architektur. Zusätzlich wird in jeder 50. Iteration das Residuum in PLSSVM für eine höhere Stabilität komplett neu berechnet und nicht einfach als DAXPY auf das vorherige Residuum aufaddiert, was in diesem Durchgang ein DAXPY durch ein GEMV ersetzt.

In einer Präsentation von Clark [Cla20] wurde ein Leitfaden für gemischte Genauigkeiten in HPC-Anwendungen vorgestellt, unter anderem auch für Krylov-Unterraum-Löser wie dem CG-Algorithmus. Demnach sollen

1. Reduktionen und Kreuzprodukte immer in hoher Genauigkeit berechnet werden,
2. Lösungen und Teillösungen ebenso in hoher Genauigkeit gehalten werden,
3. Zuverlässige Updates [CBB+10; SV96] genutzt werden und
4. für die Berechnung von beta anstatt des ursprünglichen Verfahrens von Fletcher-Reeves 4.1 die Form von Polak-Ribière [PR69] verwenden.

$$\beta^k = \frac{(\mathbf{r}^{k^\intercal}(\mathbf{r}^k - \mathbf{r}^{k-1}))}{\delta} \tag{4.1}$$

Daraus folgt: $\mathbf{x}$ und $\mathbf{r}$ werden als Zwischenlösungen in hoher Genauigkeit gehalten, δ und α werden durch DDOT beziehungsweise Reduktionen berechnet und folglich auch in hoher Genauigkeit gehalten. Dann ist bis auf die Suchrichtung $\mathbf{d}$, β und die GEMV durch diese Regeln jede Operation in hoher Genauigkeit. Da allerdings das Konvertieren für $\mathbf{d}$ und β einen vergleichbaren Aufwand hat wie das Berechnen in hoher Genauigkeit, wird alles bis auf die GEMV in FP64 berechnet.

Listing 4.2 Zuverlässige Updates nach Clark [Cla20] in C-ähnlichem Pseudocode: Ist δ^k kleiner als eine Konstante ζ und das vorherige gespeicherte δ^{old} wird ein Korrekturschritt eingeleitet. Dabei wird das Residuum neu berechnet und zu $\mathbf{b}$ gesetzt, das Zwischenergebnis $\mathbf{x}^k$ auf das Gesamtergebnis $\mathbf{y}$ addiert und anschließend auf 0 gesetzt. Darauf wird der CG-Algorithmus fortgesetzt mit $\mathbf{r}^{k+1} = \mathbf{r}^k - \mathbf{A}\mathbf{x}^k$.

```
1   if(δ^k < ζ · δ^old){
2       // Neuberechnung von r
3       vector r^k = b − Ax^k;
4       // das Residuum wird zu b gesetzt
5       vector b = r^k;
6       // Aus dem neuen r wird das nächste Korrekturkriterium bestimmt
7       float δ^old = transposed(b) · b;
8       // Zwischenergebnis x wird auf das Endergebnis y addiert.
9       vector y = y + x^k;
10      // x wird auf 0 gesetzt
11      vector x^k = 0.0;
12  }
```

Die zuverlässigen Updates sind wie in Listing 4.2 umgesetzt worden, dabei wurde sich an Clark [Cla20] orientiert. Durch die endliche Genauigkeit der Gleitkommazahlen weicht das Residuum immer weiter vom wahren Residuum mit steigender Anzahl an Iterationen ab. Für niedrige Genauigkeiten ist dieser Effekt größer. Wird das Residuum r, in diesem Falle δ, kleiner als eine Konstante ζ kleiner 1 multipliziert mit dem letzten in der Korrektur oder zu Beginn gesetzten δ^{old}, wird das Residuum neu berechnet. Für die weitere Berechnung wird $\mathbf{b} = \mathbf{r}^k$ gesetzt, also wird die Gleichung nicht mehr nach dem ursprünglichen $\mathbf{b}$ gelöst, sondern nach dem restlichen Fehler. Dafür wird $\mathbf{x}^k$ auf das Gesamtergebnis addiert und anschließend auf 0 gesetzt. Der aufgespannte Unterraum bleibt dabei erhalten.

Diese Änderungen wurden umgesetzt, allerdings entsprach das Ergebnis nicht den ersten Erwartungen. Selbst für kleine Probleme, die im Sekunden- oder gar Millisekundenbereich mit hoher Genauigkeit berechnet werden können, konvergiert die Implementierung mit einfacher Genauigkeit gar nicht und die mit gemischten Genauigkeiten nur sehr langsam, wobei gleichzeitig die schlussendliche Klassifikation durch das berechnete Modell sich verschlechtert hat, wie in Abbildung 4.1

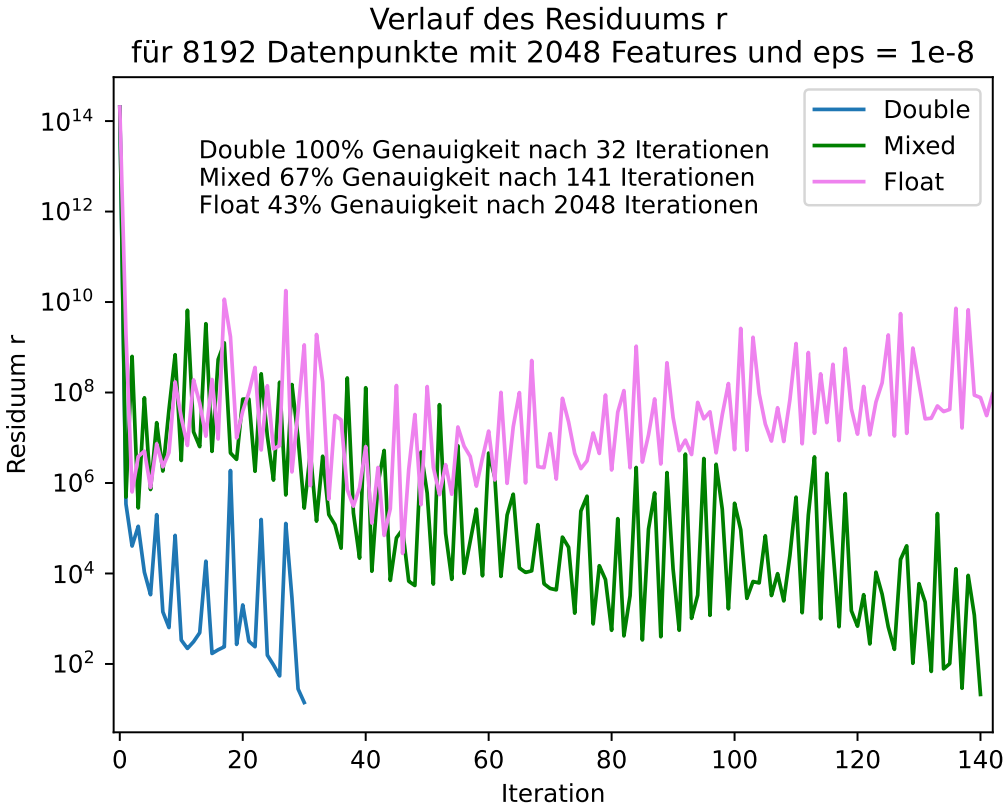


Abbildung 4.1: Verlauf des Residuums:
Dargestellt wird der Verlauf des Residuums bei der Lösung eines synthetischen Datensatzes mit 8192 Datenpunkten mit jeweils 2048 Features, erstellt wie in Listing 4.3. Die Implementierung mit doppelter Genauigkeit erreicht die vorgegebene Genauigkeit für das Abbruchkriterium nach 31 Iterationen, die mit gemischten nach 141 Iterationen und die mit nur einfacher Genauigkeit erreicht die vorgegebene Genauigkeit nach 2048 Iterationen nicht und wird abgebrochen. Die jeweiligen Modelle haben nach Ende des CG-Algorithmus eine Korrektheit bei der Klassenvorhersage auf die Trainings-Daten angewandt von 100%, 67% und 43% (doppelte, gemischte und einfache Genauigkeit).

zu sehen ist. Einerseits muss das LGS mit dem CG-Algorithmus gar nicht sehr genau berechnet werden, sodass die Hyperebene die Klassen sauber trennen kann, weshalb für die Implementierung mit doppelter Genauigkeit in Beispiel Abbildung 4.1 auch 31 Iterationen problemlos ausreichend sind, obwohl für eine mathematisch korrekte Lösung in perfekter Genauigkeit im allgemeinen Fall 8192 Iterationen notwendig wären, anderseits verschwinden diese Vorteile, wenn der Algorithmus an sich nicht mehr konvergiert und sich dem Ergebnis nicht weiter annähert. Für etwas größere Probleme konvergierte der Ansatz mit gemischten Genauigkeiten ebenso wie die Implementierung mit einfacher Genauigkeit gar nicht mehr. Abbildung 4.1 zeigt außerdem, dass der naive Ansatz, zuerst in einfacher Genauigkeit und später in doppelter Genauigkeit weiterzurechnen, in diesem Falle keinen Sinn ergibt, da die Variante in einfachen Genauigkeiten nach 2 Iterationen bereits

nicht mehr konvergiert und auf doppelte Genauigkeit gewechselt werden müsste, die vorherigen Fehler aber beibehalten werden würden. Bei der Ursachenforschung konnten dafür hauptsächlich 3 Probleme ausfindig gemacht werden, die im Folgenden vorgestellt werden:

4.1.1 Matrix-Kondition

Zur Erinnerung noch einmal die Rechenvorschrift für die Einträge der aufzustellende Matrix $\mathbf{Q} \in \mathbb{R}^{m \times m}$ zur Lösung des LGS:

$$Q_{ij} = \langle \mathbf{x}_i, \mathbf{x}_j \rangle + \frac{1}{C}\delta_{ij}. \tag{4.2}$$

Schlecht konditionierte Probleme sind numerisch besonders anfällig. Die Kondition beschreibt wie sehr Eingangsfehler oder durch die Ungenauigkeit von Gleitkommaeinheiten verursachte Fehler verstärkt werden können beziehungsweise wie empfindlich sich diese auswirken und lässt sich in diesem Falle, da die aufgebaute Matrix symmetrisch ist, berechnen durch

$$\kappa(A) = \frac{|\lambda_{max}(A)|}{|\lambda_{min}(A)|}, \tag{4.3}$$

wobei λ_{max} der größte und λ_{min} der kleinste Eigenwert ist, was der Spektralnorm entspricht. Für mehrere Datensätze, zum Großteil synthetischer Natur, aber auch Datensätze, die auf der Webseite von LIBSVM [CL11] zu finden waren, wie einige von Platt [Pla98], wurden Untersuchungen bezüglich Kondition und Eigenwertverteilung durchgeführt. Die synthetischen Datensätze wurden mit dem PLSSVM beiliegendem utility script *generate_data.py*, das in Van Craen et al. [VBP22b] näher beschrieben wird, mit folgendem Befehl erstellt:

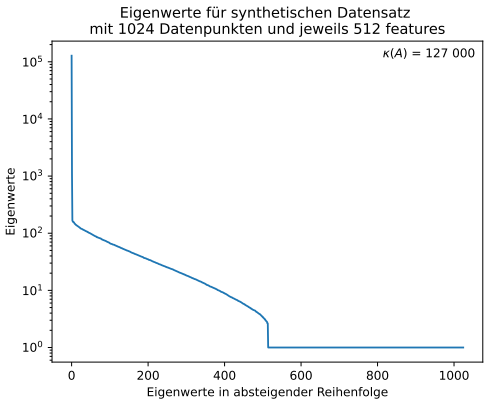
Listing 4.3 Python Utility Script zur Erstellung von synthetischen Trainingsdatensätzen.

```
python3 generate_data.py --output folder --format libsvm --problem planes/blobs
                    --samples number_datapoints --features number_features
```

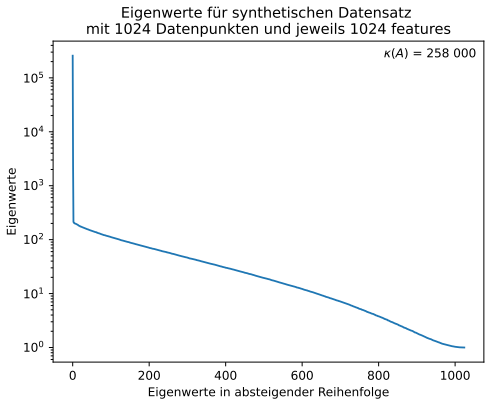
Dabei hat sich herausgestellt, wie in Abbildung 4.2 zu sehen, dass die Anzahl der Features der Anzahl an Eigenwerten ungleich $\frac{1}{C}$ entspricht und jeder Eigenwert λ sich aus den Trainingsdaten und dem Offset durch $\frac{1}{C}$ ergibt. Die Konditionszahl hängt dabei von der Anzahl der Datenpunkte und deren Features ab, so wie der Dichte und Werte der Features:

$$\kappa(\mathbf{Q}) \approx \#datapoints \cdot \#features \cdot value. \tag{4.4}$$

Die Features der Daten waren entweder zwischen $[-1, 1]$ oder $[0, 1]$, manche Datensätze waren sehr dünn besetzt und hatten nur vereinzelt Einträge mit Wert 1. Die untersuchten synthetischen Datensätze waren verteilt zwischen $[-1, 1]$, bei einer Gleichverteilung wäre der Erwartungswert des Betrages der einzelnen Features dadurch 0.5 und für das Produkt der einzelnen Berechnungen des Skalarproduktes der Matrix einen Erwartungswert von 0.25, was den gemessenen Ergebnissen in Tabelle 4.1 relativ genau entspricht. Wird dieser Wert für $value$ angegeben, ist das ziemlich genau λ_{max} beziehungsweise $\kappa(\mathbf{Q})$. Bei echten, häufig sehr dünn-besetzten Datensätzen war dieser Zusammenhang nur grob näherungsweise zu beobachten. So würde bei den Datensätzen von [Pla98] eine Kondition von 22.249 beziehungsweise 66290 erwartet werden, berechnet wurden allerdings die in Tabelle 4.1 gezeigten Werte.



(a) Eigenwerte der durch den linearen Kernel erzeugten Matrix mit 1024 Datenpunkten, jeweils 512 Features und $C = 1$



(b) Eigenwerte der durch den linearen Kernel erzeugten Matrix mit 1024 Datenpunkten, jeweils 1024 Features und $C = 1$

Abbildung 4.2: Eigenwertanalysen:
Aufgetragen sind in logarithmischer Skalierung die Eigenwerte der Matrix $\mathbf{Q}$ in absteigender Reihenfolge. Beide Matrizen haben dabei so viele von $\frac{1}{C}$ verschiedene Eigenwerte wie die Anzahl an Features. Außerdem fällt bei beiden Beispielen genau ein sehr großer Eigenwert auf.

		$\kappa(\mathbf{Q})$	$\kappa(\hat{\mathbf{Q}})$
Synth. Planes:	256 Datenpunkte mit jeweils 256 Features	19000	37000
Synth. Planes:	512 Datenpunkte mit jeweils 256 Features	35000	73000
Synth. Planes:	512 Datenpunkte mit jeweils 512 Features	72000	148000
Synth. Planes:	1024 Datenpunkte mit jeweils 512 Features	127000	258000
Synth. Planes:	1024 Datenpunkte mit jeweils 1024 Features	258000	532000
Synth. Planes:	8192 Datenpunkte mit jeweils 2048 Features	3681000	7326000
a1 [Pla98]:	1605 Datenpunkte mit jeweils 123 Features	30000	
a4 [Pla98]:	4781 Datenpunkte mit jeweils 123 Features	88000	

Tabelle 4.1: Berechnete Kondition der Matrix $\mathbf{Q}$ und $\hat{\mathbf{Q}}$ für synthetische Datensätze und 2 reale Datensätze von Platt [Pla98].
Dabei ist $\kappa(\hat{\mathbf{Q}})$ die Matrix mit Dimensionsreduktion und $\mathbf{Q}$ die Matrix ohne diese.

PLSSVM nutzt zusätzlich eine Dimensionsreduzierung, bei der die Matrix $\mathbf{Q}$ leicht modifiziert wird um die Dimension um jeweils 1 zu reduzieren. Diese Reduzierung verdoppelt laut Tabelle 4.1 die Kondition. Für große Datensätze, für die das Programm durch sein Design und der Entscheidung, die Matrix implizit zu bauen, ausgerichtet ist, können dadurch sehr hohe Konditionen entstehen von deutlich über $\kappa(\mathbf{Q}) > 10^9$. In der Präsentation von Clark [Cla20] wird das Verhalten der vorgestellten CG-Implementierung untersucht bis zu einer Kondition von $\approx 5 \cdot 10^8$, wobei bemerkt wird, dass für die Problemlösung des Programms nur Konditionen im Bereich bis $\approx 3 \cdot 10^6$ relevant sind.

Ein in ersten Versuchen gut funktionierender Ansatz war es, die Skalarprodukte wie beim polynomialen Kernel Abschnitt 3.1.3 mit einem kleinen γ durchzumultiplizieren. Dies reduziert die Kondition deutlich, da λ_{max} um ein Vielfaches kleiner wird, und führt auch zu einer verbesserten Konvergenz,

allerdings wird dabei unfreiwillig C skaliert. Wird dieser Effekt auf C ausgeglichen, wird λ_{min} sehr klein und dadurch die Kondition wieder hoch. Andernfalls kann das Ergebnis in manchen Fällen deutlich abweichen, da Fehler im Gleichungssystem nicht ausreichend berücksichtigt werden, was in einigen wenigen Testfällen passiert ist und dadurch dieser Ansatz nicht weiter verfolgt wurde.

4.1.2 Implizite Matrix

Dadurch, dass die Matrix nur implizit gespeichert wird, wird sie in jeder Iteration aufs neue im Kernel rekonstruiert, was sehr rechenintensiv ist. Dabei besteht jeder Matrixeintrag aus einem Skalarprodukt und diese sollen [Cla20] möglichst in hoher Genauigkeit berechnet werden. Wird allerdings dieses Skalarprodukt für jeden Matrixeintrag in doppelter Genauigkeit berechnet, wird der ganze Ansatz mit gemischten Genauigkeiten ad absurdum geführt: Die einzelnen Operationen im CG-Algorithmus sind in $O(\#datapoints)$ oder $O(\#datapoints^2)$, während der Aufbau der Matrix aus $\#datapoints^2$ Skalarprodukten besteht und dadurch in $O(\#datapoints^2 \cdot \#features)$. Hier kann also nur innerhalb des Kernels optimiert werden, was in Abschnitt 4.3 beachtet wird.

4.1.3 Initial Lösung

Für die erste Berechnung des Residuums muss eine am sinnvollsten gut geschätzte Lösung angegeben werden. Dazu wird bisher der gesamte Vektor $\mathbf{x} = 1$ gesetzt. Dies führt besonders bei großen Datensätzen und dadurch auch potenziell hohen $Q_{i,j}$ Werten zu einem sehr hohen Start-Residuum von über 10^{21}. Auch wenn die Fehler mit einer relativen Abweichung im Bereich von 10^{-5} bis 10^{-7} für eine Iteration in einfacher Genauigkeit und im Bereich von 10^{-6} bis 10^{-9} für gemischte Genauigkeiten liegt, sind das für Werte des Residuums von 10^{21} Fehler im Bereich von 10^{12} bis 10^{16} nach nur einer Iteration. Gleichzeitig wird durch die Orthogonalität der Suchrichtungen in die anfänglichen stark fehlerbehafteten Richtungen nicht mehr fortgeschritten. Die Implementierung mit doppelter Genauigkeit ist davon bei weitem nicht so stark betroffen, aber die Implementierung mit gemischten Genauigkeiten oder einfacher Genauigkeit hören bei größeren Problemen komplett auf zu konvergieren. Das Residuum wird berechnet durch

$$\mathbf{r} = \mathbf{b} - \mathbf{Q}\mathbf{x}, \tag{4.5}$$

dabei ist $\mathbf{b}_i \in (-1, 1)$, was sehr nahe an 0 liegt. Dies kann sich zunutze gemacht werden, indem die initiale Lösung sehr klein gewählt wird mit dem Ziel nach der Multiplikation mit $\mathbf{Q}$ ein Ergebnis nahe 0 zu bekommen. Hierfür wird $\mathbf{x} = 1/(\#datapoints \cdot \#features)$ gesetzt, als Orientierung für diese Schätzung dient die Größenordnung von λ_{max}.

4.2 Umsetzung CG

Listing 4.4 Kahans Summen Algorithmus als Inline Makro für ein Skalarprodukt.

```
1   template <typename T>
2   [[nodiscard]] inline T operator*(const transposed<T> &lhs, const std::vector<T> &rhs) {
3     // Initialisierung der Hilfsvariablen
4     T val_sum{}; // Akkumulator
5     T val_c{};   // Zwischenspeicher für eigentlich verlorene Bits
6     T val_y{};
7     T val_t{};
8     // Loop über den Vektor
9     for (typename std::vector<T>::size_type i = 0; i < lhs.vec.size(); ++i) {
10      // val_c wird von dem Produkt des Skalarproduktes abgezogen.
11      // In der ersten Iteration 0.
12      val_y = std::fma(lhs.vec[i], rhs[i], -val_c);
13      // Das Zwischenergebnis wird auf die Summe addiert,
14      // für großes val_t gehen die letzten Bits von val_y verloren.
15      val_t = val_sum + val_y;
16      // In val_c werden die verloren gegangene Bits gespeichert.
17      val_c = (val_t - val_sum) - val_y;
18      val_sum = val_t;
19    }
20    return val_sum;
21  }
```

Wie in Abschnitt 4.1.3 erwähnt, wird die Initial Lösung geändert auf $= 1/(\#datapoints \cdot \#features)$. Weiterhin werden auf der CPU, also die DDOT Operationen, in doppelter Genauigkeit berechnet. Diese fallen bezüglich ihrer Laufzeit nicht ins Gewicht und sind memory bound (Deutsch: Speichergebunden), die CPU kann also, besonders wenn die Daten nicht in den Cache passen, nicht schnell genug die Daten in die Recheneinheiten transferieren. Für ein Skalarprodukt werden für 2 double Variablen 2 Operationen ausgeführt, für eine DAXPY Operation ebenso. Aus diesem Grund wird für das bereits vorhandene Template für die Berechnung des Skalarprodukts zweier Vektoren Kahans Summen Algorithmus wie in Listing 4.4 angewandt, ohne die Laufzeit negativ zu beeinflussen. Die genannten Änderungen sind für alle Kompilier-Varianten aktiv.

Für gemischte Genauigkeiten wird ein `#define MIXED` angelegt, ebenso für die Auswahl der GPU-Kernel ein `#define TENSOR`. In Listing 4.5 werden die Auswirkungen anhand der GEMV-Operation gezeigt: Werden gemischte Genauigkeiten (`#define MIXED`) verwendet, müssen die Daten entweder auf der GPU in FP32 (line 7) oder auf der CPU in FP64 (line 40) konvertiert werden. Die Auswahl der Kernel ist zusätzlich von `#define TENSOR` abhängig, wodurch sich 4 Varianten ergeben. Im Allgemeinen steuert `#define MIXED` das Anlegen von Variablen, das Konvertieren von FP64 in FP32 und vice versa und gemeinsam mit `#define TENSOR` die Ausführung der richtigen GPU-Kernel.

Optional gibt es weiterhin, wie in Listing 4.6 zu sehen, über ein `#define POLAK_RIBIERE` die Möglichkeit β in der Form von P-R oder wie bisher in der ursprünglichen Form von F-R zu verwenden. In Kapitel 5 werden beide mögliche Varianten auf ihre Vor- und Nachteile untersucht.

Die Auswirkungen der Korrekturschemata werden in Abschnitt 5.2 untersucht, erzielen allerdings nicht den gewünschten Effekt. Dies wird mehreren Faktoren zugeschrieben: Einerseits konvergiert das Ergebnis, wenn es konvergiert, so schnell, dass kein Korrekturschritt benötigt wird, anderseits reicht durch die Natur des Problems auch ein nicht exaktes Ergebnis aus, da die Hyperebene so gewählt wird, dass der Abstand zu den Trainingsdaten maximiert wird.

