Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

GPU-beschleunigte Support-Vector Machines

Alexander Van Craen

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. rer. nat. Dirk Pflüger
Betreuer/in:	Dipl.-Inf. David Pfander
Beginn am:	10. Oktober 2017
Beendet am:	10. April 2018

Kurzfassung

Data-Mining gewinnt immer mehr an Bedeutung, denn es stehen immer mehr Daten zur Verfügung. Um der stetig wachsenden Anzahl an Daten entgegenzuwirken, werden eine immer stärkere Hardware, bessere Algorithmen und optimierte Implementierungen benötigt. Es ist beispielsweise möglich mithilfe massiver Parallelität auf Grafikkarten die Rechenzeit zu verkürzen.
Eine Möglichkeit Daten zu klassifizieren ist das überwachte maschinelle Lernen. Die Support-Vector Machine (SVM) ist eines dieser Verfahren. Sie erstellt ein Modell, worin die Trainingsdaten als Punkte im Raum interpretiert werden. Es werden diejenigen Datenpunkte gesucht, mit denen ein linearer Separator aufgestellt werden kann, der die Daten durch das breiteste Band voneinander trennt (large margin classifier). Anhand dieses gelernten Modells können dann neue Daten effizient klassifiziert werden.
In dieser Arbeit wird eine Least Square Support Vektor Maschine implementiert. Bei der Least Squares Support Vektor Maschine werden nicht die wenigen Datenpunkte gesucht, die für die Separation wichtig sind (Support Vektoren), sondern bei allen Datenpunkten der proportionale Fehler bestimmt, und daraus die separierende Hyperebene abgeleitet. Die Least Squares Support Vektor Maschine wird mithilfe von NVIDIAs CUDA Parallel Programming Models für Grafikkarten implementiert und für NVIDIAs QUADRO GP100 optimiert. Dabei wurde explizit darauf geachtet, dass die GPU-Implementierung möglichst verzweigungsfrei ist. Außerdem wird die SoA Datenstruktur verwendet, und die Zugriffe über das Shared Memory optimiert.
Damit konnte, bei ähnlicher Genauigkeit, auf der NVIDIAs QUADRO GP100 Grafikkarte, mit ca. 2,3 TFLOPS mit doppelter Genauigkeit, eine Laufzeitverbesserung um mehr als das 300-Fache gegenüber der gebräuchlichen CPU-Library for Support Vector Machines, LIBSVM, Implementierung gemessen werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis der Listings

Verzeichnis der Algorithmen

Abkürzungsverzeichnis

AoS Array of Structs. 35

CG konjugierten Gradienten. 11

CPU Central Processing Unit (Prozessor eines Computers). 6

CUDA Compute Unified Device Architecture. 5

GPU Graphics Processing Unit (engl. Grafikprozessor). 15

KI künstliche Intelligenz. 15

KKT Karush-Kuhn-Tucker. 22

LGS linearen Gleichungssystems. 15

LIBSVM Library for Support-Vector Machines. 5

LS-SVM Least Squares SVM. 5, 15

SM Streaming Multiprozessor. 27, 29

SMO Sequential Minimal Optimization. 15

SoA Structure of Arrays. 35

STL C++ Standard Template Library. 35

SVM Support-Vector Machine. 3

1. Einleitung

Es werden immer mehr Daten produziert und diese auch gesammelt. Beispielsweise generieren verknüpfte Autos, das Internet der Dinge oder auch medizinische Geräte ständig neue Daten. Um diese rasant wachsende Datenmenge analysieren und klassifizieren zu können, muss vermehrt auf stark automatisierte Verfahren zurückgegriffen werden. Mit dieser Aufgabe beschäftigt sich ein Teilgebiet der künstlichen Intelligenz (KI), das Data-Mining. Data-Mining ist „die Anwendung von Algorithmen zum Extrahieren von Mustern aus Daten“[FPSU96]. Hier werden systematische Anwendungen zusammen mit statistischen Methoden genutzt, um aus (sehr) großen Datenbeständen Erkenntnisse zu gewinnen. Mit der Suche nach Querverbindungen in diesen Daten wird in vielen Lebensbereichen nach Erkenntnisgewinn geforscht. Data Mining wird nicht nur in der Medizin und anderen lebensverbessernden/-ermöglichenden Forschungen eingesetzt, um beispielsweise die Krebsfrüherkennung zu verbessern [DWK05], sondern hat auch durchaus angenehmen Nutzen, wie zum Beispiel bei Spam Filtern [JLLH11].

Eine Aufgabenstellung des Data-Minings ist die Klassifikation. Die Support-Vector Machine (SVM) ist ein dafür verwendbarer Klassifikator. Sie versucht eine Menge von Objekten so zu unterteilen, dass um die Klassen das breiteste mögliche Band entsteht, in welchem keine Objekte/Daten liegen. Damit gehört die SVM zu der Klasse der Large Margin Classifier (engl. „Breiter-Rand-Klassifikator“). Deren Einsatzgebiet ist sehr vielfältig. Sie werden unter anderem in der Handschrifterkennung [BHB02], sowie in modernen Verfahren zur Untersuchung von Erddämmen und Deichen eingesetzt [FCK17]. Je nach Anwendungsgebiet hat sich die SVM gegenüber anderen weit verbreiteten Verfahren, wie beispielsweise neuronale Netzwerke (KNN/ANN) [AYKY17], als überlegen erwiesen.

Durch die stark wachsende Anzahl an Datenmengen bietet es sich an, Grafikkarten in die Berechnung zu integrieren. Ihre massive Parallelität und ihre starke Rechenpower sind für große Datenmengen gut geeignet. Aus diesem Grund wird in dieser Arbeit, mithilfe von NVIDIAs CUDA Parallel Programming Models, eine Least Squares SVM (LS-SVM) für NVIDIA Grafikkarten implementiert, und explizit für die NVIDIA GP100 optimiert. Dabei ist eine LS-SVM eine SVM die ihr Modell nicht auf wenige Support Vektoren stützt, sondern auf alle gelernten Daten, die proportional zu ihrer Relevanz gewichtet werden. Diese Arbeit stützt sich maßgeblich auf die Arbeit von Chu, Ong und Keerthi [COK05]. Als Grundlagen dafür dienen unter anderem die Arbeiten von Suykens et al. [SBLV02], Suykens, Gestel und Brabanter [SGB02] und Suykens et al. [SLD+99]. Die Lösung des entstehenden linearen Gleichungssystem (LGS) wird mithilfe des CG-Verfahrens, beschrieben in [She+94], berechnet.

Ziel der Arbeit ist es, die grundlegende Funktionalität für ein Drop-in Alternative für die am weitesten verbreitete SVM Bibliothek LIBSVM [CL11] bereitzustellen, um moderne Anwendungen wie zum Beispiel die Alzheimer Forschung [KRG+15] deutlich zu beschleunigen. Dabei wird, anders als in LIBSVM, keine Sequential Minimal Optimization (SMO) genutzt. Auch wird nur die binäre Klassifikation, also die Klassifikation von zwei Klassen und keine Multiklassen-Klassifikation, unterstützt.

Nach einer Auseinandersetzung mit verwandten Arbeiten wird in Kapitel 3 (Support-Vector Machine) zunächst die SVM erklärt und damit die Grundlagen für die Implementation gelegt. Das Kapitel 4 (CUDA) beschäftigt sich mit der grundlegenden Funktionalität des NVIDIA CUDA Parallel Programming Models. Dabei werden die grundlegenden Konzepte sowie für die Implementierung wesentliche Befehle erklärt. Das darauffolgende Kapitel 5 (Implementierung) befasst sich mit der Implementierung. Es werden die grundlegenden Aspekte der LS-SVM-Implementierung und deren Optimierung erklärt. Abschließend wird die Graphics Processing Unit (engl. Grafikprozessor) (GPU)-Implementierung evaluiert. Hierfür wird in Kapitel 6 (Ergebnisse) zuerst die genutzte Hardware zusammen mit den Compilern genannt. Anschließend folgen Vergleiche um die Laufzeit, sowie die erreichte Genauigkeit, einstufen zu können. Ein abschließendes Fazit, sowie ein Ausblick wie an der Implementierung weiterentwickelt werden kann, vervollständigen die Arbeit.

2. Verwandte Arbeiten

Die Grundidee der Support-Vektor Machines stammt aus dem Jahr 1992 von Boser, Guyon und Vapnik [BGV92] und wurde im Jahr 1995 von Cortes und Vapnik auf den heutigen Stand erweitert [CV95]. Darauf aufbauend wird von Chang und Lin [CL11] die heute am weitesten verbreitete SVM-Bibliothek LIBSVM veröffentlicht. Diese basiert auf der SMO [Pla98] Methode und ist ein Supervised (engl. überwachten) Learning Verfahren [HTF09]. Eine gute Darstellung der grundlegenden Theorie, der SVM und weiteren Kernel basierter Verfahren, wurde von Cristianini und Shawe-Taylor [CS00] verfasst.

Es gibt verschiedene Bibliotheken für die Anwendung einer SVM: LIBSVM wurde ursprünglich in C programmiert. Inzwischen ist sie auf verschiedene Sprachen wie zum Beispiel C++ und Java portiert worden, und bietet Schnittstellen für weitere Programmiersprachen wie Python an. Außerdem wurde von Athanasopoulos et al. [ADMK11] auch eine LIBSVM-CUDA Implementierung entwickelt. LIBSVM ist eine sehr weit verbreitete SVM Bibliothek, und basiert, anders als diese Arbeit, auf dem SMO-Verfahren.

LIBLINEAR[FCH+08] ist eine LIBSVM Erweiterung, die nur auf einen linearen Support Vektor Maschine Kernel (Abschnitt 3.1.2) ausgelegt ist. Außerdem wird hier eine Multicore Version implementiert, was bei LIBSVM nicht der Fall ist, denn diese kann nur einen Prozessorkern nutzen. LIBLINEAR basiert, anders als diese Implementierung, auch auf dem SMO-Verfahren. Ein weiterer Algorithmus, das parallele Lösen mittels des SMO Verfahrens, wurde von Zeng et al. [ZYX+08] für die CPU beschrieben.

Auf Vapnik's SVM aufbauend gibt es ein Werk von Joachims [Joa02; Joa99]. Er entwickelt eine weitere Bibliothek SVM light[1] in C. Diese CPU-Implementierung und der Artikel legen den Fokus auf die Auswahl und eine Einschränkung der Datensets.

In dieser Arbeit wird nicht das Verfahren, auf welches alle bis hier genannten Implementierungen aufbauen, sondern die von Suykens et al. entwickelte Least Square Version [SLD+99], implementiert. Die Methode der kleinsten Quadrate (Least Square) ist eine mathematische Ausgleichungsrechnung, bei der in einer Punktwolke eine Funktion gesucht wird, die möglichst nahe an diesen Datenpunkten verläuft. Dies wird durch die Minimierung des quadratischen Residuums, also dem quadratischen Abstand zwischen den Punkten und der Funktion erreicht [Wei02]. Auf den Unterschied zwischen der LS-SVM und der SVM nach Vapnik wird in Abschnitt 3.1 eingegangen. Außerdem untersuchen Ye und Xiong [YX07] genau diese Korrelation der beiden Verfahren.

Von Do, Nguyen und Poulet [DNP08] wird behandelt, wie mit sehr vielen Daten auf der Grafikkarte umgegangen werden kann. Sie legen den Fokus auf das Splitten der Daten. Diese Ausarbeitung hingegen, geht davon aus, dass alle Daten gleichzeitig auf die Grafikkarte geladen werden können, und legt den Fokus auf die Optimierung des Least Squares Verfahren für die GPU.

[1] http://svmlight.joachims.org

Catanzaro, Sundaram und Keutzer [CSK08] befasst sich ebenfalls mit einer Grafikkartenimplementierung. Anders als in dieser Arbeit implementieren sie auch das SMO Verfahren.

Nan et al. [NSC+17] untersuchen die LS-SVM genauer. Sie legen den Fokus auf die Verkleinerung der Eingabedaten. Anders als bei der Arbeit von Nan et al. wird in dieser Ausarbeitung angenommen, dass die Eingabedaten selektiv genug gewählt sind, sodass hier nicht mehr minimiert werden muss und kann.

Für die LS-SVM legt Suykens in diversen Veröffentlichungen die Grundlage. Er entwickelte die LS-SVM [SGB02], und verbessert diese in Folgearbeiten, in welchen er die Gewichtung einführt [SBLV02] und seine Implementierung durch eine dünn besetzte Datenstruktur erweitert[SLV00]. Außerdem führt er die Multiklassen Least-Square-Klassifikation ein [SV99b]. Mit diesen Veröffentlichungen legt er die Grundlage für diese Least Square GPU Implementierung, auch wenn hier nicht alles davon aufgegriffen werden kann.
Zusammen mit Xiaolin Huang und anderen, publiziert er die indefiniten Kernel in der LS-SVM [HMHS17].

Für die Optimierung von CUDA bietet Harris in seiner Präsentation [Har07] einen guten Einstieg. Neben der Lektüre von Ryoo et al. [RRB+08] die als Einstieg in die parallele Optimierung dienen kann, lohnt sich auch die, der ausführlichen NVIDIA Leitfäden [NVI18a; NVI18c].

Als Einstieg in die iterativen Löser und explizit in die konjugierten Gradienten (CG) kann die Einführung die CG von Shewchuk et al. [She+94] dienen. Die Anwendung der CG, auf die SVM, wird von Wen, Edelma und Gorsich [WEG03] für die CPU beschrieben.

3. Support-Vector Machine

Die SVM [Vap00; VGS97] ist ein sogenannter Large Margin Classifier (engl. Breiter-Rand-Klassifikator), und gehört zu dem Teilgebiet des überwachten Lernens. Ein Large Margin Classifier ist ein Klassifikator, der für jede Eingabe den informativen Abstand zu der Entscheidungsgrenze angeben kann. Außerdem wird bei dem Lernprozess der Entscheidungsgrenze dieser Klassifikatoren der informative Abstand zwischen den Daten und der Entscheidungsgrenze maximiert, sodass ein möglichst breites Band um diese Entscheidungsgrenze frei von Daten besteht.

Eine SVM lernt, im Sinne der KI - also dem Wiedererkennen und Nachbilden von Gesetzmäßigkeiten - aus einer Menge von Trainingsdaten diejenige Hyperebene, welche die Trainingsdaten bestmöglich in ihre Klassen unterteilt. Eine Hyperebene ist eine Ebene, die nicht nur im dreidimensionalen Raum, sondern in einem Raum mit beliebig vielen Dimensionen definiert ist. Anhand des Gelernten kann die SVM bei neuen Daten bestimmen, auf welcher Seite der Hyperebene diese liegen, und damit welcher Klasse der Trainingsdaten die Daten ähnlicher sind.

3.0.1. Klassifikationsmodell

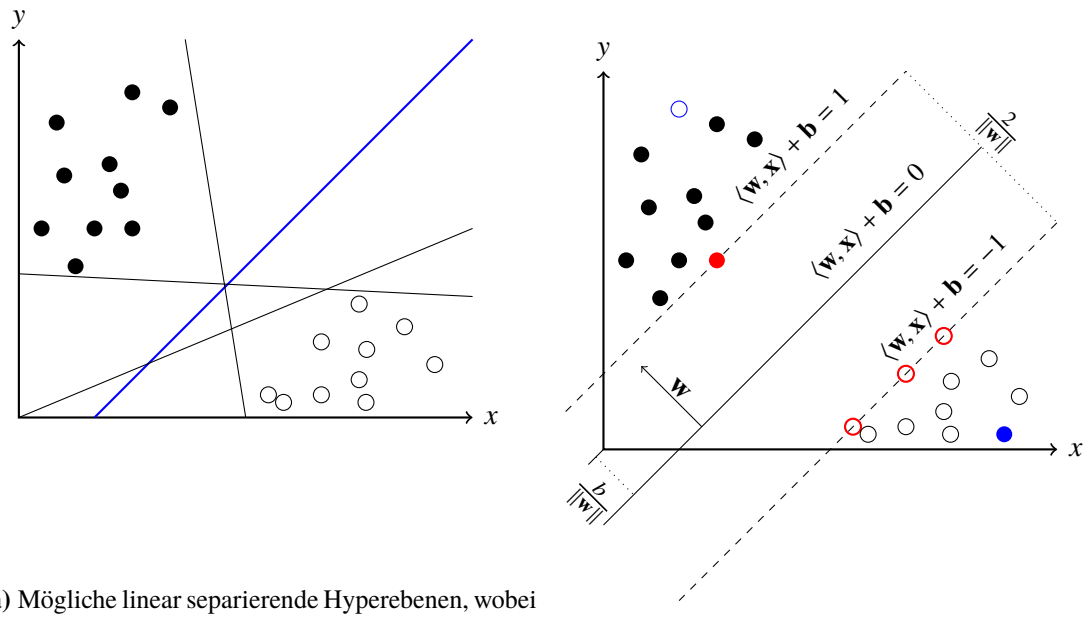
Damit die SVM Eingabedaten klassifizieren kann, muss sie erst ein Klassifikationsmodell erlernen. Dafür werden Trainingsdaten der Dimension d in Form von Vektoren $x_i \in \mathbb{R}^d$ benötigt, bei denen bekannt ist, zu welcher Klasse y_i diese gehören.

$$\text{Trainingsdaten} \coloneqq \{(x_i, y_i) | x_i \in \mathbb{R}^d, y_i \in 1, -1\}_{i=0}^{m} \tag{3.1}$$

Die SVM unterteilt immer in zwei Klassen, eine positive ($y_i = +1$) und eine negative ($y_i = -1$). Sofern die Trainingsdaten nicht so gelabelt sind, muss dies vor dem Lernen angepasst werden. Somit lässt sich die Menge aller $m \in \mathbb{Z}$ Trainingsdaten und ihre dazugehörigen Labels als Menge von Tupeln schreiben:

Das eigentliche Lernen besteht nun darin diejenige Hyperebene ($\langle \mathbf{w}, x_i \rangle + \mathbf{b} = 0$) zu finden, die die Trainingsdaten bestmöglich unterteilt. Dabei ist $\mathbf{w}$ der Normalenvektor der Hyperebene und es gilt $\mathbf{w} \in \mathbb{R}^d$. Mit dem Bias b wird die Ebene in die Richtung, in die der Normalenvektor $\mathbf{w}$ zeigt, verschoben. Die Hyperebenen ist eine wichtige Eigenschaft der SVM, denn eine SVM ist ein linearer Klassifizierer, das heißt die Trainingsdaten müssen linear separierbar sein.

Wie in dem zweidimensionalen Beispiel Abbildung 3.1a zu sehen ist, gibt es unter Umständen sehr viele verschiedene Möglichkeiten Mengen linear zu separieren. Die bestmögliche ist die Hyperebene, welche die Daten mit der breitesten Margin (engl. Band) teilt. Also im Beispiel Abbildung 3.1a die blaue Gerade. Wie in Abbildung 3.1b zu sehen ist, liegt diese genau mittig in dem breitest möglichen Band zwischen den innersten zwei (roten) Datenpunkten verschiedener Klassen. Also in unserem Beispiel mittig zwischen den Geraden $\langle \mathbf{w}, x_i \rangle + \mathbf{b} = 1$ und $\langle \mathbf{w}, x_i \rangle + \mathbf{b} = -1$. Bei der SVM wird der Bias so gewählt, dass er mit $\frac{\mathbf{b}}{\|\mathbf{w}\|}$ den Abstand der Hyperebene zum Ursprung angibt.



(a) Mögliche linear separierende Hyperebenen, wobei die blaue, die am besten separierende Hyperebene ist.