Listing 4.5 Kernel Auswahl und Konvertierungen für die GEMV Berechnung auf der CPU in Abhängigkeit von MIXED und TENSOR für potenziell mehrere Grafikkarten.

```
#pragma omp parallel for
// Aktuelles r auf die Grafikkarten kopieren
for (typename std::vector<queue_type>::size_type device = 0; device < devices_.size(); ++device) {
   r_d[device].memcpy_to_device(d, 0, dept_);
   #if defined(MIXED)
   // Für MIXED in float transferieren
   run_transformation_kernel_df(device, range_r, r_d_f[device], r_d[device]);
   #endif
}
// Ad = A * r (q = A * d)
#pragma omp parallel for
for (typename std::vector<queue_type>::size_type device = 0; device < devices_.size(); ++device) {
   #if defined(MIXED)
   // Vorherige float Ergebnisse auf 0 setzen
   Ad_d_f[device].memset(0);
   r_d_f[device].memset(0, dept_);
   // Richtigen Kernel auswählen
   #if defined(TENSOR)
   run_device_kernel_tf(device, q_d_f[device], Ad_d_f[device], r_d_f[device], 1);
   #else
   run_device_kernel_f(device, q_d_f[device], Ad_d_f[device], r_d_f[device], 1);
   #endif
   #else
   // Vorherige double Ergebnisse auf 0 setzen
   Ad_d[device].memset(0);
   r_d[device].memset(0, dept_);
   // Richtigen Kernel auswählen
   #if defined(TENSOR)
   run_device_kernel_td(device, q_d[device], Ad_d[device], r_d[device], 1);
   #else
   run_device_kernel(device, q_d[device], Ad_d[device], r_d[device], 1);
   #endif
   #endif
}
// update Ad (q)
#if defined(MIXED)
// Daten von allen GPUs in Ad_f speichern und in doppelte Genauigkeit casten
device_reduction_f(Ad_d_f, Ad_f);
for(typename std::vector<queue_type>::size_type cast_i = 0; cast_i < dept_; ++cast_i){
   Ad[cast_i] = static_cast<double>(Ad_f[cast_i]);
}
#else
// Daten von allen GPUs in Ad
device_reduction(Ad_d, Ad);
#endif
```

Listing 4.6 Über #define POLAK_RIBIERE kann bestimmt werden, ob β innerhalb des CG-Algorithmus nach Polak-Ribière (P-R) oder nach Fletcher-Reeves (F-R) berechnet werden kann.

```
#if defined(POLAK_RIBIERE)
const real_type beta = transposed<double>{ r} * (r-r_old) / delta_old;
#else
const real_type beta = delta / delta_old;
#endif
```

Listing 4.7 Verschiedene Korrekturschemata für das Residuum **r** in Abhängigkeit von `CORRECTION_SCHEME` innerhalb des CG-Algorithmus.

```
1   switch(CORRECTION_SCHEME){
2      case NoCorrectionScheme:
3      // Normale Berechnung von r und x
4      r -= alpha_cd * Ad;
5      x += alpha_cd * d;
6      break;
7      case NewRScheme:
8      // Aktuelles Korrekturschema in PLSSVM
9      x += alpha_cd * d;
10     // Jeden 50. Durchlauf Neuberechnung von r
11     if (run % 50 == 49) {
12        // gemv: r= b - A * x
13        . . .
14     } else {
15        r -= alpha_cd * Ad;
16     }
17     break;
18     case ReliableUpdate:
19     // Falls das Ergebnis ausreichend genau ist,
20     // wird ein zuverlässiges Update ausgeführt.
21     if (delta < 0.1 * b_norm) {
22        y+= x;
23        r = b_new
24        // gemv: r-= A * x
25        . . .
26        b_new = r;
27        b_norm = transposed<double>{ b_new } * b_new;
28        std::fill(x.begin(), x.end(), 0.0);
29     } else {
30        r -= alpha_cd * Ad;
31        x += alpha_cd * d;
32     }
33     break;
34  }
```

Aus diesem Grund werden die zuverlässigen Updates als auch die bereits vorhandene Fehlerkorrektur des Residuums durch eine `constexpr kernel_index_type CORRECTION_SCHEME` festgelegt, wie in Listing 4.7 zu sehen. Wird in line 2 kein Korrekturschema ausgewählt, wird in jeder Iteration $\mathbf{r}$ und $\mathbf{x}$ aktualisiert. Beim zweiten Schema in line 7 wird $\mathbf{r}$ alle 50 Schritte neu berechnet. Dies soll Rundungsfehler in $\mathbf{r}$, die sich mit der Anzahl an Iterationen immer weiter erhöhen können, zurücksetzen. Das letzte Korrekturschema in line 18 ist das der zuverlässigen Updates wie in Listing 4.2 mit $\zeta = 0.1$

4.3 GPU Kernel

Die GPU-Kernel stellen aus den Trainingsdaten und der Konstanten C die Matrix $\mathbf{Q}$ auf beziehungsweise mit den dimensionsreduzierenden Ausgleichsthermen $\hat{\mathbf{Q}}$, multiplizieren diese mit einem Vektor und addieren oder subtrahieren dieses Zwischenergebnis auf einen vorhanden Vektor. Dieser Vektor ist dann das Ergebnis des Kernels, das im weiteren Verlauf zurück auf die CPU transferiert wird. Für mehrere GPUs werden die Features gleichmäßig auf diese verteilt. Im Fall von 2 GPUs stellt die erste GPU die Matrix aus der ersten Hälfte der Features auf und die zweite aus den restlichen Features. Die Matrix Vektor Multiplikation wird dann auf jeder GPU separat, also doppelt, ausgeführt und die Ergebnisse am Ende aufaddiert. Der ersten GPU kommt dabei eine Spezialrolle zu, da sie die durch die Dimensionsreduzierung anfallende Ausgleichsterme zusätzlich verarbeitet.

Bei der Implementierung wurde besonders auf 3 Aspekte geachtet:

- Die Verwendung der Matrixeinheiten.
- Die Anwendung von Techniken, welche potenziell die Stabilität verbessern.
- Die Laufzeit an sich.

Für den Kernel in doppelter Genauigkeit werden Tensor-Cores mit Unterstützung für FP64 benötigt, dies unterstützt bis zum Erscheinen der NVIDIA H100 nur die NVIDIA A100 [NVI20; NVI22d]. Der Kernel in gemischten Genauigkeiten nutzt das TF32 Format, dies wird von GPUs ab der Ampere Architektur unterstützt [NVI20; NVI21].

4.3.1 Kernel mit doppelter Genauigkeit

In diesem Teil wird der CUDA-Kernel für doppelte Genauigkeit vorgestellt, mehrere Vorgehensweisen sind dabei identisch zum Kernel mit niedriger Genauigkeit. Auf die Unterschiede wird anschließend in Abschnitt 4.3.2 eingegangen. Die rein aus den Trainingsdaten aufzustellende Matrix wird in diesem Kontext $\mathbf{A}$, der zu multiplizierende Vektor $\mathbf{v}$ und der Ergebnisvektor, auf den $\mathbf{Av}$ addiert oder subtrahiert werden, $\mathbf{b}$ genannt, also $\mathbf{b} = \mathbf{b} \pm \mathbf{Av}$. Matrix $\mathbf{A}$ wird aus den Trainingsdaten, die als Matrix $\mathbf{X}$ abgespeichert sind, berechnet, wobei jeder Kernel nur eine Submatrix aus einem Teilset der Daten berechnet. $\mathbf{X}$ ist dabei als Structure-of-Arrays (Deutsch: Struktur von Arrays) abgespeichert, zuerst also Feature 0 aller Trainingsdaten, dann Feature 1 und so weiter. Somit ist nach Gleichung (3.8)

$$\mathbf{A} = \langle \mathbf{X}, \mathbf{X} \rangle = \mathbf{X}^T \cdot \mathbf{X}, \tag{4.6}$$

worauf die Matrixeinheiten angewandt werden können.

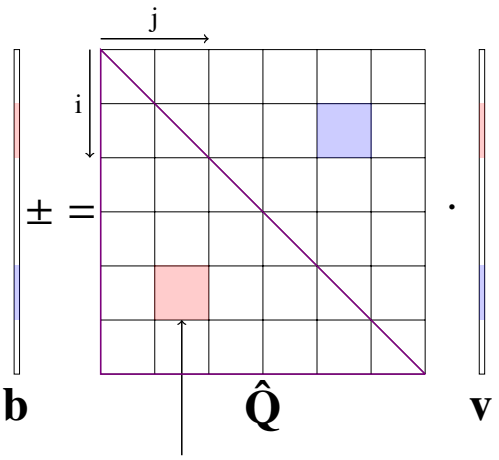


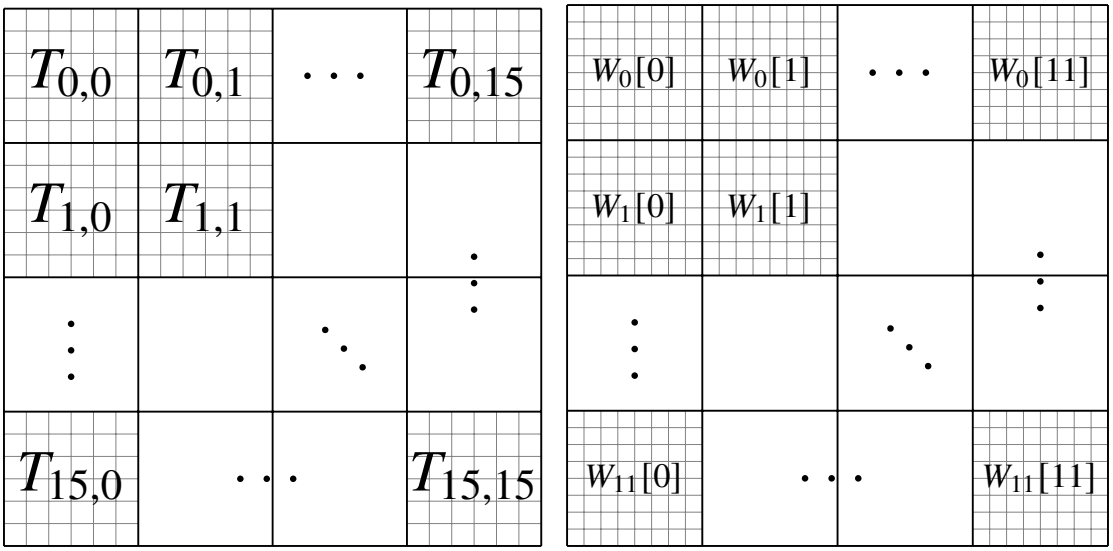
Abbildung 4.3: Aufteilung der Matrix in Blöcke, dabei wird aus Gründen der Symmetrie die Matrix nur grob zur Hälfte aufgestellt. Die in rot dargestellte Submatrix $\mathbf{Q}_{i,j}$ wird transponiert und auch für den blau dargestellten Teil benutzt. Das Aufstellen der Matrix nimmt dabei bezüglich der Laufzeit und der Anzahl an Operationen den größten Anteil ein.

Der grobe Aufbau des Kernels lässt sich dabei zusammenfassen zu:

1. Initialisierung des geteilten Speichers und einer Barriere für weitere Kopiervorgänge.
2. Dem Matrixaufbau für die Submatrix $\mathbf{A}_{i,j} = \langle \mathbf{X}_i, \mathbf{X}_j \rangle$, welche in jedem 2 dimensionalen Thread-Block i, j ausgeführt wird:
   a) Initialisierung der Fragmente für die Matrixeinheiten.
   b) Ein Loop über die Features der entsprechenden Trainingsdaten, wobei immer 8 Features pro Iteration auf einmal geladen werden, um mehrere Matrixoperationen durchführen zu können.
      i. Das asynchrone Laden der Features in den geteilten Speicher.
      ii. Das Laden aus dem geteilten Speicher in die Fragment-Register.
      iii. Der Matrixoperation, bei dem das Ergebnis auf die Akkumulatoren addiert wird.
   c) Dem Zurückspeichern der Akkumulatoren in den geteilten Speicher.
3. Der Matrix-Vektor-Multiplikation, bei der für GPU 0 aus $\mathbf{A}\ \hat{\mathbf{Q}}$ wird.

Aufteilung in Blöcke

Für den Kernel wird die Matrix wie in Abbildung 4.3 dargestellt in Blöcke aufgeteilt, dabei wird die untere Dreiecksblockmatrix $\hat{\mathbf{Q}}_{i,j}$, falls $i \geq j$, berechnet, mit $\mathbf{v}_j$ multipliziert und auf $\mathbf{b}_j$ summiert. Ist sogar $i > j$ und damit nicht auf der Diagonalen, wird $\hat{\mathbf{Q}}_{i,j}$ transponiert, mit $\mathbf{x}_i$ multipliziert und auf $\mathbf{b}_i$ summiert. Um Divergenzen innerhalb eines Warps und Blockes zu vermeiden wird



(a) Blockaufbau des Standard Matrix-Kernels. Jeder Thread in einem 16×16 Block speichert 36 Matrixeinträge in einem 6×6 Array.

(b) Blockaufbau des Matrix-Kernels. Hierbei werden pro Warp 12 jeweils 8×8 große Felder gespeichert, also 12 Ergebnisse aus Matrixoperationen.

Abbildung 4.4: Vergleich des CUDA Blockaufbaus der Standard-Implementierung und der Implementierung für Matrixeinheiten:
Im Gegensatz zur Standard-Implementierung ist die Implementierung für Matrixeinheiten bei der Speicherung Warp- anstatt Thread basiert bei der Aufteilung des Blockes. Dies ist notwendig für einfache Zuweisungen und da nicht bekannt ist, in welchem Thread welcher Teil der Matrix gehalten wird.

im Gegensatz zur normalen Implementierung für $i = j$ einfach der komplette Block berechnet und keine Transposition vorgenommen. Für die Berechnung von $\mathbf{A}_{i,j}$, also $\hat{\mathbf{Q}}$ ohne Diagonale und Ausgleichstherme, werden die Trainingsdaten $\mathbf{X}_i$ und $\mathbf{X}_j$ benötigt. Die Größe eines Blockes soll möglichst groß gewählt werden, um Speichertransfers vom globalen in den geteilten Speicher und vom geteilten Speicher in die Register zu minimieren. Wird die Matrix $\mathbf{A} \in R^{m \times m}$ in pro Dimension n-Teile der Größe m/n unterteilt, so werden $2 \cdot n^2 \cdot m/n = 2 \cdot n \cdot m$ Zahlen transferiert, wobei die Gesamtgröße m fest ist und dadurch nur auf n Einfluss genommen werden kann. Dem entgegen benötigt ein größerer Block mehr Register und mehr geteilten Speicher (Matrix-Implementierung) oder Cache (Standard-Implementierung). Es gilt also anhand theoretischer Überlegungen und von Profiling-Ergebnissen einen Mittelweg zu finden.

Eine erste Implementierung beim Speicherlayout des Blockes orientierte sich an der bereits vorhandenen Standard-Version, die in Abbildung 4.4 zu sehen ist. Dabei hat sich herausgestellt, dass diese Repräsentation nicht sinnvoll für Matrixeinheiten genutzt werden kann. Anstatt eines Kernel mit Thread-Dimension 16×16 wird nun ein Layout mit Dimensionen 12×32 genutzt, also `block.x = 32; block.y = 12;`, wobei dabei die einfache Einteilung in 12 Warps eine wesentliche Rolle spielt. Da ein SM auf der NVIDIA A100 4 SPs mit jeweils einer Matrixeinheit besitzt, wird bewusst ein Vielfaches von 4 für die Anzahl an Warps gewählt. Andernfalls wäre keine optimale Lastverteilung möglich.

Allgemeine Initialisierung

Listing 4.8 Initialisierung des dynamisch geteilten Speichers und Bindung an den Kernel. Die Größe wird hier in `dyn_sha_mem` (dynamic shared memory) festgelegt, mit `cudaFuncSetAttribute()` an die Funktion gebunden und als dritter Ausführungskonfiguration Parameter neben der Grid und Blockgröße beim Aufruf des Kernels übergeben.

```
1       // Berechnung der Größe für dyn_sha_mem
2       size_t dyn_sha_mem = ((BLOCK_SIZE + 4) * BLOCK_OFF) * sizeof(double);
3       // Als Attribut, das die maximale Größe des geteilten festlegt, syn_sha_mem
4       // über ein Error_Check-Makro an die Funktion cuda::device_kernel_linear_td binden.
5       PLSSVM_CUDA_ERROR_CHECK(cudaFuncSetAttribute(
6       cuda::device_kernel_linear_td, cudaFuncAttributeMaxDynamicSharedMemorySize, dyn_sha_mem));
7       // Aufruf des Cuda Kernels mit dyn_sha_mem als dritten Ausführungsparameter
8       cuda::device_kernel_linear_td<<<grid, block, dyn_sha_mem>>>
9       (q_d.get(), r_d.get(), x_d.get(), data_d_[device].get(), QA_cost_, 1 / cost_,
10      num_rows_, num_features, add, gamma_, device);
```

Bei der Standard-Implementierung wird ein 6×6 Array für jeden Thread im lokalen Speicher gehalten und dann zur Nutzung im L1 Speicher gecached. Da die Teilmatrix am Ende der Matrixoperationen in den geteilten Speicher geschrieben werden muss und dieses Feld potenziell größer ist als die zur Verfügung stehende 48 KB an statischem geteilten Speicher, muss mit dem Befehl `cudaFuncSetAttribute()` der L1 Cache verringert und dem geteilten Speicher hinzugefügt werden. Die genaue Vorgehensweise wird in Listing 4.8 gezeigt.

Durch dieses Vorgehen kann mit `extern __shared__` ein besonderer auf den so reservierten Speicherbereich angelegte Pointer initialisiert werden. Dieser ist 256 Bit ausgerichtet und muss wie in Listing 4.9 als Ausgangspunkt für alle weiteren Pointer in den geteilten Speicher dienen [NVI22a]. Durch den geringeren Cache wird versucht diesen möglichst nicht zu nutzen, sondern selbst die Zuweisungen über den geteilten Speicher zu steuern. Da der geteilte Speicher und der L1 Cache die gleichen Latenzen haben, hängt die Laufzeit von der effektiven Nutzung des geteilten Speichers ab.

Die Größe eines Blockes bei der Implementierung mit doppelter Genauigkeit liegt bei 96×96, diese Matrix wird als Array im geteilten Speicher über den Ausgangspointer, wie in Listing 4.9 in line 17 gezeigt, gespeichert. Dadurch hat das Array eine Größe von 96×96 Double-Variablen. Zusätzlich werden Pointer im geteilten Speicher **Is** und **Js** eingeführt, dabei werden Arrays beziehungsweise Pointer in den geteilten Speicher mit einem Suffix s versehen. Diese speichern über den Verlauf des Kernels alle für den Block notwendigen Trainingsdaten $\mathbf{X}_i$ und $\mathbf{X}_j$. Sie werden so angelegt, dass sie 16 Features dieser Trainingsdaten gleichzeitig im geteilten Speicher halten können, also 16 Features mal 96 Datenpunkte. Dieser Platz überlappt mit dem Speicher für die späteren Ergebnisse der Matrix wie in Abbildung 4.5 zu sehen ist. Allerdings wird der Platz um die Matrix in den geteilten Speicher zu schreiben erst gebraucht, nachdem keine weiteren Features mehr in **Is** oder **Js** gespeichert werden. Insgesamt haben diese Bereiche eine Größe von $96 \cdot 8 \cdot 2 \cdot 2 = 3072$ double.

Obwohl für eine einzelne Iteration über die Features mit Matrixoperationen nur jeweils 4 Features benötigt werden, wird für die Nebenläufigkeit der asynchronen Kopiervorgänge mindestens das Doppelte an Platz benötigt. Einen Bereich, in den neue Features asynchron geladen werden, und einen Bereich, in dem sich die Daten befinden, mit denen gerechnet wird. Dies wird in Abbildung 4.5 dargestellt durch die Bereiche 0 und 1. Um weniger Synchronisationen zu haben,

an denen die Fertigstellung der Kopiervorgänge sichergestellt wird, werden **Is** und **Js** gleich in doppelter Feature-Größe einer Matrixoperation transferiert, also 8 Features auf einmal. Für die Matrix Vektor Operationen am Ende werden für die Dimensionsoptimierung die Hilfsvariablen $\mathbf{q}_i$ und $\mathbf{q}_j$ und der zu multiplizierende Vektor $\mathbf{v}_i$, für die untere Dreiecksmatrix, und $\mathbf{v}_j$, für die transponierte obere Dreiecksmatrix, benötigt, was zusammen $4 \cdot 96 = 484$ double ergibt. Am Ende der Initialisierungsphase wird noch eine Barriere für die asynchrone Datentransfers angelegt und der Block synchronisiert. Um Speicherbankkonflikte, wie in Abschnitt 4.3.1 näher beschrieben, zu vermeiden, wird zusätzlich ein Offset in jeder Zeile eingeführt mit einer Länge von 4 Double. Die gesamte Größe des benötigten Speichers in Byte beträgt wie in Listing 4.8 durch die Doppelnutzung eines Bereiches insgesamt $((96+4) \cdot (96+4)) \cdot 8 = 80000$ Byte und damit etwas weniger als die Hälfte des möglichen geteilten Speichers von 163 kB.

Listing 4.9 Initialisierung der Variablen zum Start im Kernel mit Matrixeinheiten: Es wird getestet, ob der Block sich in der unteren Dreiecksmatrix befindet, falls ja, werden Hilfsvariablen angelegt und die Initialisierung der Pointer für den geteilten Speicher und einer Barriere vorgenommen, was bei der Barriere eine Synchronisierung über den Block erfordert.

```
1       // Blockzugehörigkeit in der Matrix durch i und j bestimmen
2       const int i = blockIdx.x * BLOCK_SIZE;
3       const int j = blockIdx.y * BLOCK_SIZE;
4       if (i >= j) { // if lower triangular matrix
5          // Hilfsvariablen für die späteren Kopiervorgänge
6          const int id_1d = threadIdx.y * 32 + threadIdx.x;
7          const int line = id_1d / (BLOCK_SIZE / 2);
8          const int mem_num = id_1d % (BLOCK_SIZE / 2);
9          const int transfer_line = id_1d / (BLOCK_SIZE / 4);
10         const int tranfser_offset = id_1d % (BLOCK_SIZE / 4);
11         // Innerhalb des 1kByte großen Systembuffer im geteilten Speicher
12         // einen Bereich für eine blockweite Barriere anlegen.
13         __shared__ ::cuda::barrier<::cuda::thread_scope_block> bar[1];
14
15         // Pointer auf den reservierten dynamisch geteilten Speicher,
16         // dieser hält die Matrixergebnisse
17         extern __shared__ double solution[];
18         // Start-Pointer auf Is und Js
19         double *Is = (double *) &solution[0];
20         double *Js = (double *) &solution[0] + 8 * BLOCK_OFF;
21         // Pointer auf die "letzten 4 Reihen" im geteilten Speicher
22         // für die Vektoren für GEMV
23         double *Vjs = (double *) &solution[0] + BLOCK_SIZE * BLOCK_OFF;
24         double *Vis = (double *) &solution[0] + BLOCK_SIZE * BLOCK_OFF + 1 * BLOCK_OFF;
25         double *Qis = (double *) &solution[0] + BLOCK_SIZE * BLOCK_OFF + 2 * BLOCK_OFF;
26         double *Qjs = (double *) &solution[0] + BLOCK_SIZE * BLOCK_OFF + 3 * BLOCK_OFF;
27         // Initialisierung der Barriere
28         if (threadIdx.x == 0) {
29            init(&bar[0], THREADS_PER_BLOCK);
30         }
31         // Nach der Initialisierung der Barriere,
32         // muss diese im Block synchronisiert werden.
33         __syncthreads();
```

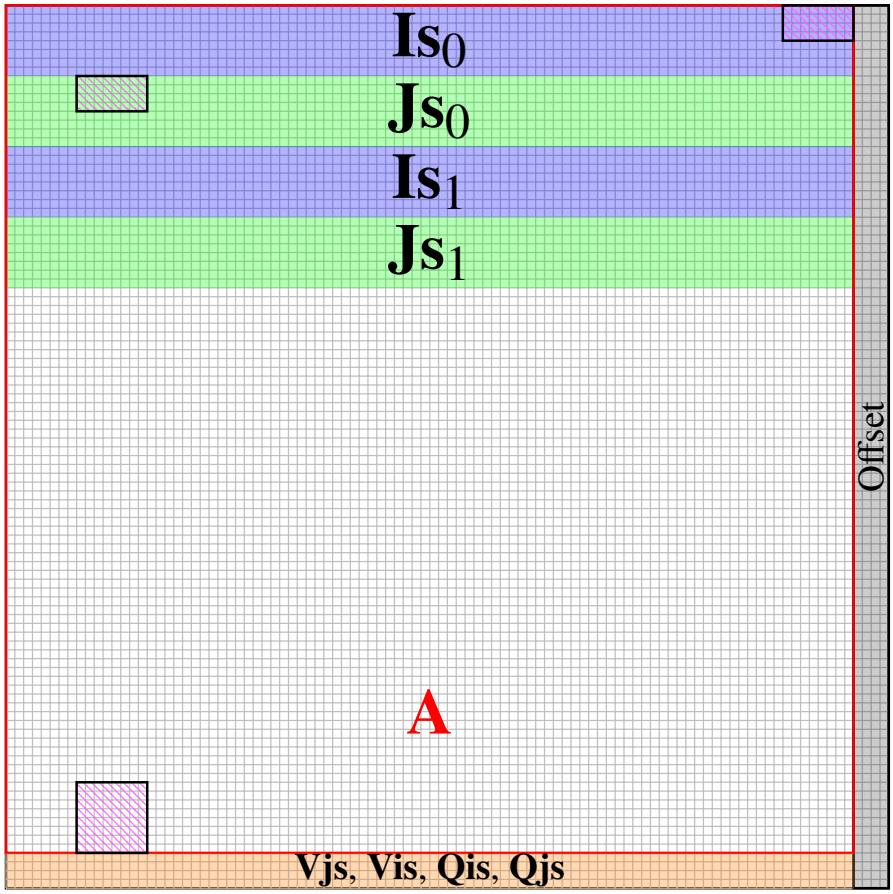


Abbildung 4.5: Layout des geteilten Speichers des Kernels für Matrixeinheiten in doppelter Genauigkeit: Zu sehen ist der gesamte belegte geteilte Speicher. Eine Zelle entspricht dabei einem FP64. Für jeweils 96 Trainingsdaten in i- und j-Richtung, siehe Abbildung 4.3, werden in den Bereichen 0 und 1 alternierend jeweils 8 Features der Trainingsdaten für die Indizes i und j blau markiert in **Is** und grün markiert in **Js** gespeichert. Die Matrix **A** wird wie in Abbildung 4.4b am Ende des Matrixaufbaus in dem rot umrandeten Bereich gespeichert. Pink schraffiert und schwarz umrandet sind die Daten der Matrixoperation für $\mathbf{W_{11}}[\mathbf{1}]$ in $\mathbf{Is}_0$ und $\mathbf{Js}_0$ und wo die Lösung nach dem Matrixaufbau in **A** gespeichert wird. Für die GEMV-Operation werden noch die in Orange gekennzeichneten Vektoren **Vjs**, **Vis**, **Qis** und **Qjs** benötigt. Um Speicherbank-Konflikte zu vermeiden, wird nach jeder Zeile ein Offset der Länge 4 angelegt, der grau markiert ist.

Matrixaufbau

Um nicht nach jeder Iteration die Zwischenergebnisse in den geteilten Speicher schreiben und kurz darauf wieder laden zu müssen, werden für den Matrixaufbau, wie in Listing 4.10 zu sehen ist, 12 Akkumulatoren und jeweils ein matrix_a- und matrix_b-Fragment angelegt und die Akkumulatoren auf 0 initialisiert. Da nach Gleichung (4.6) $\mathbf{X}_i$ und folglich auch **Is** transponiert sein müssen, wird `a_frag` in line 2 von Listing 4.10 als `nvcuda::wmma::col_major` angelegt. Dadurch muss vor der Matrixoperation nicht extra transponiert werden. Ein `matrix_a`- beziehungsweise `matrix_b`-

Fragment hat jeweils eine Dimension von 8×4 und jedes `accumulator`-Fragment von 8×8. Für jedes Matrix-Fragment wird also 1 Variable und für jeden Akkumulator 2 Variablen vom Typ Double pro Thread gespeichert. Zusammen macht dies $12 \cdot 2 + 2 \cdot 1 = 26$ Doubles und damit 52 an 32 bit Registern. Für eine zwischenzeitliche Implementierung mit einer Blockgröße von 128 und 16 Warps gab es zu wenige zur Verfügung stehende Register um jedes Zwischenergebnis in einem Akkumulator zu speichern, was zu erheblichen Geschwindigkeitseinbußen führte und der Grund für die aktuelle Größe darstellt.

Listing 4.10 Anlegen der Matrix Fragmente a und b und 12 Akkumulatoren pro Warp, die dann auf 0 gesetzt werden. Dabei wird das a-Fragment spaltenbasiert angelegt, wodurch weitere Speicher-Transformationen nicht mehr nötig sind. Für symmetrische Blöcke ist `ROLL_SIZE` die Anzahl an Warps

```
// Initialisierung von jeweils 1 matrix_a und matrix_b Fragment
nvcuda::wmma::fragment<nvcuda::wmma::matrix_a, 8, 8, 4, double, nvcuda::wmma::col_major> a_frag;
nvcuda::wmma::fragment<nvcuda::wmma::matrix_b, 8, 8, 4, double, nvcuda::wmma::row_major> b_frag;
// Initialisierung von einem Array mit 12 Akkumulatoren
nvcuda::wmma::fragment<nvcuda::wmma::accumulator, 8, 8, 4, double> c_frag[ROLL_SIZE];
// Diese werden alle mit 0.0 gefüllt
for (int frags = 0; frags < ROLL_SIZE; ++frags) {
   nvcuda::wmma::fill_fragment(c_frag[frags], 0.0f);
}
```

Der Matrixaufbau lässt sich in 2 Teile gliedern, das asynchrone Laden von $\mathbf{X}_i$ und $\mathbf{X}_j$ aus dem globalen Speicher in den geteilten Speicher **Is** und **Js** und in die Matrixoperationen. Vor der ersten Iteration über die Features werden direkt die Daten der Features 0-7 aus dem globalen Speicher geladen. Innerhalb der Schleife wird dann zuerst ein Offset berechnet, siehe Listing 4.11. Dieser prüft das *8er-Bit* auf 0 oder 1, also ob eine gerade oder ungerade Iteration läuft, und multipliziert gegebenenfalls die 8 mit dem Blockoffset von 100 um dadurch den Offset in den gedoppelten beziehungsweise zweiten Bereich des **Is** oder **Js** Arrays zu erhalten, diese liegt $8 \cdot 100 \cdot 2 = 1600$ Positionen hinter **Is** beziehungsweise **Js**, wie in Abbildung 4.5 zu sehen ist. Die darauf folgende Barriere synchronisiert alle Threads des Blockes und wartet bis alle auf diese Barriere initialisierten asynchronen Speicheroperationen abgeschlossen sind. Laut den CUDA-Leitfäden [NVI22a; NVI22b] sind 128 Bit große Speicheroperationen am effizientesten, weshalb ein double2 Datentyp übertragen wird und dafür ein `reinterpret_cast()` vorgenommen wird. Ein double2 ist ein in CUDA integrierter Vektortyp, der 2 FP64-Zahlen beinhaltet [NVI22a]. Gut zu wissen bei 128 Bit Speicheroperationen ist, dass die Zugriffe und Ausführungen geviertelt werden, zuerst Thread 0-7, dann Thread 8-15 und so weiter, insgesamt 4 Takte lang.

Für eine Matrixoperationen-Iteration über die Trainingsdaten innerhalb des Blockes müssen **Is** und **Js** mit jeweils mindestens 4 Features geladen werden, was insgesamt $2 \cdot 96 \cdot 4 = 768$ Datenpunkte ergibt. Demgegenüber stehen 384 Threads pro Block, was bei jeweils 128 Bit großen Transfers einen Kopiervorgang pro Thread aus dem globalen in den geteilten Speicher bedeutet. Da der präferierte 128×128 große Block wegen mangelnder Register nicht möglich ist, was 2 Warps an Speicherzugriffen pro Zeile in **Is** oder **Js** bedeuten würde, muss für eine optimale Zuordnung unter anderem eine teure Modulo-Operation ausgeführt werden, weshalb mehrere Hilfsvariablen am Anfang des Kernels definiert werden (Listing 4.9).

Für eine ausreichende Nebenläufigkeit, besonders um die Speichertransfers aus dem globalen in den geteilten Speicher oder in die Register zu verbergen, ist die Standard-Implementierung auf die parallele Ausführung von mindestens 2 Blöcke auf einem SM angewiesen. Während der eine Block auf die Zugriffe wartet, kann der andere Block rechnen und umgekehrt. Durch die asynchronen Kopiervorgänge, und 2 Bereiche für **Is** und **Js**, wie zum Beispiel in line 21, funktioniert der vorgestellte Kernel auch mit einem aktiven Block pro SM schnell. So wird als erster Schritt von Iteration s der Kopiervorgang der Daten für Iteration $s + 1$ gestartet. Gleichzeitig laufen, wie in Abschnitt 3.3.2 erwähnt, asynchrone Kopiervorgänge nicht über den L1 Cache und die Register, was daher also weniger Registerdruck verursacht. Ein großer Nachteil an den asynchronen Kopiervorgängen ist allerdings die benötigte Barriere nach den Transaktionen, welche die Laufzeit negativ beeinflussen. Aus diesem Grund werden die Daten für jeweils 2 Iterationen auf einmal geladen und nur jede 2. Iteration synchronisiert, was den Kernel um ungefähr 20% beschleunigt und zum in Abbildung 4.5 gezeigten Speicherlayout führt.

Listing 4.11 Asynchrone Speichertransaktionen beim Matrix-Kernel in doppelter Genauigkeit. Zu Beginn jeder Iteration über die Features wird ein Offset berechnet, welches darüber entscheidet in welchen Bereich der doppelt angelegten **Is** oder **Js** Vektoren die Daten geladen werden. Im nächsten Schritt wird mit einer Barriere sichergestellt, dass alle asynchrone Speicheroperationen aus der letzten Iteration abgeschlossen sind. Darauf wird von jedem Warp Elemente für entweder **Is** beziehungsweise **Js** aus dem globalen Speicher asynchron geladen, jeder Thread überträgt dabei 256 Bit.

```
1  ...
2  // Loop über die Features, pro Durchgang werden 8 Features bearbeitet.
3  for (int feature_it = 8; feature_it < feature_range; feature_it = feature_it + 8) {
4     // Offset um alternierend im jeweils ersten oder zweiten Bereich
5     // von Is und Js zu schreiben
6     const int off_plus = (feature_it & 8) * BLOCK_OFF * 2;
7     // Warte bis die neuen Daten in  I und J transferiert sind
8     bar[0].arrive_and_wait();
9
10    if (threadIdx.y < 6) {  // warp 0-5
11       // 6 Warps pro 8 Feature Zeilen für I
12       // --> 96*8/(32*6)= 4 Doubles pro Thread
13       // double2 Pointer an die richtige Zeile und Spalte
14       // ... in den geteilten Speicher
15       double2 *const I2s = reinterpret_cast<double2 *>
16             (&Is[transfer_line * BLOCK_OFF + tranfser_offset * 4 + off_plus]);
17       // ... in den globalen Speicher
18       const double2 *const D_i2 = reinterpret_cast<const double2 *>
19             (&in[transfer_line * points + tranfser_offset * 4 + feature_it * points + i]);
20       // Asynchroner Kopiervorgang
21       ::cuda::memcpy_async(I2s, D_i2, 2 * sizeof(double2), bar[0]);
22    } else{  // warp 6-11
23       // 6 Warps pro 8 Feature Zeilen für J, identisch zu I
24       double2 *const J2s = reinterpret_cast<double2 *>
25             (&Is[transfer_line * BLOCK_OFF + tranfser_offset * 4 + off_plus]);
26       const double2 *const D_j2 = reinterpret_cast<const double2 *>
27             (&in[(transfer_line - 8) * points + tranfser_offset * 4 + feature_it * points + j]);
28       ::cuda::memcpy_async(J2s, D_j2, 2 * sizeof(double2), bar[0]);
29    }
```

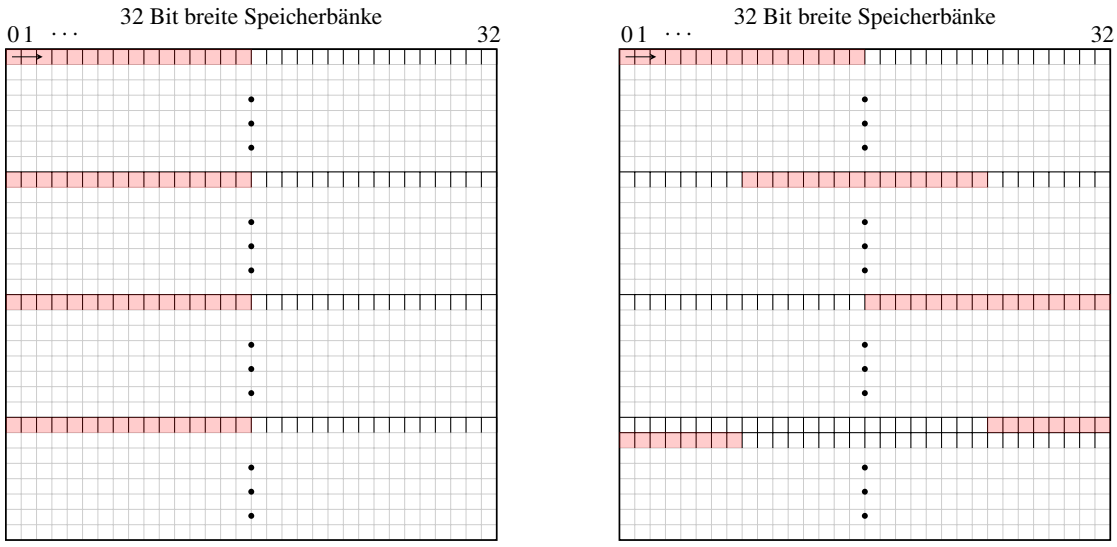


Abbildung 4.6: Darstellung der Daten eines Fragmentes (rot) im geteilten Speicher ohne (links) und mit (rechts) Offset. Ohne Offset liegt ein Fragment nur auf der Hälfte der Speicherbänke, die Bandbreite wird halbiert. Mit Offset werden alle Speicherbänke gleichmäßig ausgelastet.

Für die eigentliche Matrixoperation wie in Listing 4.12 beschrieben, haben sich 2 Dinge als zentral herausgestellt:

- Um eine sinnvolle Nebenläufigkeit bei den Matrixoperationen zu haben, müssen mindestens 12 Warps auf der SM aktiv sein. Dabei spielt es keine Rolle, ob diese in einem Block oder in mehreren Blöcken laufen. Eine Implementierung des 128×128 Blockes mit 8 Warps, dadurch gerade ausreichend viele Register um diesen umsetzen zu können, ist trotz vieler anderer Vorteile wie weniger Speicheroperationen um circa 50% langsamer als die derzeitige Matrixoperationen Implementierung mit 96×96 großen Blöcken und 12 Warps. Gleichzeitig ist eine Implementierung mit 64×64 Blöcken und wieder jeweils 8 Warps, bei der allerdings gleichzeitig 2 Blöcke auf einem SM laufen, zwar schneller als die 128×128 Variante, aber trotz insgesamt 16 aktiven Warps um circa 20% langsamer als die Version mit 12 Warps.
- Ein effizientes Laden und Speichern von und in den geteilten Speicher ist zentral. Einerseits gilt es Zugriffe möglichst zu minimieren, weshalb die Speicherstruktur innerhalb eines Blockes so aufgebaut ist, dass `matrix_a` für jeden Warp nur einmalig pro Feature geladen werden muss und danach nur noch `matrix_b` durchwechselt. Ein Beispiel welche Daten für eine Matrixoperation geladen werden müssen, ist in Abbildung 4.5 zu sehen: Die Daten aus $\mathbf{Is}_0$ bleiben dabei konstant, während über die Zeile $\mathbf{Js}_0$ iteriert wird. Zusätzlich spielt der Offset, wie in Abbildung 4.6 zu sehen, eine zentrale Rolle. Ohne diesen läuft der Kernel ungefähr 80% langsamer.

In Listing 4.12 wird der Ablauf der Matrixoperation dargestellt. Für die Auswahl des richtigen Speicherbereiches, also ob $\mathbf{Is}_0$ oder $\mathbf{Is}_1$, identisch für $\mathbf{Js}$, wie in Abbildung 4.5 zu sehen, gibt es für das Laden der Daten für die Matrixoperation einen eigenen Offset. Dieser alterniert in jeder Iteration mit dem Offset der Kopiervorgänge. Ein Beispiel für die Zuordnung der Daten einer Matrixoperation kann in Abbildung 4.5 betrachtet werden. Da `a_frag` spaltenbasiert angelegt wird,

werden die Daten beim Laden in line 8 direkt transponiert und haben dann die Dimension 8×4 Bei den Schleifen ist zu beachten, dass die Änderungen pro Iteration möglichst eindimensional und simpel sind. Beim Entrollen der Schleife kann eine falsche Operation 2 oder mehrere Register pro Iteration benötigen, was in diesem Falle 24 oder mehr Register bedeuten kann. Gleichzeitig ist der Matrixaufbau der Register-intensivste Teil, weil alle Fragmente in den Registern gehalten werden müssen.

Im Inneren der Schleife wird in line 16 `b_frag` zeilenbasiert geladen mit den Dimensionen 4×8, dann wird die Matrixoperation ausgeführt und die Zwischenergebnisse der Dimension 8×8 aber Lokal in den Registern gehalten. Dadurch ist zwar keine Zuordnung möglich, aber dies ist während des Aufsummierungsprozesses nicht notwendig. Es werden pro Iteration über 4 Features 13 Fragmente pro Warp aus dem geteilten Speicher geladen und insgesamt 12 Matrixoperationen durchgeführt. Also relativ Nahe im optimalen Verhältnis von $1:1$. Ein kleiner Vorteil der Matrixoperationen ist, dass immer 4 Zwischenergebnisse aus der Multiplikation addiert und dann auf den Akkumulator addiert werden und dabei eine Genauigkeit von einer FMA Operation erreichen beziehungsweise etwas genauer, da zuerst das Zwischenergebnis aus 4 Multiplikationen gespeichert wird. Dadurch viertelt sich die Anzahl an einzelnen Additionen. Nachdem alle Matrixoperationen ausgeführt worden sind, werden die Ergebnisse aus den Registern wie in Listing 4.13 in den geteilten Speicher in der Form von Abbildung 4.4 geschrieben und dadurch wieder zuordenbar.

Listing 4.12 Für die Matrixoperation im Kernel für Matrixeinheiten in doppelter Genauigkeit wird zuerst matrix_a aus dem geteilten Speicher in die Register geladen und bleibt für alle Matrixoperationen konstant. Dann wird über eine *Zeile* iteriert, matrix_b geladen und das Ergebnis von $A \cdot B$ auf den jeweiligen Akkumulator c_frag addiert. Dies wird dann für die nächsten 4 Features wiederholt

```
1    ...
2      // Iteriere über 2x 4 Features
3    for (int  mem_roll = 0; mem_roll < 2; ++mem_roll){
4       // Pointer auf die ersten zu ladenden Daten aus I im geteilten Speicher
5       double *const shmem_i_ptr = (double *) &Is[0]
6             + 8 * threadIdx.y + 4 * BLOCK_OFF * mem_roll + off_minus;
7       // Lade Daten aus shmemi_ptr in a_frag mit BLOCK_OFF = 100
8       nvcuda::wmma::load_matrix_sync(a_frag, shmem_i_ptr, BLOCK_OFF);
9       #pragma unroll
10      // Iteriere über die 12 8x8x4 Daten des Warps im Block
11      for (int j_roll = 0; j_roll < ROLL_SIZE; ++j_roll) {
12         // Pointer auf die ersten zu ladenden Daten aus I im geteilten Speicher
13         double *const shmem_j_ptr = (double *) &Js[0] + 8 * j_roll
14                + 4 * BLOCK_OFF * mem_roll + off_minus;
15         // Lade Daten aus shmemj_ptr in b_frag mit BLOCK_OFF = 100
16         nvcuda::wmma::load_matrix_sync(b_frag, shmem_j_ptr, BLOCK_OFF);
17
18         // TensorCoreOperation der Form C[j_roll] += A*B
19         nvcuda::wmma::mma_sync(c_frag[j_roll], a_frag, b_frag, c_frag[j_roll]);
20      }
21   }
22 } // end of feature iteration
```

Gegen Ende der Arbeit hat sich bei den Tests ein weiteres Problem herausgestellt: Bei allen CUDA-Kernel kann es bei großen Datensätzen dazu kommen, dass die Anzahl an Datenpunkten multipliziert mit den Features größer ist als die maximal darstellbare Zahl im 32 Bit Integer

oder Unsigned Zahlenformat, siehe auch Abschnitt 3.2.1. Dadurch kommt es zu einem Überlauf und falschem beziehungsweise undefiniertem Verhalten beim Laden der Trainingsdaten aus dem globalen Speicher, zum Beispiel in line 18, was zu einem kritischen Fehler führen kann. Der naive Ansatz wäre mit dem Datentyp long, der 64 Bits lang ist, zu rechnen, was sich allerdings negativ auf die Laufzeit auswirkt. Dies lässt sich einfach umsetzen, da die Integer in den Kernel als Type alias `using kernel_index_type = int;` hinterlegt sind. Eine performante Lösung, also unter Beibehaltung des Integerformates, was für Compiler-Optimierungen verwendet werden soll [NVI22a], wurde vorgeschlagen, mehr dazu in Kapitel 6.

Matrix Vektor Multiplikation

Bei der Matrix Vektor Multiplikation gibt es gegenüber der Standardimplementierung den Vorteil, dass die Teil-Matrix $\mathbf{A}_{i,j}$ als Ganzes im geteilten Speicher vorliegt. Daher kann die Operation blockweise und nicht Thread-weise gedacht und berechnet werden. Dies spart eine Menge an Divergenz und Kontrollmechaniken, stabilisiert die Berechnung und reduziert die benötigten `atomicAdd`-Befehle drastisch. Diese Funktion sperrt eine Variable im globalen Speicher vor fremden Zugriffen, transferiert diese Variable, addiert den gewünschten Wert auf sie und transferiert sie zurück, bevor sie wieder freigegeben wird. Wenn viele Threads gleichzeitig auf die gleiche Variable zugreifen wollen, erhöht dies die Laufzeit stark.