(b) Beste Hyperebene

Abbildung 3.1.: Zweidimensionales Beispiel zur Verdeutlichung der „besten“ Hyperebene

Der Normalenvektor wird so skaliert, dass die Bandbreite genau $\frac{2}{\|\mathbf{w}\|}$ ist. Dies führt dazu, dass alle Trainingsdatenpunkte auf dem Rand oder außerhalb der Margin $\langle \mathbf{w}, x_i \rangle + \mathbf{b} = \pm 1$ liegen müssen. Damit lässt sich folgende Bedingung aufstellen:

$$y_i \cdot (\langle \mathbf{w}, x_i \rangle + \mathbf{b}) \geq 1 \qquad \forall i \tag{3.2}$$

Die rot eingefärbten Datenpunkte in Abbildung 3.1b sind die Punkte, die zur eindeutigen Bestimmung der Hyperebene notwendig sind. Sie liegen direkt auf dem Rand $\langle \mathbf{w}, x_i \rangle + \mathbf{b} = \pm 1$ und heißen Support-Vektoren. Support-Vektoren sind diejenigen Daten, die ausschlaggebend für die Lage der Hyperebene sind. Das Ziel des Lernvorgangs ist es, diese zu erkennen, da anhand dieser, die Hyperebene eindeutig bestimmt ist.

3.0.2. Klassifizieren

Bei der Klassifikation eines Datenpunktes bestimmt die SVM auf welcher Seite, der zuvor bestimmten Hyperebene, dieser liegt. Daraus kann dann geschlossen werden, welcher Klasse der Datenpunkt ähnlicher ist, und damit gehört er mit einer höheren Wahrscheinlichkeit zu dieser.

Angenommen die korrekte Hyperebene ist nun aus Trainingsdaten erlernt, das heißt der Normalenvektor $\mathbf{w}$ und der Bias $\mathbf{b}$ sind bekannt. Um einen neuen Datenpunkt $\hat{x}$ zu klassifizieren, wird Gleichung (3.2) benutzt. Dazu wird $\hat{x}$ in die Formel $\langle \mathbf{w}, \hat{x} \rangle + \mathbf{b}$ eingesetzt und das Ergebnis bestimmt. Ist das Vorzeichen positiv, liegt der neue Datenpunkt $\hat{x}$ auf der Seite der Hyperebene, auf welcher beim Lernen die positive Klasse +1 lag. Ist das Ergebnis negativ, liegt der neue Datenpunkt

entsprechend auf der Seite, die beim Lernen der negativen Klasse -1 entsprach. Die Klasse $\hat{y}$ des Datenpunktes zu bestimmen, lässt sich damit mit folgender Gleichung ausdrücken:

$$\hat{y} = \text{sgn}(\langle \mathbf{w}, \hat{x} \rangle + \mathbf{b}) \tag{3.3}$$

3.0.3. Linear separierbare Daten

Die in Abschnitt 3.0.1 gestellten Anforderungen lassen sich mathematisch als eine Hyperebene beschreiben, welche die minimale quadratische Norm $\|\mathbf{w}\|_2^2$ besitzt, und gleichzeitig für alle Trainingsdaten die Gleichung (3.2) erfüllt. Die SVM lässt sich also als Minimierungsproblem auffassen:

$$\begin{gathered} \min_{\mathbf{w},\mathbf{b}} \left(\frac{1}{2} \|\mathbf{w}\|_2^2 \right), \\ \text{Nebenbedingung:} \quad y_i \cdot (\langle \mathbf{w}, x_i \rangle + \mathbf{b}) \geq 1 \qquad \forall i \end{gathered} \tag{3.4}$$

durch Anpassen von $\mathbf{w}$ und $\mathbf{b}$.

3.0.4. Linear unseparierbare Daten

Bei genauerer Betrachtung von Abbildung 3.1b stellt sich die Frage, was passiert, wenn die Trainingsdaten, beispielsweise durch ein Verrauschen der Daten, nicht mehr linear separierbar sind? In Abbildung 3.1b durch die blauen Punkte dargestellt. Dann ist die Gleichung (3.2) nicht mehr lösbar. Um auch solche Probleme lernen zu können, werden positive Schlupfvariable ξ_i eingeführt, durch die sich „Fehler“ in den Daten quantifizieren lassen.

$$y_i \cdot (\langle \mathbf{w}, x_i \rangle + \mathbf{b}) \geq 1 - \xi_i \qquad \forall i \tag{3.5}$$

Ziel ist es eine Hyperebene mit großer Bandbreite zu finden, bei der gleichzeitig die Fehler minimal sind. Die Schlupfvariable bietet jedem Trainingspaar x_i, y_i die Möglichkeit die Nebenbedingung (Gleichung (3.4)) um genau den Wert $\xi_i > 0$ zu verletzen. Die Summe der Verletzungen soll also minimal sein, weswegen der Gleichung (3.4) die Summe $\sum_{\forall i} \xi_i$ den Nebenbedingungen hinzugefügt wird. Um die Gewichtung der beiden Minimierungen einstellen zu können wird zusätzlich ein positiver konstanter Parameter C eingeführt, sodass sich die Minimierung 3.4 zu:

$$\min_{\mathbf{w},\mathbf{b},\xi} \left(\frac{1}{2} \|\mathbf{w}\|_2^2 + C \sum_{\forall i} \xi_i \right) \tag{3.6}$$

erweitern lässt. Die wieder aus Abschnitt 3.0.1 bekannte Nebenbedingung 3.5 wird um die Bedingung, dass alle Schlupfvariablen positiv sein müssen, ergänzt:

$$\begin{gathered} \min_{\mathbf{w},\mathbf{b},\xi} \left(\frac{1}{2} \|\mathbf{w}\|_2^2 + C \sum_{\forall i} \xi_i \right), \\ \text{Nebenbedingung:} \quad y_i \cdot (\langle \mathbf{w}, x_i \rangle + \mathbf{b}) \geq 1 - \xi_i \quad |\xi_i \geq 0 \qquad \forall i. \end{gathered} \tag{3.7}$$

3.1. Least Squares SVM (LS-SVM)

Suykens und Vandewalle [SV99a] haben das Problem in der abgewandelten Least Squares Form gelöst. Die Nebenbedingung in Gleichung (3.7) wird nicht mehr als Ungleichung, sondern als Gleichung aufgefasst:

$$\begin{aligned} &\min_{\mathbf{w},\mathbf{b},\xi} \left(\frac{1}{2}\|\mathbf{w}\|_2^2 + C \sum_{\forall i} \xi_i \right), \\ &\text{Nebenbedingung:} \quad y_i \cdot (\langle \mathbf{w}, x_i \rangle + \mathbf{b}) = 1 - \xi_i \quad |\xi_i \geq 0 \qquad \forall i. \end{aligned} \tag{3.8}$$

Anders als bei der klassischen SVM werden nun nicht mehr nur wenige Datenpunkte zu Support Vektoren, sondern die LS-SVM bestimmt für jeden Datenpunkt eine Gewichtung proportional zum Abstand zwischen Datenpunkt und Hyperebene. Die Gewichtung kann anders als bei der klassischen SVM nicht nur positiv, sondern auch negativ sein. Aus diesem Grund wird ξ_i quadriert, damit nur über positive Abstände addiert wird. Als Ausgleich wird die Gewichtung C halbiert.

$$\begin{aligned} &\min_{\mathbf{w},\mathbf{b},\xi} \left(\frac{1}{2}\|\mathbf{w}\|_2^2 + \frac{C}{2} \sum_{\forall i} \xi_i^2 \right), \\ &\text{Nebenbedingung:} \quad y_i \cdot (\langle \mathbf{w}, x_i \rangle + \mathbf{b}) = 1 - \xi_i \quad |\xi_i \geq 0 \qquad \forall i. \end{aligned} \tag{3.9}$$

Ein genauer Vergleich zwischen der SVM und der LS-SVM kann in [YX07] nachgelesen werden.

3.1.1. Duales Problem

Die von der SVM bestimmte Hyperebene ist abhängig von, und damit eindeutig bestimmt durch die Support-Vektoren (vgl. Abschnitt 3.0.1). Das heißt der Normalenvektor $\mathbf{w}$ lässt sich auch als Linearkombination dieser Support-Vektoren aufstellen:

$$\mathbf{w} = \sum_{i=1}^{m} \alpha_i y_i x_i \ |\alpha_i \in \mathbb{R} \tag{3.10}$$

Wobei α_i für alle x_i, die keine Support-Vektoren sind, gleich Null ist, und diese somit nicht gewichtet werden. Für die Support-Vektoren ist α_i ungleich Null. Mithilfe von den Lagrange-Multiplikatoren α_i und den Karush-Kuhn-Tucker (KKT) Bedingungen:

$$\begin{cases} \frac{\partial L}{\partial \mathbf{w}} = 0 & \rightarrow \quad \mathbf{w} = \sum_{i=1}^{m} \alpha_i x_i \\ \frac{\partial L}{\partial \mathbf{b}} = 0 & \rightarrow \quad \sum_{i=1}^{m} \alpha_i = 0 \\ \frac{\partial L}{\partial \xi_i} = 0 & \rightarrow \quad \alpha_i = C \cdot \xi_i \\ \frac{\partial L}{\partial \alpha_i} = 0 & \rightarrow \quad \xi_i = y_i - (\langle \mathbf{w}, x_i \rangle + b) \end{cases} \tag{3.11}$$

lässt sich nach Saunders, Gammerman und Vovk [SGV98] das Minimierungsproblem in seine duale Form umformulieren. Unter Elimination von ξ und $\mathbf{w}$ ergibt sich dann, nach Chu, Ong und Keerthi [COK05], folgendes Maximierungsproblem:

$$\begin{aligned} &\max_{\alpha} \left(\sum_{i=1}^{m} \alpha_{\mathbf{i}} - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j \langle x_i, x_j \rangle \right) \\ \text{Nebenbedingungen:} \quad &0 \leq \alpha_i \leq C \\ &\sum_{i=1}^{m} \alpha_i y_i = 0 \end{aligned} \tag{3.12}$$

3.1.2. Kernel-Trick

In der künstlichen Intelligenz (KI) wird oft die Linearität der Verfahren ausgenutzt und der sogenannte Kernel-Trick angewandt, um mit linearen Klassifiern auch nicht lineare Probleme klassifizieren zu können. Jede Abbildung $K : X \times X \rightarrow \mathbb{R}$ ist ein möglicher Kernel, sofern sie in einem Prähilbertraum $(F, \langle ., . \rangle)$ definiert ist, und in diesem ein Featuremapping (eine Abbildung) ϕ aus dem Eingaberaum $X \in \mathbb{R}^d$ in den Merkmalsraum $F \in \mathbb{R}^{\hat{d}}$ ($\phi : X \rightarrow F$) existiert (weiterführend [HSS08; Wer18]).

$$k(x_i, x_j) = \langle \phi(x_i), \phi(x_j) \rangle \tag{3.13}$$

Dabei ist ϕ eine Abbildung in eine höhere Dimension:

$$\phi : \mathbb{R}^{d_1} \rightarrow \mathbb{R}^{d_2}, y \rightarrow \phi(x); d_1 < d_2 \tag{3.14}$$

Dies führt, wie in Abbildung 3.2 dargestellt, dazu, dass auch nicht linear separierbare Daten sinnvoll klassifiziert werden können.

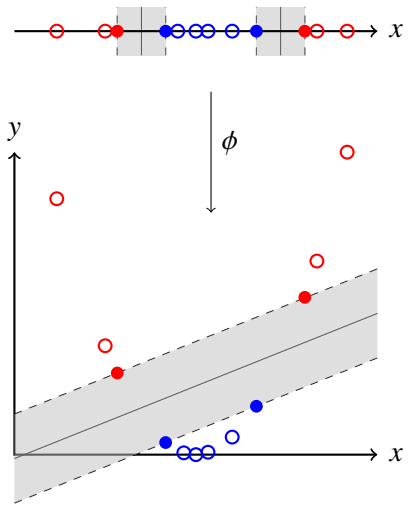


Abbildung 3.2.: Mit dem Kernel werden die Daten über die Funktion ϕ in eine höhere Dimension projiziert. Dies führt dazu, dass möglicherweise linear nicht separierbare Daten linear separierbar werden.

Die Gleichung (3.12) wird dabei folgendermaßen umformuliert. Das Skalarprodukt wird durch die Kernelfunktion ersetzt:

$$\begin{aligned} &\max_{\alpha} \left(\sum_{i=1}^{m} \alpha_{\mathbf{i}} - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j \cdot k(x_i, x_j) \right) \\ \text{Nebenbedingung:} \quad &0 \leq \alpha_i \leq C \\ &\sum_{i=1}^{m} \alpha_i y_i = 0 \end{aligned} \tag{3.15}$$

Auch die Klassifizierungsfunktion (Gleichung (3.3)) muss angepasst werden. Hier muss der Umformungsschritt mit den Lagrange-Multiplikatoren gemacht werden. Ebenso muss das Skalarprodukt durch die Kernelfunktion ersetzt werden.

$$f(x) = \operatorname{sgn}\left(k(x_i, x) + b\right) = \operatorname{sgn}\left(\sum_{i=1}^{m} \alpha_i y_i \cdot k(x_i, x) + b\right) = \operatorname{sgn}\left(\sum_{i=1}^{m} \alpha_i y_i \langle \phi(x_i), \phi(x) \rangle + b\right) \tag{3.16}$$

Es gibt verschiedene Kernel, die oft angewandt werden:

- linearer Kernel (Abschnitt 5.2.4) $k(x_i, x_j) = \langle x_i, x_j \rangle$
- polynomiale Kernel (Abschnitt 5.2.5) $k(x_i, x_j) = (\gamma \langle x_i, x_j \rangle + r)^d, \gamma > 0$
- radialer Basisfunktions Kernel (Abschnitt 5.2.6) $k(x_i, x_j) = \exp(-\gamma \|x_i - x_j\|^2), \gamma > 0$

3.1.3. Gleichungssystem

Nach Suykens und Vandewalle [SV99a] kann die Gleichung (3.15) unter Veränderung der Nebenbedingung (Gleichung (3.8)) in dieses lineare Gleichungssystem umgeformt werden:

$$\begin{bmatrix} Q & 1_m \\ 1_m^T & 0 \end{bmatrix} \cdot \begin{bmatrix} \alpha \\ b \end{bmatrix} = \begin{bmatrix} y \\ 0 \end{bmatrix} \tag{3.17}$$

Wobei $Q \in \mathbb{R}^{m \times m}$ ist und $Q_{ij} = K(x_i, x_j) + \frac{1}{C}\delta_{ij}$. Das Kronecker Delta$\delta_{ij}$ dient dazu, dass der Bruch $\frac{1}{C}$ nur auf die Diagonalelemente aufaddiert wird. Es ist überall, außer auf der Diagonalen, gleich null, das heißt $\delta_{ij} = \begin{cases} 0 & \text{wenn } i \neq j, \\ 1 & \text{wenn } i = j. \end{cases}$.
1_m ist ein Vektor mit m Einsen.

Damit würde die Anzahl der Einträge der Matrix $(\#Datenpunkte + 1)^2$ betragen. Um diese Matrix zu verkleinern haben Chu, Ong und Keerthi [COK05] sich mit diesem Gleichungssystem auseinandergesetzt. Die Matrix Q wird in vier Teile unterteilt: die untere Zeile (q^T) und die rechte Spalte q, der letzte Eintrag von der Matrix $Q_{m,m}$ und alles übriggebliebene $\bar{Q}$:

$$Q = \begin{bmatrix} \bar{Q} & q \\ q^T & Q_{m,m} \end{bmatrix} \tag{3.18}$$

Für die neue Matrix der linken Seite $\tilde{Q}$, wird nun von jedem Eintrag i, j der übrig bleibenden Matrix $\bar{Q}$ q, an der Stelle i und an der Stelle j, abgezogen. Der letzte Eintrag von $Q_{m,m}$ wird auf jeden Eintrag in $\bar{Q}$ aufaddiert. Damit ergibt sich:

$$\begin{aligned}
\tilde{Q} :&= \bar{Q} - 1_{m-1} \cdot q^T - q \cdot 1_{m-1}^T + Q_{m,m} \cdot 1_{m-1} \cdot 1_{m-1}^T = \\
&= \bar{Q} - \begin{pmatrix} q_1^T & \cdots & q_{m-1}^T \\ \vdots & \vdots & \vdots \\ q_1^T & \cdots & q_{m-1}^T \end{pmatrix} - \begin{pmatrix} q_1 & \cdots & q_1 \\ \vdots & \cdots & \vdots \\ q_{m-1} & \cdots & q_{m-1} \end{pmatrix} + \begin{pmatrix} Q_{m,m} & \cdots & Q_{m,m} \\ \vdots & \ddots & \vdots \\ Q_{m,m} & \cdots & Q_{m,m} \end{pmatrix} = \\
&= \begin{pmatrix} Q_{1,1} & \cdots & Q_{1,m-1} \\ \vdots & \ddots & \vdots \\ Q_{1,m-1} & \cdots & Q_{m-1,m-1} \end{pmatrix} - \begin{pmatrix} Q_{1,m} & \cdots & Q_{1,m-1} \\ \vdots & \vdots & \vdots \\ Q_{1,m} & \cdots & Q_{1,m-1} \end{pmatrix} - \begin{pmatrix} Q_{m,1} & \cdots & Q_{m,1} \\ \vdots & \cdots & \vdots \\ Q_{m,m-1} & \cdots & Q_{m,m-1} \end{pmatrix} + \\
&+ \begin{pmatrix} Q_{m,m} & \cdots & Q_{m,m} \\ \vdots & \ddots & \vdots \\ Q_{m,m} & \cdots & Q_{m,m} \end{pmatrix}.
\end{aligned} \tag{3.19}$$

Mit dieser Matrix $\tilde{Q}$ lässt sich jetzt die Gleichung (3.17) in einem kleineren Gleichungssystem darstellen [COK05]:

$$\begin{aligned}
\tilde{Q} \cdot \tilde{\alpha} &= \bar{y} - y_m \cdot 1_{m-1} \\
\tilde{Q} \cdot \tilde{\alpha} &= \begin{pmatrix} y_1 - y_m \\ y_2 - y_m \\ \vdots \\ y_{m-1} - y_m \end{pmatrix}
\end{aligned} \tag{3.20}$$

Wobei $\tilde{\alpha}$ gesucht wird und $\bar{y}$ definiert ist, als die ersten $m-1$ Stellen von y.

$$\bar{y} \coloneqq (y_1, y_2, \ldots, y_{d-1})^T | \bar{y} \in \mathbb{R}^{d-1}, y \in \mathbb{R}^d \tag{3.21}$$

Aus diesem deutlich kleineren Gleichungssystem 3.20 (die Anzahl der zu bestimmenden α_i hat sich mindestens halbiert [COK05]), lässt sich auch der Bias b wieder ableiten [COK05]:

$$b = y_m + Q_{m,m} \cdot (1_{m-1}^T \cdot \tilde{\alpha}) - q^T \cdot \tilde{\alpha} = y_m + Q_{m,m} \cdot \sum_{i=1}^{m-1} (\tilde{\alpha}_i) - \sum_{i=1}^{m-1} \left(q_i^T \cdot \tilde{\alpha}_i \right), \tag{3.22}$$

und

$$\mathbf{w} = \sum_{i=1}^{m} \alpha_i \phi(x_i). \tag{3.23}$$

4. CUDA

CUDA ist eine NVIDIA Architektur für parallele Berechnungen, die die Rechenleistung des Systems durch Nutzung der Leistung des Grafikprozessors deutlich steigern kann. In diesem Kapitel wird zu Beginn der grundlegende Aufbau dieser Architektur und der damit verbundenen Programmiersprache erläutert. Außerdem werden für die Erklärung der Implementierung Kapitel 5 die grundlegenden CUDA-Befehle genannt und kurz erklärt. Für weitere Erklärungen empfiehlt sich hier der CUDA Programming Guide[NVI18a] und die NVIDIA CUDA Library[Lib13]. Die Erklärungen in diesem Kapitel basieren auf dem CUDA C Programming Guide[NVI18a].

4.1. CUDA Konzept

Um die maximale Performance mit einer CUDA-Applikation erreichen zu können, ist es wichtig, die Funktionsweise und den Aufbau, einer GPU zu verstehen (Abbildung 4.1). Erst damit ist es möglich, die Multiprozessoren der GPU, und damit deren Leistungsfähigkeit, optimal auszunutzen.