Um den Ausgleich der Dimensionsreduktion und den diagonalen Teil nur einmal zu berechnen, wird unterschieden zwischen GPU 0 und möglichen anderen. In Listing 4.14 ist der Algorithmus für GPU 0 gezeigt. Noch vor dem gezeigten Code-Teil werden die Vektoren **Vis**, **Vjs**, also die benötigten Daten des Vektors **v**, und die Vektoren **Qis** und **Qjs**, die benötigten Daten für die dimensionsreduzierende Ausgleichsberechnung, aus dem globalen Speicher in den geteilten Speicher geladen. In der darauf folgenden und gezeigten GEMV Rechnung wird wieder ein Offset verwendet, um die Zugriffe auf den geteilten Speicher zu optimieren. Dies geschieht durch die `index`-Variable in line 8, die über eine Modulo-Operation einen fortlaufenden Speicherzugriff bei den Bänken gewährt. So beginnt Thread 0 in Spalte 0, Thread 1 in Spalte 1 und so weiter. Die Modulo Operation erhöht in der For-Schleife zwar deutlich die Anzahl benötigter Register, da allerdings der registerintensive Matrix-Aufbau Teil mit den ganzen Fragmenten vorbei ist, fällt dies nicht weiter ins Gewicht. Auch die Abfrage, ob der Block sich auf der Diagonalen befindet oder nicht, muss nur einmal gestellt werden. In line 12 und damit für GPU 0 wird aus **A** die durch die Dimensionsreduktion modifizierte Version $\hat{\mathbf{Q}}$ ohne die Konstante auf der Diagonale von $1/C$, die dann in line 17 hinzukommt. $\hat{\mathbf{Q}}$ wird direkt mit **Vjs** multipliziert und ergibt so ein skalares Teilergebnis der GEMV. Falls der Kernel nicht von

Listing 4.13 Die Ergebnisse der Akkumulatoren werden im Kernel für Matrixeinheiten in doppelter Genauigkeit nach der Iteration über die Features in den geteilten Speicher geschrieben.

```
1   ...
2    // Zurückschreiben der c_frag Akkumulatoren in den
3   for (int frags = 0; frags < ROLL_SIZE; ++frags) {
4      // Pointer in die richtige Stelle der Blockmatrix im geteilten Speicher
5      double *const shmem_m_ptr = (double *) &solution[0] + 8 * threadIdx.y * BLOCK_OFF + 8 * frags;
6      // Schreibvorgang mit Offset von 100.
7      nvcuda::wmma::store_matrix_sync(shmem_m_ptr, c_frag[frags],
8            BLOCK_OFF, nvcuda::wmma::mem_row_major);
9   }
```

GPU 0 aufgerufen wird, wird direkt **A** mit dem Vektor **Vjs** multipliziert. Ein weiterer Vorteil liegt an der erhöhten Stabilität, so werden hier 96 Additionen zusammengefasst und erst dann auf das Gesamtergebnis des Lösungsvektors **b** addiert. Dies führt ebenfalls dazu, dass pro Zeile nur noch eine teure `atomicAdd`-Funktion ausgeführt werden muss.

Listing 4.14 Matrix Vektor Multiplikation im Kernel für Matrixeinheiten in doppelter Genauigkeit: Von den ersten 3 Warps berechnet jeder Thread eine Zeile in der Matrix. Falls der Block auf der Diagonalen der Gesamtmatrix liegt, wird zusätzlich $1/C$ addiert, bevor das kumulierte Ergebnis am Ende in den Ergebnisvektor geschrieben wird.

```
...
// GEMV Berechnung mit Offset
if (threadIdx.y < BLOCK_SIZE / WARP_SIZE) {  // 96/32=3 --> Warp 0-2
   double sol_tmp = 0.0;
   // Jeder der 96 Threads iteriert über eine Zeile im 96x96 Block.
   for (int store_it = 0; store_it < BLOCK_SIZE; ++store_it) {
      // Index-Verschiebung für effizienteren Speicherzugriff
      const int index = (id_1d + store_it) % BLOCK_SIZE;
      // Qi ist zeilenweise konstant und wird in die Register geladen
      const double Qi = Qis[id_1d];
      // Eigentliche GEMV Berechnung
      sol_tmp += (solution[id_1d * BLOCK_OFF + index] * gamma
             - Qi - Qjs[index] + QA_cost) * Vjs[index];
   }
   if (i == j) {
      // Addiere kosten auf die Diagonale, fall i = j
      sol_tmp += cost * Vjs[id_1d];
   }
   // Atomares Aufaddieren auf den Lösungsvektor
   atomicAdd(&out[i + id_1d], sol_tmp * add);
}
```

Liegt der Block, wie in Abbildung 4.3 gezeigt, nicht auf der Diagonalen, sondern darunter, also $i > j$, behandeln Warp 3-5 die transponierte obere Dreiecksmatrix. Dabei ist das Prinzip ähnlich. Für die GPUs $1, 2, \ldots$ sieht dieser Teil fast identisch aus, er benötigt nur weder die Vektoren mit den Hilfsvariablen **Qis** und **Qjs**, die auch nicht weiter verrechnet werden, noch wird der Zusatz für die Diagonale berechnet.

Pro Kernel mit $i \neq j$ werden insgesamt $2 \cdot 96$ `atomicAdd`-Funktionen ausgeführt, ansonsten 96, wobei die Zahlen fortlaufend vorliegen. Es kann also der gesamte Warp eine `atomicAdd`-Funktion gleichzeitig ausführen und somit mit wenigen Speicherzugriffen, bei denen bedacht werden muss, dass ja nicht eine einzelne Variable gleichzeitig aus dem globalen Speicher geladen wird, sondern standardmäßig 64 Bytes, Somit werden nicht nur insgesamt deutlich weniger `atomicAdd`-Funktionen ausgeführt, sondern im Optimalfall Latenz technisch nur $192/8 = 24$ Stück, da direkt 8 Variablen gesperrt und transferiert werden können und die benötigten weiteren Variablen nicht gesperrt sind.

Die Standardimplementierung hat einerseits if-else-Konstrukte im Inneren der Schleife, anderseits werden $96 \cdot 96$ `atomicAdd`-Funktionen für die obere Dreiecksmatrix durchgeführt, die sich gegenseitig blockieren, und $96 \cdot 6$ sich gegenseitig blockierende für die untere Dreiecksmatrix.

Am Ende steht der gesuchte Vektor im globalen Speicher und der Kernel endet.

4.3.2 Kernel mit einfacher Genauigkeit

Der generelle Aufbau des Kernels mit einfacher Genauigkeit ist gleich. Die Größe der Fragmente und Akkumulatoren ist aber deutlich größer. Ein Fragment hat die Dimension 16×8 und enthält damit 4× so viele Variablen wie das Fragment mit doppelter Genauigkeit. Der Akkumulator hat eine Dimension von 16×16, also die gleichen Verhältnisse wie beim Fragment. Insgesamt also das Vierfache an Variablen mit halber Größe, folglich doppelt so viele Register. Die Blockgröße wurde auf 128×128 gesetzt. Das Problem daran ist ein enormer Registerdruck, weshalb hier pro SM nur 8 Warps gleichzeitig aktiv sein können.

Listing 4.15 Besonderheiten des TF32 Formats: Gezeigt wird die Initialisierung und Nutzung eines TF32 Fragmentes. Dabei müssen nach dem Laden in die Register alle Einträge des Fragmentes in jedem Thread von FP32 zu TF32 transformiert werden.

```
...
// Initialisierung eines TF32 Fragmentes mit den Dimensionen 16 16 8
nvcuda::wmma::fragment<nvcuda::wmma::matrix_a, 16, 16, 8,
      nvcuda::wmma::precision::tf32, nvcuda::wmma::col_major> a_frag;
...
// Befüllen der Matrix a_frag aus dem geteilten Speicher ausgehend von shmem_i_ptr
float *const shmem_i_ptr = (float *) &Is[0] + 16 * threadIdx.y + off_minus;
nvcuda::wmma::load_matrix_sync(a_frag, shmem_i_ptr, BLOCK_OFF_F);

// Die geladenene Daten müssen von FP32 zu TF32 transferiert werden.
// Da dies für alle Elemente gilt, ist die Zuordnung irrelevant.
#pragma unroll
for (int t = 0; t < a_frag.num_elements; ++t) {
   a_frag.x[t] = nvcuda::wmma::__float_to_tf32(a_frag.x[t]);
}
```

Da reine FP32 Operanden bei NVIDIA nicht als Matrixoperation unterstützt werden, müssen die Inhalte der Fragmente in das TF32 Format transformiert werden. Eine TF32 Variable hat dabei die Mantisse einer Variablen mit halber Genauigkeit (10 Bits) und den Exponenten wie eine Variable mit einfacher Genauigkeit (8 Bits), gemeinsam mit den Vorzeichen hat dieser Datentyp nur 19 Bits an Länge. Die Variablen des Fragmentes verlieren an dieser Stelle Genauigkeit. Da die Operationsart aber eine Multiplikation ist und normalerweise die Mantisse halbiert wird, was hier nicht passiert, ist der Effekt nicht ganz so stark wie erwartet. Im Gegenzug werden die Anzahl an Additionen auf dem Akkumulator beim Matrixaufbau durch 8 geteilt, ebenso in der äußeren Schleife der Matrix Vektor Multiplikation um Faktor 128 verringert. Die Genauigkeit und Stabilität nimmt dadurch wieder zu. Die kumulierten Auswirkungen werden im nächsten Kapitel betrachtet.

5 Ergebnisse

In diesem Kapitel werden die Auswirkungen der im letzten Kapitel diskutierten Änderungen auf Konvergenz, Hardwareausnutzung und Laufzeit untersucht. Zuerst wird kurz die für die Untersuchungen verwendete Hard- und Software beschrieben, um besonders die Laufzeitergebnisse besser nachzuvollziehen können. Danach wird das Konvergenzverhalten des CG-Algorithmus in Abhängigkeit verschiedener Parameter getestet. Diese beinhalten die verwendete Genauigkeit der GEMV-Operationen, die Wahl der β-Berechnungsvorschrift, dem Tensor- oder Standard-Kernel und der Wahl eines möglichen Korrekturschemas und die Größe des Problems, die sich direkt auf die Kondition und die Anzahl an Operationen auswirkt. Im folgenden Abschnitt werden die einzelnen CUDA-Kernels mittels Profiling Ergebnissen auf die Hardware-Ausnutzung und ihre Laufzeit untersucht. Ebenso wird untersucht, ob und wie sich die einzelnen Teile im CG-Algorithmus bezüglich ihrer Laufzeit ändern und welchen Anteil sie einnehmen. Im letzten Abschnitt wird dann das Zusammenspiel von Konvergenzverhalten und Laufzeit kombiniert auf Probleme angewandt, um Handlungsempfehlungen, Schwächen und Stärken der Implementierung zu evaluieren.

5.1 Setup

Im Folgenden werden die genutzten Systeme samt relevanter Software beschrieben so wie die verwendeten Trainingsdaten.

5.1.1 Hardware

Da die neuen Kernel, welche die Matrixeinheiten nutzen, nur für CUDA implementiert worden sind, werden die Tests auf die in Tabelle 5.1 gezeigte CUDA fähige GPUs und Systeme eingeschränkt. Die Änderungen außerhalb der Kernel sind allerdings für alle Kernel und Backends identisch, hier werden folglich vergleichbare Werte erwartet.

5.1.2 Software

Alle Systeme nutzen Ubuntu 20.04.1, den GNU GCC 9.4.0 Compiler und OpenMPI 4.0.3. Alle Systeme bis auf ipvsgpu1 nutzen CUDA Version 11.4 mit Treiber Version 470.141.03. Ipvsgpu1 nutzt CUDA Version 11.6 mit Treiber Version 510.85.02. Kompiliert wurde PLSSVM jeweils für das CUDA-Backend mit passender Architektur.

	Prozessor	Anzahl Kerne/Threads	Prozessor-basistakt	RAM	NVIDIA GPU
argon-gtx*	Dual Socket Intel Skylake Xeon Gold 5120	28/56	2,2 GHz	755 GB DDR3	8× GeForce GTX 1080 ti 11 GB [NVI16a]
argon-tesla1*	Dual Socket Intel Skylake Xeon Silver 4116	24/48	2,1 GHz	188 GB DDR4	2× Quadro GP100 16 GB [NVI16b]
pcsgs02	Intel Cascade Lake i9-10980XE	18/36	3,0 GHz	64 GB DDR4	GeForce RTX 3080 10 GB [NVI21]
pcsgs11	AMD Zen2 Ryzen Threadripper 3960X	24/48	3,8 GHz	128 GB DDR4	2× GeForce RTX 3090 24 GB [NVI21]
ipvsgpu1	AMD Zen2 EPYC 7742	128/256	2,25 GHz	1024 GB DDR4	4× NVIDIA A100 40 GB [NVI20]

Tabelle 5.1: Die für Laufzeit Tests verwendete Hardware
Die mit * gekennzeichneten Systeme werden über den Slurm Workload Manager gesteuert.

5.1.3 Testdaten

Solange nicht anders beschrieben, wurden die benutzten Datensätze mit dem PLSSVM beigefügten generate_data.py Script erstellt [VBP22b].

```
python3 generate_data.py --output folder --format libsvm --problem planes
--samples number_datapoints --features number_features
```

Dabei wurden für eine verschiedene Anzahl an Datenpunkten und Features Datensätze generiert, bei denen die Datenpunkte in 2 verschiedene aneinander angrenzende Cluster aufgeteilt worden sind, die sich in einigen Punkten mit geringer Wahrscheinlichkeit überschneiden können. Diese Datensätze ergeben repräsentative Ergebnisse bezüglich der Geschwindigkeit einzelner Iterationsschritte, da diese nur von der Größe abhängen. Allerdings sind die Anzahl an benötigten CG-Schritten bis zu einer Lösung, bei der die Klassifizierung zu über 90% korrekt ist, von synthetischen Datensätzen und zwischen verschiedenen echten Datensätzen sehr unterschiedlich und dadurch können die letztendliche Laufzeit und benötigten CG-Schritte für allgemeine Probleme nicht genau eingeschätzt werden. Solange nicht explizit anders erwähnt, wird für das Training der lineare Kernel verwendet: Dieser ermöglicht multi-GPU Nutzung und unterscheidet sich exklusive Initialisierung und multi-GPU-Support nur in einer Zeile Code von dem polynomialen Kernel, während für den radialen Kernel keine neuen Implementierungen erstellt worden sind. Um eine verbesserte Vergleichbarkeit zwischen den Implementierungen zu gewähren, wird in den Tests nicht der von PLSSVM genutzte relative Fehler als Abbruchkriterium genutzt, sondern das Residuum r muss einen manuell festgelegten Wert unterschreiten.

SAT-6 airborne datasets

Um einen komplexen nicht synthetischen Datensatz zu untersuchen, wird zusätzlich der SAT-6 Datensatz von Basu et al. [BGM+15] genutzt. Dieser Datensatz beinhaltet 405,000 einzelne 28×28 Pixel große in jeweils 4 Farbkanälen gespeicherte Satellitenaufnahmen, die unter 6 verschiedenen Klassen gelabelt sind. Dankenswerterweise wurde der Datensatz von meinen Betreuern mir in einem direkt für PLSSVM passenden Format zur Verfügung gestellt. Dafür wurden die Klassen „Gebäude“ und „Straßen“ auf -1 abgebildet und „Gewässer“, „Grasland“, „Bäume“ und „Ödland“ auf 1 und die insgesamt 3136 Features ($28 \cdot 28$ Pixel mit jeweils 4 Farbkanälen) auf [-1,1] skaliert [VBP22b].

5.1.4 Implementierungen

Insgesamt unterscheiden sich die hier verglichenen Implementierungen in 4 Kategorien. Dies ergibt 24 verschiedene Optionen, die noch mit der aktuellen PLSSVM Implementierung[1] mit CUDA-Backend, zukünftig Referenz-Implementierung genannt, verglichen werden. Die Referenz-Implementierung (REF) wurde für die Tests modifiziert, sodass das implementierte Korrekturschema mit der Residuums Neuberechnung nicht durchgeführt wird. Für eine bessere Lesbarkeit werden die Implementierungen nach den jeweiligen 4 Kategorien abgekürzt:

1. **Genauigkeit:** Wird ausschließlich in doppelter Genauigkeit (D) gerechnet oder mit gemischten Genauigkeiten (M).
2. **Kernel:** Wird der bereits in PLSSVM vorhandene CUDA-Kernel verwendet, im weiteren Verlauf Standard-Kernel (S) genannt, oder einer der neu implementierten Kernel mit Unterstützung für Matrixeinheiten, kurz Tensor-Kernel (T).
3. **Berechnung von β:** Wird β nach der ursprünglichen Vorschrift von F-R [HS52] (FR) oder durch die alternative Form von P-R (PR) berechnet ([PR69]).
4. **Korrekturschemata:** Wird ein Korrekturschema, also die genaue Neuberechnung des Residuums alle 50 Iterationen (R) oder die zuverlässigen Updates (ZU), genutzt oder keines (0).

So entspricht zum Beispiel M-S-FR-0 der Implementierung mit gemischten Genauigkeiten, Standard-Kernel, F-R und keinem Korrekturschema.

5.2 Konvergenz

In diesem Abschnitt wird für die unterschiedlichen Konfigurationen der Implementierungen bei verschieden großen Datensätzen untersucht, nach wie vielen Iterationen der CG-Algorithmus ein bestimmtes, kleines Residuum annimmt oder ob dieses überhaupt erreicht wird. Da es hier um das Konvergenzverhalten in Abhängigkeit von Iterationen, was nicht von der GPU an sich abhängt, und nicht um die letztendliche Laufzeit in Abhängigkeit der Zeit geht, wurden alle Tests auf dem

[1] https://github.com/SC-SGS/PLSSVM #26

ipvsgpu1 System mit 4 GPUs durchgeführt. Das zu erreichende Residuum und die Anzahl, wie oft dieser Test wiederholt worden ist, ist in jedem Beispiel separat angegeben. Dabei wird für jede Größe immer der gleiche Datensatz genutzt. Da bei paralleler Ausführung jeweils auf der CPU und GPU die Reihenfolge der Operationen nicht vorhersehbar ist und bei jedem Durchlauf höchstwahrscheinlich unterschiedlich ist, variiert die Anzahl der benötigten Iterationen bis zum Abbruchkriterium. Die daraus entstehenden Schwankungen bei gleichbleibendem Datensatz variieren unterschiedlich stark und werden vorgestellt. Um die Ergebnisse grafisch aufzubereiten, werden in diesem Abschnitt wiederholt Box-Plots genutzt. Diese zeigen in Orange den Median der Anzahl an benötigten Iterationen. Die Box um den Median entspricht dem Bereich, in dem 50% aller Auswertungen liegen. Die von der Box ausgehenden Linien nach oben und unten werden Antennen oder auch Whiskers genannt, in diesem Bereich liegen Daten des unteren und oberen Quartils, die nicht mehr als das 1.5-Fache des Interquartilsabstands (Länge der Box) entfernt sind. Datenpunkte, die nicht innerhalb dieses Bereichs liegen, werden auch als Ausreißer bezeichnet und als einzelne Kreise dargestellt. Nach 1024 Iterationen ohne Konvergenz wird der Durchlauf abgebrochen.

5.2.1 4096 Datenpunkte mit jeweils 1024 Features

Zu Beginn wird ein sehr kleines Beispiel mit 4096 Datenpunkten und jeweils 1024 Features betrachtet. Die zu erreichende Genauigkeit bis zum Abbruch wurde mit Absicht höher gesetzt als benötigt, um ein Modell zu bekommen, das die Daten auf sich selbst bezogen zu $> 95\%$ richtig klassifizieren kann.

In Abbildung 5.1 werden alle möglichen Implementierungen gezeigt, die nur schlecht oder gar nicht das gesetzte Residuum von 10^{-6} innerhalb von 1024 Iterationen, was als Maximum gesetzt worden ist, erreichen. Dies beinhaltet alle Varianten von gemischten Genauigkeiten, die das bereits in PLSSVM vorhandene Korrekturschema der Neuberechnung des Residuums alle 50 Schritte nutzt. Die gleichen Varianten mit doppelter Genauigkeit erreichen die verlangte Genauigkeit bereits in unter 50 Schritten, sodass der Korrekturschritt nicht durchgeführt wird, und entsprechen dadurch den Varianten ohne Korrekturschema. Da sich dieses Korrekturschema als sehr negativ bei bereits kleinen Problemen herausgestellt hat, wird es bei folgenden Untersuchungen nicht weiter untersucht.

Die zweite Hälfte der Varianten in Abbildung 5.1 nutzen alle die zuverlässigen Updates in Kombination mit der Berechnung von β nach Fletcher-Reeves. Hierbei ist bemerkenswert, dass in einigen Testläufen von Varianten mit doppelter Genauigkeit innerhalb von 1024 Iterationen nicht das Ziel des Residuums erreicht worden ist und dadurch schlechter konvergiert als die in Abbildung 5.2a gezeigten Varianten in gemischter Genauigkeit. Der Median dieser Varianten liegt bei ungefähr 800, während der Median bei Implementierungen mit zuverlässigen Updates und dem β nach Polak-Ribière, siehe Abbildung 5.2, um ungefähr eine Größenordnung geringer ist. Aus diesem Grund werden auch Implementierungen mit zuverlässigen Updates und β nach Fletcher-Reeves nicht weiter untersucht.

Die Implementierungen in Abbildung 5.2 werden in Varianten mit gemischter und doppelter Genauigkeit aufgeteilt. Ohne Korrekturschema benötigen die Varianten in gemischter Genauigkeit $1.7\times$ so viele Iterationen wie die Implementierungen mit ausschließlich doppelter Genauigkeit. Die Unterschiede zwischen den Varianten in doppelter Genauigkeit sind marginal, nur die REF benötigt im Schnitt 2.5 Iterationen mehr. Die M-T-PR-0 Implementierung hat 2 große negativen

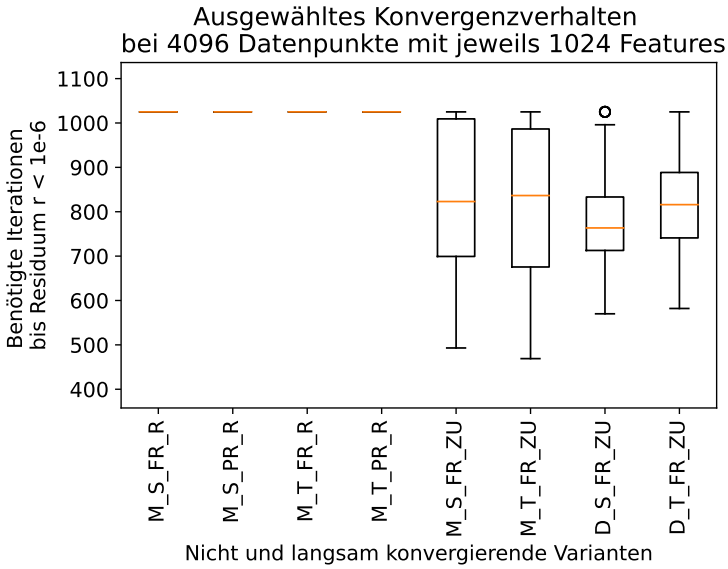
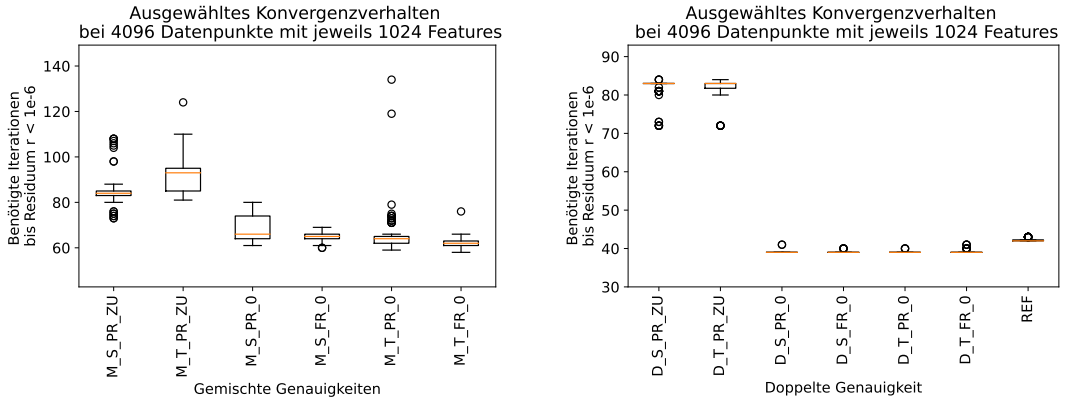


Abbildung 5.1: Dargestellt werden in Boxplots die Anzahl benötigter Iterationen bis das Residuum kleiner als 10^{-6} ist bei einer Anzahl an Durchgängen von 100 . Die ersten 4 Boxplots und Varianten entsprechen der Implementierung mit gemischten Genauigkeiten und dem Korrekturschema, das auf der Neuberechnung des Residuums beruht. Unabhängig des Kernels oder der Berechnungsvorschrift für β erreichten alle Varianten nach der maximal vorgegebenen Schrittzahl von 1024 Schritten das als Abbruchkriterium gesetzte Residuum nicht. Boxplot 5-8 stellen Varianten mit zuverlässigen Updates und der Berechnung von β nach Fletcher-Reeves, 5 und 6 sind dabei in gemischter Genauigkeit, 7 und 8 in doppelter Genauigkeit.

Abweichungen mit 119 und 134 Iterationen bis zum vorgegebenen Residuum. Bei den Varianten mit zuverlässigen Updates sind die Unterschiede bei den Iterationen bezüglich der Genauigkeiten gering, nur die M-T-PR-ZU Implementierung schneidet merklich schlechter ab. Ansonsten benötigen die zuverlässigen Updates gegenüber Varianten ohne Korrekturschema mehr als das Doppelte an Iterationen bei doppelter Genauigkeit und ungefähr 33% mehr an Iteration in gemischten Genauigkeiten. Negativ auffallend sind die Varianten M-T-PR-ZU und M-T-PR-0, beide mit Tensor-Kernel. Diese Implementierungen schneiden hier merklich schlechter ab als die Implementierungen mit Standardkernel. Hier wird vermutet, dass der Verlust an Genauigkeit durch das TF32-Datenformat, das besonders dann auftritt, wenn die Summe sich in der Größenordnung der einzelnen Summanden befindet, sich mehr bemerkbar macht als die *höhere Genauigkeit* bei der eigentlichen GEMV-Berechnung.



(a) Dargestellt werden Varianten mit gemischten Genauigkeiten. Die ersten 2 Boxplots entsprechen der Implementierung mit zuverlässigen Updates, die restlichen nutzen kein Korrekturschema. In den meisten Durchläufen werden zwischen 60 und 100 Iterationen benötigt. Die Varianten mit Tensor-Kernel haben mehr Ausreißer. M-T-FR-0 benötigt am wenigsten Iterationen, M-T-PR-ZU am meisten.

(b) Dargestellt werden Varianten in doppelter Genauigkeit. Die ersten 2 Boxplots entsprechen der Implementierung mit zuverlässigen Updates, die restlichen nutzen kein Korrekturschema. In den meisten Durchläufen werden zwischen 39 und 43 Iterationen benötigt, nur die Varianten mit zuverlässigen Updates benötigen um die 80 Iterationen. Bis auf wenige Ausnahmen wird die Anzahl des Medians an Iterationen benötigt.

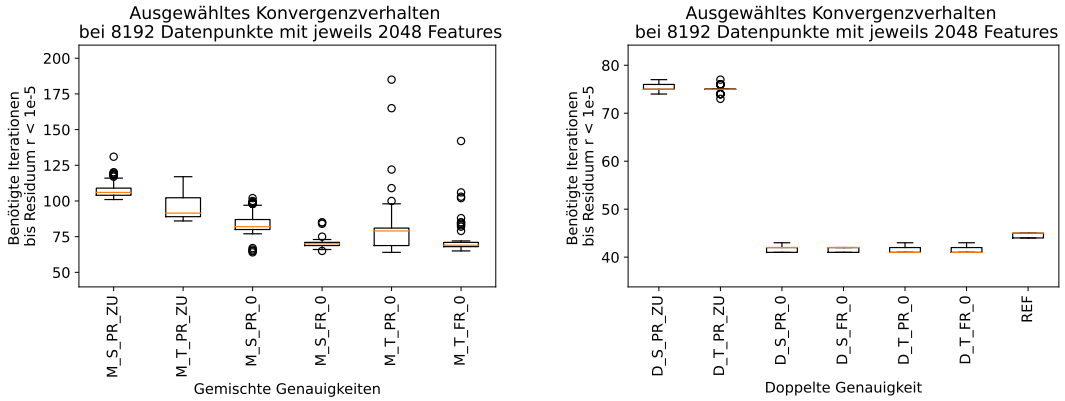
Abbildung 5.2: Dargestellt werden in Boxplots die Anzahl benötigter Iterationen bis das Residuum kleiner als 10^{-6} ist. In Abbildung 5.2a werden die Varianten in gemischter Genauigkeit gezeigt, in Abbildung 5.2b die in doppelter Genauigkeit. Es werden die Ergebnisse aus 100 Testdurchläufen dargestellt.

5.2.2 8192 Datenpunkte mit jeweils 2048 Features

Bei einem Datenset mit 8192 Datenpunkten und jeweils 2048 Features sind die Ergebnisse zum Großteil deckungsgleich gegenüber den kleineren Ergebnissen. Das Residuum als Abbruchkriterium wird etwas gelockert auf $< 10^{-5}$, da die Dimension des Residuums mit der Anzahl an Datenpunkten mitwächst, wenn auch nicht in gleicher Größenordnung wie die Lockerung. Um die Genauigkeit des vorherigen Abbruchkriteriums zu erreichen, benötigt es aber nur wenige zusätzliche Iterationen.

Bei den gemischten Genauigkeiten in Abbildung 5.3a haben weiterhin die Tensor-Kernel die meisten negativen Ausreißer, einen mit beinahe 200 Iterationen. Nur bei den zuverlässigen Updates sind die Ausreißer gegenüber dem Standardkernel geringer, ebenso die durchschnittliche Anzahl an benötigten Iterationen. Hier unterscheidet sich das Szenario von dem in Abschnitt 5.2.1, wo M-T-PR-ZU am meisten Iterationen benötigt. Die besten Varianten, M-S-FR-0 und M-T-FR-0 benutzen beide das β von F-R, was in diesem Szenario der entscheidende Faktor ist, und benötigen um die 68 Iterationen. Die Variante M-S-PR-ZU benötigt mit knapp über 100 Iterationen am längsten, bis das Abbruchkriterium erreicht wird.

Für doppelte Genauigkeiten, wie in Abbildung 5.3b zu sehen, gibt es wieder kaum Ausreißer. Die Varianten mit zuverlässigen Updates verlieren an benötigten Iterationen und brauchen nur noch um die 75 Iterationen, während die anderen Varianten etwas mehr benötigen als in Abschnitt 5.2.1,



(a) Dargestellt werden Varianten mit gemischten Genauigkeiten. Die ersten 2 Boxplots entsprechen der Implementierung mit zuverlässigen Updates, die restlichen nutzen kein Korrekturschema. In den meisten Durchläufen werden zwischen 65 und 100 Iterationen benötigt. Die Varianten mit Tensor-Kernel haben mehr und stärkere Ausreißer. Die Varianten M-T-FR-0 und M-S-FR-0 benötigen am wenigsten Iterationen, M-S-PR-ZU hingegen am meisten.

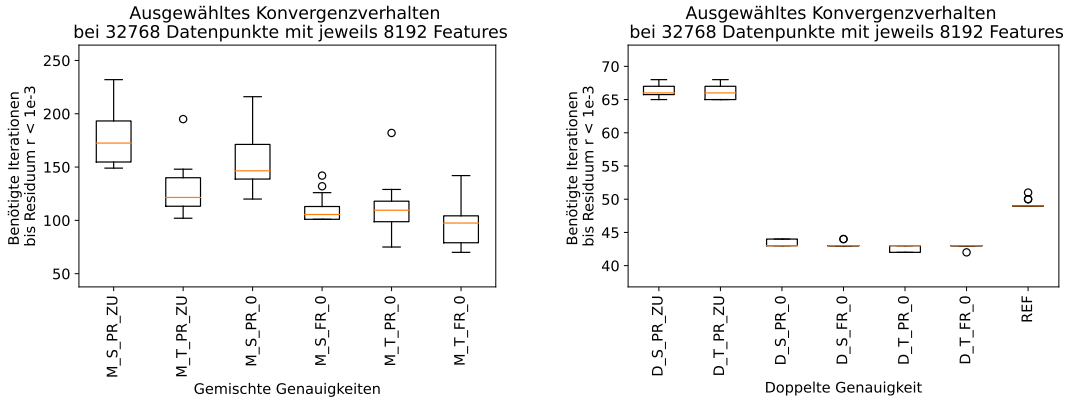
(b) Dargestellt werden Varianten in doppelter Genauigkeit. Die ersten 2 Boxplots entsprechen der Implementierung mit zuverlässigen Updates, die restlichen nutzen kein Korrekturschema. In den meisten Durchläufen werden zwischen 41 und 45 Iterationen benötigt, nur die Varianten mit zuverlässigen Updates benötigen um die 75 Iterationen. Die Abweichungen vom Median sind gering, die Implementierungen mit Tensor-Kernel schneiden am besten ab.

Abbildung 5.3: Dargestellt werden in Boxplots die Anzahl benötigter Iterationen bis das Residuum kleiner als 10^{-5} ist. In Abbildung 5.3a werden die Varianten in gemischter Genauigkeit gezeigt, in Abbildung 5.3b die in doppelter Genauigkeit. Es werden die Ergebnisse aus 64 Testdurchläufen dargestellt.

zwischen 41 und 45 Iterationen. Der Median der Tensor-Varianten ohne Korrekturschema ist dabei am niedrigsten und 10% unter dem Median der REF. Der Abstand zwischen gemischte Genauigkeiten und doppelter Genauigkeit steigt.

5.2.3 32768 Datenpunkte mit jeweils 8192 Features

Auch in diesem Beispiel wurde das Abbruchkriterium etwas gelockert, diesmal auf 10^{-3}, wobei auch dies nur wenige Iterationen weniger gegenüber dem vorherigen Abbruchkriterium bedeutet. Mit zunehmender Anzahl an Datenpunkten und Features, hier mit 32768×8192, nimmt die Genauigkeit der Tensor-Kernel gegenüber den Standard-Kernel zu. Für gemischte Genauigkeiten ist in Abbildung 5.4a zu sehen, dass bei den zuverlässigen Updates die Variante mit Tensor-Kernel ungefähr 30% weniger Iterationen als die Variante mit Standard-Kernel benötigt. Ein ähnliches, wenn auch weniger ausgeprägtes Bild zeichnet sich bei den Varianten ohne Korrekturschema ab, wobei wie in Abschnitt 5.2.1 und Abschnitt 5.2.2 die Berechnung von β nach F-R bessere Ergebnisse liefert. Für doppelte Genauigkeit sind die Unterschiede bei den Varianten mit zuverlässigen Updates weiterhin marginal, wie in Abbildung 5.4b zu sehen. Ohne Korrekturschema werden dort im Durchschnitt 43 Iterationen benötigt, die Varianten mit Tensor-Kernels mit vereinzelten Durchgängen mit 42 Iterationen, die anderen Varianten mit vereinzelt 44 Iterationen.



(a) Dargestellt werden Varianten mit gemischten Genauigkeiten. Die ersten 2 Boxplots entsprechen der Implementierung mit zuverlässigen Updates, die restlichen nutzen kein Korrekturschema. In den meisten Durchläufen werden zwischen 80 und 180 Iterationen benötigt. Die Zahl der Ausreißer nimmt gegenüber kleineren Szenarien ab. Die Varianten M-T-FR-0 benötigt mit ungefähr 100 am wenigsten Iterationen, M-S-PR-ZU hingegen mit 150-200 Iterationen am meisten.

(b) Dargestellt werden Varianten in doppelter Genauigkeit. Die ersten 2 Boxplots entsprechen der Implementierung mit zuverlässigen Updates, die restlichen nutzen kein Korrekturschema. In den meisten Durchläufen werden zwischen 42 und 50 Iterationen benötigt, nur die Varianten mit zuverlässigen Updates benötigen um die 66 Iterationen. Die Abweichungen vom Median sind gering, die Implementierungen mit Tensor-Kernel schneiden am besten ab.

Abbildung 5.4: Dargestellt werden in Boxplots die Anzahl benötigter Iterationen bis das Residuum kleiner als 10^{-3} ist. In Abbildung 5.4a werden die Varianten in gemischter Genauigkeit gezeigt, in Abbildung 5.4b die in doppelter Genauigkeit. Es werden die Ergebnisse aus 16 Testdurchläufen dargestellt.

Ab dieser Testreihe spielt die minimale Modifikation der Referenzimplementierung eine Rolle. Diese besitzt nach 50 Iterationen keine Neuberechnung des Residuums. Bei Programmdurchläufen ohne Modifikation benötigten die meisten Durchläufe 49 Iterationen, ein paar Ausreißer allerdings über 50, dadurch wurde die Neuberechnung des Residuums durchgeführt und deutlich mehr Iterationen benötigt, teilweise im 3 stelligen Bereich. Für größere Trainingsdatensätze konvergiert die Berechnung dann gar nicht mehr, vergleichbar zu den Varianten mit dem identischen Korrekturschema mit gemischten Genauigkeiten in Abschnitt 5.2.1. Daher wurde die Neuberechnung des Residuums alle 50 Iterationen deaktiviert.

Die durchschnittliche Anzahl an benötigten Iterationen der REF hat sich bei dieser Größe und diesem Problem auf +14% gegenüber den neuen Varianten in doppelter Genauigkeit vergrößert. Besonders in doppelter Genauigkeit nehmen die Unterschiede zwischen zuverlässigen Updates und den anderen Implementierungen weiter ab, aber auch in gemischten Genauigkeiten benötigt M-T-PR-ZU beinahe gleich wenig Iterationen wie die anderen Varianten ohne zuverlässige Updates und ist sogar merklich besser als Variante M-S-PR-0.

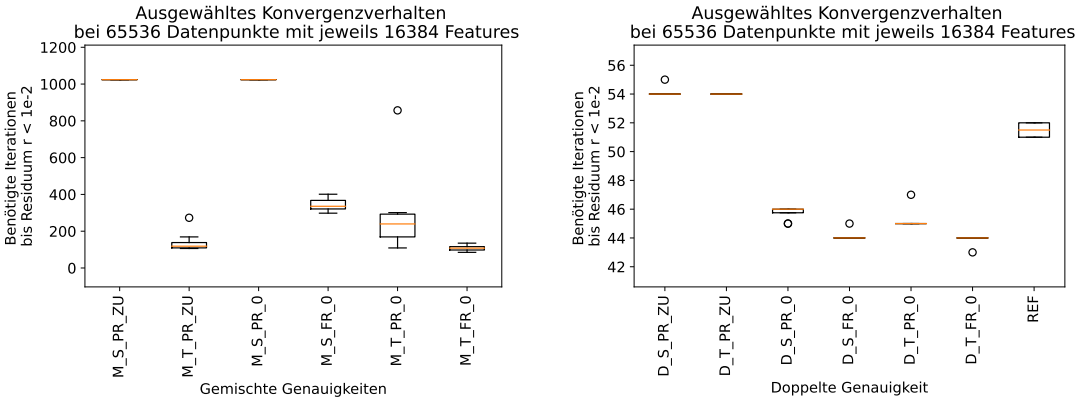
5.2.4 65536 Datenpunkte mit jeweils 16384 Features

Auch in diesem Beispiel wurde das Abbruchkriterium etwas gelockert, diesmal auf 10^{-2}, wobei auch dies nur wenige Iterationen weniger gegenüber dem vorherigen Abbruchkriterium bedeutet. In Abbildung 5.5a werden Ergebnisse von Varianten mit gemischten Genauigkeiten dargestellt: Die Implementierungen M-S-PR-ZU und M-S-PR-0 erreichen in diesem Szenario nicht mehr ein ausreichend genaues Residuum für das Abbruchkriterium. Ohne Tensor-Kernel konvergiert nur noch M-S-FR-0, allerdings mit 300-400 Iterationen deutlich langsamer als die Varianten mit Tensor-Kernel. Im Gegensatz zu den kleineren Problemgrößen benötigt die M-T-PR-ZU Variante mit einem Median von 112 weniger Iterationen als die meisten anderen Varianten , nur M-T-FR-0 hat einen minimal besseren Median von 111. Dafür ist der Durchschnitt von M-T-FR-0 mit 108 Iterationen merkbar niedrigeren gegenüber M-T-PR-ZU mit 141 Iterationen. Der Tensor-Kernel erzeugt hier bessere Ergebnisse, ebenso β nach F-R. Der Kernel mit gemischten Genauigkeiten stößt hier an seine Grenzen. Interessant, aber nicht in der Abbildung zu sehen, ist, dass das Residuum von M-S-PR-ZU in den Durchläufen einen um eine Zehnerpotenz niedrigeren Wert angenommen hat als M-S-PR-0. So wurden Werte für das Residuum von bis zu 0.2 angenommen.

Bei den Varianten in doppelter Genauigkeit, zu sehen in Abbildung 5.5b, sind die Unterschiede pro Durchlauf bei den einzelnen Varianten gering und unterscheiden sich maximal um 2 Iterationen. Allerdings werden bei diesen Varianten Unterschiede zwischen F-R und P-R sichtbar: D-T-FR-0 benötigt im Durchschnitt 43.875 Iterationen, Doppelte Genauigkeit, Standard-Kernel, β nach F-R und keinem Korrekturschema (D-S-FR-0) 44.125 Iterationen, D-T-PR-0 45.25 Iterationen und D-S-PR-0 45.75 Iterationen. Die Varianten mit zuverlässigen Updates verringern mit zunehmender Problemgröße den Abstand bezüglich der Anzahl ihrer Iterationen zu den anderen Varianten. Mit durchschnittlich 54 und 54.125 Iterationen sind D-T-PR-ZU und D-S-PR-ZU näher an der Referenzimplementierung als an den anderen neuen Implementierungen. Die modifizierte REF benötigt im Schnitt 51.5 Iterationen und damit gegenüber D-S-FR-0 ungefähr 17% mehr Iterationen.

5.2.5 131072 Datenpunkte mit jeweils 32768 Features

Durch die niedrige Anzahl an Durchläufen nimmt die Aussagekraft der Daten in Abbildung 5.6 ab, da Ausreißer einen größeren Einfluss haben können und nicht unbedingt als solche wahrgenommen werden könnten. Variante D-S-PR-ZU hat einen positiven Ausreißer mit 51 Iterationen, ansonsten liegt die Anzahl an benötigten Iterationen bei 57 und in einem Durchgang bei 56. D-T-PR-ZU benötigt in jedem Durchgang 56 Iterationen und damit im Durchschnitt das erste Mal mehr Iterationen als D-S-PR-ZU. Die neuen Implementierungen ohne Korrekturschema benötigen alle im Median 47 Iterationen mit jeweils einem Ausreißer auf 48 bei D-S-PR-0 und D-S-FR-0, nur D-T-FR-0 schneidet wieder besser ab mit dreimal 45 und einmal 44 Iterationen. Die REF benötigt 54, 53, 52 und 52 Iterationen. Im Durchschnitt sind das 8 Iterationen mehr als D-T-FR-0, was 18% entspricht. Dadurch festigt sich der Trend aus den vorherigen Szenarien, dass D-T-FR-0 am schnellsten konvergiert.



(a) Dargestellt werden Varianten mit gemischten Genauigkeiten. Die ersten 2 Boxplots entsprechen der Implementierung mit zuverlässigen Updates, die restlichen nutzen kein Korrekturschema. Variante M-S-PR-ZU und M-S-PR-0 konvergieren nicht mehr und sind nach 1024 Iterationen abgebrochen worden. Die Zahl der Ausreißer nimmt gegenüber kleineren Szenarien ab. Die Variante M-T-FR-0 benötigt teilweise nur 85 Iterationen, ihr Median liegt bei 111, die zweitbeste Variante ist M-T-PR-ZU mit einem Median von 112.

(b) Dargestellt werden Varianten in doppelter Genauigkeit. Die ersten 2 Boxplots entsprechen der Implementierung mit zuverlässigen Updates, die restlichen nutzen kein Korrekturschema. In den Durchläufen werden zwischen 43 und 55 Iterationen benötigt, wobei die Varianten mit zuverlässigen Updates näher an der REF sind als an den neuen Implementierungen ohne Korrekturschema. D-T-FR-0 benötigt mit durchschnittlich 43.875 am wenigsten Iterationen, D-S-PR-ZU hingegen mit 54.125 am meisten.

Abbildung 5.5: Dargestellt werden in Boxplots die Anzahl benötigter Iterationen bis das Residuum kleiner als 10^{-2} ist. In Abbildung 5.5a werden die Varianten in gemischter Genauigkeit gezeigt, in Abbildung 5.5b die in doppelter Genauigkeit. Es werden die Ergebnisse aus 8 Testdurchläufen dargestellt.

5.2.6 SAT-6 airborne datasets

Aus zeitlichen Gründen ist auch hier die Stichprobengröße gering und beträgt nur 2, während nicht alle Varianten getestet worden sind. Die Varianten mit gemischter Genauigkeit erreichen nicht das gewünschte Residuum, bei den Varianten mit doppelter Genauigkeit gibt es deutliche Unterschiede. Die Ergebnisse aus Tabelle 5.2 bekräftigen die Erkenntnisse aus den Tests mit synthetischen Datensätze. Die modifizierte Referenzimplementierung benötigt im arithmetischen Mittel 1379 Iterationen, die Variante D-T-PR-ZU benötigt im Durchschnitt 12% mehr Iterationen. Die Tensor-Kernel ohne Korrekturschema benötigen deutlich weniger Iterationen, auch hier ist die Variante mit F-R am besten mit durchschnittlich 1008 benötigten Iterationen, das sind 27% weniger als die Referenzimplementierung, und auch die Variante mit P-R mit 1067 Iterationen kommt noch immer mit 23% weniger Iterationen aus.

5.2.7 Fazit

Das Korrekturschema über die Neuberechnung des Residuums verschlechtert das Konvergenzverhalten schon bei sehr einfachen Problemen deutlich. Es ist in jedem Falle von einer Nutzung dessen abzusehen. Auch die zuverlässigen Updates können nicht überzeugen. In Verbindung mit der

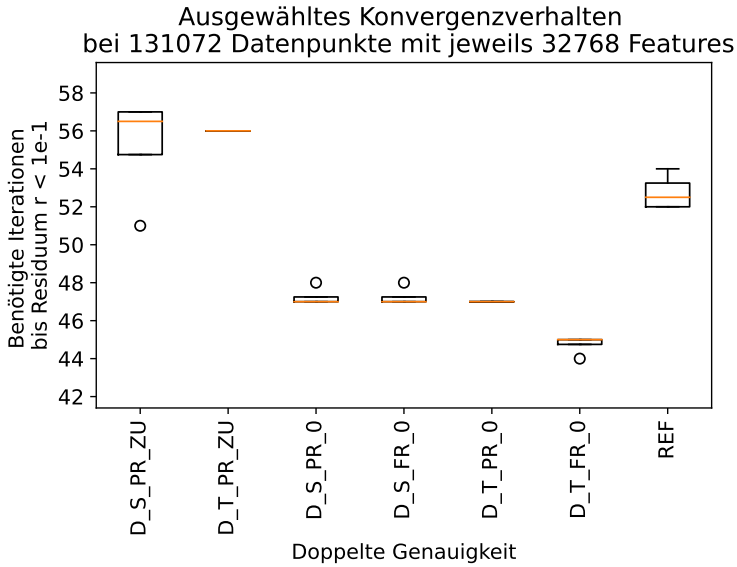


Abbildung 5.6: Dargestellt werden in Boxplots die Anzahl benötigter Iterationen bis das Residuum kleiner als 10^{-1} ist für Varianten in doppelter Genauigkeit. Es werden die Ergebnisse aus 4 Testdurchläufen dargestellt. D-T-FR-0 benötigt am wenigsten Iterationen bis das Abbruchkriterium erreicht wird. Die Varianten mit zuverlässigen Updates benötigen am meisten Iterationen.