4.1.1. CUDA Programmiermodell

Das CUDA Programmiermodell basiert, wie in der logischen Sicht von Abbildung 4.1 zu sehen ist, auf einer hierarchischen Struktur. Ein CUDA-Kernel Aufruf besteht aus mehreren CUDA-Blöcken und diese Blöcke wiederum bestehen aus CUDA-Threads. Diese Struktur ist insofern interessant, da Threads und Blöcke auf unterschiedliche und teilweise verschiedene Speicher der GPU zugreifen können.

Die Blöcke bestehen also aus mehreren `Threads` und mehrere `Blöcke` zusammen ergeben ein `Grid`. Alle `Threads` eines `Grids` führen immer denselben `Kernel` aus. Für die Grid-Dimension können genau wie für die Block-Dimension dreidimensionale (3D) Werte definiert werden. Dies kann von Vorteil sein, wenn mit mehrdimensionalen Datenstrukturen gearbeitet wird. Damit kann direkt auf diese mehrdimensionalen Strukturen zugegriffen werden, ohne dass eine Umrechnung auf eindimensionale Indizes notwendig ist.

```
dim3 dimBlock(4,2,1);
dim3 dimGrid(2,2,1);
MULT<<<dimGrid,dimBlock>>>(...);
```

Es stehen unterschiedliche Systemvariablen ("Built-in variables") zur Verfügung. Sie dienen als Index, und damit ist es Threads und Blöcken möglich sich zur Laufzeit selber zu identifizieren, da sie sich damit voneinander zu unterscheiden.

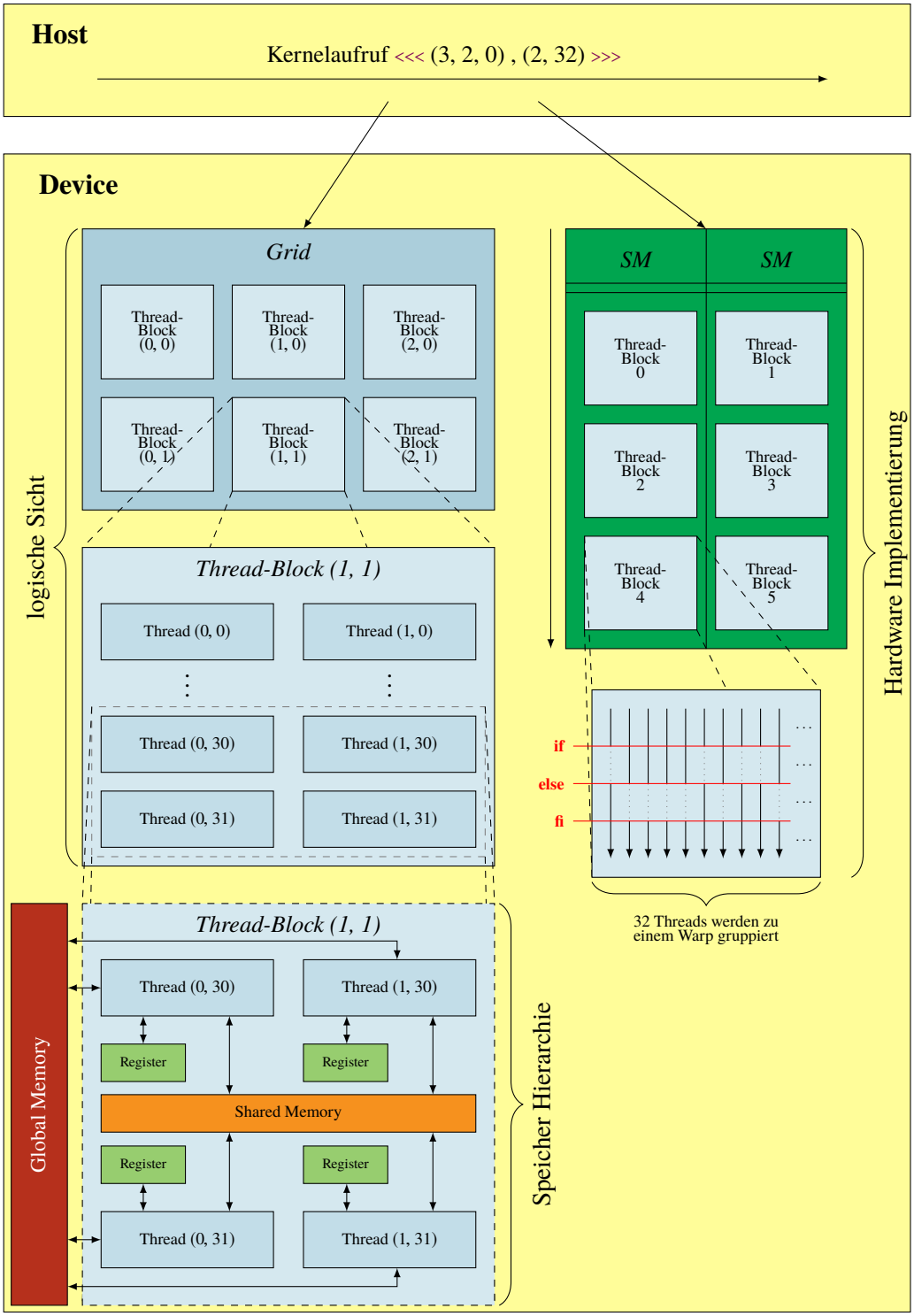


Abbildung 4.1.: CUDA Device Aufrufe werden in Grids (bestehend aus Blöcken von Threads) abgearbeitet. Threads können auf unterschiedliche Speicher zugreifen. Jeder Thread Block wird einem Streaming Multiprozessor (SM) zugewiesen, und innerhalb eines Blockes parallel abgearbeitet. Branch-Divergence wird über Masked-Execution aufgelöst.
(Erstellt in Zusammenarbeit mit Marcel Breyer)

```
blockIdx.x
blockIdx.y
blockIdx.z
threadIdx.x
```

Zu beachten ist, dass die Thread Indizes nur pro Block eindeutig sind, das heißt ein Thread ist über das Tupel von Block Index und Thread Index eindeutig identifiziert.

4.1.2. Kernels

CUDA basiert auf dem Konzept von Kernels. Ein Kernel beschreibt dabei eine Funktion, die auf mehreren Blöcken und Threads ausgeführt wird. Ein Kernel-Aufruf kann mit einem normalen Funktions-Aufruf verglichen werden, dabei wird der Kernel bei seinem Aufruf parallel auf der GPU ausgeführt.

```
kernel<<<dim3 dimGrid,dim3 dimBlock, size_t memory, cudaStream_t stream>>>(Funktion paramList);
```

Die CUDA-Kernelaufrufe unterscheiden sich in der Syntax von normalen Funktionsaufrufen darin, dass nach dem Funktionsnamen in drei Spitzen klammern „<<<>>>“ die CUDA Parameter stehen. Darin eingeschlossen ist die Grid Dimension `dimGrid` und die Block Dimension. Die Dimension kann dabei wahlweise direkt als integer angegeben werden (1D) oder mehrdimensional als `dim3`. Außerdem muss, sofern verwendet, mit `memory` die pro Block benötigte Shared Memory Größe angegeben werden. Optional lässt sich mit `stream` der CUDA-Stream angeben.

```
MULT<<<4,1024>>>(a,x,b);
```

In diesem Beispiel wird ein CUDA-Kernel „`MULT`“ aufgerufen. Er wird von vier CUDA Blöcken mit jeweils 1024 Threads parallel ausgeführt. Das heißt 4096 Threads führen den Kernel gleichzeitig auf der GPU aus.

Im Vergleich zu normalen Thread-Aufrufen auf der CPU sind diese CUDA-Threads viel leichtgewichtiger, können schneller erzeugt, und vom Scheduler zugeteilt, werden.

4.1.3. Streaming Multiprozessoren

Die CUDA-Architektur basiert auf einem Array wobei jede Array Stelle ein Streaming Multiprozessor (SM) ist. Wenn ein Kernel-Grid aufgerufen wird, dann werden die Blöcke dieses Grids in eine Warteschlange gesetzt. Wird ein SM frei, dann wird der nächste freie Thread-Block aus der Warteschlange auf diesem gestartet.

Die SM basieren auf der SIMT (Single-Instruction, Multiple-Thread) Architektur. In dieser werden aufeinanderfolgende Threads eines Blocks logisch in Gruppen zu je 32 Threads, so genannten Warps, zusammengefasst. Ist ein Warp durch einen Block nicht ausgelastet, können hier auch mehrere Blöcke zusammengefasst werden. Alle Threads in einem Warp starten an der gleichen Stelle und führen die Instruktionen auf allen 32 Threads parallel aus. Dabei führt jeder Thread diese Instruktionen auf seinem eigenen Speicherbereich aus. Bei Verzweigung innerhalb des Programmcodes kann es zu einer Verzweigungsdivergenz (branch divergence) kommen, also dazu, dass die Threads eines Warps unterschiedliche Programmpfade durchlaufen müssen. In diesem Fall führt der Warp

die Instruktionen der unterschiedlichen Verzweigungen nacheinander aus und deaktiviert dabei alle Threads, die nicht auf dem aktuell betrachteten Pfad liegen (masked execution). Der Bereich Hardware Implementierung in Abbildung 4.1 verdeutlicht dies nochmals grafisch.

4.1.4. CUDA-Speichermodell

Die Hardware der GPU stellt verschiedene Speicher zur Verfügung. Die verschiedenen Speicher unterscheiden sich nicht nur in ihrer Zugehörigkeit, sondern auch in ihrer Zugriffsgeschwindigkeit:

Register On-Chip Register (sehr performant)

Shared Memory On-Chip Register (performant)

Global Memory off-chip Memory (langsam)

Local Memory Ein Thread privater Bereich im Global Memory (langsam)

Constant Memory Cached off-chip Memory (performant)

Die verschiedenen Speicher haben, wie in dem Bereich der Speicherhierarchie von Abbildung 4.1 dargestellt, nicht nur unterschiedliche Zugriffsbereiche, sondern müssen im Kernel auch unterschiedlich initialisiert werden:

- Lesen/Schreiben in Register Zugriff pro Thread: `int var;`
- Lesen/Schreiben in Local Memory Zugriff pro Thread: `int var[100];`
- Lesen/Schreiben in Shared Memory Zugriff pro Block: `__shared__ int s_var;`
- Lesen/Schreiben in Global Memory Zugriff pro Grid: `__device__ int g_var;`
- Lesen in Konstant Memory Zugriff pro Grid: `__constant__ int c_var;`

Außerdem steht ab CUDA 4.0, mit dem Unifed Memory, ein geteilter Adressraum zwischen Hauptspeicher und Global Memory zur Verfügung. Außerhalb des Unifed Memory muss der Programmierer dafür sorgen, dass die Daten zwischen Grafikkarte und Hauptspeicher hin und her kopiert werden.

4.1.5. Implementierungsziele

Zusammenfassend lassen sich von der Hardware einige Punkte ableiten, die zu beachten sind, um die GPU möglichst optimal auszunutzen.

- Mit möglichst vielen Threads gleichzeitig auslasten. Dies kaschiert Latenzen.
- Divergente Branches minimieren.
- Abhängigkeiten zwischen Threads minimieren, da diese ein Synchronisationsoverhead mit sich bringen.
- Unterschiedlichen Speicher richtig nutzen.
- Speicherzugriffe optimieren.

4.2. CUDA Speichermanagement

In CUDA müssen Speicherbereiche ähnlich wie in C manuell allokiert (reserviert) und zu gegebenem Zeitpunkt auch wieder freigegeben werden. Neben den Befehlen dafür stellt CUDA auch Befehle zur Kommunikation zwischen dem Hauptspeicher und dem globalen Grafikkarten Speicher bereit.

cudaMalloc

```
cudaError_t cudaMalloc(void** devPtr, size_t size)
```

Die `cudaMalloc` Funktion allokiert einen linearen Speicherbereich auf dem Device (der Grafikkarte). Der allokierte Speicherbereich hat die `size` in Byte und der Device-Pointer `devPtr` zeigt danach auf die erste Stelle des allokierten Speicherbereichs. Tritt dabei ein Fehler auf, so wird ein entsprechender Fehlercode vom Typ `cudaError_t` zurückgegeben.

cudaFree

```
cudaError_t cudaFree(void* devPtr)
```

Mithilfe der `cudaFree`-Funktion wird der, zuvor mit `cudaMalloc` allokierte, Speicherbereich auf den der Device-Pointer `devPtr` zeigt, freigegeben. Tritt dabei ein Fehler auf, so wird ein entsprechender Fehlercode vom Typ `cudaError_t` zurückgegeben.

cudaMemset

```
cudaError_t cudaMemset(void* devPtr, int value, size_t count)
```

`cudaMemset` setzt die ersten `count` Bytes, beginnend mit dem Byte, auf welches der Device-Pointer `devPtr`zeigt, auf die Binärrepräsentation von `value`. Hierbei muss beachtet werden, dass cudaMemset die Werte byteweise setzt. Das kann z. B. dazu führen, dass bei `value` Werten ungleich Null, auf den ersten Blick, unerwartete Ergebnisse entstehen. Beispiel: Bei einem Integers (mit 32 bit Länge) ergibt `cudaMemset(devPtr, 2, 4)` nicht wie möglicherweise falsch angenommen eine 2_{10} für `*device ptr`, sondern viel mehr $00000010\ 00000010\ 00000010\ 00000010_2 = 33686018_{10}$.

Tritt dabei ein Fehler auf, so wird ein entsprechender Fehlercode vom Typ `cudaError_t` zurückgegeben.

4.2.1. cudaMemcpy

```
cudaError_t cudaMemcpy(void * dst, const void * src, size_t count, enum cudaMemcpyKind kind)
```

`cudaMemcpy` kopiert `count` Bytes, ab dem Byte auf das `src` zeigt, bis zu dem Speicherbereich, auf den der Pointer `dst` zeigt. Mit `kind` wird angegeben von und zu welchem Gerät kopiert wird. Dabei ist folgende Konstellationen möglich: von dem Hauptspeicher in den Hauptspeicher mit `cudaMemcpyHostToHost`, von dem Hauptspeicher in den globalen Grafikspeicher mit `cudaMemcpyHostToDevice`, von dem globalen Grafikkartenspeicher in den Hauptspeicher mit `cudaMemcpyDeviceToHost`, und innerhalb des globalen Grafikkartenspeichers mit `cudaMemcpyDeviceToDevice` möglich. Tritt dabei, oder bei einem asynchronen `cudaMemcopy` davor, ein Fehler auf, so wird ein entsprechender Fehlercode vom Typ `cudaError\_t` zurückgegeben.

4.3. CUDA Schlüsselworte

In CUDA gibt es einige für das Verständnis der Implementierung grundlegende Schlüsselworte:

__syncthreads()

Dieses Schlüsselwort dient als Barriere innerhalb eines Blockes. Damit werden alle Threads eines Blockes synchronisiert. Erst wenn alle Threads in einem Block mit der Ausführung bis zum Syncthreads Aufruf gekommen sind, darf die Barriere überschritten werden.

__shared__

Shared ist ein Schlüsselwort, welches vor Shared Memory Allokationen steht. Damit wird innerhalb der CUDA Kernel festgelegt, dass der Speicherbereich nicht von jedem Thread angelegt wird, sondern der Speicherbereich wird pro Block nur einmal angelegt. Alle Threads dieses Blockes haben darauf Zugriff. Die Operationen auf dem geteilten Speicher sind nicht atomar.

__global__

Global dient dazu Funktionen zu kennzeichnen, die sowohl vom Device, als auch vom Host aus, aufgerufen werden können. Es muss vor der Funktionsdefinition stehen.

__device__

Device dient im Vergleich zu `__global__` dazu, Funktionen zu kennzeichnen, die ausschließlich von dem Device aus aufgerufen werden können. Dieser Befehl muss vor der Funktionsdefinition stehen. Außerdem werden damit im Kernel Variablen im Global Memory der Grafikkarte definiert.

#pragma unroll

Dieses Pragma dient dazu den Compiler anzuweisen, die nachfolgende Schleife um einen gesetzten Wert zu entrollen.

4.4. atomicAdd

```
double atomicAdd(double* address, double val)
```

Mit `atomicAdd` wird auf eine Speicheradresse `adress` im Shared Memory atomar der Wert von `value` aufaddiert. Für 64-bit Datentypen steht `atomicAdd` erst seit CUDA compute capability 6.0 zur Verfügung. Davor musste es selber implementiert werden. Diese Implementierung ist laufzeittechnisch aber deutlich schlechter.

5. Implementierung

In diesem Kapitel geht es um die Implementierung des LS-SVM Algorithmus 5.1 sowie um die Implementierung für einen Klassifizierer nach dem LIBSVM Modell Algorithmus 5.6. Dabei wird das eigentliche LGS-Lösen diskutiert und optimiert. Es wird genauer auf die Dateiformate, sowie auf das Einlesen an sich, eingegangen. Außerdem wird die Klassifikation, erklärt.

Algorithmus 5.1 Der LS-SVM Algorithmus für die GPU

procedure LS-SVM-TRAIN(Datendatei, ϵ, i_{max}, kernel)
  $data \leftarrow$ EINLESEN($Datendatei$)
  $\tilde{y} \leftarrow$ RECHTE SEITE BESTIMMEN($data$)
  AUF GPU LADEN($data, \tilde{y}$)
  $\tilde{\alpha} \leftarrow$ LGS LÖSE($data, \tilde{y}, \epsilon, i_{max}, kernel$)
  $\alpha \leftarrow \tilde{\alpha}.append - \sum \tilde{\alpha}$
  SPEICHERN($\alpha, kernel, data$)
end procedure

5.1. Interne Datenrepräsentation (SoA)

Intern werden alle Daten in einem großen Vektor gehalten. Der Vektor wird aber nicht durch Aneinanderreihung der einzelnen Punkte erstellt (Array of Structs (AoS)), sondern wird dimensionsweise aufgebaut. Dies bedeutet, dass dabei zuerst von allen Daten das erste Feature abgespeichert ist. Danach wird in der gleichen Reihenfolge das folgende Feature aller Daten abgespeichert und so weiter(vergleiche Abbildung 5.1). Da die Daten im Programmverlauf dimensionsweise benötigt werden, hat diese nicht ganz intuitive Speicherstruktur (Structure of Arrays (SoA)) den Vorteil, dass sie deutlich effizienter ist.

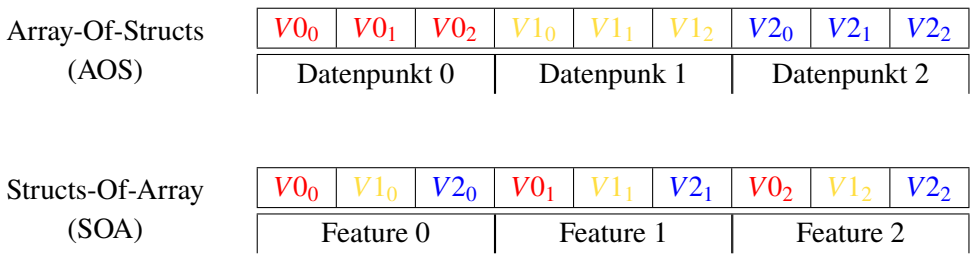


Abbildung 5.1.: Array-Of-Structs vs. Structs-Of-Array

CUDA unterstützt die C++ Standard Template Library (STL) nicht. Deshalb müssen die Vektoren als C++ Arrays auf die GPU geladen werden, um dort damit rechnen zu können. Da alle Elemente in einem Vektor hintereinanderliegen, stellt dies kein Problem dar. Der Vektor kann, um auf die GPU geladen zu werden, gleich behandelt werden wie ein Array, und somit wie ein Array auf die GPU geladen werden. Ein Zeiger auf das erste Element im Array dient als Start. Beginnend mit diesem wird ein Speicherbereich der Größe des Vektors/Arrays entsprechend auf die Grafikkarte kopiert.