Anzahl Iterationen	Durchgang 1	Durchgang 2	Durchschnitt
D-T-PR-ZU	1552	1548	1550
D-T-PR-0	1070	1063	1067
D-T-FR-0	1002	1014	1008
REF	1371	1387	1379

Tabelle 5.2: Benötigte Anzahl an Iterationen um den SAT-6 Trainingsdatensatz mit 324000 Datenpunkten und 3137 Features zu lösen mit Residuum $r < 0.0011$ als Abbruchkriterium. Es wurden jeweils nur 2 Programmdurchläufe gestartet mit den Varianten D-T-PR-ZU, D-T-PR-0, D-T-FR-0 und REF.

Vorschrift für β von F-R konvergierte diese Methodik ähnlich schlecht wie das Neuberechnungskorrekturschema. Mit P-R werden zwar deutlich bessere Ergebnisse erhalten, in Ausnahmefällen sogar solche, welche einzelne Variante ohne Korrekturschema übertreffen, aber es gibt auch Varianten ohne Korrekturschema, welche in jedem Testfall mit weniger Iterationen das Abbruchkriterium erreichen. Eine Ausnahme können Fälle darstellen, die sich im Grenzbereich der numerischen Stabilität bewegen. In dem einen Szenario in Abschnitt 5.2.4 generierte M-S-PR-ZU etwas bessere Ergebnisse als M-S-PR-ZU. Der Unterschied war allerdings gering und die Messung ein Einzelfall. Was sich aber über die Testszenarien herausgestellt hat, ist, dass mit wachsender numerischer Instabilität die Differenzen zwischen zuverlässigen Updates und Varianten ohne Korrekturschema abnehmen. Ebenso verhält es sich mit der Vorschrift für β von P-R, diese erreicht zwar deutlich bessere Ergebnisse mit zuverlässigen Updates, aber gegenüber den Varianten ohne Korrekturschema sind die Ergebnisse schlechter. Dies rechtfertigt aber nicht die Empfehlung im allgemeinen Fall.

Für sehr kleine Probleme wirkt sich der Standardkernel bei gemischten Genauigkeiten positiv auf die Anzahl benötigter Iterationen aus, bei größeren Problemen schneidet der Tensor-Kernel aber besser ab. Bis M-T-FR-0 nicht mehr konvergiert, werden bei kleinen Residuen zwischen 1.6 und 2.5 × so viele Iterationen wie bei D-T-FR-0 benötigt. Das Limit bei den getesteten synthetischen Datensätzen lag bei 65536 × 16384, was alleine für die Trainingsdaten 4,3 GB für FP32 und 8,6 GB für FP64 an Speicher bedeutet. Die nächstgrößere getestete Stufe würde das vierfache an Speicher benötigen und würde damit den Speicher der meisten Konsumer-GPUs überschreiten. Ebenso würde man bei einem Single-GPU-Setup das aktuell noch durch den Datentyp Integer gesetzte Limit überschreiten (Abschnitt 4.3.1). Ob die Implementierung aber für einen spezifischen Echtwelt-Datensatz funktioniert, lässt sich nicht vorhersagen.

Die Varianten mit doppelter Genauigkeit konvergieren mit den in Kapitel 4 durchgeführten Änderungen schneller als die Referenz-Implementierung und es wurde bisher kein Fall im möglichen durch die Integer bedingte Größe eingeschränkten Bereich gefunden ($4 \times (2^{31} - 1)$), der nicht konvergierte, auch wenn bei wachsender Problemgröße und Komplexität die Anzahl an benötigten Iterationen zwischen den Varianten stieg. Die meisten Iterationen werden bei der REF benötigt, die wenigsten Iterationen bei der Variante mit Tensor-Kernel und F-R.

Gegenüber der ersten Implementierung in gemischten Genauigkeiten, siehe Abschnitt 4.1 und Abbildung 4.1, bei der die Konvergenz bei 8192 Datenpunkten und 2048 Features am Limit war, können synthetische Probleme mit achtmal so vielen Datenpunkten und achtmal so vielen Features gelöst werden. Sollten sich die in der Arbeit durchgeführten Änderungen ähnlich auf die Stabilität der Implementierung mit doppelter Genauigkeit auswirken wie auf die Implementierung mit gemischten Genauigkeiten, also jeweils Faktor 8 in Punkten und Features, so sollten auch Probleme, nachdem das Überlauf-Problem gefixt worden ist, mit mindestens 1 Million Punkten und 250.000 Features oder mit 2.5 Millionen Datenpunkten und 24.000 Features gelöst werden können, unter der Annahme, dass die Referenz-Implementierung in den Testszenarien am Limit war. Potenziell also auch deutlich höher.

5.3 Laufzeiten

In diesem Abschnitt werden die einzelnen CUDA-Kernel, einzelne CG-Iterationen und komplette Durchläufe auf ihre Laufzeit und Skalierung untersucht. Im ersten Unterabschnitt werden die linearen Kernel theoretisch anhand von Profiling Ergebnissen untersucht. Diese wurden alle auf dem ipvsgpu1-System mit dem *Nsight Compute*-Analyseprogramm erstellt [NVI22e], da nur auf diesem System die Rechte dafür vorhanden waren. Darauf werden einzelne CG-Iterationen auf Skalierung untersucht, während zuletzt die kompletten Laufzeiten für verschiedene Systeme und Probleme gemessen werden.

Von den komplexen Kernel werden 4 lineare Versionen genauer betrachtet:

- `device_kernel_linear_td`: Der lineare Tensor-Kernel in doppelter Genauigkeit.
- `device_kernel_linear_tf`: Der lineare Tensor-Kernel mit gemischten Genauigkeiten.
- `device_kernel_linear<double>`: Der lineare Standard-Kernel in doppelter Genauigkeit.
- `device_kernel_linear<float>`: Der lineare Standard-Kernel in einfacher Genauigkeit.

	double_to_float	float_to_double
Dauer [µs]	7,01	6,56
Rechenauslastung [%]	1,22	1,37
Speicherdurchsatz [%]	2,82	3,48

Tabelle 5.3: Dauer, Rechen- und Speicherdurchsatz der Kernel `device_kernel_cast_double_to_float` und `device_kernel_cast_float_to_double` nach *Nsight Compute*. `device_kernel_cast_double_to_float` transferiert einen Array auf der GPU von FP64 zu FP32 und `device_kernel_cast_float_to_double` genau umgekehrt. Die Auslastung beider Kernel ist sehr gering.

5.3.1 Kernel-Profiling

Die Profiling Ergebnisse basieren alle, wenn nicht anders geschrieben, auf einer Problemgröße von 32769 Datenpunkten und 4096 Features, 4 NVIDIA A100, Profiling Ergebnisse von Device 0 und dem linearen Kernel. Die daraus wichtigsten Größen werden in Tabellenformat wiedergegeben.

Konvertierungskernel

In jeder CG-Iteration wird bei der Nutzung von gemischten Genauigkeiten ein Vektor der Größe der Anzahl an Datenpunkten von FP64 zu FP32 transformiert und nach der Berechnung wieder zurück. Auch beim Setup werden Daten transferiert. Dafür gibt es zwei eigene Kernel `device_kernel_cast_double_to_float` und `device_kernel_cast_float_to_double`.

Diese Kernel sind sehr einfach und lasten die GPUs nicht aus. Die in Tabelle 5.3 gezeigten Werte variieren von Durchgang zu Durchgang in der Größenordnung von einer Mikrosekunde. Da die gesamte gemessene Zeit allerdings im Bereich von ungefähr 7 Mikrosekunden liegt, sind die Kernel im Bezug auf Optimierungen oder der gesamten Laufzeit zu vernachlässigen, auch wenn die GPU Auslastung sehr niedrig ist.

GEMV-Kernel in doppelter Genauigkeit

Der Tensor-Kernel mit doppelter Genauigkeit diente als Entwicklungs- und Optimierungsgrundlage. Daher wurde in diesen die meiste Energie bei der Optimierung gesteckt.

Im Vergleich mit dem Standardkernel in Tabelle 5.4 ist der Tensor-Kernel deutlich schneller. Laut Herstellerangaben hat die Nvidia A100 [NVI20] unter Ausnutzung der Matrixeinheiten theoretisch die doppelte Leistung in doppelter Genauigkeit. Die Vorteile des neu-implementierten Kernels gehen also nicht nur auf die Ausnutzung der Matrixeinheiten zurück.

Die Rechenauslastung ist um $\approx 2/3$ höher, während die Spitzenlast beim Speicherdurchsatz einigermaßen identisch bleibt. Der angegebene Speicherdurchsatz entspricht dem Maximum der einzelnen in der Tabelle aufgezeigten Durchsätze, frühere Versionen des Tensor-Kernels waren sehr stark durch die hohe Auslastung beziehungsweise Überlastung des geteilten Speichers begrenzt. Entgegen ersten Vermutungen spielt bei beiden Kernels die Bandbreite zwischen SM und globalen Speicher noch keine Rolle, wie anhand des DRAM Durchsatzes zu sehen ist.

	device_kernel_linear_td	device_kernel_linear<double>
Dauer [ms]	105,00	351,38
Rechenauslastung [%]	70,20	41,78
Speicherdurchsatz [%]	67,01	69,34
L1/TEX Cache Durchsatz [%]	67,22	69,41
L2 Cache Durchsatz [%]	24,49	9,79
DRAM Durchsatz [%]	27,89	8,35

Tabelle 5.4: Überblick über wichtige Größen der Kernel `device_kernel_linear_td` und `device_kernel_linear<double>` nach *Nsight Compute*. Dargestellt werden die durchschnittliche Dauer, die Rechenauslastung und der Speicherdurchsatz, wobei der Speicherdurchsatz nochmals aufgeteilt wird in die Kategorie L1/TEX Cache Durchsatz, L2 Cache Durchsatz und DRAM Durchsatz. Der `device_kernel_linear_td` hat eine höher Rechenauslastung und benötigt mehr als 3 Mal weniger Zeit für einen durchschnittlichen Aufruf.

Pipeline	device_kernel_linear_td	device_kernel_linear<double>
Top 1 [%]	Tensor (DP) 69,87	FP64 41,83
Top 2 [%]	LSU 35,22	LSU 41,79
Top 3 [%]	ALU 8,52	ALU 8,13
Top 4 [%]	FMA 5,51	ADU 4,64

Tabelle 5.5: Die jeweils am meisten beanspruchten Hardware-Pipelines von `device_kernel_linear_td` und `device_kernel_linear<double>` nach *Nsight Compute*. Tensor (DP) steht dabei für die Beanspruchung der Matrixeinheiten im FP64 Format, Load Store Unit (Deutsch: Laden-Speichern-Einheit) (LSU), die für das Laden in und aus dem L1/Tex und geteilten Speicher verantwortlich ist, Arithemitc Logic Unit (Deutsch: Arithemitische Logikeinheit) (ALU) für Integer Berechnungen, FMA für die bereits erwähnten FMA Berechnungen in diesem Falle von Integer-Zahlen, FP64 für die Berechnung mit Gleitkommazahlen mit doppelter Genauigkeit und Address Divergence Unit (Deutsch: Adressen-Divergenzeinheit) (ADU), welche die Warp-Divergenzen verwaltet.

Bei der in Tabelle 5.5 zu sehenden Auslastung der einzelnen Pipelines gibt *Nsight Compute* an, dass für den Tensor-Kernel nach weiterer Optimierung die Auslastung der Tensor-Cores zum Flaschenhals werden kann. Bei beiden Kernels werden die LSU und Gleitkommazahlen Einheiten (Tensor und Skalar) am meisten beansprucht. Die ADU wird hingegen bei dem Standardkernel deutlich mehr genutzt mit 4.64%, was es möglichst zu vermeiden gilt, während diese bei Tensor-Kernel nur eine Auslastung von 0.06% hat.

Die größte Herausforderung stellte die Optimierung des geteilten Speichers dar. Die aktuellen Zahlen werden in Tabelle 5.6 dargestellt. Die Auslastung dessen war zu Anfang der Optimierung bei über 90% und es gab über $40 \cdot 10^9$ viele Bank-Konflikte.

Geteilter Speicher	device_kernel_linear_td	device_kernel_linear<double>
Instruktionen	2 933 738 080	6 486 552 576
Wavefronts	8 047 182 532	26 095 016 300
Maximalauslastung	64,81 %	62,78 %
Speicherbank-Konflikte	755 209 059	6 514 200 792

Tabelle 5.6: Die Nutzung des geteilten Speichers für die Kernel `device_kernel_linear_td` und `device_kernel_linear<double>` nach *Nsight Compute*. In der ersten Zeile wird die Anzahl an Instruktionen verglichen. In Zeile 2 wird verglichen, wie viele Wavefronts, also einzelne serielle Zugriffe, auf den geteilten Speicher notwendig sind, bevor in Zeile 3 die maximale Auslastung und zuletzt die Speicherbank-Konflikte verglichen werden.

Gründe für das Anhalten von Warps [cycles instruction^{-1}]	device_kernel_linear_td	device_kernel_linear<double>
Top 1 [cycles]	Stall Math Pipe Throttle 4,33	Stall Long Scoreboard 41,50
Top 2 [cycles]	Stall Wait 2,86	Stall Barrier 2,98
Top 3 [cycles]	Selected 1,0	Stall Wait 2,46

Tabelle 5.7: Die wichtigsten Gründe für das Anhalten eines Warps in den Kernel von `device_kernel_linear_td` und `device_kernel_linear<double>`. Stall Math Pipe Throttle bedeutet, dass eine Pipeline für eine bestimmte mathematische Funktion (Tensor (FP64)) gerade ausgelastet ist und auf die Freigabe gewartet wird, Stall Wait, dass auf eine Abhängigkeit gewartet wird, und Selected, dass der Warp für eine Operation des Micro Scheduler angehalten worden ist. Bei Stall Long Scoreboard wird auf auf Daten aus dem L1/TEX, L2, globalen oder lokalen Speicher gewartet und bei Stall Barrier wird an einer Barriere auf die anderen Warps des Blockes gewartet.

Die Anzahl der Konflikte konnte wie in Tabelle 5.6 zu sehen deutlich reduziert werden und Konflikte kommen nur noch beim Laden aus dem globalen Speicher in den geteilten Speicher vor, beim Laden aus dem geteilten Speicher in die Register kommen keine Konflikte mehr vor, während der Standardkernel eine vergleichbare Anzahl an Konflikten beim Laden aus dem globalen Speicher hat, aber zusätzlich 5 775 529 000 Konflikte beim Laden aus dem geteilten Speicher in die Register. Die am Ende ausschlaggebende Größe ist die der Wavefronts: Diese gibt an, wie oft effektiv auf den geteilten Speicher zugegriffen werden musste, ein Speicherkonflikt wird seriell abgearbeitet und erhöht somit die Anzahl an Wavefronts. Interessant ist die Korrelation zwischen dem Verhältnis der Wavefronts und der Dauer für einen Kernel-Durchlauf: Mit abnehmenden Wavefronts ist durch die Limitierung der Speicherzugriffe proportional die Dauer pro Kernel-Aufruf gesunken. Der Faktor zwischen den Laufzeiten beider Kernel beträgt 3.3, der Faktor zwischen den Wavefronten beträgt 3.2.

Auf der anderen Seite kann auch untersucht werden, warum ein Warp zwischen Instruktionen angehalten wird. Dafür gibt es in *Nsight Compute* einen eigenen Abschnitt dessen wichtigste Gründe in Tabelle 5.7 veranschaulicht werden. Von *Nsight Compute* werden auch mögliche Ursachen und Lösungsvorschläge, wie die Ursachen behoben werden können, angegeben. Für die dominierenden Gründe Stall Math Pipe Throttle und Stall Wait wird vorgeschlagen die Anzahl an Warps zu

Belegung	device_kernel_linear_td	device_kernel_linear<double>
Aktive Warps pro SM	12	16
Block Limit Register	1	2
Block Limit geteilter Speicher	2	65

Tabelle 5.8: Eine Übersicht über die Belegung der SM-Ressourcen mit den Kernel von `device_kernel_linear_td` und `device_kernel_linear<double>`. Die aktiven Warps pro SM geben an, wie viele Warps gleichzeitig aktiv sind auf einem SM und ergeben sich aus der Anzahl aktiver Blöcke multipliziert mit der Anzahl an aktiven Warps pro Block. Dabei können theoretisch viele Blöcke parallel aktiv sein. Dies kann aber durch Ressourcen wie die Anzahl an Register oder die Größe des freien Speichers limitiert werden.

erhöhen. Bei Stall Long Scoreboard, was den Standardkernel stark limitiert, wird vorgeschlagen die Speicherzugriffe zu optimieren, was sich mit den Ergebnissen aus dem geteilten Speicher deckt. Eine hohe Anzahl an Zyklen für Stall Barrier Wait kann laut *Nsight Compute* durch divergierende Pfade vor einer Synchronisation verursacht werden, was beim Standardkernel allerdings nicht der Fall ist.

Als letzter Punkt wird die Belegung an Warps, Threads und Blöcken untersucht. Theoretisch können pro SM 2048 Threads gleichzeitig aktiv sein in maximal 32 Blöcken oder insgesamt 64 Warps.

In Tabelle 5.8 ist zu sehen, dass der Standardkernel gleichzeitig mehr Warps auf einem SM gleichzeitig aktiv hat. Da kein asynchrones Laden verwendet wird, braucht dieser auch mindestens 2 Blöcke um performant zu sein. Der Tensor-Kernel hat hingegen nur 12 aktive Warps in einem Block. Gleichzeitig wird empfohlen gegen die häufigsten Gründe den Warp anzuhalten, mehr Warps aktiv zu halten. Der limitierende Faktor sind hier die Register, wobei jeder Thread 138 Register benötigt. Um einen zweiten Block parallel laufen zu lassen, dürfte ein Block allerdings nur 80 Register benötigen. Eine weitere Möglichkeit wäre die Anzahl der Elemente, die ein Block berechnet, zu erhöhen, was zusätzlich die Anzahl an Speichertransaktionen verringern würde. Durch die Aufteilung in 4 SP pro SM wäre beim aktuellen Blockdesign die nächstbeste Größe 128×128. Dies würde allerdings nicht nur 4 neue Warps und somit 128 neue Threads bedeuten, sondern auch pro Thread 4 weitere Akkumulatoren und folglich 16 weitere Register. Damit reichen die Register aber nicht mehr aus um einen einzelnen Block laufen zu lassen.

Zusammengefasst ist der Tensor-Kernel durch die Ausnutzung der Matrixeinheiten und eine effiziente Ausnutzung des geteilten Speichers in dem betrachteten Beispiel um Faktor ≈ 3.3 schneller als der Standardkernel.

GEMV-Kernel in einfacher Genauigkeit

In diesem Abschnitt werden nur die wichtigsten Daten zusammengetragen oder Unterschiede zum FP64-Kernel erläutert.

Tabelle 5.9 zeigt, dass beide Kernel mit einfacher Genauigkeit ungefähr doppelt so schnell wie die jeweiligen Varianten in doppelter Genauigkeit sind. Allerdings ist besonders beim Tensor-Kernel die Rechenauslastung deutlich niedriger als bei der Variante in doppelter Genauigkeit. Für den

	device_kernel_linear_tf	device_kernel_linear<float>
Dauer [ms]	50,61	179,30
Rechenauslastung [%]	42,10	81,88
Speicherdurchsatz [%]	42,68	84,72
L1/TEX Cache Durchsatz [%]	42,85	84,82
L2 Cache Durchsatz [%]	18,89	10,56
DRAM Durchsatz [%]	20,62	7,84

Tabelle 5.9: Dauer, Rechen- und Speicherdurchsatz der Kernel `device_kernel_linear_tf` und `device_kernel_linear<float>` nach *Nsight Compute*. Dargestellt werden die durchschnittliche Dauer, die Rechenauslastung und der Speicherdurchsatz, wobei der Speicherdurchsatz nochmals aufgeteilt wird in die Kategorie L1/TEX Cache Durchsatz, L2 Cache Durchsatz und DRAM Durchsatz. Der `device_kernel_linear_tf` hat ungefähr die Hälfte an Rechenauslastung und benötigt mehr als 3.5 Mal weniger Zeit für einen durchschnittlichen Aufruf.

Pipeline	device_kernel_linear_tf	device_kernel_linear<float>
Top 1 [%]	LSU 33,36	LSU 81,97
Top 2 [%]	FMA 26,94	FMA 44,48
Top 3 [%]	ALU 17,17	ALU 18,23
Top 4 [%]	Tensor (FP) 9,04	ADU 9,10

Tabelle 5.10: Die jeweils am meisten beanspruchten Hardware-Pipelines von `device_kernel_linear_tf` und `device_kernel_linear<float>` nach *Nsight Compute*. Tensor (FP) steht dabei für die Beanspruchung der Matrixeinheiten im FP32 Format, LSU, die für das Laden in und aus dem L1/Tex und geteilten Speicher verantwortlich ist, ALU für Integer Berechnungen, FMA für die bereits erwähnten FMA Berechnungen mit Gleitkommazahlen und Integer-Zahlen, und ADU, welche die Warp-Divergenzen verwaltet.

Tensor-Kernel lässt sich das dadurch erklären, dass sich bei der genutzten Genauigkeit mit einer Matrixoperation ein achtfaches an skalaren Operationen ersetzen lässt. Wäre der Tensor-Kernel daher limitiert durch die Rechenoperationen, könnte er bis zu achtmal schneller sein als der Standard-Kernel in gemischten Genauigkeiten. Auch sind viele Speicheroperationen beim Tensor-Kernel so ausgelegt, dass sie ein Vielfaches an 128 bit auf einmal transferieren unabhängig vom Datentyp, was die Belastung des geteilten Speichers schont. Durch die Größe eines Blockes von 128×128 werden außerdem weniger Transaktionen vom geteilten Speicher in die Register benötigt.

Tabelle 5.10 zeigt die genannten Effekte auf die Auslastung der Matrixeinheiten Pipeline. Beim Standard-Kernel ergibt sich ein ähnliches Bild zum Kernel mit doppelter Genauigkeit, alle Werte haben sich ungefähr verdoppelt, bis auf die FMA Instruktionen, die einerseits die FP64 Instruktionen ablösen und doppelt vorhanden sind gegenüber den FP64 Einheiten.

Die Gründe, warum der Tensor-Kernel trotz der schlechteren Auslastung der Rechenpipelines schneller ist, liegen am geteilten Speicher. Durch den kleineren Datentyp und die größere Blockgröße benötigt dieser Kernel nur 2 451 623 444 viele Wavefronts, was mehr als 3× weniger ist als beim Tensor-Kernel in doppelter Genauigkeit. Der Standardkernel hingegen reduziert die Wavefronts von 26 095 016 300 auf 16 720 027 100, also nur um Faktor 1.55.

Die Gründe für das Anhalten der Kernels sind ebenfalls ähnlich: Beim Tensor-Kernel fällt die hohe Auslastung der Matrixeinheiten weg, dafür dominiert Stall Wait stärker. Die restliche Reihenfolge ist identisch. Beim Standardkernel wird proportional weniger lange auf die Daten gewartet, dafür länger an den Barrieren.

Der Grund für den verkürzten Stillstand während des Wartens beim Standardkernel ist, dass durch den geänderten Datentyp von FP32 weniger Register benötigt werden und daher auf das Register genau 4 Blöcke gleichzeitig auf einer SM aktiv sein können. Dadurch können auch ineffiziente Datenzugriffe besser versteckt werden. Beim Tensor-Kernel stellt sich das Gleiche Problem in einfacher Genauigkeit wie bei doppelter Genauigkeit. Derzeit werden 160 von 256 möglichen Register pro Thread benötigt, für 2 Blöcke parallel dürften dadurch maximal 128 Threads gebraucht werden. Auf der anderen Seite kann durch die Dimension der Matrixoperation ($16 \times 16 \times 8$) bei 4 SP die Blockgröße bei einem quadratischen Block nur in 64er Schritten angehoben werden. Dies würde einerseits 50% mehr Threads bedeuten (Vereinfacht gerechnet), also $160 \cdot 1.5 = 240$ Register benötigen und gleichzeitig 50% mehr Akkumulatoren, was mindestens 32 weitere Register bedeuten würde und somit mehr als die 256 möglichen. Paradoxerweise ist der Kernel primär von der Anzahl an Registern limitiert, während zum gleichen Zeitpunkt nicht einmal Zweidrittel aller Register genutzt werden, da durch den quadratischen Aufbau die nächste passende Stufe bei 192×192 liegen würde, und wie gerade überschlagen, dafür nicht die Register ausreichen. Eine Möglichkeit, diese Limitierung zu umgehen, wäre eine nicht quadratische Blockgröße zu verwenden.

5.3.2 Kernel Skalierung

In diesem Abschnitt werden die 4 linearen Kernel, die von der aktuellen Implementierung genutzt werden können, auf ihre Laufzeit untersucht. Um eine bessere Vergleichbarkeit zu anderen PLSSVM-Backends und SVM-Programmen zu gewähren, wird sich bezüglich den Testszenarien, Größe und Layout an den Arbeiten von Breyer et al. [BCP22] und Van Craen et al. [VBP22b] orientiert. Der in diesen Arbeiten genutzte CUDA-Kernel entspricht dem Standard-Kernel in doppelter Genauigkeit und kann als Vergleichspunkt dienen.

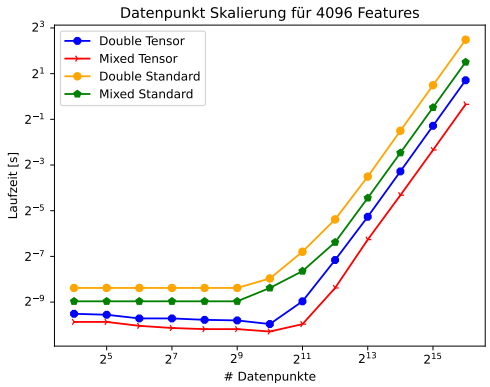
Wie bereits in Abschnitt 4.3 erwähnt, werden für die Ausführung der Tensor-Kernel in doppelter Genauigkeit FP64-Matrixeinheiten benötigt, welche aktuell (Stand: 4. Oktober 2022) bei NVIDIA nur die A100 [NVI20] enthält. Der Kernel für gemischte Genauigkeiten benötigt mindestens eine GPU mit Ampere-Architektur oder Neuer, da erst diese das TF32-Format unterstützen. Für die Laufzeitmessungen wurden auf verschiedenen Systemen, die genauer in Abschnitt 5.1 beschrieben werden, 100 Iterationen ohne ein Korrekturschema oder einem Abbruchkriterium mit jeweils nur einer einzelnen GPU ausgeführt und gemittelt. Somit ist die Anzahl der Rechenoperationen in jeder Iteration für die Größe des Datensatzes komplett identisch. Die Ergebnisse sind in Abbildung 5.7 zu sehen: Auf der linken Seite für 2^{12} feste Features und 2^4 bis 2^{16} Datenpunkten. Auf der rechten Seite für 2^{15} feste Datenpunkte und 2^4 bis 2^{14} Features.

Ab einer gewissen Größe wird für die doppelte Anzahl an Datenpunkten ungefähr das Vierfache an Laufzeit benötigt. Das deckt sich mit der Theorie, bei doppelter Anzahl an Datenpunkten ist die aufgestellte Matrix viermal so groß und es werden viermal so viele Berechnungen zur Aufstellung der Matrix und folgenden Multiplikationen durchgeführt. Bei den Features verhält es sich so, dass bei doppelter Anzahl an Features für das Aufstellen der Matrix das doppelte an Berechnungen durchgeführt werden müssen, sich also die Laufzeit annähernd verdoppelt, da die GEMV-Berechnung nicht mit skaliert. Auch das wird durch die Messungen bestätigt.

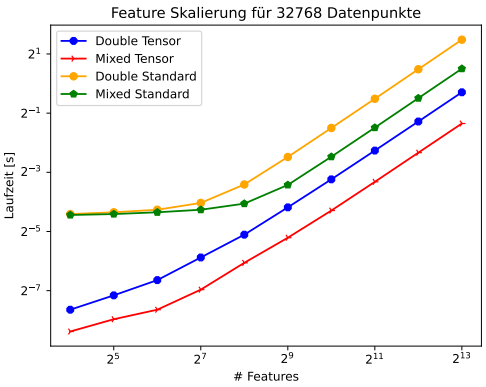
Bei der A100 in Abbildung 5.7a und Abbildung 5.7b sind die Verhältnisse zwischen den Kernel deckungsgleich mit den Ergebnissen aus der theoretischen Betrachtung aus Abschnitt 5.3.1: Der Standard-Kernel in FP32 ist ziemlich genau doppelt so schnell wie der Kernel mit FP64, ist selbst allerdings um 73 % langsamer als der Tensor-Kernel mit doppelter Genauigkeit. Der Tensor-Kernel in gemischter Genauigkeit ist 2.07 mal schneller als der in doppelter Genauigkeit. Insgesamt ist der Tensor-Kernel in doppelter Genauigkeit ungefähr 3.4 mal schneller als der Standard-Kernel. Interessant wird es für kleine Größen: Bei 2^{15} Datenpunkten und 2^4 Features ist der Aufwand für den Matrixaufbau sehr gering. Die Tensor-Kernel benötigen für diese Berechnung durchschnittlich 3 respektive 5 ms und skalieren noch relativ lange für kleinere Probleme, während die Standardimplementierungen knappe 50 ms benötigen. Diese Zeit wird folglich beinahe ausschließlich für die GEMV-Berechnung verwendet. Für größere Probleme kann der Compiler diesen Overhead allerdings fast komplett verbergen. Beim Schritt von 2^9 zu 2^{10} Features ergibt sich zum Beispiel in [ms]: $t[2^{10}] - t[2^9] = 700 - 353 = 347$, also eine Differenz von 6 ms, die für die GEMV-Berechnung übrig bleibt von ursprünglich fast 50 ms. Die nicht lineare Skalierung bei den Datenpunkten zu Beginn lässt sich einfach erklären, da bei einer geringen Anzahl an Datenpunkten nicht genügend Blöcke entstehen, um alle SMs auszulasten.

Die Ampere Generation mit CUDA Compute Capability 8.6 besitzt nur $1/64$ so viel FP64 Leistung wie FP32 [NVI21]. Da besonders die Standard-Kernel limitiert sind durch Speicherzugriffe, sind die Laufzeiten dessen in doppelter Genauigkeit für die RTX 3080 und RTX 3090 (Abbildung 5.7c bis Abbildung 5.7f) *nur* um Faktor 8 langsamer als der Standard-Kernel in einfacher Genauigkeit. Der Tensor-Kernel mit gemischten Genauigkeiten ist, obwohl bei der RTX 3080 die normale FP32 Performance gleich hoch ist wie die TF32 Performance mit Matrixeinheiten [NVI21], viermal so schnell wie der Standard-Kernel in FP32, also 32× schneller als bei der FP64 Referenz-Implementierung. Bei der RTX 3090 sieht das Bild beinahe identisch aus. Die Unterschiede zwischen den Standard-Kernel ist weiterhin bei Faktor 8, aber zwischen dem Tensor-Kernel und dem Standard-Kernel in gemischter Genauigkeit liegt der Unterschied auf diesem System nur bei 3.8. Dies könnte mit der etwas höheren Speicherbandbreite und -Frequenz im Verhältnis zu der Anzahl an CUDA-Cores gegenüber der RTX 3080 zusammenhängen: Die Recheneinheiten der RTX 3090 sind etwas niedriger getaktet, der Speicher allerdings höher. Ohne Profiling kann dies aber nicht abschließend bewertet werden. Das Verhalten bei kleinen Größen ist identisch zu den Ergebnissen der A100, wobei auch hier der Tensor-Kernel in den Bereich von µs liegen kann und daher größere Messungenauigkeiten auftreten können.

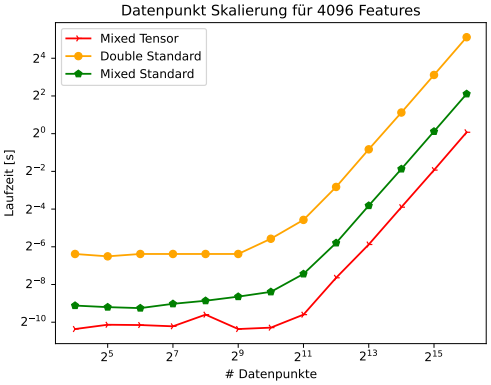
Beim Vergleich zwischen RTX 3080 und 3090 mit der A100 ist in doppelter Genauigkeit die A100 5-7 mal schneller mit dem Standard-Kernel und 18-22 mal schneller als die Tensor-Implementierung. Bei gemischten Genauigkeiten, besonders mit Tensor-Kernel, sind die Unterschiede allerdings sehr gering. So benötigt das Testsystem mit RTX 3080 und Tensor-Kernel nur 33% länger als das System mit einer A100.



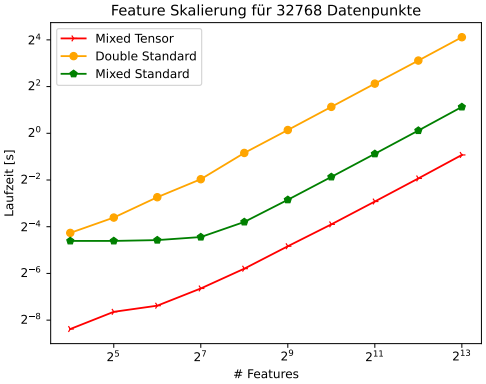
(a) Datenpunkt Skalierung A100 auf ipvsgpu1.



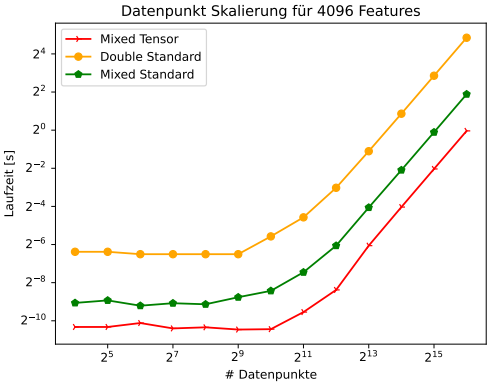
(b) Feature Skalierung A100 auf ipvsgpu1.



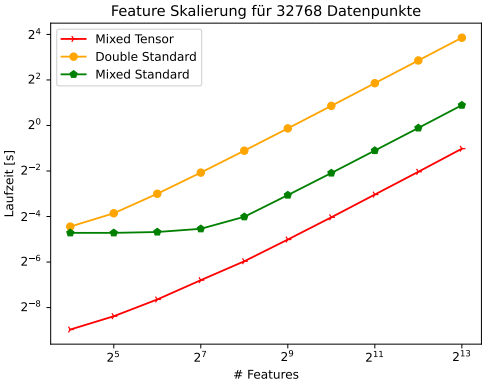
(c) Datenpunkt Skalierung RTX 3080 auf pcsgs2.



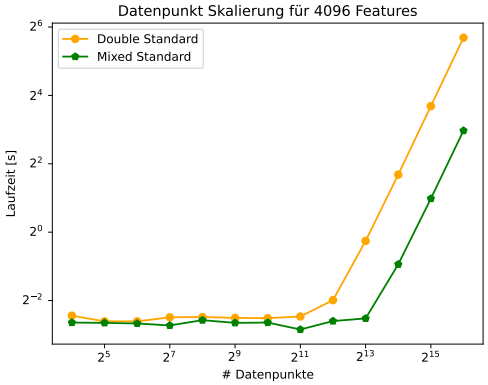
(d) Feature Skalierung RTX 3080 auf pcsgs2.



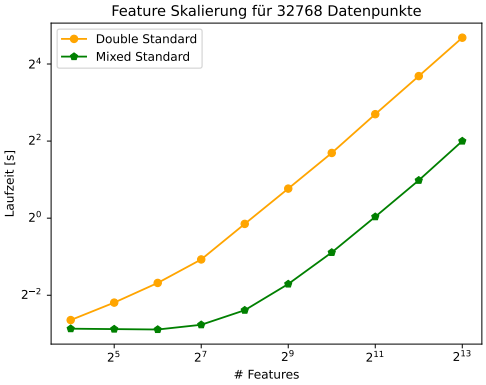
(e) Datenpunkt Skalierung RTX 3090 auf pcsgs11.



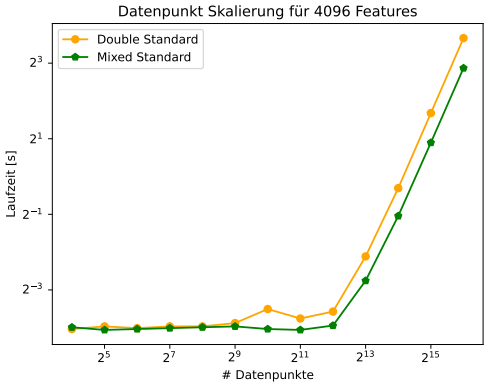
(f) Feature Skalierung RTX 3090 auf pcsgs11.



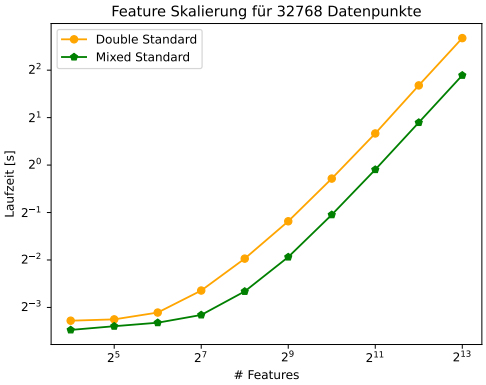
(g) Datenpunkt Skalierung GTX 1080 ti auf Argon-gtx.



(h) Feature Skalierung GTX 1080 ti auf argon-gtx.



(i) Datenpunkt Skalierung P100 auf argon-tesla1.



(j) Feature Skalierung P100 auf argon-tesla1.

Abbildung 5.7: Dargestellt werden Feature und Datenpunkt Skalierungen auf verschiedenen Server- und Desktop-GPUs. Auf der linken Seite werden die Features auf $2^{12} = 4096$ gesetzt und die Anzahl an Datenpunkten variiert von 16-65536. Auf der rechten Seite werden Datenpunkte auf $2^{15} = 32768$ gesetzt und die Features variieren von 16-8192.

Für größere Probleme verhält sich die GTX 1080 ti in Abbildung 5.7g und Abbildung 5.7h wie zu erwarten: Eine annähernde Vervierfachung der Laufzeiten bei doppelter Anzahl an Punkten und eine annähernde Verdopplung der Laufzeit bei den Features. Bei sehr kleinen Problemen hingegen sind die Laufzeiten unverhältnismäßig hoch gegenüber zum Beispiel einer RTX 3080. Die GTX 1080 ti basiert auf der Pascal Architektur mit CUDA Compute Capability 6.1, die ein Verhältnis von FP64 zu FP32 von 1 zu 32 besitzt. Da die GTX 1080 ti keine Matrixeinheiten besitzt, werden hier nur die Standard-Kernel verwendet. Der Standardkernel in gemischten Genauigkeiten ist dabei 6.6 Mal schneller als in doppelter Genauigkeit. Im Vergleich zu der neueren RTX 3080 ist die 1080 ti in doppelter Genauigkeit 50% langsamer als die RTX 3080, in einfacher Genauigkeit 80% beziehungsweise 757% (Tensor-Kernel) langsamer.

relative Geschwindigkeit	FP64 Standard-Kernel	FP32 Standard-Kernel	FP64 Tensor-Kernel	FP32 Tensor-Kernel
A100	1	1.97	3.42	7.11
RTX 3080	0.16	1.30		5.32
RTX 3090	0.19	1.52		5.76
GTX 1080 ti	0.11	0.71		
P100	0.44	0.77		
absolute Geschwindigkeit [s]				
A100	5.60	2.84	1.64	0.79
RTX 3080	34.67	4.32		1.05
RTX 3090	28.80	3.70		0.97
GTX 1080 ti	51.57	7.84		
P100	12.67	7.31		

Tabelle 5.11: Relative Geschwindigkeit (oben) und absolute Geschwindigkeit (unten) der Kernel für 65536 Datenpunkte und 4096 Features für verschiedene GPUs. Die Laufzeit des Standard-Kernels in doppelter Genauigkeit wird als Ausgangspunkt für die relative Geschwindigkeit festgelegt. Bei einem Wert größer 1 ist der Kernel schneller, bei kleiner 1 langsamer. Bei den leeren Feldern sind keine für das Zahlenformat unterstützten Matrixeinheiten vorhanden.

Die P100 ist wie die A100 eine GPU für Datenzentren beziehungsweise wissenschaftliche Berechnungen und hat dadurch ein Verhältnis von 2:1 bezüglich FP64 zu FP32. Auch die P100 hat keine Matrixeinheiten, weshalb nur der Standardkernel in beiden Genauigkeiten in Abbildung 5.7i und Abbildung 5.7j untersucht wird. Wie auch die GTX 1080 ti benötigt die P100 mindestens 4096 Datenpunkten um bei 4096 Features ausgelastet zu sein, um dann wie erwartet bei Verdopplung der Datenpunkte das Vierfache an Laufzeit zu benötigen. In doppelter Genauigkeit ist der Kernel nur 73% langsamer als mit gemischten Genauigkeiten, was der kleinste gemessene Abstand dieser Kernel in allen Tests war. Damit ist die P100 4.1 Mal schneller in doppelter Genauigkeit als die GTX 1080 ti und 2.7 Mal schneller als die RTX 3080, dafür 2.3 Mal langsamer als die A100 mit Standardkernel oder 7.7 Mal langsamer mit Tensor-Kernel. Mit gemischten Genauigkeiten ist die P100 7% schneller als die GTX 1080 ti, dafür 69% langsamer als die RTX 3080 mit Standardkernel und 6.9 Mal langsamer mit Tensor-Kernel. Gegenüber der A100 dann 2.6 beziehungsweise 9.3 Mal langsamer für Standard- und Tensor-Kernel.

Werden die Laufzeiten pro Iteration isoliert betrachtet wie in Tabelle 5.11, erweitert sich einmal das nutzbare Spektrum an linearen CUDA-Kernel von 1 auf 2-4, anderseits ist bereits eine RTX 3080 oder RTX 3090 5.32 bis 5.76 Mal schneller mit dem Tensor-Kernel in gemischten Genauigkeiten als bisher eine A100. Auch die ältere 1080 ti und P100 unterscheiden sich nur noch um 40 beziehungsweise 30% vom FP64 Standard-Kernel auf der A100. Dies ermöglicht besonders mit Desktop-GPUs ein deutlich effizienteres Arbeiten mit PLSSVM.

5.4 Kombinierte Tests

In diesem Abschnitt werden die Veränderungen am Kernel und am CG-Algorithmus kombiniert betrachtet. Die Vorteile ergeben sich wie in den vergangenen Abschnitten behandelt aus der Möglichkeit durch eine niedrigere Genauigkeit die Hardware deutlich effizienter zu nutzen, einen schnelleren Kernel zu nutzen, stabiler zu sein oder mit weniger Iterationen zu konvergieren. Da die Vorteile multiplikativ sind, also beispielsweise folgt aus der doppelten Anzahl an Iterationen bei einem zweiunddreißigstel der Laufzeit ein insgesamt 16 mal schnellerer Lernvorgang, werden hier nur exemplarisch 2 Beispiele in Kürze behandelt: Ein synthetisches Beispiel mit 65536 Datenpunkten und 16384 Features aus Abschnitt 5.2.4 und für den komplexen Fall das bereits erwähnte SAT-6 Problem aus Abschnitt 5.1.3.

5.4.1 Synthetischer Test mit 65536 Datenpunkten und 16384 Features

Die Problemgröße von 65536 Datenpunkten mit 16384 Features nähert sich dem Limit für gemischte Genauigkeiten an, wie in Abschnitt 5.2.4 gezeigt worden ist. In diesem Fall kann es sein, dass für instabilere Kernel das Residuum niedriger gewählt werden muss.

In Tabelle 5.12 wird die Trainingszeit und die dazu gemessene Genauigkeit gezeigt. Die Variante mit Tensor-Kernel in doppelter Genauigkeit erzeugt das beste Modell und ist ungefähr 4.3 Mal so schnell wie die Referenz-Implementierung. Je instabiler die Implementierung und je größer der Trainingsdatensatz beziehungsweise je näher am Limit der numerischen Stabilität, desto niedriger muss das Residuum sein, um die gleiche Modell-Genauigkeit zu erreichen. Ab einem gewissen kleinen Residuum nimmt bei genauerem Ergebnis die Vorhersagewahrscheinlichkeit des Modells, also wie viele Datenpunkte richtig klassifiziert werden, nicht weiter zu. Bei den synthetischen Datensätzen hat sich ein Epsilon zwischen 0.2 und 0.01 als sinnvoll erwiesen bei der hier vorgestellten CG-Implementierung. Um auf das gleiche Residuum zu kommen, ist die Mixed-Tensor Variante am schnellsten, allerdings ist das daraus gewonnene Modell nicht genauso präzise, was nicht verwunderlich ist, da sich in den Konvergenztests gezeigt hat, dass diese Trainingsdatengröße am Limit der numerischen Stabilität ist. Dies kann ausgeglichen werden durch ein niedrigeres Residuum, was allerdings die Laufzeit deutlich erhöhen kann. Die Referenz-Implementierung mit 4 $\times$ NVIDIA A100 ist in diesem Beispiel langsamer beim Training als eine RTX 3080 mit Mixed-Tensor-Kernel, allerdings ist das gewonnene Modell etwas besser. Das berechnete Modell der beiden Referenz-Implementierungen ist auf beiden Testsystemen gleich gut, aber bei der RTX 3080 ist der Mixed-Tensor-Kernel 26.8 Mal schneller als die Referenz-Implementierung. Anstatt 2 Minuten und 14 Sekunden wird bei dieser Variante fast eine ganze Stunde, 59 Minuten und 46 Sekunden, gewartet. Durch die Nutzung nur einer Grafikkarte ist der Matrixaufbau bei der RTX 3080 Mixed-Tensor Variante minimal ungenauer als bei den 4 A100 Karten mit selben Kernel, da hier 16384 Zwischenergebnisse aufsummiert werden und bei den A100 pro GPU nur 4096. Dieser Unterschied zeigt sich auch in unterschiedlich genauen Modellen und einer schlechteren Konvergenz.

	Trainingszeit [s]	Genauigkeit [%]
4 × A100 Double-Tensor	34.457	97.1
4 × A100 Mixed-Tensor	19.793	95.9
4 × A100 Mixed-Tensor*	45.127	96.9
4 × A100 Referenz	145.828	96.4
RTX 3080 Mixed-Tensor	134.068	95.2
RTX 3080 Referenz	3586.394	96.4

Tabelle 5.12: Trainingszeit und Genauigkeit der Vorhersage bei einem Datensatz mit 65536 Datenpunkten und 16384 Features. Die angegebenen Werte sind der Durchschnitt aus jeweils 2 Läufen. Das Abbruchkriterium liegt bei $r < 5000$ (bei* $r < 5$). Die Tests wurden auf ipvsgpu1 und pcsgs02 ausgeführt.

	# Iterationen	Iterationsdauer [s]	Trainingszeit [s]
D-T-PR-ZU	1550	8.076	12518
D-T-PR-0	1067	8.032	8570
D-T-FR-0	1008	8.035	8099
REF	1379	26.682	36794

Tabelle 5.13: Laufzeitanalyse für das Lernen des SAT-6 Trainingsdatensatz mit 324000 Datenpunkten und 3137 Features mit einem Abbruchkriterium des Residuums $r < 0.0011$. Die angegebenen Werte sind der Durchschnitt aus jeweils 2 Läufen.

5.4.2 SAT-6 Laufzeit-Tests

In diesem Szenario wird der lineare Kernel auf 4 NVIDIA A100 GPUs auf der ipvsgpu1 genutzt. Die in Tabelle 5.13 gezeigten Daten stammen aus den Konvergenztests aus Abschnitt 5.2.6, nur diesmal mit dem Fokus auf der Trainingszeit.