5.2. LGS lösen

Zur Lösung des linearen Gleichungssystems $\tilde{Q} \cdot \tilde{\alpha} = \bar{y} - y_m \cdot 1_{m-1}$ (Gleichung (3.20)) wird das Verfahren der konjugierten Gradienten (CG) benutzt. Dies ist möglich, da die Matrix Q,wie in [COK05] beschrieben, symmetrisch und positiv definit ist. Implementiert wird die Variante von Shewchuk et al. [She+94] (Algorithmus 5.2).

Algorithmus 5.2 CG Algorithmus

```
1:  procedure CG(Vektor b,Matrix Q̃, int i_max,)
2:      d ← 1
3:      r ← b − Q̃d
4:      d ← r
5:      δ ← r^T r
6:      δ_start ← δ
7:      for i ← 0; i < i_max; ++i do
8:          q ← Q̃d
9:          x ← δ / (d^T q)
10:         α̃ ← α̃ + α̃d
11:         r ← b − Q̃d
12:         δ_alt ← δ
13:         δ ← r^T r
14:         Test auf Abbruch
15:         d ← r + (δ / δ_alt) d
16:     end for
17:     return α̃
18: end procedure
```

Zur Bestimmung des Residuums, in Zeile 3 und 11, sowie in Zeile 8, werden Matrix-Vektor Produkte gelöst. In den Zeilen 5, 9 und 13 werden Skalarprodukte bestimmt.

Da $\tilde{Q}$ $(m-1)^2 = (\#Datenpunkte - 1)^2$ viele Einträge hat, ist es für große Trainingsdaten nicht möglich die Matrix weder im Hauptspeicher noch auf der Grafikkarte komplett zu speichern. Aus diesem Grund wird die Matrix implizit dargestellt. Das heißt jeder Eintrag $\tilde{Q}_{i,j}$ mit Gleichung (5.1) wird für jede Verwendung nach Gleichung (3.19) neu berechnet.

$$\begin{aligned} i \neq j : \tilde{Q}_{i,j} &= \mathrm{k}(x_i, x_j) \qquad -\mathrm{k}(x_m, x_j) - \mathrm{k}(x_i, x_m) + \mathrm{k}(x_m, x_m) + \frac{1}{C} \\ i = j : \tilde{Q}_{i,i} &= \mathrm{k}(x_i, x_i) + \frac{1}{C} -\mathrm{k}(x_m, x_i) - \mathrm{k}(x_i, x_m) + \mathrm{k}(x_m, x_m) + \frac{1}{C} \end{aligned} \tag{5.1}$$

Algorithmus 5.3 Matrix Vektor Produkt $r = \tilde{Q} \cdot d$

for $i \leftarrow 0;\ i < m-1;++i$ **do**
  for $j \leftarrow 0;\ j < m-1;++j$ **do**
    if $i == j$ **then** $r[i] \leftarrow r[i] + (\mathrm{k}(x_i, x_i) + \frac{1}{C} - \mathrm{k}(x_m, x_i) - \mathrm{k}(x_i, x_m) + \mathrm{k}(x_m, x_m) + \frac{1}{C}) \cdot d[i]$
    else $r[i] \leftarrow r[i] + (\mathrm{k}(x_i, x_j) - \mathrm{k}(x_m, x_j) - \mathrm{k}(x_i, x_m) + \mathrm{k}(x_m, x_m) + \frac{1}{C}) \cdot d[i]$
    end if
  end for
end for

Algorithmus 5.4 Residuum $r = b - \tilde{Q}d$

$r \leftarrow b$
for $i \leftarrow 0;\ i < m-1;++i$ **do**
  for $j \leftarrow 0;\ j < m-1;++j$ **do**
    if $i == j$ **then** $r[i] \leftarrow r[i] - (\mathrm{k}(x_i, x_i) + \frac{1}{C} - \mathrm{k}(x_m, x_i) - \mathrm{k}(x_i, x_m) + \mathrm{k}(x_m, x_m) + \frac{1}{C}) \cdot d[i]$
    else $r[i] \leftarrow r[i] - (\mathrm{k}(x_i, x_j) - \mathrm{k}(x_m, x_j) - \mathrm{k}(x_i, x_m) + \mathrm{k}(x_m, x_m) + \frac{1}{C}) \cdot d[i]$
    end if
  end for
end for

In der Implementierung lassen sich die Matrix-Vektor Produktberechnung (Algorithmus 5.3) und die Residuumsberechnung (Algorithmus 5.4) in die gleiche Funktion auslagern. Nach der Zuweisung von `r = b` unterscheiden sich Algorithmus 5.3 und Algorithmus 5.4 nur noch darin, dass in Algorithmus 5.3 in jedem Schritt auf $r[i]$ aufaddiert wird, und bei der Residuumsberechnung in Algorithmus 5.4 von $r[i]$ subtrahiert wird. Durch das Ausklammern des Vorzeichens mit $\mathrm{sgn} := \pm 1$ können beide in Algorithmus 5.5 zusammengefasst werden.

Algorithmus 5.5 $r = (b-)\tilde{Q} \cdot d$

1: **procedure** PROD(in x,in C, in m, in sgn, inout r)
2: **for** $i \leftarrow 0;\ i < m-1;++i$ **do**
3: **for** $j \leftarrow 0;\ j < m-1;++j$ **do**
4: **if** $i == j$ **then** $r[i] \leftarrow r[i] + (\mathrm{k}(x_i, x_i) + \frac{1}{C} - \mathrm{k}(x_m, x_i) - \mathrm{k}(x_i, x_m) + \mathrm{k}(x_m, x_m) + \frac{1}{C}) \cdot \mathrm{sgn} \cdot d[i]$
5: **else** $r[i] \leftarrow r[i] + (\mathrm{k}(x_i, x_j) - \mathrm{k}(x_m, x_j) - \mathrm{k}(x_i, x_m) + \mathrm{k}(x_m, x_m) + \frac{1}{C}) \cdot \mathrm{sgn} \cdot d[i]$
6: **end if**
7: **end for**
8: **end for**
9: **end procedure**

5.2.1. Blocking

Die Matrix-Vektor Produktberechnung ist die aufwändigste Berechnung des Algorithmus 5.2. Aus diesem Grund wird diese zusammen mit den Vorbedingungen Algorithmus 5.5 auf der Grafikkarte berechnet. Dafür wird das Matrix-Vektor Produkt in einzelne Blöcke unterteilt, die parallel berechnet werden. In Gleichung (5.2) beispielsweise mit Blockgröße 2.

$$\begin{pmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \end{pmatrix} \pm = \left(\begin{array}{cc|cc|c} x & x & x & x & x \\ x & x & x & x & x \\ \hline x & x & x & x & x \\ x & x & x & x & x \\ \hline x & x & x & x & x \end{array}\right) \cdot \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \end{pmatrix} \tag{5.2}$$

Damit möglichst wenig Randbedingungen überprüft werden müssen, wird immer auf die volle Blockgröße erweitert (padding):

$$\begin{pmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ x \end{pmatrix} \pm = \left(\begin{array}{cc|cc|cc} x & x & x & x & x & 0 \\ x & x & x & x & x & 0 \\ \hline x & x & x & x & x & 0 \\ x & x & x & x & x & 0 \\ \hline x & x & x & x & x & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right) \cdot \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ 0 \end{pmatrix} \tag{5.3}$$

Damit lässt sich jeder Block bei beliebiger Datengröße gleich behandeln. Nachdem die Matrix nach [SV99a] symmetrisch ist, kann der Berechnungsaufwand halbiert werden, indem nur das obere rechte / untere linke Dreieck berechnet wird. Die fehlenden Einträge können dann auf die andere Seite der Diagonale projiziert werden.

$$\left(\begin{array}{cc|cc|cc} x & x & x & x & x & 0 \\ & x & x & x & x & 0 \\ \hline & & x & x & x & 0 \\ & & & x & x & 0 \\ \hline & & & & x & 0 \\ & & & & & 0 \end{array}\right)$$

Da CUDA mit Blöcken von Threads arbeitet, bietet es sich an, jeden Matrixeintrag parallel durch einen Thread berechnen zu lassen. Ein CUDA Block darf, bei CUDA Compute Capability 6.0, maximal 1024 Threads beinhalten. Dies führt zu einer maximalen Blockgröße von 32x32 Threads (`blocksize`). Damit ergibt sich ein Gitter (`grid`) aus jeweils $\left\lfloor \frac{\text{Anzahl der Eingabedaten}}{\text{Blockgröße}} \right\rfloor + 1 = \left\lfloor \frac{\texttt{Ndatas\_data}}{\texttt{blocksize}} \right\rfloor + 1$ vielen Spalten und Zeilen von Thread Blöcken der jeweiligen Blockgröße. Dies wird, wie in Listing 5.1 gezeigt, in CUDA realisiert.

Zur Vereinfachung wird ab hier erstmal alles nur anhand des linearen SVM Kernel behandelt. Es werden Threads für die ganze Matrix erzeugt und jeder ermittelt seine Position in der Matrix (Listing 5.2 Zeile 2 und Zeile 3). Nur die Threads oberhalb der Diagonalen werden für die Berechnung genutzt (Listing 5.2 Zeile 5). Dieses Vorgehen ist möglich, da die reine Thread Erstellung auf der

Listing 5.1 Kernelaufruf ohne Shared Memory

```
1  // Block groesse
2  unsigned bloksize = 32;
3
4  //Matrix in Gitter aufteilen
5  dim3 grid((int)Ndatas_data/bloksize + 1,(int)Ndatas_data/bloksize + 1);
6  dim3 block(bloksize,bloksize);
7
8  //gekuertzter Aufruf
9  cuda_kernel<<<grid,block>>>(...);
```

Listing 5.2 Linearer CUDA Kernel

```
1  __global__ void kernel_f(...){
2    int index_i = blockIdx.x * blockDim.x + threadIdx.x;
3    int index_j = blockIdx.y * blockDim.y + threadIdx.y;
4
5    if(index_i >= index_j){
6        //Q̃_{i,j}
7        double Q = kernel_linear(data_d, data_d, index_i, index_j, Nfeatures_data, Ndatas_data ,
             Ndatas_data)
8          - kernel_linear(data_d, data_d, index_i, index_m, Nfeatures_data, Ndatas_data ,
               Ndatas_data)
9          - kernel_linear(data_d, data_d, index_j, index_m, Nfeatures_data, Ndatas_data ,
               Ndatas_data)
10         + kernel_linear(data_d, data_d, index_m, index_m, Nfeatures_data, Ndatas_data ,
               Ndatas_data) +1/C;
11       if(index_i == index_j){
12         atomicAdd(&ret[ii + i], (Q + 1/C)* d[ii + i] * sgn);
13       }else{
14         atomicAdd(&ret[jj + j], Q * d[ii + i] * sgn);
15         atomicAdd(&ret[ii + i], Q * d[jj + j] * sgn);
16       }
17     }
18   }
19 }
```

Grafikkarte sehr leichtgewichtig ist. Liegt er oberhalb der Diagonalen, wird das entsprechende Q berechnet.

$$\begin{array}{rccccccc} \tilde{Q} = & k(x_i, x_j) & - & k(x_m, x_j) & - & k(x_i, x_m) & + & k(x_m, x_m) + \dfrac{1}{C} \\ \textit{Listing } 5.2: & \text{Zeile 7} & & \text{Zeile 8} & & \text{Zeile 9} & & \text{Zeile 10} \end{array} \tag{5.4}$$

Sofern der jeweilige Thread einen Matrixeintrag auf der Diagonalen berechnet, muss noch einmal $\frac{1}{C}$ auf den Eintrag aufaddiert werden, ansonsten kann er direkt mit dem entsprechenden Eintrag des Vektors d multipliziert werden. Die Variable `sgn` bestimmt, ob das Produkt auf den ursprünglichen Vektor r aufaddiert oder davon subtrahiert wird. Wenn `sgn` gleich plus Eins gewählt wird, wird der berechnete Matrix-Vektor Produkt Eintrag auf r aufaddiert, bei minus Eins wird dieser subtrahiert.

Wichtig hierbei ist, dass die eigentliche Addition auf r atomar geschehen muss. Seit CUDA Compute Capability 6.0 wird `atomicAdd` auch für Werte doppelter Genauigkeit bereitgestellt [NVI18c]. Davor musste diese Atomizität durch einen sehr zeitintensiven Code erkauft werden (Abschnitt 4.4). Die Atomizität ist notwendig, da unter Umständen sehr viele Threads gleichzeitig versuchen auf einen Eintrag etwas auf zu addieren. Wenn dies nicht atomar geschieht, können beispielsweise zwei Threads gleichzeitig den Wert $d[i]$ lesen. Danach führen beide auf den gleichen Wert ihre Addition aus und schreiben das Ergebnis zurück in $d[i]$. Dabei entsteht eine Race Condition, das heißt jede Ausführung kann andere Ergebnisse hervorbringen. Das Ergebnis des schnelleren Threads, jener, welcher zuerst zurückgeschrieben hat, wird einfach überschrieben und in der schlussendlichen Summe nicht beachtet.

Eine mögliche Skalarprodukt Implementierung, für den linearen Kernel, ist mit Listing 5.3 gegeben.

Listing 5.3 Skalarprodukt

```
__device__ double kernel_linear(const double* vec1, const double* vec2, const int start_vec1,
    const int startvec_2, const int dim, const int Ndatas_vec1, const int Ndatas_vec2){
  double result = 0.0;
    for(int i = 0; i < dim ; ++i){
        result += vec1[i * Ndatas_vec2 + start_vec1] * vec2[i * Ndatas_vec2 + start_vec2];
    }
  return result;
};
```

5.2.2. Caching

Da für jede Berechnung von $Q_{i,j}$ die zwei Einträge q_i und q_j gebraucht werden lohnt es sich, diese zwischenzuspeichern. Dies ist möglich, da q nur $\#Datenpunkte - 1$ double Werte enthält. Auch die Berechnung ist beispielsweise mit Listing 5.4 möglich.

Damit reduziert sich die Anzahl an Skalarprodukten pro zu berechnendem Matrixelement von drei auf eins. Die zwei eliminierten Skalarprodukte müssen so einmal vorberechnet werden, und werden dann mehrfach genutzt. Dies führt zu einer erheblichen Laufzeitverkürzung.

Listing 5.4 Parallele GPU beschleunigte Berechnung der q_i

```
__global__ void kernel_q(...){
  int index = blockIdx.x * blockDim.x + threadIdx.x;
    double temp = 0;
    for(int feature = 0; feature < Nfeatures_data ; ++feature){
       temp += data_d[feature * Ndatas_data + index] * data_d[feature * Ndatas_data +
            Ndatas_data - 1];
    }
    q[index] = temp;
  }
}
```

Listing 5.5 Linearer Kernel q vorberechnet

```
1  __global__ void kernel_f(...){
2    int index_i = blockIdx.x * blockDim.x + threadIdx.x;
3    int index_j = blockIdx.y * blockDim.y + threadIdx.y;
4
5    if(index_i >= index_j){
6      //Q̄_{i,j}
7      double temp = multd(data_d, data_d, index_i, index_j, Nfeatures_data, Ndatas_data ,
           Ndatas_data)
8        - q[i]
9        - q[j]
10       + QA_cost;
11     if(index_i == index_j){
12       atomicAdd(&ret[ii + i], (temp + 1/C)* d[ii + i] * sgn);
13     }else{
14       atomicAdd(&ret[jj + j], temp * d[ii + i] * sgn);
15       atomicAdd(&ret[ii + i], temp * d[jj + j] * sgn);
16     }
17   }
18 }
```

5.2.3. CUDA Caching

Einem Thread stehen unterschiedliche Speicher zur Verfügung (siehe Abbildung 4.1). Die sehr schnellen Register, getrennt für jeden Thread, und der etwas langsamere Shared Memory. Wobei der Shared Memory aber immer noch viel schneller ist, als der Global Memory. Aus diesem Grund ist die Blockstruktur sinnvoll. Pro Block werden $2 \cdot blocksize$ viele Eingabedaten benötigt, um $blocksize^2$ viele Skalarprodukte zu berechnen.

$$\begin{array}{c} \\ x_i \\ x_{i+1} \\ \vdots \\ x_{i+blocksize} \end{array} \begin{array}{c} \begin{array}{cccc} x_j & x_{j+1} & \dots & x_{j+blocksize} \end{array} \\ \begin{pmatrix} \langle x_i, x_j \rangle & \langle x_i, x_{j+1} \rangle & \dots & \langle x_i, x_{j+blocksize} \rangle) \\ \langle x_{i+1}, x_j) & k(x_{i+1}, x_{j+1} \rangle & \dots & \langle x_{i+1}, x_{j+blocksize} \rangle \\ \vdots & \vdots & & \vdots \\ \langle x_{i+blocksize}, x_j \rangle & \langle x_{i+blocksize}, x_{j+1} \rangle & \dots & \langle x_{i+blocksize}, x_{j+blocksize} \rangle \end{pmatrix} \end{array} \tag{5.5}$$

Da das Shared Memory eine so viel höhere Bandbreite als das Global Memory hat, ist es aus Performance Sicht nützlich, von jedem Block die benutzten Daten zuerst einmal in den Shared Memory zu laden. Der Shared Memory ist pro Multiprozessor SM allerdings auf 64 kB begrenzt [NVI18a]. Das heißt, jeder Block darf maximal 64 kB Shared Memory benötigen. Nach dem NVIDIA Pascal Tuning Guide [NVI18c] sollten für hohe Performance sogar maximal 32 kB Shared Memory pro Block belegt werden. Bei $2 \cdot 32$ Datenvektoren würden also pro Datum 1 kB bzw. 512 B zur Verfügung stehen. Damit könnte also maximal ein Feature Raum von 125 Features bzw. 62 Features, bei Darstellung mit doppelter Genauigkeit (32 bit), abgedeckt werden, was viel zu gering ist. Deshalb wird in der Implementierung nicht mit ganzen „Vektoren“ gecached, sondern es wird Feature-weise gecached. Damit dies möglich ist, werden erstmal Listing 5.3 und Listing 5.5 zusammengefasst in Listing 5.6. Dies hat zwar zur Folge, dass es zur Anpassung des SVM Kernels

nicht mehr reicht, die Skalarprodukt Funktion durch eine andere SVM Kernel Funktion zu ersetzen, aber es ermöglicht eine starke Laufzeitverbesserung.

In Listing 5.7 lädt nicht mehr jeder Thread getrennt seine Daten aus dem Hauptspeicher, sondern es werden immer erst alle benötigten Features für einen Rechenschritt geladen und dann damit gerechnet. Wichtig hierbei ist, dass darauf geachtet wird, das Einlesen gleichmäßig über alle CUDA Warps zu verteilen. In dem in Listing 5.7 gezeigten Ansatz, lädt jeder Thread der 0ten Zeile, den seinem Spaltenindex entsprechenden x_i Wert (Zeile 9) und jeder Thread der 1ten Zeile den entsprechenden x_j Wert (Zeile 10) in den Shared Memory. Erneut ist wichtig, dass im Vergleich zu Listing 5.6 die If-Bedingung in Zeile 5 bzw. 8 angepasst werden muss. Hier kann jetzt nicht mehr direkt überprüft werden, ob der Thread oberhalb, beziehungsweise auf, der Diagonale liegt, sondern nur noch blockweise, ob der entsprechende Threadblock im zu berechnenden Bereich liegt. Dies ist notwendig, da ansonsten bei Blöcken auf der Diagonalen nicht mehr alle benötigten Daten in den Shared Memory geladen werden. Nach dieser Aufweichung der Bedingung, muss mit Zeile 25 sichergestellt werden, dass nur für die über, und auf, der Diagonalen berechneten Werte, die restliche Berechnung durchgeführt wird. Außerdem ist das `__syncthreads();` in Zeile 18 sehr wichtig. Damit die Daten vollständig und korrekt im Shared Memory sind, bevor auf diese zugegriffen wird. Damit müssen die Threads, die nichts laden, so lange pausieren wie die anderen Threads Daten vom globalen in den geteilten Speicher laden.

Listing 5.6 Linearer Kernel mit Skalarproduktberechnung

```
1  __global__ void kernel_f(...){
2    int index_i = blockIdx.x * blockDim.x + threadIdx.x;
3    int index_j = blockIdx.y * blockDim.y + threadIdx.y;
4
5    if(index_i >= index_j){
6        //Q̃_{i,j}
7        double temp = 0.0;
8        for(int feature = 0; feature < Nfeatures_data; ++Nfeatures_data){
9          temp += data_d[feature * Ndatas_data + index_i] * data_d[feature * Ndatas_data + index_j]
10       }
11       temp += QA_cost
12         - q[index_i]
13         - q[index_j];
14       if(index_i == index_j){
15         atomicAdd(&ret[ii + i], (temp + 1/C)* d[ii + i] * sgn);
16       }else{
17         atomicAdd(&ret[jj + j], temp * d[ii + i] * sgn);
18         atomicAdd(&ret[ii + i], temp * d[jj + j] * sgn);
19       }
20     }
21   }
22 }
```

Listing 5.7 Linearer Kernel, Daten cached in Shared Memory

```
__global__ void kernel_f(...){
  int index_i = blockIdx.x * blockDim.x ;
  int index_j = blockIdx.y * blockDim.y ;

  __shared__ double data_intern_i [blocksize];
  __shared__ double data_intern_j [blocksize];

  if(index_i >= index_j){
    index_i += threadIdx.x * BLOCKING_SIZE_THREAD;
    index_j += threadIdx.y * BLOCKING_SIZE_THREAD;
    const int ii = blockIdx.y * blockDim.y + threadIdx.x ;
    double temp = 0.0;

    for(int vec_index = 0; vec_index < Ncols * Nrows ; vec_index += Nrows){
      if(threadIdx.y == 0) data_intern_i[threadIdx.x] = data_d[data_index + index_i ];
      if(threadIdx.y == 1) data_intern_j[threadIdx.x] = data_d[data_index + ii];

      __syncthreads();

      temp += data_intern_i[threadIdx.x] * data_intern_j[threadIdx.y]
    }

    temp += QA_cost - q[i + x] - q[j + y];

    if(index_i> j + y){
      atomicAdd(&ret[index_i], temp * d[index_j] * sgn);
      atomicAdd(&ret[index_j], temp * d[index_i] * sgn);
    }else if(index_i == index_j){
      atomicAdd(&ret[index_i], (temp + 1/C ) * d[index_i] * dgn);
    }
  }
}
```

5.2.4. Thread-level Blocking

Für die schlussendliche Implementierung des linearen Kernels werden die, bis hierhin genannten Optimierungen, auch auf Thread-Ebene angewandt. Das heißt, jetzt behandelt nicht nur jeder CUDA Block einen Block der Skalarprodukte, sondern jeder Thread bestimmt einen eigenen kleinen Block, und jeder CUDA Block damit einen größeren Block.

$$\left(\begin{array}{cc|cc|cc|cc} t_0 & t_0 & t_1 & t_1 & t_2 & t_2 & t_3 & t_3 \\ t_0 & t_0 & t_1 & t_1 & t_2 & t_2 & t_3 & t_3 \\ \hline & & t_4 & t_4 & t_5 & t_5 & t_6 & t_6 \\ & & t_4 & t_4 & t_5 & t_5 & t_6 & t_6 \\ \hline & & & & t_7 & t_7 & t_8 & t_8 \\ & & & & t_7 & t_7 & t_8 & t_8 \\ & & & & & & t_9 & t_9 \\ & & & & & & t_9 & t_9 \end{array}\right)$$

Die finale Implementierung des linearen Kernels ist in Listing 5.8 zu sehen.

Auch hier wird, wie in Abschnitt 5.2.3 erklärt, Feature-weise vorgegangen. In Zeile 12 wird über den Feature Index iteriert. Da die Daten Feature-weise linear gespeichert sind, wird in jeder Iteration Anzahl an Datenpunkte weiter gesprungen und ab hier mit der For-Schleife, beginnend in Zeile 14, linear die zur Berechnung benötigten Daten, in den Shared Memory geladen. Dies geschieht parallel für alle Datenblöcke, die von den Threads benötigt werden. Die For-Schleife in Zeile 21 cached dann Thread-weise die Werte Daten j in Register. Ab Zeile 26 wird dann spaltenweise das Zwischenergebnis berechnet. Dazu wird für jede Spalte i der entsprechende Feature Wert aus dem Shared Memory in ein Register geladen und anschließend mit der kompletten Spalte verrechnet. Die Matrix `matr` speichert die Werte des Threads zwischen und wird Feature-weise aufgebaut. Ab Zeile 35 wird mit dem in `matr` zwischengespeicherte Skalarprodukt und Gleichung (5.1) in `temp` der entsprechende $Q_{i,j}$ Wert berechnet und je nach Addition oder Subtraktion mit Eins oder Minus Eins multipliziert. In Zeile 39 wird überprüft, ob der berechnete $Q_{i,j}$ Wert sich oberhalb der Diagonalen befindet. Wenn dies so ist, dann wird die, dem Index entsprechende, Multiplikation, der Matrix-Vektor Multiplikation, durchgeführt und auf die entsprechenden Stellen im Ergebnis Array aufaddiert. In Zeile 42 wird überprüft, ob das Skalarprodukt auf der Diagonalen, der gesamten Matrix, liegt. Ist dies der Fall, muss nach Gleichung (5.1) zusätzlich $\frac{1}{cost}$ auf das Zwischenergebnis aufaddiert werden. Auch hier wird durch Multiplikation mit `sgn` berücksichtigt, ob es eine Addition oder Subtraktion ist.

Da von allen For-Schleifen die Iterationsanzahlen zur Compilezeit bekannt sind, ist es sinnvoll diese zu Entrollen, da so Abfragen zur Laufzeit entfallen. Dies geschieht mit dem Pragma `#pragma unroll()` direkt vor dem jeweiligen For.

5.2.5. Polynomialer Kernel

Der polynomiale Kernel unterscheidet sich bei der Skalarproduktberechnung nicht vom linearen Kernel. Das heißt, er unterscheidet sich erst ab Zeile 35 von Listing 5.8. Formal wird dabei nun nicht mehr nur das Skalarprodukt $\langle x_i, x_j \rangle$ berechnet, sondern Gleichung (5.6).

$$k(x_i, x_j) \coloneqq (gamma \cdot \langle x_i, x_j \rangle + coef0)^{degree} \tag{5.6}$$

Damit ändert sich in Listing 5.9 sogar ausschließlich die Zeile 5. Das in `matr` berechnete Skalarprodukt bleibt gleich. Dieses Skalarprodukt wird aber nicht, wie im linearen Kernel, direkt zur Berechnung des Matrix-Vektor Produktes genommen, sondern in Gleichung (5.6) eingesetzt. Und damit ergibt sich direkt Listing 5.9. Außerdem müssen auch die außerhalb des Kernels vorberechneten q anders berechnet werden. Auch hier muss das in Listing 5.4 berechnete Skalarprodukt, erst in Gleichung (5.6) eingesetzt werden, bevor es gespeichert wird. Damit ergibt sich Listing 5.10. Die komplette Implementierung des polynomialen Kernels ist im Anhang A.1 mit Listing A.2 dokumentiert.

Listing 5.8 Finale Implementierung des linearen Kernels

```
__global__ void kernel_linear(...){
  int i = blockIdx.x * blockDim.x * BLOCKING_SIZE_THREAD;
  int j = blockIdx.y * blockDim.y * BLOCKING_SIZE_THREAD;
  __shared__ double data_intern_i [CUDABLOCK_SIZE][BLOCKING_SIZE_THREAD];
  __shared__ double data_intern_j [CUDABLOCK_SIZE][BLOCKING_SIZE_THREAD];
  double matr[BLOCKING_SIZE_THREAD][BLOCKING_SIZE_THREAD] = {};
  double data_j[BLOCKING_SIZE_THREAD];
  if(i >= j){
    i += threadIdx.x * BLOCKING_SIZE_THREAD;
    const int ji = j + threadIdx.x * BLOCKING_SIZE_THREAD;
    j += threadIdx.y * BLOCKING_SIZE_THREAD;
    for(int vec_index = 0; vec_index < Ncols * Nrows; vec_index += Nrows){
      #pragma unroll(BLOCKING_SIZE_THREAD)
      for(int block_id = 0; block_id < BLOCKING_SIZE_THREAD; ++block_id){
        const int data_index = vec_index + block_id;
        if(threadIdx.y == block_id) data_intern_i[threadIdx.x][block_id] = data_d[data_index + i
            ];
        if(threadIdx.y == block_id * 2) data_intern_j[threadIdx.x][block_id] = data_d[data_index
            + ji];
      }
      __syncthreads();
      #pragma unroll(BLOCKING_SIZE_THREAD)
      for(int data_index = 0; data_index < BLOCKING_SIZE_THREAD; ++data_index){
        data_j[data_index] = data_intern_j[threadIdx.y][data_index];
      }
      __syncthreads();
      #pragma unroll(BLOCKING_SIZE_THREAD)
      for(int x = 0; x < BLOCKING_SIZE_THREAD; ++x){
        const double data_i = data_intern_i[threadIdx.x][x];
        #pragma unroll(BLOCKING_SIZE_THREAD)
        for(int y = 0; y < BLOCKING_SIZE_THREAD; ++y){
          matr[x][y] += data_i * data_j[y];
        }
      }
    }
    #pragma unroll(BLOCKING_SIZE_THREAD)
    for(int x = 0; x < BLOCKING_SIZE_THREAD; ++x){
      #pragma unroll(BLOCKING_SIZE_THREAD)
      for(int y = 0; y < BLOCKING_SIZE_THREAD; ++y){
        const double temp = (matr[x][y] + QA_cost - q[i + x] - q[j + y]) * add;
        if(i + x > j + y){
          atomicAdd(&ret[i + x], temp * d[j + y]);
          atomicAdd(&ret[j + y], temp * d[i + x]);
        }else if(i + x == j + y){
          atomicAdd(&ret[j + y], (temp + cost * add) * d[i + x]);
        }
      }
    }
  }
}
```

Listing 5.9 Polynomialer Kernel nach der Skalarproduktberechnung

```
1  #pragma unroll(BLOCKING_SIZE_THREAD)
2  for(int x = 0; x < BLOCKING_SIZE_THREAD; ++x){
3    #pragma unroll(BLOCKING_SIZE_THREAD)
4    for(int y = 0; y < BLOCKING_SIZE_THREAD; ++y){
5      const double temp = (pow(gamma * matr[x][y] + coef0, degree) + QA_cost - q[i + x] - q[j +
           y]) * sgn;
6      if(i + x > j + y){
7        atomicAdd(&ret[i + x], temp * d[j + y]);
8        atomicAdd(&ret[j + y], temp * d[i + x]);
9      }else if(i + x == j + y){
10       atomicAdd(&ret[j + y], (temp + cost * sgn) * d[i + x]);
11     }
12   }
13 }
```

Listing 5.10 Kernel zur Bestimmung der q_i

```
1  __global__ void kernel_q(...){
2    int index = blockIdx.x * blockDim.x + threadIdx.x;
3      double temp = 0;
4      for(int feature = 0; feature < Nfeatures_data ; ++feature){
5         temp += data_d[feature * Ndatas_data + index] * data_d[feature * Ndatas_data +
               Ndatas_data - 1];
6      }
7      q[index] = pow((gama * temp + cost0), degree);
8    }
9  }
```

5.2.6. Radialer Basisfunktion Kernel

Bei der radialen Basisfunktion wird, im Vergleich zu dem linearen SVM-Kernel, nicht mehr das Skalarprodukt zweier Vektoren berechnet, sondern der quadratische euklidische Abstand. Dieser wird mit $-gamma$ durch multipliziert und in den Exponenten zur Eulerschen Zahl genommen.

$$k(x_i, x_j) := \exp(-gamma * |u - v|^2) \tag{5.7}$$

Da die benötigten Daten identisch zu den beiden vorherigen Kernel sind, bleibt das Caching gleich. Im Vergleich zu Listing 5.8 ändert sich nur Zeile 30. Hier werden nicht mehr die einzelnen Summanden des Skalarproduktes berechnet, sondern der quadratische Abstand aufaddiert.

```
1  matr[x][y] += (data_i - data_j[y]) * (data_i - data_j[y]) ;
```

Auch bei diesem Kernel muss die Q_{ij} Berechnung angepasst werden.

```
1  const double temp = (exp(-gamma * matr[x][y]) + QA_cost - q[i + x] - q[j + y]) * add;
```

Da bei dem quadratischen euklidischen Abstand, die Wurzel von dem euklidischen Abstand mit Quadrat gekürzt werden kann, wird der berechnete Abstand direkt mit minus Gamma multipliziert und in die Exponentialfunktion eingesetzt.

Auch bei diesem Kernel muss die Vorberechnung der q's angepasst werden: Hier wird auch, anstelle des Skalarproduktes, der quadratische euklidische Abstand für jedes q_i berechnet.

5.2.7. Kernel Zusammenfassung

Aus mathematischer Sicht ist der lineare Kernel identisch zu dem polynomialen Kernel von Grad 1 mit `gamma` gleich Eins und `coef0` gleich Null. Aus Performance Sicht hingegen, ist es durchaus sinnvoll die Kernel getrennt voneinander zu implementieren, und bei ihrem Aufruf zu unterscheiden. Da in diesem linearen Kernel die *pow*- Funktion komplett weggelassen wird und damit Laufzeit gespart werden kann (Vergleiche Anhang A.1)

5.3. Klassifizieren

Die Implementierte SVM Klasse bietet aber auch eine grafikkartengestützte Klassifikation, für bereits geladene Daten, an. Bei der Klassifikation muss pro Datenpunkt nur ein Skalarprodukt

Algorithmus 5.6 Pseudocode klassifizieren

```
procedure LS-SVM-PREDICT(Modeldatei, Datendatei)
    data ← EINLESEN(Datendatei)
    model ← EINLESEN(Modeldatei)
    AUFGPULADEN(data, model)
    LEARN W(model)
    KLASSIFIZIEREN(data, w)
    SPEICHERN(α, data)
end procedure
```

berechnet werden. Damit ist die Performance stark durch die Einlesegeschwindigkeit limitiert.

Die vollständigen Klassifikationsfunktionen sind im Anhang aufgeführt Anhang A.2

5.4. Datei Formate

Die Implementierung der SVM unterstützt zwei Datenformate. Einmal das Standard LIBSVM Format, wobei trotz des dünn-besetzten Eingabeformat nicht dünn-besetzt gerechnet wird. Es werden beim Einlesen der Daten alle ausgelassenen Werte in der internen Datenrepräsentation gleich Null gesetzt, und dann auf den vollen Daten gerechnet. Des Weiteren werden alle Dateien mit der Dateiendung „.arff“ als ARFF Dateien interpretiert. Damit ist auch das Einlesen von Komma getrennten Dateien (CSV) möglich. Damit Arff Dateien als solche erkannt werden, muss die Dateiendung „.arff“ lauten.

5.4.1. LIBSVM Format

LIBSVM nutzt ein sogenanntes „sparse“ (engl. dünn besetzt) Format [CL11]. Dieses Format ist für dünn besetzte Datenstrukturen geeignet, da in diesem nur alle von Null verschiedene Features angegeben werden. Jede Zeile repräsentiert einen Datenpunkt. Die Zeile beginnt mit der Klasse des jeweiligen Datenpunktes (± 1). Anschließend folgen die einzelnen Features als Paare von Index und dem jeweiligen Feature Wert. Zwischen Feature Index und Wert wird mit einem Doppelpunkt getrennt, zwischen den Paaren mit einem Leerzeichen. Als Beispiel ein Datenpunkt der Klasse -1 und den Features $1, 20, -25$: `-1 1:1 2:20 3:-25`

5.4.2. ARFF Format

Das ARFF Format ist ein sogenanntes „dense“ (engl. dicht) Format, und wurde von der Machine Learning Group an der Universität von Waikato für das Programm Weka entwickelt [Gar95]. ARFF Dokumente werden in Header und Daten unterteilt. Der Header enthält den Namen der Relation, eine Liste aller Attribute zusammen mit deren Typen und eine Liste aller Klassen. Auch hier werden die Datenpunkte zeilenweise dargestellt. Wobei im Vergleich zum LIBSVM Format keine Stellen angegeben werden. Damit müssen alle Feature-Werte explizit angegeben werden. Die einzelnen Werte werden durch Kommata getrennt, weshalb das Dezimaltrennzeichen zwingend ein Punkt sein muss. Beispiel:

```
1  %HEADER-------------------------
2  @RELATION Relationsmane
3
4  @ATTRIBUTE Attribut1  NUMERIC
5  @ATTRIBUTE Attribut2  NUMERIC
6  @ATTRIBUTE Attribut3  NUMERIC
7  @ATTRIBUTE class {-1,1}
8
9  %DATA---------------------------
10 @DATA
11 1,10,-25,-1
12 1.2,1.8,.2,1
```

Derzeit werden beim Einlesen nur numerische Attribute und die Klassen ± 1 unterstützt. Das heißt, es muss sichergestellt sein, dass alle Attribute numerischen Typs sind und genau die Klassen -1 und 1 vertreten sind.

5.4.3. Ausgabeformat Modell

Nach dem Lernen werden die Support-Vektoren in eine Modell-Datei geschrieben. Diese besteht aus zwei Teilen, Header und Support-Vektoren. In dem Header steht als erstes der SVM-Typ: `svm_type c_svc\n`, in der zweiten Zeile der Kernel-Typ: `kernel_type linear/poly/radial\n`. In der dritten Zeile die Gesamtanzahl der Klassen `nr_class 2\n`. Die vierte Zeile gibt die Gesamtanzahl der Support-Vektoren an: `total_sv #SV \n`. In der nächsten Zeile steht Bias, sie werden in LIBSVM *rho* genannt: `rho bias \n`. Die darauffolgende Zeile enthält die Label beider Klassen: `label 1 -1 \n`. Hierauf folgt die Anzahl der Support-Vektoren der jeweiligen Klasse:`nr_sv #SV1 #SV2\n`. Der Header wird abgeschlossen, mit einer Zeile, in der `sv\n` steht.

Darauf folgen erst alle Support-Vektoren der ersten Klasse und darauf die der zweiten Klasse. Jeder Support-Vektor steht in einer eigenen Zeile. Zu Beginn der jeweiligen Zeile steht die Gewichtung, daran anschließend folgen Paare von Feature Index und Feature Wert. Feature Index und Feature Wert werden durch einen Doppelpunkt voneinander getrennt, und werden nur für Feature Werte ungleich Null aufgezählt. Zwischen der Gewichtung und den Paaren, sowie zwischen diesen, dient ein Leerzeichen als Trennzeichen.

5.4.4. Ausgabe Klassifizieren

Die Ausgabedatei der Klassifikation enthält in jeder Zeile genau die Klasse, die dem Vektor in der Eingabedatei nach dem Modell entspricht. Dabei enthält Zeile eins die Klasse für den ersten Vektor in der Eingabedatei, Zeile zwei die Klasse für den zweiten usw.

5.5. Speichern

Beim Speichern wird das gelernte Modell in die Model Datei, nach den Vorgaben aus Abschnitt 5.4.3, zurückgeschrieben. Hierbei wird darauf geachtet, dass nur Vektor Einträge ungleich Null geschrieben werden.

6. Ergebnisse

In diesem Kapitel geht es darum die Laufzeit und die Genauigkeit genauer darzulegen. Dabei wurden alle Laufzeitmessungen mit den Standard Parametern (außer epsilon) von LIBSVM ausgeführt.

$$\begin{aligned} \text{gamma} &= \frac{1}{\text{Anzahl Features}} \\ \text{coef0} &= 0 \\ \text{degree} &= 3 \end{aligned}$$

Zur Verbesserung der Vergleichbarkeit wurde das epsilon bei der Grafikkartenimplementierung angepasst (genauer erläutert in Abschnitt 6.5). Die Laufzeiten der LIBSVM Implementierung wurden mit einem epsilon von $0,0001 = 10^{-4}$ gemessen, die Laufzeiten der Grafikkartenimplementierung mit einem epsilon von $0,00000001 = 10^{-8}$. Dies führt dazu, dass beide Implementierungen in etwa die gleiche Genauigkeit erreichen, und damit die Ergebnisse vergleichbarer sind.

Als Datengrundlage zur Zeitmessung dienen Dateien, die mit dem sklearn Single-Label Generator `make_classification` erstellt wurden [PVG+11]. Dieser Generator erzeugt zunächst Cluster von normalverteilten Punkten über die Eckpunkte eines n_informativ-dimensionalen Hyperwürfel und weist jeder Klasse eine gleiche Anzahl von Clustern zu. Er führt die Interdependenz zwischen diesen Merkmalen ein und fügt den Daten verschiedene Arten von weiterem Rauschen hinzu [PVG+11]. Es wurden, bis auf die Anzahl der Datenpunkte und der Features, alle Parameter auf den Standardwerten belassen. Zur besseren Vergleichbarkeit sind die angegebenen Zeiten der Mittelwert aus mehreren Messdurchgängen. Dabei ist jedoch auffällig, dass, obwohl für jeden Durchgang neue Klassifikationsdateien generiert wurden, die Laufzeiten, in jedem Durchlauf, sehr ähnlich waren.

Zu Beginn wird auf die verwendete Hardware und die Compiler eingegangen, danach die Laufzeiten der einzelnen Abschnitte diskutiert. Es wird ein Vergleich mit der Genauigkeit und den Laufzeiten zu LIBSVM gezogen und abschließend werden die Ergebnisse diskutiert.

6.1. Verwendete Hardware

Die Zeitmessungen wurden auf verschiedenen Maschinen durchgeführt. Die deutlich höhere Taktrate der CPU, bei den CPU Tests, schränkt dabei die Vergleichbarkeit nicht stark ein, da bei der GPU Implementierung die Grafikkarte den Hauptanteil des Lösens berechnet. Außerdem wird das Einlesen und das Schreiben der Modelldatei auf die Festplatte stark durch die Performance dieser beschränkt.

6.1.1. Grafikkarten Tests

Alle Tests der GPU-Implementierung wurden auf einer Dual-Sockel Maschine, mit einer NVIDIA®Quadro®GP100, sowie zwei Intel®Xeon®E5-2620, durchgeführt. Es standen 188GiB Arbeitsspeicher zur Verfügung.

NVIDIA Quadro GP100

Architketur:	NVIDIA Pascal
NVIDIA CUDA Kerne:	3584
Double-Precision:	5,3 TeraFLOPS
Single-Precision:	10,6 TeraFLOPS
Speicher:	16 GiB
Compute Capability:	6.0

Für eine vollständige Beschreibung siehe NVIDIA®'s Datenblatt [NVI17b] und NVIDIA®'s Whitepaper [NVI17a].

Intel®Xeon®CPU E5-2620

Codename:	Sandy Bridge
Anzahl Kerne:	6
Anzahl Threads:	12
Grundtaktfrequenz:	2,00 GHz
Taktrate:	2,50 GHz
Cache:	15 MB SmartCache

Für eine vollständige Beschreibung siehe Intel®'s Datenblatt [Int12].

6.1.2. CPU Tests

Aus Zeitgründen wurden die Zeitmessungen, die nur CPU abhängig waren, auf einem Cluster durchgeführt. Das Cluster besteht aus 16 Nodes mit jeweils einem Intel®Xeon®E3-1585v5 und 32 GB RAM. Pro Zeitmessung wurde nur ein Node mit einem Thread genutzt.

Intel®Xeon®E3-1585v5

Codename:	Skylake
Anzahl Kerne:	4
Anzahl Threads:	8
Grundtaktfrequenz:	3,50 GHz
Taktrate:	3,90 GHz
Cache:	8 MB

Für eine vollständige Beschreibung siehe Intel®'s Datenblatt [Int16].

6.2. Verwendete Compiler

Der Grafikkarten Compiler ist der NVIDIA®CUDA Compiler nvcc in der Version 9.0. Dieser unterstützt C++ nur bis zur Version 14. Weshalb das komplette Projekt nur in C++ 14 kompiliert wird. Außerdem wird von der Version 9.0 nicht die aktuelle GCC Version 7.x unterstützt. Mit dem Flag „`-gencode arch=compute_60,code=sm_60`“ wird für Grafikgarten mit CUDA Compute Capability 6.0 kompiliert und mit „`-Xcompiler '-fopenmp'`“ OpenMP im Host Code ermöglicht. Für mehr Informationen siehe NVIDIA CUDA Compiler Driver NVCC [NVI18b]

Der C++ Compiler ist der g++ (GCC) Compiler in der Version 6.4.0. Kompiliert wird nach dem Stand C++ 14 und mit openmp. Für mehr Informationen siehe GCC Compiler Manual [SG17].

6.3. Einzellaufzeiten Lernen

6.3.1. CG

Beim Vergleich der Laufzeiten, der verschiedenen Abschnitte im Lernen, fällt auf, dass der CG Abschnitt eindeutig die größte Komplexität der vier Abschnitte hat. Es fällt aber auch auf, dass die Konstanten der anderen Abschnitte sehr groß sind, sodass die Laufzeit des eigentlichen Lösens bei den Tests noch nicht so stark ins Gewicht fällt. Außerdem ist hier deutlich zu sehen, dass die Anzahl der Features nicht so stark ins Gewicht fällt, wie die Anzahl der Datenpunkte. Die Laufzeit des CG-Algorithmus wächst deutlich stärker mit der Anzahl an Datenpunkten, als mit der Anzahl an Features. Die mittlere gemessene maximale Laufzeit beträgt bei einer Testdatei mit 2^{19} Datenpunkten mit jeweils 2^{11} Features, also $2^{19} \cdot 2^{11} = 2^{30} \approx 10^9$ Informationen, nur ca. 1549 s.

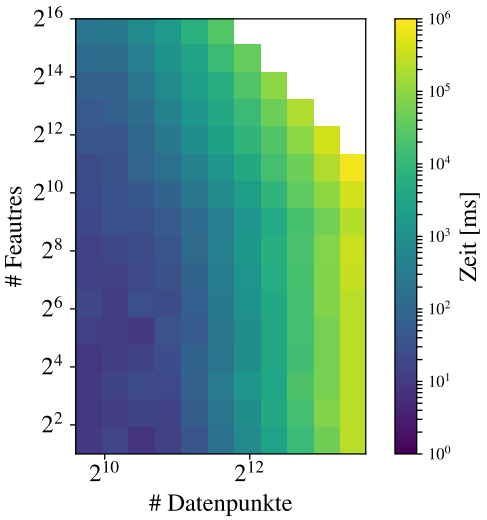


Abbildung 6.1.: Laufzeit CG

6.3.2. Einlesen

Bei den Laufzeiten des Einlesevorgangs Abbildung 6.2 ist ein gewisser Overhead sichtbar. Sobald jedoch mehr größere Dateien eingelesen werden müssen, steigt die Laufzeit wie zu erwarten linear mit der Datengröße. Hier beträgt die längste gemessene Zeit für das Einlesen im Mittel ca. 413 s bei einer Datei von ca. 20 GB Größe und einer Datengröße von $2^{19} \cdot 2^{11} \cdot (8+4)\text{B} = 12{,}88\,\text{GB}$.

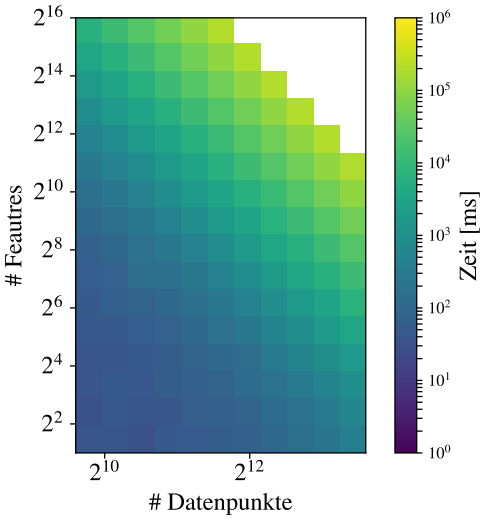


Abbildung 6.2.: Laufzeit einlesen LIBSVM Format

6.3.3. Transfer

Hier fällt auf, dass das Übertragen der Daten auf die Grafikkarte einen gewissen Overhead hat. Diese ca. 200ms fallen bei dem ersten Ansprechen der Grafikkarte an. Sobald aber mehr als $2^{11} \cdot 2^{16} * sizeof(double) \approx 0{,}5\,\text{GB}$ auf die Grafikkarte übertragen wird, steigt die benötigte Zeit proportional zu der Größe, der zu übertragenden Daten. Das mittlere gemessene Maximum beträgt ca. 25 s bei 12,88 GB

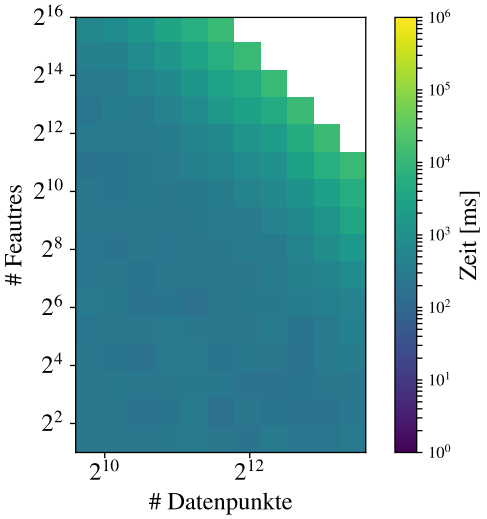


Abbildung 6.3.: Laufzeit Daten auf Quadro GP100 laden

6.3.4. Schreiben

Es fällt auf, dass das Schreiben im Vergleich ein großer Anteil ist. Das mittlere gemessene Maximum beträgt ca. 490 s, damit geht ein großer Anteil der Gesamtlaufzeit auf diesen Abschnitt. Da bei der LS-SVM auch in der Modelldatei alle Vektoren stehen, wird hier mehr Zeit benötigt, als bei dem Standardverfahren. Hier wäre evtl. denkbar, die Trainingsdatei zu kopieren und danach zu manipulieren, anstatt alles selber zu schreiben. Allerdings würde dies nur bei LIBSVM Dateien sinnvoll sein, da hier immer nur die erste „Spalte“ zu verändern ist. Bei den ARFF Dateien müsste jede „Spalte“ angepasst werden.

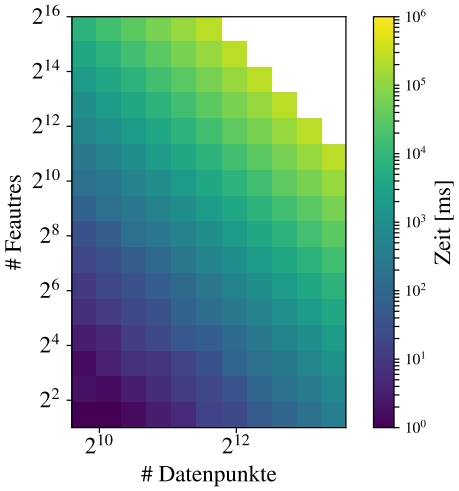


Abbildung 6.4.: Laufzeit Modell schreiben

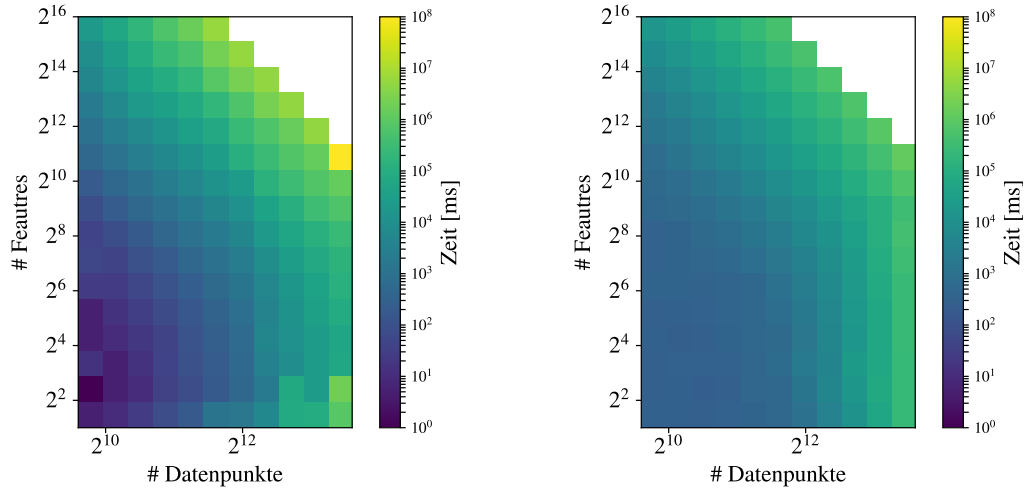
6.4. Vergleich LIBSVM

LIBSVM von Chang und Lin [CL11] ist der Standard für die SVM und ist die am weitesten verbreitete SVM Bibliothek. Aus diesem Grund werden die drei implementierten Kernel mit der C-SVM Implementierung in C++ mit der LIBSVM Implementierung verglichen. LIBSVM wurde außerdem nach Java portiert und bietet auch für andere Programmiersprachen wie Python oder Matlab ein Interface. Ein genereller Vorteil ist, dass LIBSVM auf dem Prinzip des Sequential Minimal Optimization (SMO) basiert, dies führt dazu, dass die Anzahl der Support Vektoren geringer ist. Für weitere Unterschiede vergleiche [YX07].

Hier wird nur die CPU Variante verglichen, da die CUDA Implementierung von Athanasopoulos et al. [ADMK11] für die stark veraltete CUDA Compute Capability 1.1 optimiert wurde. Außerdem wurden keine konkreten Laufzeiten veröffentlicht, sodass ein direkter Vergleich nicht möglich wäre. Nach den Laufzeitdiagrammen liegen die zu erwartenden Laufzeiten aber über denen dieser Implementierung.

Es ist deutlich auffällig, dass der lineare Kernel mehr Support Vektoren hat, als die anderen beiden. Hier lag die Laufzeit bei gleicher Anzahl an Datenpunkten und gleicher Anzahl an Features im Schnitt deutlich höher. Außerdem ist ein deutlicher Unterschied im Laufzeitverhalten bei steigender Feature Anzahl sichtbar. Hier skaliert die GPU Implementierung deutlich besser mit steigender Anzahl an Dimensionen, als die CPU Implementierung von LIBSVM.

6.4.1. Vergleich LIBSVM linear

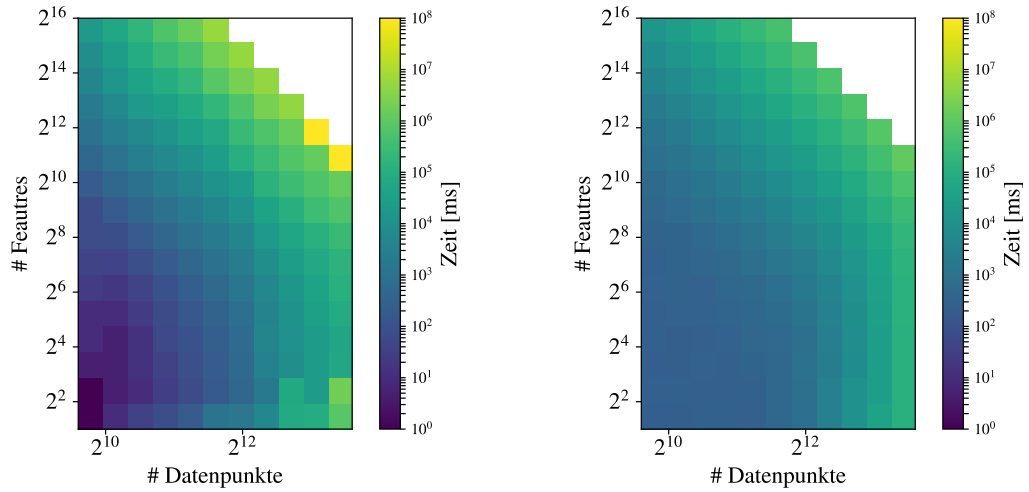


(a) LIBSVM Laufzeit linearer Kernel auf der CPU **(b)** Laufzeit linearer Kernel auf der Quadro GP100

Abbildung 6.5.: Laufzeitvergleich zwischen der Implementierung und LIBSVM bei einer Klassifikation mit dem linearen Kernel

Beim Vergleich der linearen SVM Kernel Abbildung 6.5 ist ein unterschiedliches Verhalten zu beobachten. Bei der LIBSVM Version steigt die Laufzeit mit steigender Feature Anzahl deutlich stärker an, als bei der Grafikkarten Implementierung. Diese hat allerdings den Nachteil, dass sie bei kleinen Trainingsdaten deutlich schlechtere Laufzeiten aufweist. Zusätzlich zu dem Overhead des Einlesens (vergleiche Abschnitt 6.3.2) dominiert der Transfer der wenigen Daten auf die Grafikkarte die Laufzeit (Abschnitt 6.3.3). Auch ist zu beobachten, dass die Laufzeit nicht nur deutlich geringer auf der Grafikkarte ist, sondern auch langsamer ansteigt. Der lineare Kernel hat eine mittlere maximale Laufzeit von ca. 692 079 s = 8 d 14 min mit der LIBSVM Version. Die Grafikkarten Implementierung hat eine mittlere maximale Laufzeit von ca. 2447 s ≈ 41 min. Das ist eine maximale Performance Verbesserung um einen Faktor $\frac{692079}{2447} \approx 283$. Die maximale Laufzeit wird bei 524288 Datenpunkten mit jeweils 2048 Features erreicht.

6.4.2. Vergleich LIBSVM polynomial



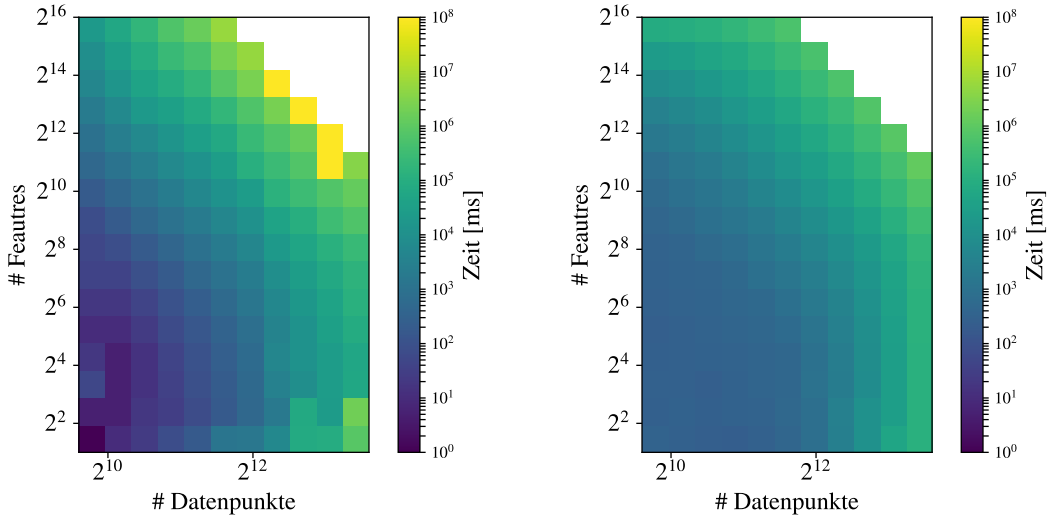
(a) LIBSVM Laufzeit polynomialer Kernel auf der CPU

(b) Laufzeit polynomialer Kernel auf der Quadro GP100

Abbildung 6.6.: Laufzeitvergleich zwischen der Implementierung und LIBSVM bei einer Klassifikation mit dem polynomialen Kernel

Auch bei dem Vergleich des polynomialen SVM Kernels Abbildung 6.6 fällt auf, dass die Anzahl der Features unterschiedliche Auswirkungen auf die Laufzeit hat. Das Laufzeitverhalten ist bei allen drei Kernel sehr ähnlich. Allerdings ist der Unterschied der maximalen Laufzeit interessant. Bei der LIBSVM Variante ist die durchschnittliche maximale Laufzeit, mit 697 218 s $\approx$ 8 d sehr ähnlich mit der des linearen Kernels. Die Grafikkarten Implementierung hat eine durchschnittliche maximale Laufzeit von 2194 s $\approx$ 36 min und liegt damit etwas unter der des linearen Kernels. Damit ist die GPU-Implementierung um Faktor $\frac{697218}{2194} \approx 317$ schneller.

6.4.3. Vergleich LIBSVM Radialer Kernel



(a) LIBSVM Laufzeit radialer Kernel auf der CPU **(b)** Laufzeit radialer Kernel auf der Quadro GP100

Abbildung 6.7.: Laufzeitvergleich zwischen der Implementierung und LIBSVM bei einer Klassifikation mit dem radialen Kernel

Der radiale LIBSVM Kernel ist bei den getesteten Daten der schnellste. Mit einer durchschnittlichen maximalen Laufzeit von 450 830 s, also ungefähr 3 d 14 h, ist der radiale Kernel der LIBSVM Implementierung fast doppelt so schnell, als der lineare beziehungsweise polynomiale Kernel. Die Laufzeit der GPU Implementierung, mit 2463 s, also ungefähr 41 min, ist nahezu identisch mit der des linearen Kernels. Bei diesem Vergleich ist die GPU Implementierung dennoch nur um Faktor $\frac{450830}{2463} \approx 183$ besser. Generell ist der Laufzeitunterschied zwischen den unterschiedlichen Kerneln prozentual auf der GPU nicht so groß wie bei der LIBSVM-Implementierung. Dies liegt wahrscheinlich daran, dass die eigentliche Kernellaufzeit (vergleiche Kapitel 6) die Gesamtlaufzeit der GPU Version noch nicht stark genug dominiert.

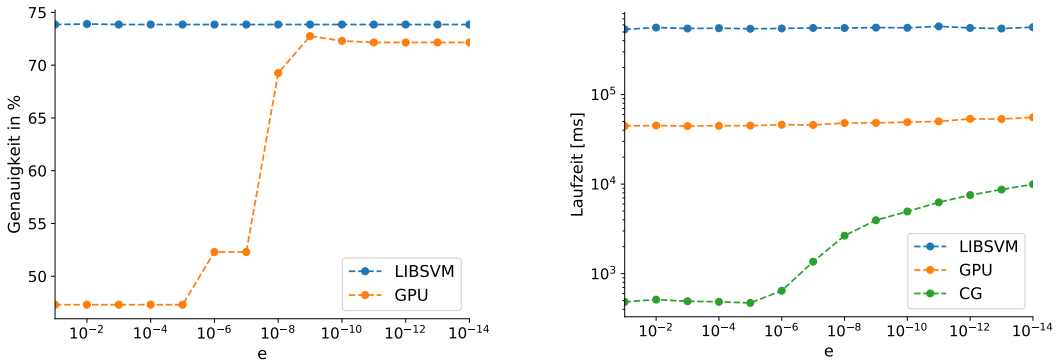
6.4.4. Einschränkung der Vergleichbarkeit

Bei dem Vergleich von LIBSVM und der LS-SVM Grafikkarten Implementierung ist außerdem aufgefallen, dass die LIBSVM Laufzeit stark von der Form der Eingabedaten abhängt. Wenn mehr Support Vektoren gefunden werden müssen, ist die Laufzeit höher. Die Grafikkarten Implementierung hingegen ist bei gleicher Anzahl an Datenpunkten immer etwa gleich schnell. Aus diesem Grund werden, bei unterschiedlichen Klassifizierungsdaten, unterschiedliche Laufzeiten gemessen, sodass je nach Daten der Unterschied der beiden Versionen, größer oder kleiner ausfallen kann.

Außerdem macht, selbst bei den größten Testdateien, das eigentliche Lösen nur etwa 50 Prozent der Laufzeit aus. Darauf muss je nach Vergleich Rücksicht genommen werden. Auch die Endbedingung des iterativen Lösers, sowie die beiden SVM-Verfahren an sich, sind unterschiedlich. Aus diesem

Grund werden unterschiedliche Genauigkeiten erreicht (siehe Abschnitt 6.5). Je nach Trainingsdaten hat die SMO Implementierung Vorteile in der Laufzeit oder die LS-SVM Vorteile bei der Genauigkeit (vergleiche [YX07]).

6.5. Vergleich Genauigkeiten



(a) Vergleich der Genauigkeiten

(b) Vergleich der Laufzeiten

Abbildung 6.8.: Vergleich der Genauigkeiten für unterschiedliche epsilon e und der dazugehörigen Laufzeiten, für das lineare Lernen einer Datei mit 5000 Datenpunkten und 10000 Features.

In diesem Abschnitt wird die Qualität der Ergebnisse untersucht. Als Grundlage dient hierzu eine, mit sklearn [PVG+11] erstellte, Testdatei mit 7000 Datenpunkten und 10000 Features. Die Datei wird in zwei Dateien gesplittet. Eine mit 5000 Datenpunkten, zum Lernen, und für die Überprüfung des Ergebnisses, eine mit den restlichen 2000 Datenpunkten, die klassifiziert wird.

Untersucht wird die Genauigkeit beispielhaft für den linearen SVM-Kernel. Dabei ist das Beispiel allerdings deutlich zu klein, um den Performancevorteil der Grafikkarte voll darstellen zu können, da diese bei den Tests nicht voll ausgelastet ist. Bei einem größeren Test würden sich noch deutlich gravierendere Laufzeitunterschiede darstellen.

Bei der Betrachtung von Abbildung 6.8a fällt auf, dass die LIBSVM-Implementierung selbst für sehr kleine epsilon in diesem Beispiel etwas genauer als die hier untersuchte LS-Implementierung ist. Bemerkenswert ist, dass das epsilon in der LIBSVM Implementierung so gut wie keine Auswirkung auf die Qualität des Ergebnisses hat. Bei der Laufzeit hingegen ist eine minimale Steigerung zu beobachten. Die LS-GPU Implementierung hingegen hat eine deutliche Abhängigkeit von einem wohl gewählten epsilon. Die Güte der Hyperebene hängt maßgeblich von einem klein genug gewählten epsilon ab. Allerdings hängt auch die Laufzeit von der Wahl des epsilon ab. Umso genauer das LGS gelöst werden muss, umso mehr CG Iterationen müssen durchgeführt werden, was, wie in Abbildung 6.8 zu sehen ist, zu einer längeren Laufzeit führt. Bei diesem vergleichsweise kleinen Datensatz mit 5000 Datenpunkten und jeweils 10000 Features wirkt sich dieser Mehraufwand zur genaueren Berechnung des LGS noch sehr gering auf die Gesamtzeit aus. Bei größeren Datensätzen wäre hier allerdings eine deutliche Laufzeitsteigerung sichtbar, sodass das epsilon mit Bedacht gewählt werden muss. Es muss klein genug gewählt werden, damit die Ergebnisse genau genug sind,

aber es sollte nicht zu klein gewählt werden, sodass die Laufzeiten so gering wie möglich bleiben. In Abbildung 6.8 ist gut zu sehen, dass ab einer gewissen Genauigkeit (bei diesem Datensatz ca. 10^{-9}) eine genauere Berechnung des LGS nur zu einer Laufzeitverlängerung und nicht zu einer qualitativ genaueren Hyperebene führt.

Dieser deutliche Verhaltensunterschied kann einerseits mit der unterschiedlichen Interpretation von epsilon begründet werden, andererseits damit, dass die LIBSVM Implementierung die Güte der Ergebnisse überprüft [CL11].

6.6. Erreichte Leistung

Es ist eine gewaltige Performance Steigerung, mit einem Faktor von über 300 (Abschnitt 6.4.2), im Vergleich zu der LIBSVM Implementierung festzustellen. Dabei werden auf der Nvidia Quadro GP100 in den SVM-Kernels ca. 2,3TFLOPS doppelter Genauigkeit(Double-Percission) erreicht. Damit ist bei der Implementierung, trotz den in Kapitel 5 getroffenen Verbesserungen, theoretisch noch weiteres Verbesserungspotenzial möglich. Während der Berechnung wurden jedoch Temperaturen von bis zu 80 °C erreicht. Da die Grafikkarte ab einer Temperatur von 80 °C beginnt die Taktrate herabzusenken (Thermal Throttling), sinkt auch die maximal erreichbare Anzahl der Fließkomma Berechnungen pro Sekunde (FLOPS), und dies indiziert, dass die Belastungsgrenze der Grafikkarte erreicht wird. Im Mittel wurde die Taktrate auf ca. 1150 MHz abgesenkt, bei längeren Tests teilweise auf bis zu 1101 MHz.

Es lässt sich als Fazit ziehen, dass je nach Datensatz eine deutliche Performance Steigerung durch die GPU erreicht wird. Gerade bei größeren Datensätzen spielt diese Implementierung ihre Vorzüge aus. Die Datensätze dürfen allerdings nur so groß sein, dass sie noch auf die Grafikkarte passen. Die größten Testdatensätze, mit bis zu $2^{19} \cdot 2^{11} = 2^{30}$, haben bis zu 15 GB Grafikkartenspeicher belegt.

7. Fazit

Die hier untersuchte CUDA LS-SVM-Implementierung bietet im Vergleich zu der LIBSVM-Implementierung einen deutlichen Performance Vorteil. Um die gleiche Genauigkeit zu erreichen, müssen allerdings die Parameter genauer eingestellt werden. Sofern die Parameter aber gut gewählt sind, erreicht die CUDA LS-SVM-Implementierung eine vergleichbare Genauigkeit zu der LIBSVM-Implementierung in einem Bruchteil der Laufzeit.

Mit 2,3TFLOPS bei doppelter Genauigkeit erreicht diese LS-SVM-Implementierungen durch die Grafikkarte eine extreme Performance Steigerung, die mit einer reinen CPU Implementierungen auf einer CPU nicht möglich wäre. Bei der weiteren Untersuchung der Laufzeiten fällt auf, dass das Einlesen, sowie das Schreiben, bei kleineren Datensätzen sehr lange die dominierenden Faktoren sind. Beim Schreiben des Modells hat LS-SVM im Vergleich zum Standard SVM-Verfahren den Nachteil, dass alle Vektoren Support Vektoren sind. Damit entsteht hier ein großer Mehraufwand.

Das Einlesen der Daten, so wie das Übertragen und die Berechnung auf der GPU geschieht sequentiell. Hier gäbe es also aus Performance Sicht eine weitere Verbesserungsmöglichkeit, wobei dafür der Algorithmus angepasst werden müsste.

Eine Einschränkung dieser LS-Implementierung ist, dass der komplette Datensatz auf die Grafikkarte geladen werden muss. Dies führt, zusammen damit, dass nicht auf einer dünn besetzten Datenstruktur gerechnet wird, dazu, dass es eine Maximalgröße an gleichzeitig berechenbaren Eingabedaten gibt. Diese Maximalgröße lässt sich durch eine Unified Memory Implementierung auf die Größe von Haupt- und Grafikkartenspeicher anheben.

Ausblick

Um die starke Beschleunigung durch die Grafikkarte noch besser nutzen zu können, wäre es sinnvoll die GPU Implementierung so zu erweitern, dass dünn besetze Datenstrukturen nicht nur beim Einlesen, sondern auch in der Berechnung, in ihrer komprimierten Form voll unterstützt werden. Außerdem kann mithilfe der Regression die Anzahl an Support Vektoren minimiert werden. Damit würde die Klassifikation sowie das Schreiben stark beschleunigt werden.

Eine weitere sinnvolle Erweiterung wäre die Multiklassen Klassifikation. Außerdem könnte die Laufzeit über CUDA Streams noch optimiert werden, da hier zum Beispiel das Übertragen auf die Grafikkarte parallel zu den ersten Berechnungen stattfinden kann. Dies wäre auch eine einfache Möglichkeit, mehr als eine Grafikkarte zu nutzen und zusammen mit dem Unified Memory Konzept von CUDA würde das auch die Größenbeschränkung der Daten auf die Hauptspeichergröße anheben.

Eine gute Erweiterung wäre eine Anpassung der Abbruchbedingung oder eine automatische Wahl von epsilon.

Auch weitere Vergleiche in Laufzeit und Genauigkeit zu beispielsweise einer Sequential Minimal Optimization GPU Implementierung wären möglich.

A. Anhang

A.1. Kernel

Nachfolgend werden alle SVM-Kernel aufgelistet.

Listing A.1 Linearer Kernel

```
__global__ void kernel_linear(...){
  int i = blockIdx.x * blockDim.x * BLOCKING_SIZE_THREAD;
  int j = blockIdx.y * blockDim.y * BLOCKING_SIZE_THREAD;
  __shared__ double data_intern_i [CUDABLOCK_SIZE][BLOCKING_SIZE_THREAD];
  __shared__ double data_intern_j [CUDABLOCK_SIZE][BLOCKING_SIZE_THREAD];
  double matr[BLOCKING_SIZE_THREAD][BLOCKING_SIZE_THREAD] = {};
  double data_j[BLOCKING_SIZE_THREAD];
  if(i >= j){
    i += threadIdx.x * BLOCKING_SIZE_THREAD;
    const int ji = j + threadIdx.x * BLOCKING_SIZE_THREAD;
    j += threadIdx.y * BLOCKING_SIZE_THREAD;
    for(int vec_index = 0; vec_index < Ncols * Nrows; vec_index += Nrows){
      #pragma unroll(BLOCKING_SIZE_THREAD)
      for(int block_id = 0; block_id < BLOCKING_SIZE_THREAD; ++block_id){
        const int data_index = vec_index + block_id;
        if(threadIdx.y == block_id) data_intern_i[threadIdx.x][block_id] = data_d[data_index + i
            ];
        if(threadIdx.y == block_id * 2) data_intern_j[threadIdx.x][block_id] = data_d[data_index
            + ji];
      }
      __syncthreads();
      #pragma unroll(BLOCKING_SIZE_THREAD)
      for(int data_index = 0; data_index < BLOCKING_SIZE_THREAD; ++data_index){
        data_j[data_index] = data_intern_j[threadIdx.y][data_index];
      }
      __syncthreads();
      #pragma unroll(BLOCKING_SIZE_THREAD)
      for(int x = 0; x < BLOCKING_SIZE_THREAD; ++x){
        const double data_i = data_intern_i[threadIdx.x][x];
        #pragma unroll(BLOCKING_SIZE_THREAD)
        for(int y = 0; y < BLOCKING_SIZE_THREAD; ++y){
          matr[x][y] += data_i * data_j[y];
        }
      }
    }
    #pragma unroll(BLOCKING_SIZE_THREAD)
    for(int x = 0; x < BLOCKING_SIZE_THREAD; ++x){
      #pragma unroll(BLOCKING_SIZE_THREAD)
      for(int y = 0; y < BLOCKING_SIZE_THREAD; ++y){
        const double temp = (matr[x][y] + QA_cost - q[i + x] - q[j + y]) * add;
        if(i + x > j + y){
          atomicAdd(&ret[i + x], temp * d[j + y]);
          atomicAdd(&ret[j + y], temp * d[i + x]);
        }else if(i + x == j + y){
          atomicAdd(&ret[j + y], (temp + cost * add) * d[i + x]);
        }
      }
    }
  }
}
```

Listing A.2 Polynomialer Kernel

```
__global__ __forceinline__ void kernel_poly(double *q, double *ret, double *d, double
    *data_d,const double QA_cost, const double cost,const int Ncols,const int Nrows,const int
    add, const double gamma, const double coef0 ,const double degree){
  int i = blockIdx.x * blockDim.x * BLOCKING_SIZE_THREAD;
  int j = blockIdx.y * blockDim.y * BLOCKING_SIZE_THREAD;
  __shared__ double data_intern_i [CUDABLOCK_SIZE][BLOCKING_SIZE_THREAD];
  __shared__ double data_intern_j [CUDABLOCK_SIZE][BLOCKING_SIZE_THREAD];
  double matr[BLOCKING_SIZE_THREAD][BLOCKING_SIZE_THREAD] = {};
  double data_j[BLOCKING_SIZE_THREAD];
  if(i >= j){
    i += threadIdx.x * BLOCKING_SIZE_THREAD;
    const int ji = j + threadIdx.x * BLOCKING_SIZE_THREAD;
    j += threadIdx.y * BLOCKING_SIZE_THREAD;
    for(int vec_index = 0; vec_index < Ncols * Nrows ; vec_index += Nrows){
      {
        #pragma unroll(BLOCKING_SIZE_THREAD)
        for(int block_id = 0; block_id < BLOCKING_SIZE_THREAD; ++block_id){
          const int data_index = vec_index + block_id;
          if(threadIdx.y == block_id ) data_intern_i[threadIdx.x][block_id] = data_d[data_index
              + i ];
          if(threadIdx.y == block_id * 2 ) data_intern_j[threadIdx.x][block_id] =
              data_d[data_index + ji];
        }

      }
      __syncthreads();
      #pragma unroll(BLOCKING_SIZE_THREAD)
      for(int data_index = 0; data_index < BLOCKING_SIZE_THREAD; ++data_index){
        data_j[data_index] = data_intern_j[threadIdx.y][data_index];
      }
      __syncthreads();
      #pragma unroll(BLOCKING_SIZE_THREAD)
      for(int x = 0; x < BLOCKING_SIZE_THREAD; ++x){
        const double data_i = data_intern_i[threadIdx.x][x];
        #pragma unroll(BLOCKING_SIZE_THREAD)
        for(int y = 0; y < BLOCKING_SIZE_THREAD; ++y){
          matr[x][y] += data_i * data_j[y];
        }
      }
    }
    #pragma unroll(BLOCKING_SIZE_THREAD)
    for(int x = 0; x < BLOCKING_SIZE_THREAD; ++x){
      #pragma unroll(BLOCKING_SIZE_THREAD)
      for(int y = 0; y < BLOCKING_SIZE_THREAD; ++y){
        const double temp = (pow(gamma * matr[x][y] + coef0, degree) + QA_cost - q[i + x] - q[j
            + y]) * sgn;
        if(i + x > j + y){
          atomicAdd(&ret[i + x], temp * d[j + y]);
          atomicAdd(&ret[j + y], temp * d[i + x]);
        }else if(i + x == j + y){
          atomicAdd(&ret[j + y], (temp + cost * sgn) * d[i + x]);
        }
      }
    }
  }
}
```

Listing A.3 Radialer Kernel

```
__global__ __forceinline__ void kernel_radial(...){
  int i = blockIdx.x * blockDim.x * BLOCKING_SIZE_THREAD;
  int j = blockIdx.y * blockDim.y * BLOCKING_SIZE_THREAD;
  __shared__ double data_intern_i [CUDABLOCK_SIZE][BLOCKING_SIZE_THREAD];
  __shared__ double data_intern_j [CUDABLOCK_SIZE][BLOCKING_SIZE_THREAD];
  double matr[BLOCKING_SIZE_THREAD][BLOCKING_SIZE_THREAD] = {};
  double data_j[BLOCKING_SIZE_THREAD];
  if(i >= j){
    i += threadIdx.x * BLOCKING_SIZE_THREAD;
    const int ji = j + threadIdx.x * BLOCKING_SIZE_THREAD;
    j += threadIdx.y * BLOCKING_SIZE_THREAD;
    for(int vec_index = 0; vec_index < Ncols * Nrows ; vec_index += Nrows){
      {
        #pragma unroll(BLOCKING_SIZE_THREAD)
        for(int block_id = 0; block_id < BLOCKING_SIZE_THREAD; ++block_id){
          const int data_index = vec_index + block_id;
          if(threadIdx.y == block_id ) data_intern_i[threadIdx.x][block_id] = data_d[data_index
              + i ];
          if(threadIdx.y == block_id * 2 ) data_intern_j[threadIdx.x][block_id] =
              data_d[data_index + ji];
        }
      }
      __syncthreads();
      #pragma unroll(BLOCKING_SIZE_THREAD)
      for(int data_index = 0; data_index < BLOCKING_SIZE_THREAD; ++data_index){
        data_j[data_index] = data_intern_j[threadIdx.y][data_index];
      }
      __syncthreads();
      #pragma unroll(BLOCKING_SIZE_THREAD)
      for(int x = 0; x < BLOCKING_SIZE_THREAD; ++x){
        const double data_i = data_intern_i[threadIdx.x][x];
        #pragma unroll(BLOCKING_SIZE_THREAD)
        for(int y = 0; y < BLOCKING_SIZE_THREAD; ++y){
          matr[x][y] += (data_i - data_j[y]) * (data_i - data_j[y]) ;
        }
      }
    }
    #pragma unroll(BLOCKING_SIZE_THREAD)
    for(int x = 0; x < BLOCKING_SIZE_THREAD; ++x){
      #pragma unroll(BLOCKING_SIZE_THREAD)
      for(int y = 0; y < BLOCKING_SIZE_THREAD; ++y){
        const double temp = (exp(-gamma * matr[x][y]) + QA_cost - q[i + x] - q[j + y]) * sgn;
        if(i + x > j + y){
          atomicAdd(&ret[i + x], temp * d[j + y]);
          atomicAdd(&ret[j + y], temp * d[i + x]);
        }else if(i + x == j + y){
          atomicAdd(&ret[j + y], (temp + cost * sgn) * d[i + x]);
        }
      }
    }
  }
}
```

A.2. Klassifikation

Nachfolgend werden alle Klassifikations-Kernel und deren Aufruf aufgelistet.

Listing A.4 w auf der GPU generieren

```
__global__ void kernel_w(double* w_d, double* data_d, double* alpha_d, int count ){
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  double temp = 0;
  for(int dat = 0; dat < count ; ++dat){
    temp += alpha_d[index] * data_d[dat * count + index];
  }
  w_d[index] = temp;
}
```

Listing A.5 Aufruf Kernel w

```
void CSVM::load_w(){
  cudaMalloc((void **) &w_d, Nfeatures_data * sizeof(double));
  double *alpha_d;
  cudaMalloc((void **) &alpha_d, Nfeatures_data * sizeof(double));
  cudaMemcpy(alpha_d, &alpha[0], Nfeatures_data* sizeof(double), cudaMemcpyHostToDevice);
  kernel_w<<<((int)Nfeatures_data/1024) + 1, std::min((int)Nfeatures_data, 1024)>>>(w_d, data_d,
      alpha_d, Ndatas_data);
  cudaDeviceSynchronize();
  cudaFree(alpha_d);
}
```

Listing A.6 GPU Klassifikationskernel

```
__global__ void kernel_predict(double *data_d, double *w, int dim, double *out){
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  double temp = 0;
  for(int feature = 0; feature < dim ; ++feature){
    temp += w[feature] * data_d[index * dim + feature];
  }
  if(temp > 0) {
    out[index] = 1;
  }else{
    out[index] = -1;
  }
}
```

Listing A.7 Aufruf GPU Klassifikationskernel

```
1  std::vector<double> CSVM::predict(double *data, int dim , int count){
2    double *data_d, *out;
3    cudaMalloc((void **) &data_d, dim * count * sizeof(double));
4    cudaMalloc((void **) &out, count * sizeof(double));
5    cudaMemcpy(data_d, data, dim * count * sizeof(double), cudaMemcpyHostToDevice);
6    kernel_predict<<<((int)count/1024) + 1, std::min(count, 1024)>>>(data, w_d, dim, out);
7    std::vector<double> ret(count);
8    cudaDeviceSynchronize();
9    cudaMemcpy(&ret[0], out, count * sizeof(double), cudaMemcpyDeviceToHost);
10   cudaFree(data_d);
11   cudaFree(out);
12   return ret;
13 }
```

Literaturverzeichnis

[ADMK11] A. Athanasopoulos, A. Dimou, V. Mezaris, I. Kompatsiaris. „GPU acceleration for support vector machines“. In: *Procs. 12th Inter. Workshop on Image Analysis for Multimedia Interactive Services (WIAMIS 2011), Delft, Netherlands*. 2011. URL: http://mklab.iti.gr/files/wiamis11.pdf (zitiert auf S. 17, 55).

[AYKY17] R. M. Adnan, X. Yuan, O. Kisi, Y. Yuan. „Streamflow forecasting using artificial neural network and support vector machine models“. In: *American Scientific Research Journal for Engineering, Technology, and Sciences (ASRJETS)* 29.1 (2017), S. 286–294. ISSN: 2313-4402. URL: http://asrjetsjournal.org/index.php/American_Scientific_Journal/article/view/2814 (zitiert auf S. 15).

[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. Pittsburgh, Pennsylvania, USA: ACM, 1992, S. 144–152. ISBN: 0-89791-497-X. DOI: 10.1145/130385.130401. URL: http://doi.acm.org/10.1145/130385.130401 (zitiert auf S. 17).

[BHB02] C. Bahlmann, B. Haasdonk, H. Burkhardt. „Online handwriting recognition with support vector machines-a kernel approach“. In: *Frontiers in handwriting recognition, 2002. proceedings. eighth international workshop on*. IEEE. IEEE Comput. Soc, 2002, S. 49–54. DOI: 10.1109/IWFHR.2002.1030883 (zitiert auf S. 15).

[CL11] C.-C. Chang, C.-J. Lin. „LIBSVM: A library for support vector machines“. In: *ACM Transactions on Intelligent Systems and Technology* 2.3 (3 Apr. 2011). Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm, S. 1–27. DOI: 10.1145/1961189.1961199. URL: https://www.csie.ntu.edu.tw/~cjlin/papers/libsvm.pdf (zitiert auf S. 15, 17, 48, 55, 60).

[COK05] W. Chu, C. J. Ong, S. Keerthi. „An Improved Conjugate Gradient Scheme to the Solution of Least Squares SVM“. In: *IEEE Transactions on Neural Networks* 16.2 (März 2005), S. 498–501. DOI: 10.1109/tnn.2004.841785 (zitiert auf S. 15, 23–25, 36).

[CS00] N. Cristianini, J. Shawe-Taylor. *An introduction to support vector machines and other kernel-based learning methods*. Cambridge university press, 2000. DOI: 10.1017/cbo9780511801389 (zitiert auf S. 17).

[CSK08] B. Catanzaro, N. Sundaram, K. Keutzer. „Fast support vector machine training and classification on graphics processors“. In: *Proceedings of the 25th international conference on Machine learning*. ACM. ACM Press, 2008, S. 104–111. DOI: 10.1145/1390156.1390170 (zitiert auf S. 18).

[CV95] C. Cortes, V. Vapnik. „Support-vector networks“. In: *Machine Learning* 20.3 (Sep. 1995), S. 273–297. ISSN: 1573-0565. DOI: 10.1007/BF00994018. URL: https://doi.org/10.1007/BF00994018 (zitiert auf S. 17).

[DNP08] T.-N. Do, V.-H. Nguyen, F. Poulet. „Speed Up SVM Algorithm for Massive Classification Tasks“. In: *Advanced Data Mining and Applications*. Springer Berlin Heidelberg, 2008, S. 147–157. ISBN: 978-3-540-88192-6. DOI: 10.1007/978-3-540-88192-6_15 (zitiert auf S. 17).

[DWK05] D. Delen, G. Walker, A. Kadam. „Predicting breast cancer survivability: a comparison of three data mining methods“. In: *Artificial intelligence in medicine* 34.2 (Juni 2005), S. 113–127. DOI: https://doi.org/10.1016/j.artmed.2004.07.002 (zitiert auf S. 15).

[FCH+08] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, C.-J. Lin. „LIBLINEAR: A library for large linear classification“. In: *Journal of machine learning research* 9.Aug (2008), S. 1871–1874 (zitiert auf S. 17).

[FCK17] W. D. Fisher, T. K. Campa, V. V. Krzhizhanovskaya. „Anomaly detection in earth dam and levee passive seismic data using support vector machines and automatic feature selection“. In: *Journal of Computational Science* 20 (Mai 2017), S. 143–153. DOI: 10.1016/j.jocs.2016.11.016 (zitiert auf S. 15).

[FPSU96] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, R. Uthurusamy. *Advances in Knowledge Discovery and Data Mining (American Association for Artificial Intelligence)*. AAAI Press, 1. Feb. 1996. ISBN: 978-0262560979. DOI: 10.1007/978-3-642-37456-2. URL: https://www.amazon.com/Knowledge-Discovery-Association-Artificial-Intelligence/dp/0262560976?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0262560976 (zitiert auf S. 15).

[Gar95] S. R. Garner. „Weka: The waikato environment for knowledge analysis“. In: *Proceedings of the New Zealand computer science research students conference*. Citeseer. 1995, S. 57–64 (zitiert auf S. 48).

[Har07] M. Harris. „Optimizing cuda“. In: *SC07: High Performance Computing With CUDA* (2007) (zitiert auf S. 18).

[HMHS17] X. Huang, A. Maier, J. Hornegger, J. A. K. 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. 18).

[HSS08] T. Hofmann, B. Schölkopf, A. J. Smola. „Kernel methods in machine learning“. In: *The Annals of Statistics* 36.3 (Juni 2008), S. 1171–1220. DOI: 10.1214/009053607000000677 (zitiert auf S. 23).

[HTF09] T. Hastie, R. Tibshirani, J. Friedman. „Unsupervised learning“. In: *The elements of statistical learning*. Springer, 2009, S. 485–585. DOI: 10.1007/978-1-4614-7138-7_10 (zitiert auf S. 17).

[Int12] Intel. *Intel® Xeon® Prozessor E5-2620*. 2012. URL: https://ark.intel.com/de/products/64594/Intel-Xeon-Processor-E5-2620-15M-Cache-2_00-GHz-7_20-GTs-Intel-QPI (zitiert auf S. 52).

[Int16] Intel. *Intel® Xeon® Prozessor E3-1585 v5*. 2016. URL: https://ark.intel.com/de/products/93742/Intel-Xeon-Processor-E3-1585-v5-8M-Cache-3_50-GHz (zitiert auf S. 52).

[JLLH11] X. Jin, C. Lin, J. Luo, J. Han. „A data mining-based spam detection system for social media networks“. In: *Proceedings of the VLDB Endowment* 4.12 (2011), S. 1458–1461 (zitiert auf S. 15).

[Joa02] T. Joachims. *Learning to Classify Text Using Support Vector Machines – Methods, Theory, and Algorithms*. Kluwer/Springer, 2002. ISBN: 978-0792376798. URL: https://www.amazon.com/Learning-Classify-Machines-International-Engineering/dp/079237679X?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=079237679X (zitiert auf S. 17).

[Joa99] T. Joachims. „Making large-Scale SVM Learning Practical“. In: *Advances in Kernel Methods - Support Vector Learning*. Hrsg. von B. Schölkopf, C. Burges, A. Smola. Cambridge, MA: MIT Press, 1999. Kap. 11, S. 169–184. URL: http://hdl.handle.net/10419/77178 (zitiert auf S. 17).

[KRG+15] L. Khedher, J. Ramírez, J. M. Górriz, A. Brahim, F. Segovia, A. D. N. Initiative et al. „Early diagnosis of Alzheimer's disease based on partial least squares, principal component analysis and support vector machine using segmented MRI images“. In: *Neurocomputing* 151 (März 2015), S. 139–150. DOI: 10.1016/j.neucom.2014.09.072 (zitiert auf S. 15).

[Lib13] D. N. C. Library. *NVIDIA CUDA Library Documentation*. Version 3.2 Beta. 2013. URL: https://www.cs.cmu.edu/afs/cs/academic/class/15668-s11/www/cuda-doc/html/index.html (zitiert auf S. 27).

[NSC+17] S. Nan, L. Sun, B. Chen, Z. Lin, K.-A. Toh. „Density-Dependent Quantized Least Squares Support Vector Machine for Large Data Sets“. In: *IEEE Transactions on Neural Networks and Learning Systems* 28.1 (Jan. 2017), S. 94–106. DOI: 10.1109/tnnls.2015.2504382 (zitiert auf S. 18).

[NVI17a] NVIDIA. *NVIDIA Tesla P100. The Most Advanced Datacenter Accelerator Ever Built, Featuring Pascal GP100, the World's Fastest GPU*. Whitepaper. 2017. URL: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf (zitiert auf S. 52).

[NVI17b] NVIDIA. *UNMATCHED POWER. UNMATCHED CREATIVE FREEDOM. NVIDIA® QUADRO® GP100*. 2017. URL: http://images.nvidia.com/content/pdf/quadro/data-sheets/302049-NV-DS-Quadro-Pascal-GP100-US-NV-27Feb17-HR.pdf (zitiert auf S. 52).

[NVI18a] NVIDIA. *CUDA C PROGRAMMING GUIDE. Design Guide*. Version PG-02829-001_v9.1. NVIDIA, März 2018. URL: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf (zitiert auf S. 18, 27, 41).

[NVI18b] NVIDIA. *CUDA COMPILER DRIVER NVCC. Reference Guide*. Version TRM-06721-001_v9.1. März 2018. URL: http://docs.nvidia.com/cuda/pdf/CUDA_Compiler_Driver_NVCC.pdf (zitiert auf S. 53).

[NVI18c] NVIDIA. *TUNING CUDA APPLICATIONS FOR PASCAL. Application Note*. Version DA-08134-001_v9.1. März 2018. URL: http://docs.nvidia.com/cuda/pdf/Pascal_Tuning_Guide.pdf (zitiert auf S. 18, 40, 41).

[Pla98] J. Platt. *Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines*. Techn. Ber. MSR-TR-98-14. Microsoft Research, 21. Apr. 1998, S. 21. URL: https://www.microsoft.com/en-us/research/publication/sequential-minimal-optimization-a-fast-algorithm-for-training-support-vector-machines/ (zitiert auf S. 17).

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay. „Scikit-learn: Machine Learning in Python“. In: *Journal of Machine Learning Research* 12 (2011), S. 2825–2830 (zitiert auf S. 51, 59).

[RRB+08] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, W.-m. W. Hwu. „Optimization principles and application performance evaluation of a multithreaded GPU using CUDA“. In: *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM. ACM Press, 2008, S. 73–82. DOI: 10.1145/1345206.1345220 (zitiert auf S. 18).

[SBLV02] J. A. K. 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. 15, 18).

[SG17] R. M. Stallman, the GCC Developer Community. *Using the GNU Compiler Collection*. For gcc version 6.4.0. Free Software Foundation, 2017. URL: https://gcc.gnu.org/onlinedocs/gcc-6.4.0/gcc.pdf (zitiert auf S. 53).

[SGB02] J. A. K. Suykens, T. V. Gestel, J. D. Brabanter. *Least Squares Support Vector Machines*. World Scientific, Nov. 2002. DOI: 10.1142/9789812776655 (zitiert auf S. 15, 18).

[SGV98] C. Saunders, A. Gammerman, V. Vovk. „Ridge regression learning algorithm in dual variables“. In: *Proceedings of the15th International Conference on Machine Learning* (1998) (zitiert auf S. 23).

[She+94] J. R. Shewchuk et al. *An introduction to the conjugate gradient method without the agonizing pain*. 1994. URL: https://courses.cs.washington.edu/courses/cse558/02sp/projects/painless-conjugate-gradient.pdf (zitiert auf S. 15, 18, 36).

[SLD+99] J. A. K. Suykens, L. Lukas, P. V. Dooren, B. D. Moor, J. Vandewalle. „Least squares support vector machine classifiers: a large scale algorithm“. In: *European Conference on Circuit Theory and Design, ECCTD*. Bd. 99. Citeseer. 1999, S. 839–842. URL: http://hdl.handle.net/2078.1/80729 (zitiert auf S. 15, 17).

[SLV00] J. A. K. Suykens, L. Lukas, J. Vandewalle. „Sparse approximation using least squares support vector machines“. In: *Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on*. Bd. 2. IEEE. Presses Polytech. Univ. Romandes, 2000, S. 757–760. DOI: 10.1109/ISCAS.2000.856439 (zitiert auf S. 18).

[SV99a] J. A. K. Suykens, J. Vandewalle. „Least Squares Support Vector Machine Classifiers“. In: *Neural Processing Letters* 9.3 (Juni 1999), S. 293–300. ISSN: 1573-773X. DOI: 10.1023/A:1018628609742. URL: https://doi.org/10.1023/A:1018628609742 (zitiert auf S. 22, 24, 38).

[SV99b] J. A. K. Suykens, J. Vandewalle. „Multiclass least squares support vector machines“. In: *Neural Networks, 1999. IJCNN'99. International Joint Conference*. Bd. 2. IEEE. IEEE, 1999, S. 900–903. DOI: 10.1109/IJCNN.1999.831072 (zitiert auf S. 18).

[Vap00] V. N. Vapnik. *The nature of statistical learning theory*. Deutsch. 2. Aufl. Statistics for engineering and information science. UB Vaihingen. New York ; Berlin ; Heidelberg [u.a.]: Springer, 2000. ISBN: 0-387-98780-0. DOI: 10.1007/978-1-4757-3264-1. URL: http://swbplus.bsz-bw.de/bsz084496711cov.htm (zitiert auf S. 19).

[VGS97] V. Vapnik, S. E. Golowich, A. Smola. „Support vector method for function approximation, regression estimation and signal processing“. In: *Advances in neural information processing systems*. 1997, S. 281–287 (zitiert auf S. 19).

[WEG03] T. Wen, A. Edelma, D. Gorsich. „A Fast Projected Conjugate Gradient Algorithm for Training Support Vector Machines“. In: *Contemporary Mathematics*. Hrsg. von V. Olshevsky. Boston, MA, USA: American Mathematical Society, 2003. Kap. A Fast Projected Conjugate Gradient Algorithm for Training Support Vector Machines, S. 245–263. ISBN: 0-8218-3177-1. URL: http://dl.acm.org/citation.cfm?id=1139327.1139341 (zitiert auf S. 18).

[Wei02] E. W. Weisstein. *Least Squares Fitting*. Hrsg. von M.-.-.-A. W. W. Resource. 2002. URL: http://mathworld.wolfram.com/LeastSquaresFitting.html (zitiert auf S. 17).

[Wer18] D. Werner. *Funktionalanalysis*. Deutsch. 8. Aufl. Springer-Lehrbuch. Berlin, Heidelberg: Springer Spektrum, 2018, Online–Ressource (XIII, 586 Seiten 21 Abb, online resource). ISBN: 978-3-662-55407-4. DOI: 10.1007/978-3-662-55407-4. URL: http://dx.doi.org/10.1007/978-3-662-55407-4 (zitiert auf S. 23).

[YX07] J. Ye, T. Xiong. „SVM versus Least Squares SVM“. English (US). In: *Journal of Machine Learning Research* 2 (2007), S. 644–651. ISSN: 1532-4435. URL: http://www.scopus.com/inward/record.url?scp=84862277678&partnerID=8YFLogxK (zitiert auf S. 17, 22, 55, 59).

[ZYX+08] Z.-Q. Zeng, H.-B. Yu, H.-R. Xu, Y.-Q. Xie, J. Gao. „Fast training support vector machines using parallel sequential minimal optimization“. In: *2008 3rd International Conference on Intelligent System and Knowledge Engineering*. Bd. 1. IEEE. IEEE, Nov. 2008, S. 997–1001. DOI: 10.1109/iske.2008.4731075 (zitiert auf S. 17).

Alle URLs wurden zuletzt am 08. 04. 2018 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