Die Tensor Kernel sind in diesem Fall ungefähr 3.3 Mal schneller als der Standard-Kernel. Die größere Abweichung bei der Variante mit zuverlässigen Updates entsteht durch die zusätzliche Ausführung des Kernel beim Korrekturschritt. Insgesamt ist der Tensor-Kernel mit F-R ungefähr 4.5 Mal schneller, benötigt folglich anstatt über 10h nur noch 2h und 15 Minuten. Das Problem lässt sich mit radialen Kernel deutlich schneller lösen [VBP22b], besonders da das im Testfall genutzte Abbruchkriterium deutlich strenger als benötigt gesetzt worden ist. Allerdings ist die hier erreichte Genauigkeit bei allen Varianten bei der Vorhersage des Testdatensatzes bei 96% und somit um 1% höher als bisher von Van Craen et al. [VBP22b].

6 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde die PLSSVM Bibliothek im Hinblick auf gemischte Genauigkeiten und der Nutzung von spezialisierten GPU-Matrixeinheiten analysiert. Die hohe Relevanz von Klassifikatoren wie SVMs und wie versucht wird mit gemischten Genauigkeiten moderne Hardware optimal auszunutzen, wurde zu Beginn in Kapitel 2 erörtert. Darauf wurden die Grundlagen für die weitere Arbeit erläutert.

Bei der Implementierung stellte sich besonders die hohe Konditionszahl des LGS als Problem heraus, das auch nicht abschließend gelöst werden konnte. Auch die DDOT Einträge der zu erstellenden Matrix erschwerten die Implementierung bezüglich ihrer Stabilität. Durch eine verbesserte Initiallösung, höhere Genauigkeit auf der CPU durch Kahans-Summations-Algorithmus und zwei neuen Kernel für den rechenintensiven Matrixaufbau mit folgender GEMV Berechnung, konnte die Stabilität erhöht werden. Eine Herausforderung bei den Kernel war, dass der Flaschenhals nicht bei der Anzahl an Berechnungen lag, sondern bei Speichertransaktionen, worauf die Nutzung von Matrixeinheiten keinen Einfluss hat und dass die Matrixeinheiten außerhalb der Nutzung von doppelter Genauigkeit zusätzlich mit einer niedrigeren Genauigkeit als Float rechnen. Daher wurde die Aufteilung innerhalb eines Blockes komplett überarbeitet und für die Nutzung der Matrixeinheiten optimiert.

Als Ergebnis hat sich gezeigt, dass die vorgestellte Implementierung in gemischter Genauigkeit im Testszenario bis zu einer Datensatzgröße von 65536 Datenpunkten mit jeweils 16874 Features konvergiert und gute Ergebnisse liefert. Die Implementierung, bei der auch der Kernel vollständig in doppelter Genauigkeit gerechnet wird, verbessert gegenüber der bisherigen Implementierung von PLSSVM die Stabilität, sodass bis zu 27% weniger Iterationen notwendig waren um die gleiche Genauigkeit zu erreichen. Die Laufzeiten der Kernel mit Matrixeinheiten war in den Testszenarien zwischen 3-4 Mal niedriger als die bisherigen CUDA Kernel. Es wird erwartet, dass durch die Änderungen noch deutlich größere und komplexere Trainingsdaten verarbeitet werden können als bisher getestet. Werden die Vorteile aus neuen Kernel und gemischten Genauigkeiten vereint, kann ein System mit einer einzelnen RTX 3080 ein System mit 4 NVIDIA A100 GPUs in der Trainingszeit übertreffen. Dies kommt einer Laufzeitreduktion um Faktor 28 für die RTX 3080 gleich und ist ein großer Erfolg, besonders unter Berücksichtigung der Strom- und Anschaffungskosten. Mit einer weitverbreiteten Desktop-GPU wird somit die Leistung eines Rechenzentrumsknotens erreicht. Aber auch für komplexere und größere Probleme konnte unter Ausnutzung des neuen Kernels in doppelter Genauigkeit und des stabileren CG-Algorithmus die Trainingszeiten um einen Faktor zwischen $4 - 4.5$ verringert werden, was mehr als das Doppelte an Verbesserung der Laufzeit ist bezüglich der theoretischen Rechenleistung durch Matrixoperationen. Durch das Ausnutzen von gemischter Genauigkeit und den Matrixeinheiten konnte nicht nur aktuelle Hardware wie GPUs besser ausgenutzt werden und die Laufzeit deutlich verkürzt werden, sondern auch die Stabilität bei gleichbleibend genauen Zahlenformaten massiv erhöht werden.

Ausblick

Aktuell sind die neuen Kernel nur für moderne CUDA-fähige GPUs verfügbar. Hier würde es sich anbieten diese auch auf HIP zu übertragen. Zusätzlich können mit einem ähnlichen Kernel-Aufbau, der die gezeigte Block-Struktur nutzt, auch Kernel optimiert werden, die keine Matrixeinheiten nutzen. Unter anderem könnte dadurch ein neuer radialer Kernel geschrieben werden und auch bereits andere vorhandene Kernel könnten von dem Aufbau der Matrixeinheiten-Kernel profitieren. Eine weitere Möglichkeit wäre ein Kernel mit Matrixeinheiten, bei dem die Trainingsdaten in halber Genauigkeit gehalten werden. Dies halbiert die Speicherbandbreite gegenüber einfacher Genauigkeit, erspart die Transformation in TF32, welche eine gleich große Mantisse besitzen wie FP16, und akkumuliert mit dem richtigen Tensor-Layout das Ergebnis ebenso in FP32 [NVI22a]. Hier wäre interessant, wie weit man mit unterschiedlichen Genauigkeiten gehen kann. Auch die vorgestellten Kernel können noch verbessert werden. Eine Möglichkeit wäre, wie bereits kurz erwähnt (4.3.1 5.3.1), die Blöcke nicht in Quadrate zu unterteilen, sondern in Rechtecke. Dies würde eine bessere Granularität ermöglichen und dadurch entweder mehr Warps oder weniger Speichertransaktionen unter Ausnutzung der restlichen Register ermöglichen. Eine performante Umgehung des Integer-Limits könnte implementiert werden. Eine fortgeschrittene Idee besteht dabei daraus, die große Schleife über Features im Matrix-Aufbau in eine weitere Schleife zu packen. Diese äußere Schleife iteriert dann über Zeiger auf die Trainingsdaten, die voneinander in der Größenordnung von 2^{31} entfernt sind.

Eine Limitierung stellt derzeit die Kondition des LGS dar. Hier könnte ein CG-Verfahren mit Vorkonditionierung Abhilfe schaffen. Mit geringerer Kondition dürfte die Implementierung mit gemischten Genauigkeiten auch für größere Datensätze stabil sein und somit auch auf GPUs, die nicht für Datenzentren ausgelegt sind, schnelle und ausreichend genaue Ergebnisse liefern. Weitere Ansätze könnten darauf basieren, bei ausreichendem Speicher die Matrix einmal in hoher Genauigkeit aufzubauen und dann entweder auf GPU oder CPU weiterzuverwenden. Da besonders bei großen Datensätzen fast die komplette Dauer einer CG-Iteration auf dem Aufbau der Matrix liegt, würde hier eine Menge an Operationen eingespart werden. Ein weiterer Ansatz wäre es, den Datensatz bei der Berechnung aufzuteilen, um Teilmatrizen des Gesamtproblems zu erhalten, die dann entweder komplett im Speicher gehalten werden können oder bezüglich ihrer Kondition ausreichend stabil sind. Die so gelösten Teilmatrizen könnten mehrere Hyperebenen ergeben, die dann mit einem geeigneten Verfahren entweder zu einer Hyperebene verrechnet werden oder nach einem Mehrheitsprinzip arbeiten könnten.

Ein weiteres Feld, welches genauer betrachtet werden könnte, wären die Korrekturschemata. Das bereits vorhandene Korrekturschema über die Neuberechnung des Residuums liefert keine brauchbaren Ergebnisse. Die zuverlässigen Updates zerstören zwar nicht das Ergebnis, aber die Konvergenzgeschwindigkeit nimmt ab. Ebenso soll β nach P-R die Stabilität erhöhen, was es auch tut für die zuverlässigen Updates, allerdings wirkt sich die Vorschrift negativ auf die Varianten ohne Korrekturschema aus. Hier könnten die genauen Ursachen untersucht werden.

Literaturverzeichnis

[AMD21] AMD. *INTRODUCING AMD CDNA™ 2ARCHITECTURE*. 2021. URL: https://www.amd.com/system/files/documents/amd-cdna2-white-paper.pdf (besucht am 24. 09. 2022) (zitiert auf S. 15).

[AMD22] AMD. *HIP Programming Guide v4.5*. 2022. URL: https://rocmdocs.amd.com/en/latest/Programming_Guides/HIP-GUIDE.html (besucht am 10. 09. 2022) (zitiert auf S. 26).

[BBD+09] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, S. Tomov. „Accelerating scientific computations with mixed precision algorithms“. In: *Computer Physics Communications* 180.12 (Dez. 2009), S. 2526–2533. DOI: 10.1016/j.cpc.2008.11.005 (zitiert auf S. 18).

[BCP22] M. Breyer, A. V. Craen, D. Pflüger. „A Comparison of SYCL, OpenCL, CUDA, and OpenMP for Massively Parallel Support Vector Machine Classification on Multi-Vendor Hardware“. In: *International Workshop on OpenCL*. ACM, Mai 2022. DOI: 10.1145/3529538.3529980 (zitiert auf S. 18, 26, 84).

[BGM+15] S. Basu, S. Ganguly, S. Mukhopadhyay, R. DiBiano, M. Karki, R. Nemani. *DeepSat - A Learning framework for Satellite Imagery*. 2015. DOI: 10.48550/ARXIV.1509.03602 (zitiert auf S. 69).

[BGV92] B. E. Boser, I. M. Guyon, V. N. Vapnik. „A training algorithm for optimal margin classifiers“. In: *Proceedings of the fifth annual workshop on Computational learning theory - COLT '92*. ACM Press, 1992. DOI: 10.1145/130385.130401 (zitiert auf S. 17).

[BT69] M. L. Balinski, A. W. Tucker. „Duality Theory of Linear Programs: A Constructive Approach with Applications“. In: *SIAM Review* 11.3 (Juli 1969), S. 347–377. DOI: 10.1137/1011060 (zitiert auf S. 24).

[CBB+10] M. Clark, R. Babich, K. Barros, R. Brower, C. Rebbi. „Solving lattice QCD systems of equations using mixed precision solvers on GPUs“. In: *Computer Physics Communications* 181.9 (Sep. 2010), S. 1517–1528. DOI: 10.1016/j.cpc.2010.05.002 (zitiert auf S. 16, 18, 45).

[CL11] C.-C. Chang, C.-J. Lin. „LIBSVM“. In: *ACM Transactions on Intelligent Systems and Technology* 2.3 (Apr. 2011), S. 1–27. DOI: 10.1145/1961189.1961199 (zitiert auf S. 17, 47).

[Cla20] K. Clark. „EFFECTIVE USE OF MIXED PRECISION FOR HPC“. In: GTC2020, 2020. URL: https://developer.nvidia.com/gtc/2020/video/s21725 (zitiert auf S. 19, 44, 45, 48, 49).

[CS00] N. Cristianini, J. Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, März 2000. DOI: 10.1017/cbo9780511801389 (zitiert auf S. 17).

[CV95] C. Cortes, V. Vapnik. „Support-vector networks“. In: *Machine Learning* 20.3 (Sep. 1995), S. 273–297. DOI: 10.1007/bf00994018 (zitiert auf S. 17, 21).

[Gol91] D. Goldberg. „What Every Computer Scientist Should Know About Floating Point Arithmetic“. In: *ACM Computing Surveys* 23.1 (1991), S. 5–48 (zitiert auf S. 28–30).

[Gor14] A. M. Gorelik. „Statements, data types and intrinsic procedures in the Fortran standards (1966-2008)“. In: *ACM SIGPLAN Fortran Forum* 33.3 (Dez. 2014), S. 5–17. DOI: 10.1145/2701654.2701655 (zitiert auf S. 27).

[Gro22a] K. Group. *OpneCL*. 2022. URL: https://www.khronos.org/opencl/ (besucht am 10. 09. 2022) (zitiert auf S. 26).

[Gro22b] K. Group. *SYCL*. 2022. URL: https://www.khronos.org/sycl/ (besucht am 10. 09. 2022) (zitiert auf S. 26).

[GST05] D. Göddeke, R. Strzodka, S. Turek. *Accelerating double precision FEM simulations with GPUs*. Univ., 2005 (zitiert auf S. 18).

[GST07] D. Göddeke, R. Strzodka, S. Turek. „Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations“. In: *International Journal of Parallel, Emergent and Distributed Systems* 22.4 (Aug. 2007), S. 221–256. DOI: 10.1080/17445760601122076 (zitiert auf S. 18).

[GWX+12] A. W. Götz, M. J. Williamson, D. Xu, D. Poole, S. L. Grand, R. C. Walker. „Routine Microsecond Molecular Dynamics Simulations with AMBER on GPUs. 1. Generalized Born“. In: *Journal of Chemical Theory and Computation* 8.5 (Apr. 2012), S. 1542–1555. DOI: 10.1021/ct200909j (zitiert auf S. 18).

[HBT+20] A. Haidar, H. Bayraktar, S. Tomov, J. Dongarra, N. J. Higham. „Mixed-precision iterative refinement using tensor cores on GPUs to accelerate solution of linear systems“. In: *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 476.2243 (Nov. 2020), S. 20200110. DOI: 10.1098/rspa.2020.0110 (zitiert auf S. 19).

[Hig93] N. J. Higham. „The Accuracy of Floating Point Summation“. In: *SIAM Journal on Scientific Computing* 14.4 (Juli 1993), S. 783–799. DOI: 10.1137/0914050 (zitiert auf S. 31).

[HMHS17] X. Huang, A. Maier, J. Hornegger, J. A. Suykens. „Indefinite kernels in least squares support vector machines and principal component analysis“. In: *Applied and Computational Harmonic Analysis* 43.1 (Juli 2017), S. 162–172. DOI: 10.1016/j.acha.2016.09.001 (zitiert auf S. 17).

[HS52] M. R. Hestenes, E. Stiefel. „Methods of conjugate gradients for solving“. In: *Journal of research of the National Bureau of Standards* 49.6 (1952), S. 409 (zitiert auf S. 25, 69).

[Kah65] W. Kahan. „Pracniques: further remarks on reducing truncation errors“. In: *Communications of the ACM* 8.1 (1965), S. 40 (zitiert auf S. 31).

[KSH17] A. Krizhevsky, I. Sutskever, G. E. Hinton. „ImageNet classification with deep convolutional neural networks“. In: *Communications of the ACM* 60.6 (Mai 2017), S. 84–90. DOI: 10.1145/3065386 (zitiert auf S. 38).

[KT13] H. W. Kuhn, A. W. Tucker. „Nonlinear Programming“. In: *Traces and Emergence of Nonlinear Programming*. Springer Basel, Juli 2013, S. 247–258. DOI: 10.1007/978-3-0348-0439-4_11 (zitiert auf S. 24).

[LS20] A. Li, S. M. Su. „Accelerating Binarized Neural Networks via Bit-Tensor-Cores in Turing GPUs“. In: *IEEE Transactions on Parallel and Distributed Systems* (2020), S. 1–1. DOI: 10.1109/tpds.2020.3045828 (zitiert auf S. 19).

[MAM18] M. Martineau, P. Atkinson, S. McIntosh-Smith. „Benchmarking the NVIDIA V100 GPU and Tensor Cores“. In: *Lecture Notes in Computer Science*. Springer International Publishing, Dez. 2018, S. 444–455. DOI: 10.1007/978-3-030-10549-5_35 (zitiert auf S. 18).

[MCL+18] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, J. S. Vetter. „NVIDIA Tensor Core Programmability, Performance & Precision“. In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, Mai 2018. DOI: 10.1109/ipdpsw.2018.00091 (zitiert auf S. 19).

[MNA+17] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, H. Wu. *Mixed Precision Training*. 2017. DOI: 10.48550/ARXIV.1710.03740 (zitiert auf S. 18).

[MR16] M. N. Murty, R. Raghava. „Kernel-Based SVM“. In: *Support Vector Machines and Perceptrons*. Springer International Publishing, 2016, S. 57–67. DOI: 10.1007/978-3-319-41063-0_5 (zitiert auf S. 24).

[NG47] J. von Neumann, H. H. Goldstine. „Numerical inverting of matrices of high order“. In: *Bulletin of the American Mathematical Society* 53.11 (1947), S. 1021–1099. DOI: bams/1183511222. URL: https://doi.org/ (zitiert auf S. 27).

[NVI16a] NVIDIA. *NVIDIA GeForce GTX 1080 Whitepaper*. 2016. URL: https://www.es.ele.tue.nl/~heco/courses/ECA/GPU-papers/GeForce_GTX_1080_Whitepaper_FINAL.pdf (besucht am 04. 10. 2022) (zitiert auf S. 68).

[NVI16b] NVIDIA. *NVIDIA Tesla P100 Whitepaper*. 2016. URL: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf (besucht am 04. 10. 2022) (zitiert auf S. 68).

[NVI20] NVIDIA. *NVIDIA A100 Tensor Core GPU Architecture*. 2020. URL: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf (besucht am 10. 09. 2022) (zitiert auf S. 15, 31, 32, 39, 53, 68, 79, 84).

[NVI21] NVIDIA. *NVIDIA Ampere GA102 GPU Architecture whitepaper*. 2021. URL: https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf (besucht am 21. 09. 2022) (zitiert auf S. 15, 53, 68, 85).

[NVI22a] NVIDIA. *CUDA C programming guide*. 2022. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/ (besucht am 10. 09. 2022) (zitiert auf S. 19, 37, 40, 56, 59, 63, 92).

[NVI22b] NVIDIA. *CUDA C++ Best Practices Guide*. 2022. URL: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/ (besucht am 10. 09. 2022) (zitiert auf S. 33, 36, 37, 59).

[NVI22c] NVIDIA. *CUDA Zone*. NVIDIA. 2022. URL: https://developer.nvidia.com/cuda-zone (besucht am 10. 09. 2022) (zitiert auf S. 26, 31).

[NVI22d] NVIDIA. *NVIDIA Hopper Architecture In Depth*. 2022. URL: https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/ (besucht am 10. 09. 2022) (zitiert auf S. 29, 33, 53).

[NVI22e] NVIDIA. *The user manual for NVIDIA Nsight Compute*. 2022. URL: https://docs.nvidia.com/nsight-compute/NsightCompute/index.html (besucht am 18. 09. 2022) (zitiert auf S. 78).

[Ope22] A. R. B. OpenMP. *OpenMP*. 2022. URL: https://www.openmp.org/about/openmp-faq/#WhatIs (zitiert auf S. 26).

[Pla98] J. Platt. *Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines*. Techn. Ber. MSR-TR-98-14. Microsoft, Apr. 1998. URL: https://www.microsoft.com/en-us/research/publication/sequential-minimal-optimization-a-fast-algorithm-for-training-support-vector-machines/ (zitiert auf S. 17, 23, 47, 48).

[PR69] E. Polak, G. Ribiere. „Note sur la convergence de méthodes de directions conjuguées“. In: *Revue française d'informatique et de recherche opérationnelle. Série rouge* 3.16 (1969), S. 35–43 (zitiert auf S. 45, 69).

[RMN09] R. Raina, A. Madhavan, A. Y. Ng. „Large-scale deep unsupervised learning using graphics processors“. In: *Proceedings of the 26th Annual International Conference on Machine Learning - ICML '09*. ACM Press, 2009. DOI: 10.1145/1553374.1553486 (zitiert auf S. 38).

[SBLV02] J. Suykens, J. D. Brabanter, L. Lukas, J. Vandewalle. „Weighted least squares support vector machines: robustness and sparse approximation“. In: *Neurocomputing* 48.1-4 (Okt. 2002), S. 85–105. DOI: 10.1016/s0925-2312(01)00644-0 (zitiert auf S. 17).

[SGB+02] J. A. K. Suykens, T. V. Gestel, J. D. Brabanter, B. D. Moor, J. Vandewalle. *Least Squares Support Vector Machines*. WORLD SCIENTIFIC, Okt. 2002. DOI: 10.1142/5089 (zitiert auf S. 15, 17).

[She94] J. R. Shewchuk. *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*. Techn. Ber. USA, 1994 (zitiert auf S. 25).

[SLV] J. Suykens, L. Lukas, J. Vandewalle. „Sparse approximation using least squares support vector machines“. In: *2000 IEEE International Symposium on Circuits and Systems. Emerging Technologies for the 21st Century. Proceedings (IEEE Cat No.00CH36353)*. Presses Polytech. Univ. Romandes. DOI: 10.1109/iscas.2000.856439 (zitiert auf S. 17).

[SV96] G. L. G. Sleijpen, H. A. van der Vorst. „Reliable updated residuals in hybrid Bi-CG methods“. In: *Computing* 56.2 (Juni 1996), S. 141–163. DOI: 10.1007/bf02309342 (zitiert auf S. 45).

[SV99a] J. A. K. Suykens, J. Vandewalle. „Least Squares Support Vector Machine Classifiers“. In: *Neural Processing Letters* 9.3 (1999), S. 293–300. ISSN: 1573-773X. DOI: 10.1023/A:1018628609742. URL: https://doi.org/10.1023/A:1018628609742 (zitiert auf S. 17, 23).

[SV99b] J. Suykens, J. Vandewalle. „Multiclass least squares support vector machines“. In: *IJCNN'99. International Joint Conference on Neural Networks. Proceedings (Cat. No.99CH36339)*. IEEE, 1999. DOI: 10.1109/ijcnn.1999.831072 (zitiert auf S. 17).

[TCJM21] J. Tu, M. A. Clark, C. Jung, R. Mawhinney. „Solving DWF Dirac Equation Using Multi-splitting Preconditioned Conjugate Gradient with Tensor Cores on NVIDIA GPUs“. In: (2021). DOI: 10.48550/ARXIV.2104.05615 (zitiert auf S. 19).

[TOP22] TOP500.org. *TOP500 LIST - JUNE 2022*. Juni 2022. URL: https://www.top500.org/lists/top500/list/2022/06/ (besucht am 25. 09. 2022) (zitiert auf S. 15).

[Van18] A. Van Craen. *GPU-beschleunigte Support-Vector Machines*. de. 2018. DOI: 10.18419/OPUS-10148 (zitiert auf S. 17, 21).

[Vap06] V. Vapnik. „A Method of Minimizing Empirical Risk for the Problem of Pattern Recognition“. In: *Estimation of Dependences Based on Empirical Data*. Springer New York, 2006, S. 139–180. DOI: 10.1007/0-387-34239-7_6 (zitiert auf S. 15, 17).

[VBP22a] A. Van Craen, M. Breyer, D. Pflüger. „PLSSVM—Parallel Least Squares Support Vector Machine“. In: *Software Impacts* 14 (Nov. 2022), S. 100343. DOI: 10.1016/j.simpa.2022.100343 (zitiert auf S. 17).

[VBP22b] A. Van Craen, M. Breyer, D. Pflüger. „PLSSVM: A (multi-)GPGPU-accelerated Least Squares Support Vector Machine“. In: (2022). DOI: 10.48550/ARXIV.2202.12674 (zitiert auf S. 15, 18, 47, 68, 69, 84, 90).

[Wil94] J. H. Wilkinson. *Rounding errors in algebraic processes*. Courier Corporation, 1994 (zitiert auf S. 18).

[WSH+21] X. Wang, T. Schneider, M. Hersche, L. Cavigelli, L. Benini. „Mixed-Precision Quantization and Parallel Implementation of Multispectral Riemannian Classification for Brain-Machine Interfaces“. In: *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, Mai 2021. DOI: 10.1109/iscas51556.2021.9401564 (zitiert auf S. 18).

Alle URLs wurden zuletzt am 10. 09. 2022 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift