Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer ZFI-BSC-2011-13

Bachelorarbeit

im Studiengang „Angewandte Informatik“

Cuda-basiertes maschinelles Lernen

(mit Anwendung auf die Klassifizierung neuronaler Aktivität)

Mohammed Ibrahim

am Institute für

Numerische und Angewandte Mathematik

Bachelor- und Masterarbeiten
des Zentrums für Informatik
an der Georg-August-Universität Göttingen

23. September 2011

Georg-August-Universität Göttingen
Zentrum für Informatik

Goldschmidtstraße 7
37077 Göttingen
Germany

Tel.	+49 551 39 172000
Fax	+49 551 39 14403
Email	office@cs.uni-goettingen.de
WWW	www.informatik.uni-goettingen.de

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Göttingen, den 23. September 2011

Danksagung

An diese Stelle möchte ich mich besonders bei Professor Dr. Gert Lube bedanken, der mich bei der Anfertigung dieser Arbeit geduldig und mit einem großen Verständnis betreut und intensiv unterstützt hat.

Des Weiteren möchte ich mich bei Dipl.-Phys. Dipl.-Math. Stephan Kramer bedanken, der bei Ausarbeitung und der Implementierung der in diese Arbeit entstandene Applikation umfangreich geholfen hat.

Ein besonderer Dank geht an Stephan Keim, der mich bei der Korrektur der Arbeit zur Seite stand.

Außerdem möchte ich einen herzlichen Dank an alle ausrichten, die mich bei der Anfertigung dieser Arbeit weiter motiviert und unterstützt haben.

I. Inhaltsverzeichnis

II. Abbildungsverzeichnis

1 Einleitung

Mit dem Hauptziel die menschliche Gehirnstruktur zu simulieren, führt das Max-Planck-Institut für Dynamik und Selbstorganisation Göttingen ein Experiment durch, bei dem die Aktivität von neuronalen Zellen unter Stimulation gemessen wird. Die Zellen wurden genetisch so verändert, dass sie auf verschiedenfarbiges Licht reagieren. Diese Reaktion wird durch sogenannte „multi-electrode Arrays“ (MEAs) registriert. In einem MEA sind Mess-Elektroden auf einem Feld verteilt, sodass sie das gesamte Feld abdecken, aber auch gleichzeitig nicht zu nahe aneinander kommen (vgl. Abb. 1).

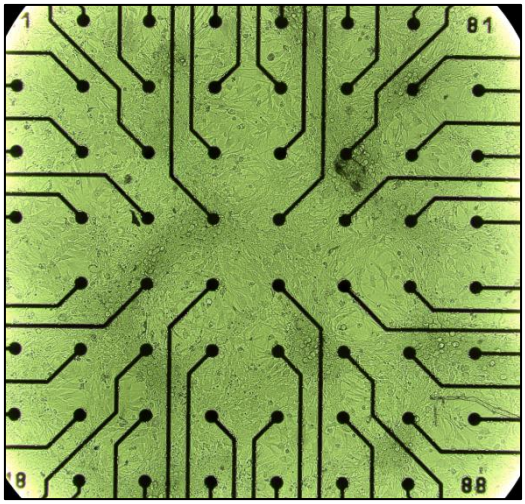

Abbildung 1: MEA mit 60 Mess-Elektroden über einer Neuronenkultur

Die Aktivitäten der neuronalen Zellen werden in verschiedenen Mustern gemessen, wobei jedes dieser Aktivitätsmuster als eine Waveform (vgl. Abb. 2) registriert wird.

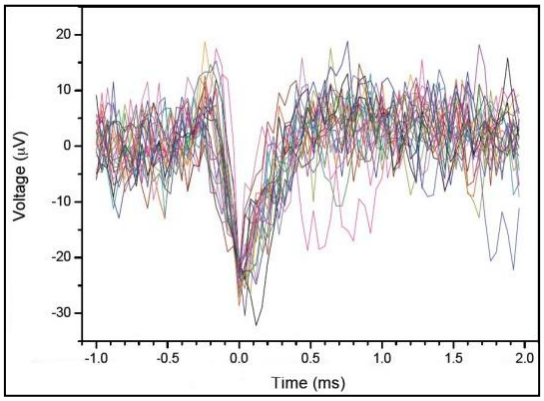

Abbildung 2:Sammlung von Waveforms vgl. Abb. 3

Um das zu erreichen, werden die Zellen auf einem MEA Gerät kultiviert und durch einen konstanten Fluss von Nährstofflösung am Leben erhalten. Jede einzelne Elektrode im „multi-electrode Array“ misst, beziehungsweise verfolgt das Verhalten der Neuronen in ihrer Umgebung (vgl. Abb. 3). Dies führt dazu, dass eine Analyse benötigt wird, um eine Neuron-zu-Elektrode Beziehung zu finden.

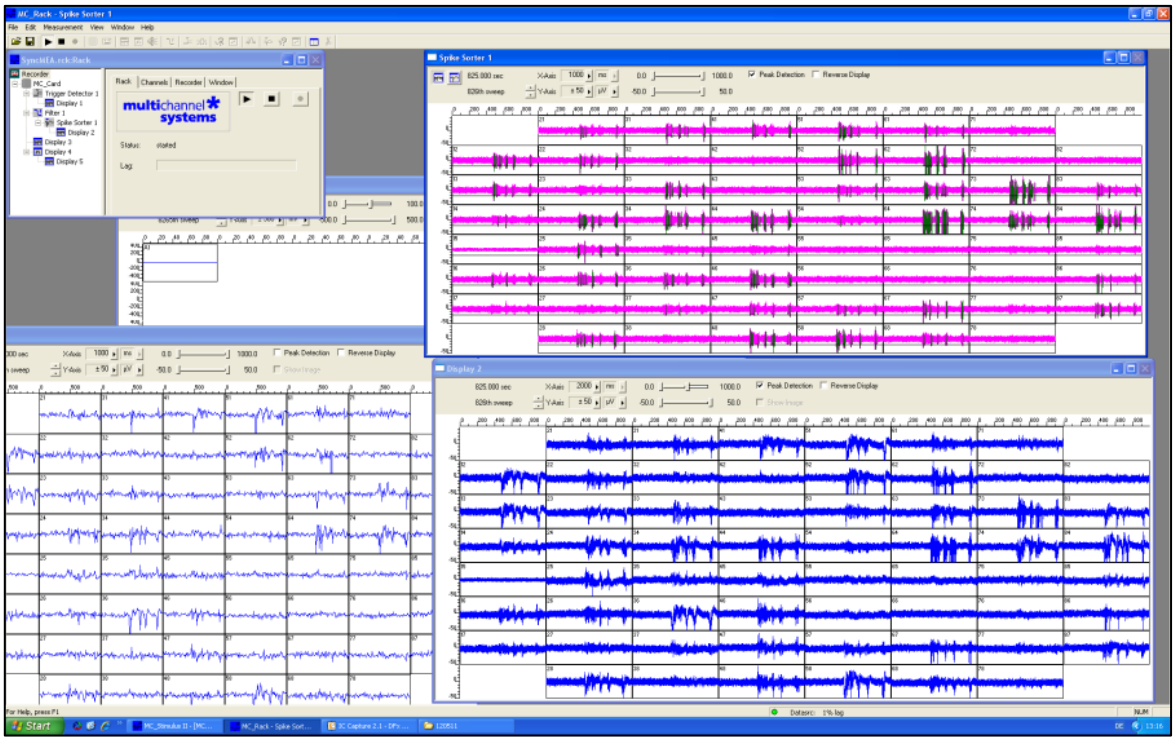

Abbildung 3: Screenshot von der Ausgabe des Messgeräts. Dabei sind die Messungen der einzelnen Elektroden zu erkennen. In Pink sind die verrauschten Waveforms zusehen und in Blau die gesäuberten Daten.

1.1 Motivation und Ziel

Jede MEA Elektrode nimmt Signale aus mehreren Zellen auf. Somit ist die Zuordnung Zelle-Elektrode nicht klar bestimmt. Hier soll das Programm, das wir in dieser Bachelorarbeit erarbeiten, ansetzen und eine Neuron-zu-Elektrode Abbildung liefern.

Aufgrund der großen Menge an Daten aus dem Experiment untersuchen wir die Möglichkeit, diese Aufgabe mit Hilfe von Grafikhardware zu parallelisieren. Dies bedeutet, dass wir existierende Verfahren zur Clusteranalyse, die meistens seriell arbeiten, modifizieren und in eine Form bringen, die die Parallelisierung auf Grafikkarten ermöglicht. Im Detail bedeutet das, möglichst viele Einzelschritte auf BLAS-Routinen abzubilden, um den Entwicklungsaufwand möglichst gering zu halten.

Somit ist der Ziel dieser Arbeit, ein Analyseprogramm für das Experiment anzufertigen und die Möglichkeiten der CUDA Architektur von Nvidia Corporation in diesem Bezug zu erforschen.

1.2 Methodik und Aufbau der Arbeit

Die erhaltenen Daten stellen eine Sammlung von Waveforms in Matrizen mit m Zeilen und n Spalten dar, wobei jede Zeile eine Waveform repräsentiert und die Spalten die Zeitspanne darstellen. Da es sich hierbei um eine große Menge an Daten handelt, führen wir eine QR Zerlegung durch, um die Datenmenge zu verringern ohne relevante Informationen zu verlieren. Durch die QR Zerlegung erhalten wir eine Matrix Q, welche die Größe unserer Eingangsmatrix besitzt und dazu die obere Dreiecksmatrix R, die wir im weiteren Verlauf nutzen werden.
Im nächsten Schritt wenden wir die Hauptkomponentenanalyse auf die Matrix R an. Dabei nutzen wir den NIPALS Algorithmus, um die Matrix R in die Matrizen U und V , welche die Eigenvektoren beinhalten und weitergehend die Matrix Σ mit den dazugehörigen Eigenwerten.
Im Anschluss folgt die Clusteranalyse, welche wir mit Hilfe des K-means beziehungsweise Algorithmus von Shy Shoham (vgl. Abschn. 3.2) durchführen.

In Kapitel 2 werden wir die Grundlagen behandeln, die die Basis für die verwendeten Algorithmen bilden. Des Weiteren gehen wir in Kapitel 3 auf die Idee der Benutzung von alternativen Clusteralgorithmen, die auf der t-Verteilung aufbauen anstatt die Gauss-Verteilung und diskutieren die Implementierung diese Algorithmen in CUDA.
Abschließend führen wir in Kapitel 4 Effizienztestes um festzustellen, ob es sich der erhöhter Programmieraufwand sich lohnt.

2 Grundlagen

In diesem Kapitel behandeln wir die theoretischen Grundlagen der einzelnen Schritte der Arbeit, zum Beispiel die QR-Zerlegung zur Reduktion der Datenmenge, oder auch die verschiedenen EM-Algorithmen, die die Basis für den in dieser Arbeit verwendeten Algorithmus von Shy Shoham (vgl. Abschn. 3.2) bilden.

2.1 QR Zerlegung

2.1.1 Definition

Unter einer QR-Zerlegung versteht man in der linearen Algebra und numerische Mathematik die Faktorisierung einer Matrix A in das Produkt

$$A = Q \cdot R, \tag{2.1}$$

wobei

$$Q \in \mathbb{R}^{m \times m} \tag{2.2}$$

eine orthogonale (d.h. $Q^T Q = I$) bzw. unitäre Matrix ist und

$$R = \begin{bmatrix} r_{11} & \cdots & r_{1n} \\ \vdots & \ddots & \vdots \\ 0 & \cdots & r_{nn} \\ \hline 0 & \cdots & 0 \end{bmatrix} \in \mathbb{K}^{m \times n} \tag{2.3}$$

eine rechte obere Dreiecksmatrix ist. Dabei sind $r_{11}, \cdots, r_{nn}$ jeweils von Null verschieden [6].

Eine solche Zerlegung existiert für jede Matrix $A \in \mathbb{R}^{m \times n}$ mit $m \geq n$ und $Rang(A) = n$ und ist eindeutig bis auf die Vorzeichen der Diagonaleinträge von R [3].

Die QR-Zerlegung kann mit verschiedenen Algorithmen berechnet werden, unter anderem mit den folgenden Verfahren:

- Householder-Transformationen.
- Givens-Rotationen.
- Gram-Schmidtsches Orthogonalisierungsverfahren.

Wir betrachten hier die Householder-Transformation.

2.1.2 QR Zerlegung mit Householder-Transformation

Die Faktorisierung durch die Householder-Transformation folgt, indem die zu faktorisierende Matrix A in jedem Schritt von links mit einer Householder-Matrix der Form

$$P = I - \frac{2}{v^*v} vv^* \in \mathbb{K}^{m\times m} \quad \text{mit} \quad v \in \mathbb{K}^m \setminus \{0\} \tag{2.4}$$

multipliziert wird, um sukzessiv die Spalten von R zu erhalten. Daraus folgt die Darstellung

$$P_n \dots P_1 A = R. \tag{2.5}$$

Hieraus ergibt sich die QR-Faktorisierung

$$A = QR \quad \text{mit} \quad Q = P_1^* \dots P_n^* = P_1 \dots P_n. \tag{2.6}$$

Zuerst wird $A_1 = A$ gesetzt und für $x = (x_1, \dots, x_n)$ die erste Spalte a_1 von A_1. Die Householder-Transformation $P_1 \in \mathbb{K}^{m\times m}$, die einen beliebigen vorgegebenen Vektor $x \in \mathbb{K}^m \setminus\{0\}$ auf ein Vielfaches von $e_1 \in \mathbb{K}^m$ spiegelt, wird bestimmt mit

$$v = \frac{1}{\| x \|_2}\left(x + \frac{x_1 \| x \|_2}{|x_1|} e_1\right) = \frac{1}{|x_1| \| x \|_2}(|x_1|x + x_1 \| x \|_2 e_1) \tag{2.7}$$

für $x_1 \neq 0$ bzw.

$$v = \frac{x}{\| x \|_2} + \frac{x_1}{|x_1|} e_1 \tag{2.8}$$

für $x_1 = 0$, wobei x_1 die erste Komponente von x bezeichnet.

Es folgt

$$P_1 a_1 = r_{11} e_1 \quad \text{mit} \quad |r_{11}| = \| a_1 \|_2 \neq 0 \tag{2.9}$$

beziehungsweise

$$P_1 A = \begin{bmatrix} r_{11} & | & \cdots \\ - & - & - \\ 0 & | & A_2 \end{bmatrix} \quad \text{mit} \quad A_2 \in \mathbb{K}^{(m-1)\times(n-1)}. \tag{2.10}$$

Wir nehmen an, dass nach i Schritten die Householder-Transformationen $P_1, \dots, P_i$ konstruiert wurden mit

$$P_i \dots P_1 A = \begin{bmatrix} r_{11} & \cdots & r_{1i} & | & \\ & \ddots & \vdots & | & R_i' \\ 0 & & r_{ii} & | & \\ - & - & - & - & - \\ 0 & \cdots & 0 & | & A_{i+1} \end{bmatrix} \tag{2.11}$$

mit $|r_{i+1,i+1}| = \| a_{i+1} \|_2 \neq 0$ und $A_{i+2} \in \mathbb{K}^{(m-i-1)\times(n-i-1)}$, und es folgt

$$\begin{bmatrix} I & 0 \\ 0 & P'_{i+1} \end{bmatrix} P_i \dots P_1 A = \left[\begin{array}{ccccccc} r_{11} & \cdots & r_{1i} & & & & \\ & \ddots & \vdots & & & R'_i & \\ 0 & & r_{ii} & & & & \\ \hline 0 & \cdots & 0 & r_{i+1,i+1} & \cdots & & \\ \hline & & & & & & \\ 0 & \cdots & 0 & 0 & & A_{i+2} & \\ & & & & & & \end{array}\right] \tag{2.12}$$

Man beachte, dass sich in diesem Schritt die ersten i Zeilen nicht verändern. P_{i+1} kann selbst wieder als Householder-Transformation mit dem Vektor $v \in \mathbb{K}^m$ der Form $v^T = [0, v'^T]$ aufgefasst werden.
Durch die vollständige Induktion erhält man die Faktorisierung in Formel (2.6) [7].

2.1.3 Anwendungen

Die QR-Zerlegung ist grundlegend für viele Verfahren in der numerischen Mathematik. Sie wird beispielsweise zur Lösung linearer Ausgleichsprobleme eingesetzt oder um orthogonale oder unitäre Basen zu bestimmen. Weiterhin wird die QR-Zerlegung zur Berechnung aller Eigenwerte einer Matrix als Teil des QR-Algorithmus eingesetzt.
In dieser Arbeit verwenden wir die QR-Zerlegung, um einen kleineren Datensatz aus einer großen Menge Daten zu erhalten. Der Grund hierfür ist, dass sich die relevanten Daten nach der Faktorisierung in der oberen Dreiecksmatrix R befinden, die wir für das weitere Vorgehen nutzen können. Die Matrixgröße wird in der Matrix Q erhalten bleiben, welche wir zu einem späteren Zeitpunkt wieder verwenden, um die Größe der Ursprungsmatrix wiederherzustellen. Dabei ist noch zu beachten, dass die Q Matrix in der Praxis rechteckig ist.

2.2 Hauptkomponentenanalyse (PCA)

Unter der Hauptkomponentenanalyse versteht man in der multivariaten Statistik ein Verfahren, bei dem große Datensätze vereinfacht und strukturiert werden, um die Analyse dieser Daten zu erleichtern. Dies wird erreicht, indem eine große Anzahl an statistischen Variablen durch eine kleinere Anzahl von aussagekräftigen Hauptkomponenten genähert wird.
Die Grundidee der Hauptkomponentenanalyse besteht darin, die Dimension der Datensätze, die aus einer Vielzahl von zusammenhängenden Variablen bestehen, zu reduzieren. Dabei behält man so viele Informationen der Datenmenge wie nötig. Um dies zu erreichen, werden die Daten in eine neue Menge von Variablen transformiert, die als „Hauptkomponenten“ (principal components – PCs) bekannt sind. Diese sind unkorreliert und so angeordnet, dass die ersten Komponenten den Großteil der Informationen aus der ursprünglichen Datenmenge beinhalten [9].

2.2.1 Konzeption

Typischerweise ist der zugrundeliegende Datensatz eine Menge von n Punkten (Gegenstände) mit den dazugehörigen p gemessenen Merkmalen, d.h. eine Menge von n Punkten im Raum $\mathbb{R}^p$. Dies kann als Matrix mit n Zeilen und p Spalten veranschaulicht werden. Die Hauptkomponentenanalyse arbeitet mit dem Ziel, eine Projektion dieser Datenpunkte in einen q-dimensionalen Unterraum $\mathbb{R}^q \subset \mathbb{R}^p$ zu bewirken, sodass möglichst wenige Informationen verloren gehen und Redundanz in Korrelation von Datenpunkten zusammengefasst werden.

2.2.2 Verfahren

Das Verfahren beinhaltet die Zerlegung der Datenmatrix X in „Structure“ und „Noise“. Dabei ist der Strukturteil das Matrixprodukt TP^T mit „scores“ T (Trefferzahl) und „loadings“ P (Ladungen). Es wird angenommen, dass die Datenmatrix X in eine Summe des Matrixprodukts TP^T und der Residualmatrix E gespalten werden kann, sodass T und P anstatt X zur weiteren Berechnung verwendet werden können (vgl. Abb. 4) [22].

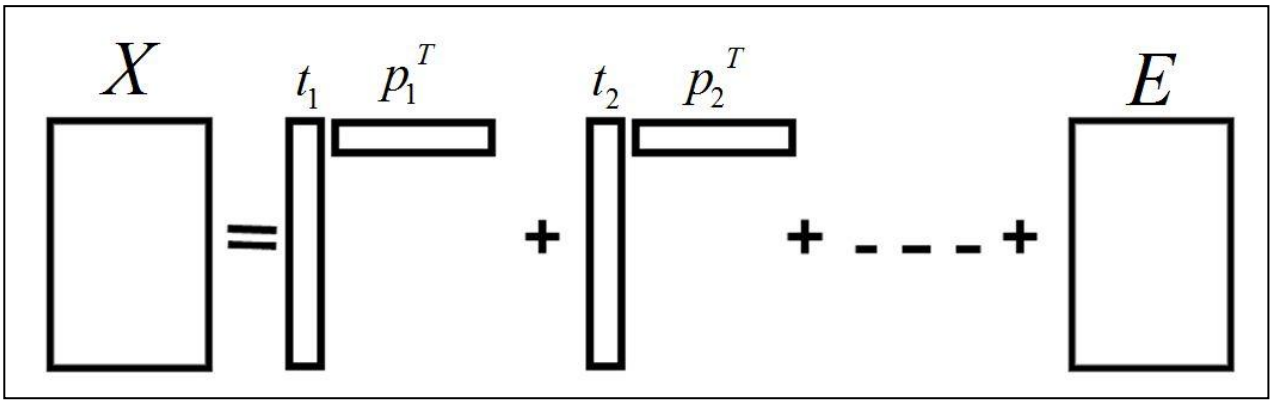

Abbildung 4: Das PC-Modell

Die Anzahl der Hauptkomponenten hängt von der Anzahl der Variablen ab, weshalb maximal p PCs existieren können [1].

Scores T

Die Scores-Matrix T beinhaltet eine Zusammenfassung der Originalvariablen aus X, wobei deutlich wird, in welcher Beziehung Zeilen aus X zueinander stehen.
Die T-Matrix ist so geordnet, dass die erste Spalte t_1 die Scores der ersten Hauptkomponente darstellen, die zweite Spalte die Scores der zweiten Hauptkomponente, usw.

Loadings P

Die Loadings-Matrix P beinhaltet die Gewichte bzw. den Einfluss der Variablen aus X auf die Werte in der Score-Matrix T. Aus der Loadings-Matrix wird sichtbar, welche Variablen die Verantwortung für Muster tragen, die wir in der Scores-Matrix finden.

Residual E

Die Residual-Matrix E ist eine $n \times p$ Matrix und beinhaltet den „Noise"-Teil. Diese Matrix sollte normalerweise klein sein, da eine große Residual-Matrix bedeuten würde, dass eine große Menge an Informationen entfernt wurde, was zu einem weniger effizienten PC-Modell führen wird.
Die Matrix E wird verwendet, um die optimale Anzahl an Hauptkomponenten zu finden. Dabei wird die Veränderung des Residuums betrachtet.

2.2.3 NIPALS Algorithmus

Um die Hauptkomponentenanalyse durchzuführen, stehen mehrere Algorithmen zur Verfügung. Einer davon ist der „Non-Linear Iterative Partial Least Squares" (NIPALS) Algorithmus [27]. Der NIPALS-Algorithmus wird in der Praxis öfter als die Singulärwertzerlegung (SVD) genutzt, um die Hauptkomponentenanalyse durchzuführen. Der Grund hierfür liegt in den numerisch genaueren Ergebnissen. Dafür benötigt der NIPALS-Algorithmus im Vergleich zur SVD mehr Zeit [22].

Algorithmus

i. Aus der Matrix X wird ein Spaltenvektor x_i ausgewählt und in den Vektor t kopiert.

$$t := x_i \tag{2.13}$$

ii. Die Matrix X wird auf t projiziert, um die dazu gehörigen loadings p zu finden.

$$p := \frac{X't}{t't} \tag{2.14}$$

iii. Der Loading-Vektor p wird auf die Länge 1 normiert.

$$p := \frac{p}{|p|} \tag{2.15}$$

iv. Der aktuelle Score-Vektor t wird zwischengespeichert und die Matrix X wird auf p projiziert, um den dazugehörigen Score-Vektor t zu finden.

$$t_{old} := t \qquad t := \frac{Xp}{p'p} \tag{2.16}$$

v. Der Differenzvektor d wird als Differenz zwischen dem alten und neuen Score-Vektor berechnet, um die Konvergenz des Algorithmus zu überprüfen. wenn $|d|$ größer als der vorgegebene Schwellwert ist → weiter bei (ii).

$$d := t_{old} - t \tag{2.17}$$

vi. Das Produkt aus Scores- und Loadings-Vektor bilden die abgeschätzten Hauptkomponenten. Dies wird von der Ursprungsmatrix X subtrahiert.

$$E := X - tp' \tag{2.18}$$

vii. Der Algorithmus wird wieder bei (i) gestartet, um die restlichen Hauptkomponenten zu finden. Hierbei wird aber E als das neue X verwendet.

$$X := E \tag{2.19}$$

Schritte (ii-vi) entsprechen dem Power Iteration Algorithmus zur Eigenwert-Berechnung [11].

2.3 K-means Clusterung

Unter K-means versteht man ein Verfahren zur Clusteranalyse. Dabei werden n Beobachtungen auf k Cluster verteilt.

K-means wird häufig für das Data-Mining und zur Gruppierung von Objekten verwendet, da es mit K-means möglich ist, schnell die Zentren der Cluster zu ermitteln [26].

2.3.1 Voraussetzungen

- Die Anzahl der Cluster muss von Anfang an bekannt sein. Es besteht jedoch die Möglichkeit, diese experimentell zu bestimmen.
- Ein Datensatz sollte möglichst wenig Rauschen, beziehungsweise Extremwerten enthalten.

2.3.2 Algorithmus

Um K-means durchzuführen wird häufig der Lloyd-Algorithmus verwendet. Hierbei wird versucht, ein globales Minimum der Funktion

$$? \; S_j, \mu_i : J = \sum_{i=1}^{k} \sum_{x_j \in S_j} \| x_j - \mu_i \|^2 \rightarrow min \tag{2.20}$$

mit k Clustern $S_1, \dots, S_k$ und Mittelwerten $\mu_1, \dots, \mu_k$ zu finden [12].

Der Algorithmus besteht aus folgenden Schritten:

i. **Initialisierung**: Setzte k zufällige Mittelwerte

$$\mu_1^{(1)}, \dots \, , \mu_k^{(1)} \tag{2.21}$$

ii. **Zuweisung**: Jeder Datenpunkt wird einem Cluster zugewiesen

$$S_i^{(t)} = \{x_j : \| \; x_j - \mu_i^{(t)} \| \leq \| \; x_j - \mu_i^{(t)} \| \;\; \forall \, i^* = 1 \, , \dots \, , k \, \} \tag{2.22}$$

iii. **Update: Berechne** neue Mittelpunkte für die gebildeten Cluster

$$\mu_i^{(t+1)} = \frac{1}{|S_i^{(t)}|} \sum_{x_j \in S_i^{(t)}} x_j \tag{2.23}$$

iii. **Konvergenz:** Der Algorithmus wird solange wiederholt, bis sich die Zuweisungen nicht mehr ändern

$$S_i^{(t+1)} = \; S_i^{(t)} \tag{2.24}$$

2.3.3 Probleme

Es ist garantiert, dass K-means eine Lösung findet [2]. Doch dies muss nicht zwangsläufig heißen, dass es auch die bestmögliche Lösung ist. Die gefundene Lösung ist stark abhängig von den gewählten Startpunkten und der Anzahl der Clusterzentren, da bei der Eingabe einer unterschiedlich großen Anzahl von Clusterzentren verschiedene Lösungen gefunden werden können, welche nicht unbedingt richtig sind.
Weiterhin kann K-means stets, bedingt durch die Minimierung des Abstands zum Mittelpunkt, nur sphärische Cluster finden.
Um die Genauigkeit von K-means zu erhöhen, existieren Variationen vom Algorithmus, wie zum Beispiel K-means++.

2.3.4 K-means++

K-means++ ist ein Algorithmus von David Arthur und Sergei Vassilvitskii aus dem Jahre 2007, der dafür verwendet wird, Anfangswerte für den K-means Algorithmus zu finden und damit das Clustering von K-means zu verbessern [2].

2.3.5 Algorithmus:

i. Wähle zufällig ein Clusterzentrum aus der Datenmenge.
ii. Für jeden Datenpunkt x berechne $D(x)$. Dabei ist $D(x)$ die Distanz zwischen x und dem am nächstliegenden Clusterzentrum, das schon ausgewählt wurde.
iii. Wähle einen neuen Datenpunkt zufällig als neues Clusterzentrum, dabei wird eine gewichtete Wahrscheinlichkeitsverteilung verwendet. Weiterhin wird ein Datenpunkt x mit einer Wahrscheinlichkeit ausgewählt, welche proportional zu $D(x)^2$ ist.
iv. Schritt (ii) und (iii) werden solange wiederholt, bis k Clusterzentren gefunden werden.

2.4 Gaussian Mixture Model (GMM)

Ähnlich wie K-means ist GMM eine Methode zur Clusteranalyse, die als Summe von gewichteten Normalverteilungen betrachtet wird. Im Unterschied zu K-means, wo nur sphärische Cluster gefunden werden, berechnet GMM auch die Ausdehnung der Wahrscheinlichkeitsdichten (vgl. Abb. 5) und die Wahrscheinlichkeiten, ob ein Datenpunkt zu einem Cluster gehört.
Zur Durchführung von GMM wird häufig der iterative Expectation-Maximization-Algorithmus (EM) verwendet [20]. Dabei wird ein statistisches Modell gebildet, d.h. es wird eine Wahrscheinlichkeitsverteilung vorgegeben und die Aufgabe besteht darin, ihre Parameter zu schätzen. Um die gesuchten Parameter zu bestimmen und um die bestmöglichen Parameter zu schätzen, wird die Likelihood Funktion

$$\mathcal{L} = \prod_n P(x_n) \tag{2.25}$$

maximiert mit

$$P(x_n) = \sum \exp\left[-\frac{1}{2}(\mathrm{x} - \mu) \cdot \Sigma^{-1} \cdot (\mathrm{x} - \mu)\right] P(k). \tag{2.26}$$

und

$$P(k) = \frac{1}{\mathrm{N}} \sum_n P_{nk} \tag{2.27}$$

$\mathcal{L}$ ist so definiert, dass es proportional zur Wahrscheinlichkeit der Datenmenge mit allen angepassten Parametern ist. Dabei ist $P(x_n)$ als „mixture weight“ des Datenpunkts x_n zu betrachten und $P(k)$ die Wahrscheinlichkeit für den k-ten Cluster [19]. Weiterhin stellt μ den Mittelwert und Σ die Kovarianzmatrix dar.

Die nachfolgende Abbildung soll den Unterschied zwischen K-means und GMM verdeutlichen.

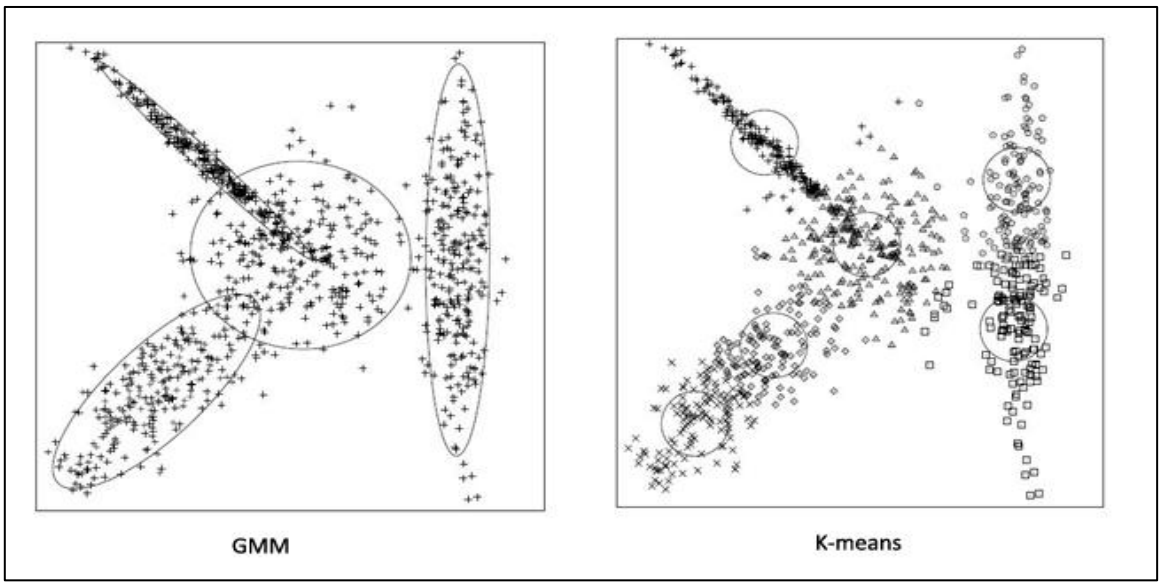

Abbildung 5: Unterschied GMM zu K-means [19]

Im Gegensatz zu K-means berechnet GMM die Ausdehnung der Wahrscheinlichkeitsdichten und erstellt Σ-Ellipsoide.

2.4.1 Algorithmus

i. Es sind Startwerte für die $\mu_k's, \Sigma_k's$ und $P(k)$ zu bestimmen. Dabei sind μ_k die Mittelwerte, $\Sigma_k \in \mathbb{K}^{m\times m}$ die Kovarianzmatrix und $P(k)$ die Wahrscheinlichkeit für den k-ten Cluster.

ii. Wiederhole Folgendes:

I. E-Schritt (Expectation) um neue $P_{nk}'s$ und neues $\mathcal{L}$ zu berechnen mit

$$\mathcal{L} = \prod_n \prod P(x_n) \tag{2.28}$$

$$P(x_n) = \sum_k N\ (x_n|\mu_k, \Sigma_k)P(k) \tag{2.29}$$

$$N(x|\mu, \Sigma) = \frac{1}{(2\pi)^{M/2}\det(\Sigma)^{1/2}} \exp\left[-\frac{1}{2}(x-\mu)^T \cdot \Sigma^{-1} \cdot (x-\mu)\right] \tag{2.30}$$

$$P_{nk} \equiv P(k|n) = \frac{N(x_n|\mu_k, \Sigma_k)P(k)}{P(x_n)} \tag{2.31}$$

II. M-Schritt (Maximization) um neue $\mu_k's, \Sigma_k's$ und $P(k)$ zu berechnen mit

$$\mu_k = \sum_n \frac{P_{nk}(x_n - \mu_k)}{\sum_n P_{nk}} \tag{2.32}$$

$$\Sigma_k = \sum_n P_{nk}(x_n - \mu_k) \otimes (x_n - \mu_k) / \sum_n P_{nk} \tag{2.33}$$

$$P(k) = \frac{1}{N}\sum_n P_{nk} \tag{2.34}$$

iii. Der Algorithmus wird erst beendet, wenn sich der Wert von $\mathcal{L}$ nicht mehr ändert.

2.4.2 Anwendungen

GMM wird unter anderem in der Bildverarbeitung zur Bildsegmentierung und auch in der Tontechnik zur Lautsprecheridentifikation verwendet.

2.5 Studentsche t-Verteilung

Die Studentsche t-Verteilung wurde im Jahre 1908 von William Sealy Gosset entwickelt. Er veröffentlichte die Herleitung erstmals 1908 unter dem Pseudonym Student, da sein damaliger Arbeitsgeber die Veröffentlichung nicht gestattete.

Eine stetige t-verteilte Zufallsvariable X hat die Wahrscheinlichkeitsdichte

$$f_n(X) = \frac{\Gamma\left(\frac{n+1}{2}\right)}{\sqrt{n\pi}\Gamma\left(\frac{n}{2}\right)}\left(1+\frac{X^2}{n}\right)^{-\frac{n+1}{2}} \quad . \tag{2.35}$$

Die t-Verteilung weist eine größere Breite und Flankenbetonung für kleinere Werte des Parameters n auf als die Normalverteilung, vgl. Abb. 6 [1].

2.5.1 Charakteristik

Die Studentsche t-Verteilung ist eine Wahrscheinlichkeitsverteilung mit dem Verhältnis

$$\frac{X}{\sqrt{Y/k}} = X\sqrt{\frac{k}{Y}} \quad . \tag{2.36}$$

Dabei sind:

- X eine normalverteilte Zufallsvariable mit Mittelwert 0 und Varianz 1 → $\mathcal{N}(0,1)$
- Y eine Chi-Quadrat verteilte Zufallsvariable mit k Freiheitsgraden.
- X und Y sind unabhängig voneinander.

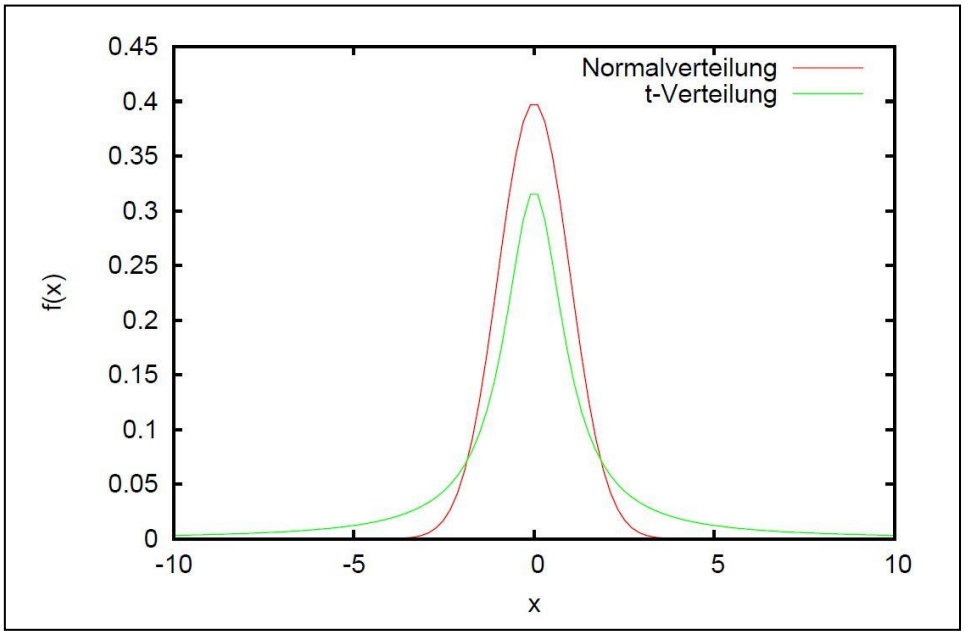

Abbildung 6: t-Verteilung (rot), mit einem Freiheitsgrad, im Vergleich zur Normalverteilung (blau)

2.5.2 Eigenschaften

- **Wendepunkt**

  Die Dichte der t-Verteilung mit n Freiheitsgraden, also der Anzahl der Parameter im System, besitzt einen Wendepunkt bei

$$x = \pm\sqrt{\frac{n}{n+2}} \ . \tag{2.37}$$

- **Median**

  Der Median liegt bei

$$\tilde{x} = 0 \ . \tag{2.38}$$

- **Erwartungswert**

  Für $n > 1$ erhält man den Erwartungswert

$$E(X) = 0 \ . \tag{2.39}$$

  Für $n = 1$ existiert kein Erwartungswert.

- **Varianz**

  Zu $n > 2$ ergibt sich die Varianz

$$Var(X) = 0 \ . \tag{2.40}$$

2.6 CUDA

Die Compute Unified Device Architecture (CUDA) ist eine Technik, die von der NVIDIA Corporation mit dem Ziel entwickelt wurde, Grafikprozessoren (GPU) für nicht-graphische Berechnung zu nutzen.
CUDA ist einer Softwareumgebung, welche es Entwicklern erlaubt, C als High-level Programmiersprache zu verwenden. Weiterhin bietet CUDA auch Schnittstellen zu anderen Programmiersprachen, wie zum Beispiel OpenCL oder auch CUDA Fortran [15].

2.6.1 Hardware Architektur

Die aktuelle Fermi Architektur besitzt bis zu 512 CUDA Rechenkerne. Ein CUDA Kern führt pro Taktzyklus eine Fließkomma- beziehungsweise eine Ganzzahl-FMAD Operation (fused multiply-add) durch. Unter einem *thread* versteht man in der Informatik einen Ausfühungsstrang in der Abarbeitung eines Programms. Die 512 Kerne sind in 16 Streaming Multiprocessors (SMs) mit 32 Kernen pro SM eingeteilt. Jeder CUDA-Rechenkern hat eine eigene arithmetisch-logische Einheit (ALU) und eine Gleitkommaeinheit (FPU). Er kann somit die oben genannten Operationen ausführen. Weiterhin besitzt die GPU 6 Speicherpartitionen und kann bis zu 6 GB GDDR5 DRAM unterstützen.

Eine weitere Neuerung der Fermi Architektur ist die Implementierung einer einheitlichen Speicheranforderung für Speicher- und Lade-Operationen mit einem L1 Cache für jedes SM und einem einheitlichen L2 Cache für alle Operationen (laden, speichern und Textur). Der L1 Cache kann so konfiguriert werden, dass shared- beziehungsweise lokale und globale Speicheroperationen oder Teilmengen davon gepuffert werden. Der einheitliche L2 Cache bietet eine effiziente und schnelle Datenverteilung innerhalb der GPU. Algorithmen, die beispielsweise benötigen, dass verschiedene SMs auf die gleichen Daten zugreifen, profitieren stark von dieser Cache-Hierarchie [17].

2.6.2 Programming Model

Ein CUDA Programm besteht grundsätzlich aus zwei Teilen: einem *Host*- und einem *Device*-Teil. Der *Host* Programmteil ist normalerweise sequenziell und wird auf der CPU ausgeführt. Der *Device*-Programmteil wird im Gegensatz zum *Host* über die GPU ausgeführt. Dies beinhaltet Funktionen, welche als *„Kernels“ bekannt sind.*

Wenn ein *„Kernel“* aufgerufen wird, dann führt er die Berechnung über N verschiedene CUDA *threads aus,* die parallel zueinander ausgeführt werden. Jeder *thread* läuft auf einem CUDA-Kern.

2.6.3 Thread Hierarchie

Ein CUDA *thread* kann mit einem ein-, zwei- oder drei-dimensionalen *thread-index* identifiziert werden. Dies ermöglicht es, einen ein- zwei oder drei-dimensionalen *thread* block zu bilden, um Berechnungen auf der Vektor-, Matrix- oder Volumen-Ebene durchzuführen.

Die Anzahl der *threads* pro *thread* block ist allerdings limitiert, denn alle *threads* eines *Blocks* werden auf einem Multiprozessor ausgeführt. Diese Anzahl liegt auf aktuellen GPUs bei 1024 *threads* pro *Block*. Weiterhin werden *thread blocks* in ein-, zwei- oder drei-dimensionalen Grids organisiert, siehe Abb. 7.

Im Gegensatz zur Blockgröße hängt die Anzahl der *blocks* in einem *Grid* von der Größe der zu bearbeitenden Daten.

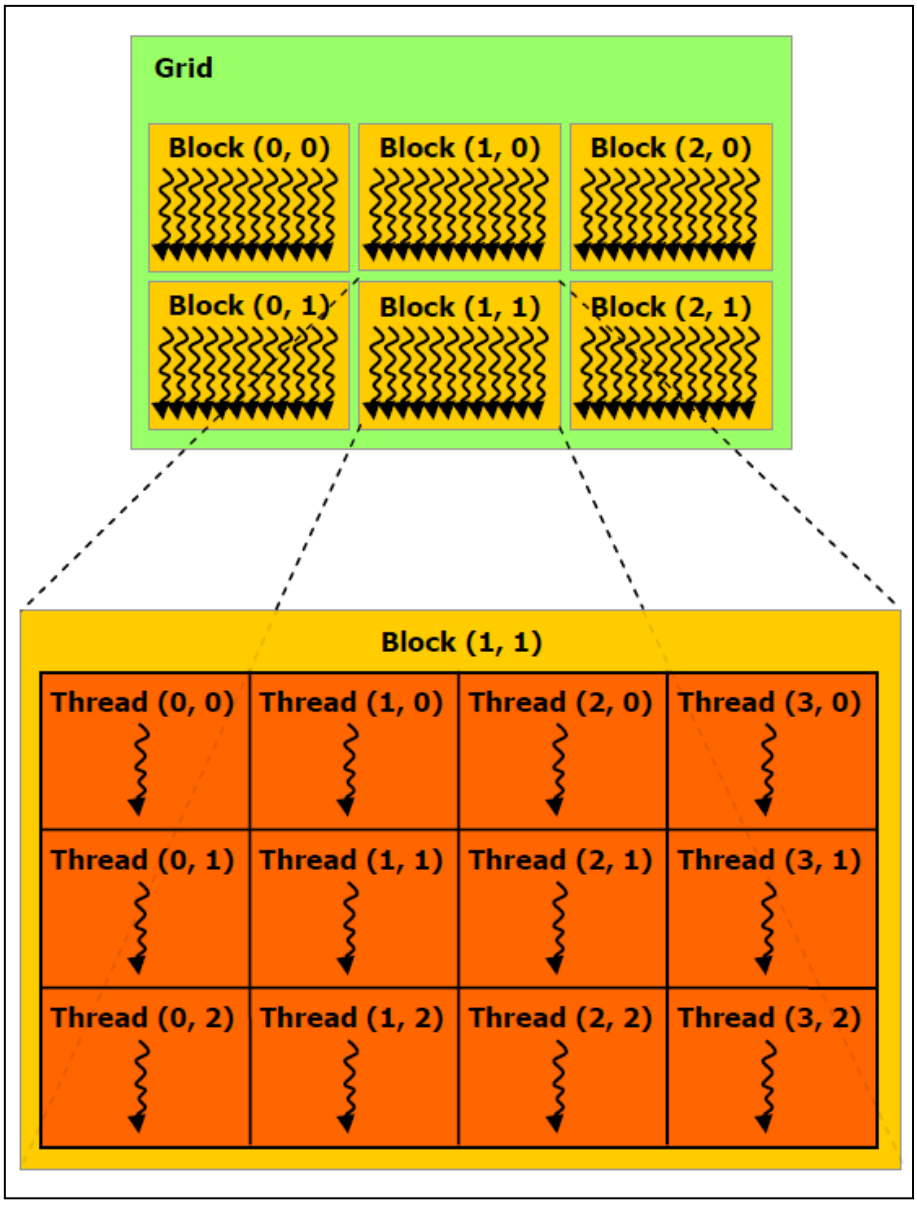

Abbildung 7 : CUDA Thread Hierarchie

2.6.4 Speicherhierarchie

Jeder *thread* hat einen privaten lokalen Speicher. Jeder *thread* block hat einen gemeinsamen Speicher sichtbar für alle *threads* in diesem Block. Alle *threads* haben Zugriff auf denselben globalen Speicher (vgl. Abb. 8).

Weiterhin existieren weitere read-only Speicherbereiche, der „constant memory“ und der „texture memory“. Alle *threads* können auf diese beiden Speicherbereiche zugreifen.

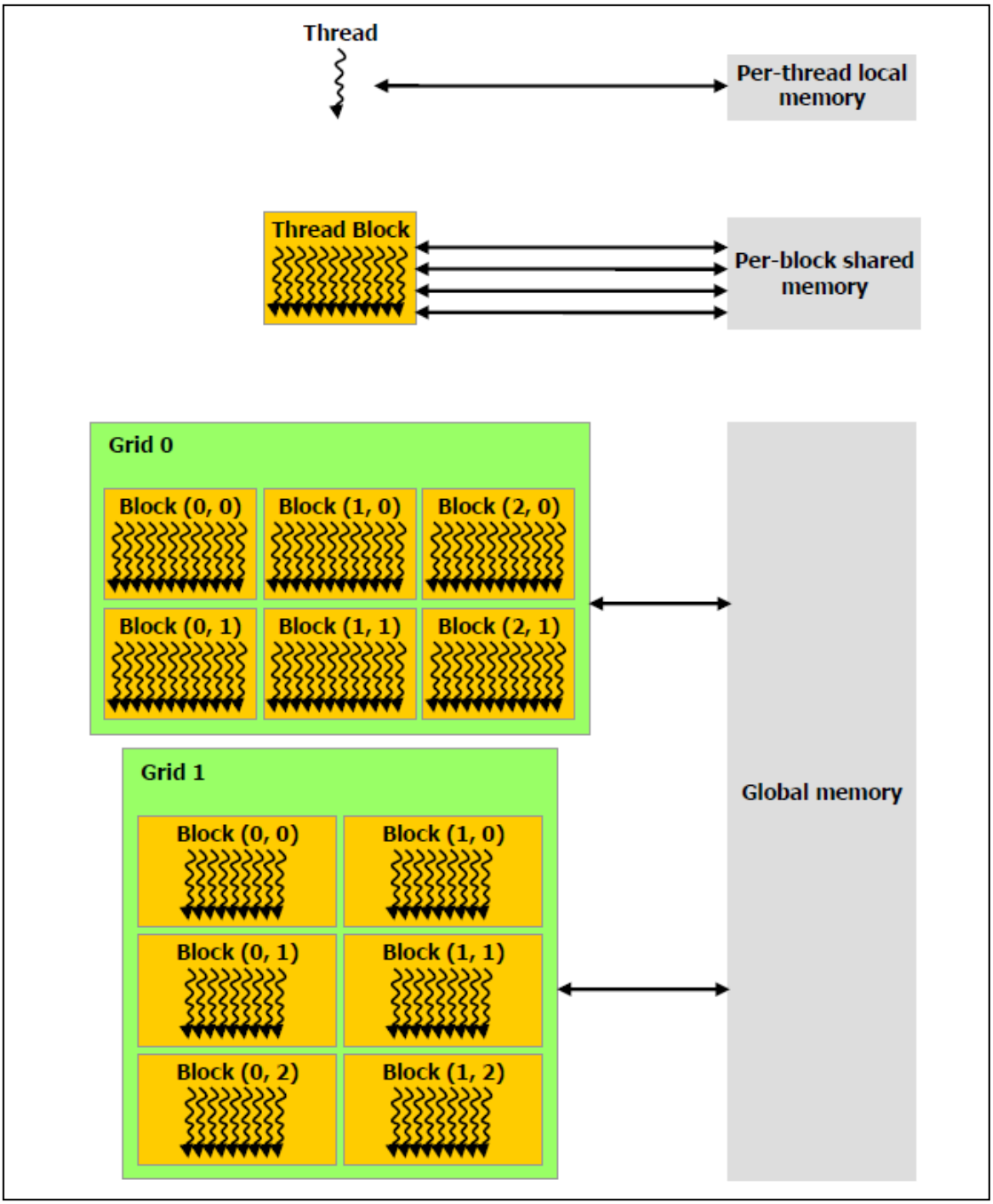

Abbildung 8: Speicher Hierarchie

2.6.5 Hetrogenes Programmieren

Das CUDA Programming Modell gibt vor, dass CUDA-*threads* auf einem separaten *device ausgeführt werden*, der als Coprozessor agiert zu dem *host*, der das C-Programm ausführt (vgl. Abb. 9).

Weiterhin wird vorgegeben, dass *host* und *device* eigene Speicherbereiche im Hauptspeicher belegen.

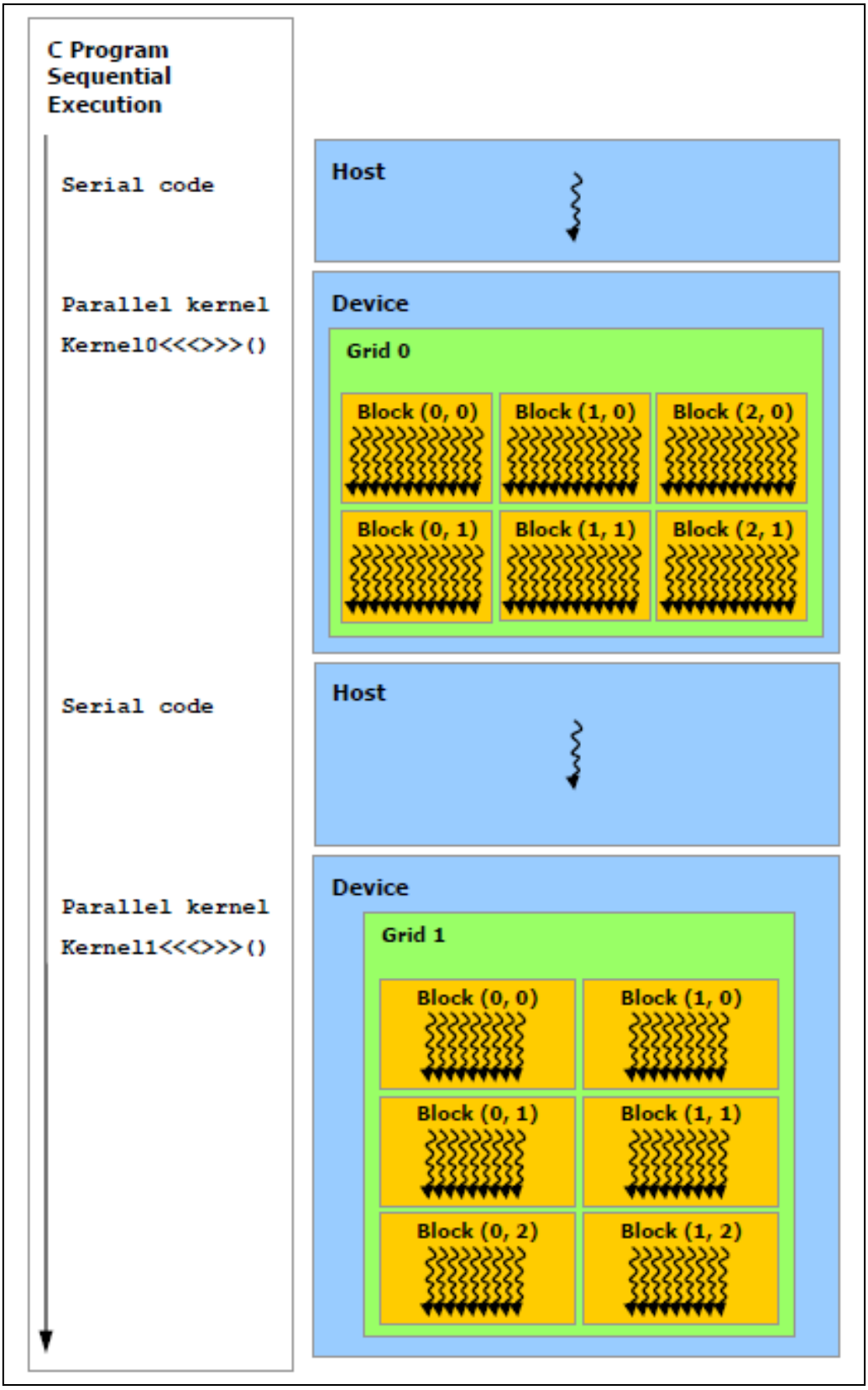

Abbildung 9: Programming Mode

3 Clustering durch Mischung multivariater t-Verteilungen

3.1 Datengenerierung

Für die Simulation werden Testdaten benötigt, um die Korrektheit beziehungsweise die Genauigkeit des implementierten Algorithmus zu verifizieren. Diese Testdaten müssen so generiert werden, dass diese nährungsweise den experimentellen Daten entsprechen.

Um Daten für den K-means Algorithmus (vgl. Abschn. 2.3) zu generieren, werden Gauß-verteilte Datenpunkte um die vom Benutzer eingegebenen Mittelpunkte angelegt. Dies erzeugt eine Matrix mit einer Anzahl von Zeilen, welche der Anzahl der Datenpunkte entspricht. Die Anzahl der Spalten der Matrix entspricht der Dimension der Datenpunkte. Diese Matrix kann dann als Parameter für den K-means Algorithmus eingegeben werden. Als Endergebnis wird erwartet, dass die Mittelpunkte, die der Benutzer angegeben hat, wieder rekonstruiert werden.

Als nächstes werden Testdaten für den von Shy Shoham vorgeschlagenen Algorithmus [23] benötigt. Diesen Daten sollen die typischen Waveforms, die bei einer Messung von neuronalen Aktivitäten vorkommen, entsprechen. Dazu wählen wir verschiedene Funktionen, die diese Wellenformen simulieren. Es wurden die folgenden drei Funktionen ausgewählt:

Die Exponentialfunktion

$$Y = \frac{1}{\tau}(X - X_0)e^{-\left(\frac{X-X_0}{\sigma_0\sqrt{2}}\right)^2} + S \qquad X \in [0,1] \tag{3.41}$$

mit X_0 als Mittelpunkt, σ_0 als Standardabweichung, τ als Streckungsfaktor und S als relative Rauschstärke (vgl. Abb. 10).

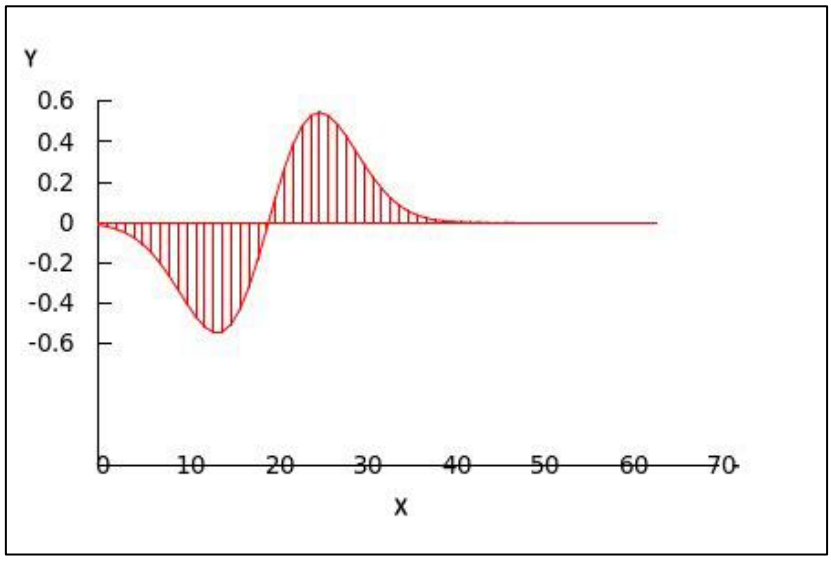

Abbildung 10 : Formel (3.41) mit X=0.3 und σ_0= 0.09

Die Sinusfunktion

$$Y = \begin{cases} 0 \\ (1+S) + sin^2(\sigma_0 \pi (X + noise)) \end{cases} \tag{3.42}$$

mit S als relative Rauschstärke, σ_0 als Frequenz und $noise$ als dazu addiertem Rauschen (vgl. Abb. 11).

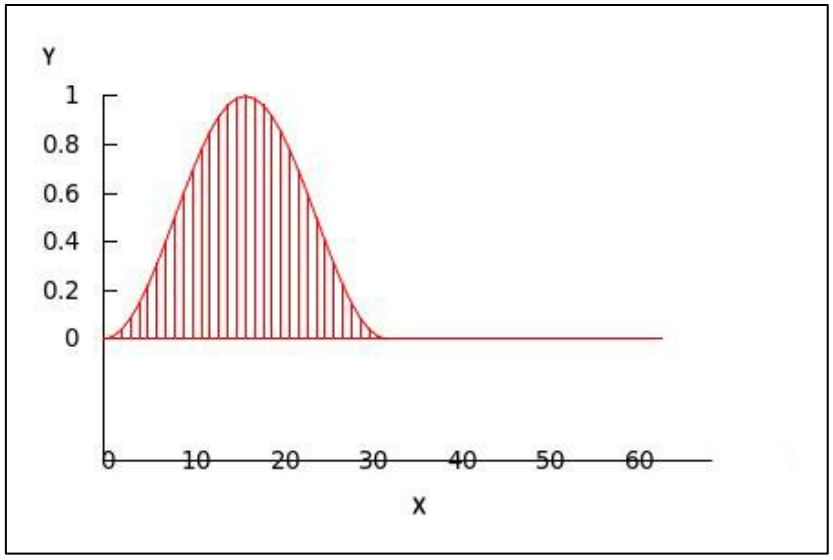

Abbildung 11: : Formel (3.42) mit σ_0= 2

Die Funktion

$$Y = \frac{1}{\tau}\left(\frac{1}{1+\left(\frac{X-X_0}{\sigma_0}\right)^2} - \frac{1}{1+\left(\frac{X-X_1}{\sigma_1}\right)^2}\right) + S \qquad X \in [0,1] \tag{3.43}$$

mit X_0 und X_1 als Mittelpunkten, σ_0 und σ_1 als Halbwertsbreiten, τ als Streckungsfaktor und S als Rauschstärke (vgl. Abb. 12).

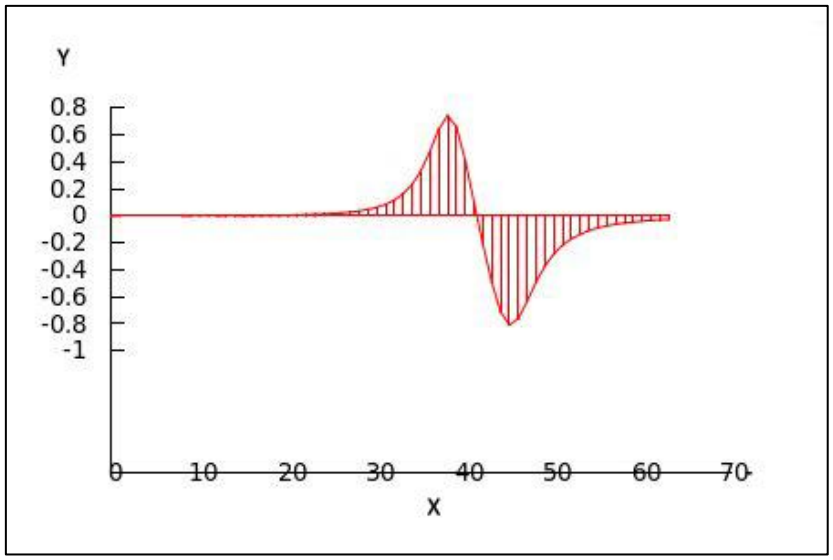

Abbildung 12: : Formel (3.43) mit X_0=0.6, X_1=0.7 und σ_0= 0.05, σ_1= 0.06

Um die Messungenauigkeit zu simulieren, werden alle Parameter, die vom Benutzer angegeben werden, additiv verrauscht. Der Benutzer hat noch die Möglichkeit, die Anzahl der zu generierenden Formen und auch der Anzahl der Datenpunkte jeder dieser Formen zu bestimmen. Zum Beispiel können insgesamt 100 Datenpunkte verteilt werden auf 20 aus der ersten Form, 30 aus der zweiten Form und 50 aus der dritten Form etc.

Das Verrauschen der Mittelpunkte X_0und X_1 soll die Messungenauigkeit bei der „Fensterung“ des Spikes (vgl. Abb. 2), darstellen. Weiterhin soll das Verrauschen von Y die Messungenauigkeit der Elektrode beziehungsweise den thermischen Rauschen in der Nährstofflösung simulieren.

Aus der Häufigkeit des Vorkommens einer Form wird durch ein Auswahlverfahren bestimmt, welche Form in eine Zeile der Eingangsdaten-Matrix geschrieben wird. Somit entsteht als Ergebnis eine zufällig Datenmatrix mit einer zufälligen Folge von waveforms.

Die Datenmatrix wird am Ende noch um die x Achse zentriert, sodass alle Zeilen der waveform-Matrix Null als Mittelpunkt haben.

3.2 K-means Implementierung

Der in den Grundlagen beschriebene K-means wird verwendet, um Startmittelpunkte für den Shoham-Algorithmus zu produzieren [23]. Der K-means Algorithmus baut darauf auf, dass viele elementweise Zugriffe durchgeführt werden. Dies führt dazu, dass der Algorithmus streng seriell arbeitet. Bei dieser Implementierung wird versucht, den Algorithmus aus Kapitel 16 der „Numerical Recepies“ [19] soweit wie möglich zu parallelisieren und ihn auf der Grafikhardware auszuführen.

Der gesamte Algorithmus wird in drei Schritte aufgeteilt. Der erste Schritt ist die Implementierung des K-means++ Algorithmus (vgl. Abschn. 2.3.4), um geeignete Startpunkte für das eigentliche K-means zu finden. Dies geschieht ohne Nutzung der Grafikhardware, denn es hat sich herausgestellt, dass das elementweise Berechnen der Abweichungen der einzelnen Datenpunkte atomare Operationen erfordert, wenn CUDA zum Einsatz kommen sollte. Solche Operationen werden benötigt, um die gesamte Berechnung zu synchronisieren. Der Nachteil an solchen Operationen ist, dass sie viel Rechenzeit benötigen und somit den Geschwindigkeitsgewinn durch die Parallelisierung zunichte machen. Bei den atomaren Operationen handelt es sich um blockierende Operationen, die aus einem Verbund von Einzeloperationen bestehen, die entweder als Ganzes erfolgreich ablaufen oder fehlschlagen. Aus diesem Grund wird der K-means++ Algorithmus mit Hilfe von OpenMP implementiert, um einen Geschwindigkeitsgewinn zu erhalten.

Nach Erzeugung der Startmittelpunkte mittels K-means++ kann der eigentliche K-means Algorithmus beginnen. Er besteht aus zwei Schritten:

1. Jedem Datenpunkt wird ein Clustermittelpunkt zugewiesen, basierend auf der Entfernung zu diesem Mittelpunkt.
2. Basierend auf den zugewiesenen Datenpunkten werden für jeden Cluster die Mittelpunkte neu berechnet.

Es hat sich erwiesen, dass die Zuweisung der Datenpunkte auf die verschiedenen Clustermittelpunkte gut auf der Grafikhardware realisierbar ist. So werden die Datenpunkte und die Clustermittelpunkte in den Grafikspeicher geladen und dann mit Hilfe der Multiprozessoren die oben genannte Zuweisung parallel durchgeführt. Dabei wird aber an einer Stelle die atomare Addition angewendet, um die Clusterzuweisungen zu zählen und damit die Berechnung der verschiedenen Programm-Threads zu synchronisieren.

Zuletzt werden die Daten zurück aus dem Grafikspeicher kopiert, um den letzten Schritt, die Berechnung der neuen Clustermittelpunkte, durchzuführen. Dies geschieht wieder auf der CPU, da es sich hierbei um eine kleine Menge an Daten handelt, die das Parallelisieren durch die Grafikhardware nicht benötigen. Somit wird der K-means Algorithmus als CPU-GPU-Hybrid betrieben.

Es ist jedoch möglich, den K-means Algorithmus auch in seiner Gesamtheit auf der CUDA Architektur zu betreiben. Dabei wird eine parallele Variante des Algorithmus benötigt, die weniger Zugriffe auf die einzelnen Elemente benötigt. So war es für den Software Entwickler Serban Giuroiu [5] möglich, diesen Algorithmus nach eigenen Angaben einen Geschwindigkeitsschub von 650% gegenüber dem sequenziellen Programm bei einer Clusteranzahl von 128 Clustern zu erzielen.
Da die Arbeit von Serban Giroiu erst später im Verlauf der Abarbeitung entdeckt wurde, war es aus Zeitgründen nicht mehr möglich, die parallele Variante selbst zu implementieren.

3.3 t-Verteilung als Basis des Clustering-Algorithmus

Betrachtet man die gemessene Aktivität eines Neurons, so beobachtet man, dass jedes Neuron eine individuelle, reproduzierbare Form erzeugt. Diese Form ist verunreinigt durch eine Störung, die hauptsächlich additiv ist. Quellen für die Störung können unter anderem Störungen in der Elektronik und Messgeräten oder auch Aktivitäten von anderen entfernten Neuronen sein.
Somit stellt sich heraus, dass das Problem der Klassifizierung von neuronalen Aktivitäten ein Clustering-Problem ist. Hierzu gibt es verschiedene bekannte Klassifizierungsalgorithmen, zum Beispiel K-means (vgl. Abschnitt 2.3) oder das „Gaussian Mixture Model“ (GMM Algorithmus; vgl. Abschnitt 2.4). Im Allgemeinen wird angenommen, dass der additive Störungsanteil, welcher Gauß-verteilt ist, dazu führt, dass die erzeugten Waveformen der einzelnen Neuronen eine Auswahl aus multidimensionalen Gauß-Verteilungen mit einem bestimmten Mittelwert und eine Kovarianzmatrix bilden. Aus diesem Grund beschäftigten sich Forscher in den letzten Jahren mit dem GMM Algorithmus, um das Klassifizierungsproblem zu lösen. Es bietet eine leistungsfähige Methode für solch ein statistisches Modell [23].
Obwohl die Ergebnisse, die aus dem Einsatz von GMM erzeugt werden, gut studiert sind, hat man aus kürzlich erschienenen Studien Hinweise erhalten, dass diese Ergebnisse eine ungenaue Beschreibung der Spike-Statistik, also die Verteilung der einzelnen Spikes, beziehungsweise Cluster, liefern. Betrachtet man die Verteilung der Mahalanobis Quadrat-Abstände (Distanzmaß zwischen Punkten in einem mehrdimensionalen Vektorraum) der Spikes, die aus einer Wellenform produziert werden, so findet man eine Abweichung zwischen der erwarteten Verteilung und der empirisch messbaren Verteilung. Dies führt zur Annahme, dass der GMM Algorithmus, der auf einer vermuteten Gauß-Verteilung aufbaut, möglicherweise nicht für diese Aufgabe angemessen ist [18].
In dem Artikel [23], wird ein weiterer Beweis erbracht für die Nicht-Gauß Natur von Spike-Form Statistiken. Desweiteren wird ein alternatives Modell vorgestellt, das die multivariate t-Verteilung anstatt der multivariaten Gauß-Verteilung zur Modellierung des Problems verwendet.
Der hier vorgestellte Algorithmus baut auf dem Standard expectation-maximization (EM) Algorithmus auf, so wie der GMM-Algorithmus. Jedoch benötigt er doppelt so viel Variablen im Vergleich zu GMM

und zusätzlich einen weiteren Berechnungsschritt, um den Freiheitsgrad der t-Verteilung anzupassen. Darüber hinaus muss der Algorithmus eine Anzahl von Aspekten ansprechen, wie zum Beispiel die Bestimmung der Anzahl der gesuchten Cluster bzw. Komponenten, die Dateninitialisierung und die Vermeidung der Konvergenz der lokalen Likelihood Maxima.
Nach Angaben des Autors ist der Algorithmus statistisch plausibel sowie einfach und effektiv bei vielen reellen Datensätzen [23].

Algorithmus:

Der Algorithmus basiert auf den EM-Algorithmus zur Mischung von Gauß-Verteilungen. Ein erster direkter EM-Algorithmus zur Schätzung der Parameter aus Mischungen von multivariaten t-Verteilungen wurde im Jahre 2000 präsentiert [18]. Jedoch setzt der von Shoham vorgeschlagene Algorithmus die EM-Methode in Verbindung mit einem effizienten Modellauswahlschema ein, anstatt EM direkt einzusetzen. Hier wird das „penalized log-likelihood“ mit einem Strafterm der auf dem „minimum message length“-Kriterium aufbaut, maximiert [25]. Daraus ergibt sich für den Logarithmus der Plausibilität der Modellparameter:

$$L = \sum_{i=1}^{n} \log \sum_{j=1}^{g} P_{ij}\pi_j - \left[\frac{N}{2} \sum_{j=1}^{g} \log \frac{n\pi_j}{12} + \frac{g}{2} \log \frac{n}{2} + \frac{g(N+1)}{2} \right]. \tag{3.44}$$

Dabei ist N die Anzahl der Parameter pro Mischkomponente. Dieses L führt dazu, dass im M-Schritt die einzelnen Mischkomponenten gegenseitig um die Datenpunkte konkurrieren. Weiterhin werden diese Komponenten eliminiert, sobald sie singulär werden. Eine Matrix ist singulär, wenn ihre Determinante gleich 0 ist.
Zusätzlich wurde festgestellt, dass die direkte Maximierung der log-likelihood nach π_j den gewünschten Effekt, nämlich die Konvergenz des Algorithmus, mit sich bringt. Dabei sind $\pi_1 \dots \pi_g$ die Wahrscheinlichkeiten des Auftretens der einzelnen Komponenten.

Die Maximierung von L muss zuerst die Bedingung $\sum_{j=1}^{g} \pi_j = 1$ erfüllen. Um das zu erreichen, wird zuerst der Lagrange-Multiplikator angewendet. Somit muss folgendes maximiert werden:

$$L = \sum_{i=1}^{n} \log \sum_{j=1}^{g} P_{ij}\pi_j - \left[\frac{N}{2} \sum_{j=1}^{g} \log \frac{n\pi_j}{12} + \frac{g}{2} \log \frac{n}{2} + \frac{g(N+1)}{2} \right] + \lambda \left[\sum_{j=1}^{g} \pi_j - 1 \right]. \tag{3.45}$$

Durch die Ableitung nach π_j erhalten wir:

$$\sum_{j}\left(\sum\nolimits_{i=1}^{n}\frac{P_{ij}}{\sum_{l=1}^{g}P_{il}\pi_l}-\frac{N}{2\pi_j}+\lambda\right)=0\,. \tag{3.46}$$

Multiplikation mit $\frac{\pi_j}{g}$ und aufsummieren über j ergibt:

$$\frac{1}{g}\sum_{i=1}^{n}\frac{\sum_j P_{ij}\pi_j}{\sum_{l=1}^{g}P_{il}\pi_l}-\sum_{j=1}^{g}\frac{N}{2g}+\frac{\lambda}{g}\sum_{j=1}^{g}\pi_j=\frac{n}{g}-\frac{N}{2}+\frac{\lambda}{g}=0\,. \tag{3.47}$$

Zuletzt wird λ aus der Gleichung (3.49) in der Gleichung (3.48) substituiert. Somit folgt die Formel für die Proportionen:

$$\pi_j=\left(\sum_{i=1}^{n}\frac{P_{ij}\,\pi_j}{\sum_{j=1}^{g}P_{ij}\pi_j}-\frac{N}{2}\right)\Big/\left(n-\frac{gN}{2}\right)\,. \tag{3.48}$$

Dies kann iterativ berechnet werden. Hinzu wird die Bedingung gefügt, dass während jeder Iteration $\pi_j \geq 0\,, j=1\ldots g$ gelten muss. Der Grund hierfür wird aus der Tatsache deutlich, dass es sich bei π_j um Wahrscheinlichkeiten handelt. Daher werden negative Werte nicht akzeptiert [4].

Der gesamte Algorithmus besteht somit aus Wiederholungen des EM-Algorithmus zur Mischung von t-Verteilungen [23] und dem modifizierten M-Schritt zum Maximieren der Gleichung (3.46). Dazu werden zwei Hilfsvariablen angewendet:

z_{ij} – Zuweisung von Spike i zur Komponente d.h. Neuron j $(0 \leq z_{ij} \leq 1)$.

u_{ij}– Gewichte, zeigen die charakteristische Eigenart von Spike i zur Komponente j.

Diese Variablen werden im E-Schritt ausgerechnet und dazu verwendet, neue Parameter im M-Schritt abzuschätzen.

k-ter Schritt des Algorithmus

Um die Parallelisierung zu erleichtern, wird die vorgeschlagene Reihenfolge der einzelnen Berechnungen verändert. Die Funktionalität des Algorithmus wird jedoch nicht verändert, da es sich hierbei um eine Schleife handelt.

Legende:

- $p :=$ Raumdimension.
- $g :=$ aktuelle Anzahl der Cluster.
- $g_{min} :=$ Minimale Anzahl an Cluster Mittelpunkten.
- $g_{max} :=$ Maximale Anzahl an gesuchten Clustermittelpunkten.
- $n :=$ Anzahl der Datenpunkte.
- $N :=$ Anzahl der Parameter pro Mischkomponente.
- $\pi_j :=$ relative Häufigkeit von Cluster j bzw. Clusterproportionen.
- $\Sigma_j :=$ Kovarianzmatrix von Cluster j.
- $\Delta_j := \sqrt{\det(\Sigma_j)}$.
- $v :=$ Freiheitsgradparameter.
- $L :=$ log-likelihood Funktion.

Initialisierung:

- Clustering Methode z.B. K-means verwenden, um die Cluster Mittelpunkte $\vec{\mu}_0, \dots, \vec{\mu}_{g-1}$ zu finden.
- $g = g_{max}$
- Setze :
  - $\pi_1, \dots, \pi_g = \frac{1}{g_{max}}$ $\quad \pi \in \mathbb{R}^g$
  - $\Sigma_1, \dots, \Sigma_g = I$ $\quad \Sigma_j \in \mathbb{R}^{g \times g}$ $\Sigma \in (\mathbb{R}^{g \times g})^g$
  - $v = 50$
  - $L_{max} = -\infty$
  - N
  - $x = \{\vec{x}_0, \dots, \vec{x}_{n-1}\}$ $\quad x \in \mathbb{R}^{p \times n}$
  - $\mu = \{\vec{\mu}_0, \dots, \vec{\mu}_{g-1}\}$ $\quad x \in \mathbb{R}^{p \times g}$
  - $\delta_{ij} := \delta\left(\vec{x}_i, \vec{\mu}_j; \Sigma_j\right) = \left(\vec{x}_i - \vec{\mu}_j\right)^T I^{-1} (\vec{x}_i - \vec{\mu}_j)$
- **Solange** $g \geq g_{min}$
- **E-Schritt**
- Gewichtetes Abstandsquadrat zwischen i-ten Datenpunkt und j-tem Cluster
- (Mahalanobis Abstand)

$$\delta_{ij} := \delta\left(x_i, \mu_j; \Sigma_j\right) = \left(\vec{x}_i - \vec{\mu}_j\right)^T \Sigma_j^{-1} (\vec{x}_i - \vec{\mu}_j) \tag{3.49}$$

- Wahrscheinlichkeit, dass der i-te Datenpunkt zum j-ten Cluster gehört.

$$P_{ij} = \frac{\Gamma\left(\frac{v+p}{2}\right)}{\Gamma\left(\frac{v}{2}\right)(\pi v)^{p/2} \Delta_j} \frac{1}{\left(1 + \frac{\delta_{ij}}{v}\right)^{(v+p)/2}} \tag{3.50}$$

- L aktualisieren mit

$$L_p = \sum_{i=1}^{n} \log \sum_{j=1}^{g} P_{ij}\pi_j - \left[\frac{N}{2}\sum_{j=1}^{g} \log\frac{n\pi_j}{12} + \frac{g}{2}\log\frac{n}{2} + \frac{g(N+1)}{2}\right] \tag{3.51}$$

- **Falls** $(\Delta L < 0.1 \;\&\; \Delta v < 10^{-2})$: Konvergenz erreicht

  Return;

  Zugehörigkeiten aktualisieren

1)
$$\hat{z}_{ij} = \frac{P_{ij}\pi_j}{\sum_{l=1}^{g} P_{il}\pi_l} \tag{3.52}$$

und die Gewichte

2)
$$\hat{u}_{ij} = \frac{p + v}{\delta_{ij} + v} \tag{3.53}$$

Um den Parameter v zu bestimmen wird y benötigt

$$y \equiv -\sum_{i=1}^{n}\sum_{j=1}^{g} \hat{z}_{ij}\left[\psi\left(\frac{p + v_{old}}{2}\right) + \log\left(\frac{2}{\delta_{ij} + v_{old}}\right) - \hat{u}_{ij}\right]/n \tag{3.54}$$

M-Schritt

Solange $\left|\sum_{\mathrm{j}=1}^{\mathrm{g}} \pi_{\mathrm{j}} - 1\right| > 10^{-4}$

Für $\mathrm{j} = 1{:}\,\mathrm{g}$

Aktualisiere π_j mit

$$\pi_j^{(k)} = \frac{\max\left(\sum_{i=1}^{n} \frac{P_{ij}\pi_j{}^{(k-1)}}{\sum_{l=1}^{g} P_{il}\,\pi_l{}^{(k-1)}} - \frac{N}{2}, 0\right)}{n - \frac{gN}{2}} \tag{3.55}$$

Aktualisiere g mit der Anzahl alle $\pi_j > 0$

Ende

Alle Komponenten mit $\pi_j = 0$ entfernen

Bestimme neue Clusterzentren

$$\mu_j = \frac{\sum_{i=1}^{n} \hat{z}_{ij}\hat{u}_{ij}\vec{x}_i}{\sum_{i=1}^{n} \hat{z}_{ij}\hat{u}_{ij}} \tag{3.56}$$

und die ihnen zugeordneten Standardabweichungen

$$\Sigma_j = \frac{\sum_{i=1}^{n}\left(\hat{z}_{ij}\hat{u}_{ij}\right)(\vec{x}_i - \vec{\mu}_j)\left(\vec{x}_i - \vec{\mu}_j\right)^T}{\sum_{i=1}^{n}\hat{z}_{ij}\hat{u}_{ij}} \tag{3.57}$$

An dieser Stelle verwendet man die Cholesky Zerlegung und somit folgt

$$\Sigma_j = L^T L \tag{3.58}$$

und

$$\Delta_j = \sqrt{\det\left(\Sigma_j\right)} = \prod_i L_{ii} \tag{3.59}$$

Parameter v aktualisieren

$$v_{new} = \frac{2}{y + \log y - 1} + 0.0416\left(1 + \operatorname{erf}\left(0.6594 * \log\left(\frac{2.1971}{y + \log y - 1}\right)\right)\right) \tag{3.60}$$

Sei $\mathrm{L} > \mathrm{L}_{\max}$

- $L_{max} = L$
- Parameter $\{\pi, \mu, \Sigma\}$ als „optimal“ speichern
- Die kleinste Komponente auf null setzen.
- $g = g - 1.$

Sonst

Break;

Ende

3.4 Schritte zur CUDA Parallelisierung

Um eine effiziente Implementierung des Algorithmus auf der Grafikhardware zu ermöglichen, müssen Elementzugriffe soweit wie möglich eliminiert werden. Die Elementzugriffe sollten durch High-Level Operationen, wie zum Beispiel Matrix-Vektor oder besser Matrix-Matrix Operationen ersetzt werden. Solche Operationen sind in dem „Basic Linear Algebra Subprograms“ (BLAS) Standard enthalten und somit auch in der CUDA Implementierung CUBLAS, die den Zugriff auf Rechenmittel der NVIDIA GPUs erlaubt.

Das Grundmodell zum Einsatz der CUBLAS-Bibliothek, ist die Erstellung von Matrix- und Vektorobjekten, die auf dem GPU-Speicher allokiert werden. Diese Objekte werden mit Daten gefüllt und durch eine Serie von CUBLAS-Funktionen werden die gewünschten Berechnungen durchgeführt. Zuletzt werden die Daten aus dem GPU-Speicher in den Hostspeicher zurückkopiert. Dafür werden für CUBLAS Wrapperfunktionen

geschrieben, um Objekte zu erstellen beziehungsweise zu zerstören. Weiterhin bietet CUBLAS Hilfsfunktionen für das Kopieren der Daten. Aus Kompatibilitätsgründen mit bestehenden FORTRAN Umgebungen werden in der CUBLAS-Bibliothek eine spaltenweise Abspeicherung von mehrdimensionalen Arrays und 1-basierte Indizierung aller Arrays eingesetzt. Somit können C und C++Programme, die row-major Speicherung benutzen, nicht direkt die CUBLAS Methoden anwenden, es sei denn man transponiert alle vorliegenden Matrizen [16].

3.4.1 Umwandlung von Einzelelementzugriffen in High-Level Operationen

Betrachten wir Gleichung (3.49) aus dem vorherigen Algorithmus

$$\delta_{ij} \coloneqq \delta\left(x_{i,}\mu_j; \Sigma_j\right) = \left(\vec{x}_i - \vec{\mu}_j\right)^T \Sigma_j^{-1}\left(\vec{x}_i - \vec{\mu}_j\right).$$

Diese Gleichung kann in drei Teile aufgeteilt werden:

1. Die Berechnung von $E_1 = \left(\vec{x}_i - \vec{\mu}_j\right)$, wobei $x \in \mathbb{R}^{n\times p}$ und $\mu \in \mathbb{R}^{g\times p}$ als CUBLAS-Matrizen vorliegen müssen, wobei n die Anzahl der Datenpunkte, g der Anzahl der Cluster und p die Raumdimension, sind.
2. Multiplikation von Σ_j^{-1} mit der Ergebnismatrix E_1 aus $(\vec{x}_i - \vec{\mu}_j)$.
   - $E_2 = \Sigma_j^{-1}\left(\vec{x}_i - \vec{\mu}_j\right), \quad E_2 \in \mathbb{R}^{p\times n}$
3. Skalarmultiplikation der Zeilen aus der transponierten Ergebnismatrix aus (1) mit den Spalten der Ergebnismatrix aus (2).
   - $\delta_{ij} = {E_1^T}_i \cdot {E_2}_j$

Als nächstes betrachten wir die Gleichung (3.55):

$$\hat{z}_{ij} = \frac{P_{ij}\pi_j}{\sum_{l=1}^{g} P_{il}\pi_l}.$$

Deutlich erkennen wir das Matrix-Vektor Produkt aus den Matrizen P und π, das sich im Nenner befindet. Hier sind die Einzelelementzugriffe in diesem Falle nicht nötig, da die Spaltensumme gebildet werden muss. Dies wird durch das Matrix-Vektor Produkt erledigt.

Daraus folgt:

1. $C_i = P_{ij} \cdot \pi_j$

und Gleichung (3.52) wird zu:

$$\hat{z}_{ij} = \frac{P_{ij}\pi_j}{C_i}. \tag{3.61}$$

Das gleiche Prinzip kann auch auf den Zähler der Gleichung (3. 56) angewendet werden:

$$\vec{\mu}_j = \frac{\sum_{i=1}^{n} \hat{z}_{ij}\hat{u}_{ij}\vec{x}_i}{\sum_{i=1}^{n} \hat{z}_{ij}\hat{u}_{ij}} .$$

Hierbei berechnen wir

1. das Ergebnis der elementweisen Multiplikation von $\hat{z}_{ij}$ mit $\hat{u}_{ij}$. Diese Berechnung liefert uns $q_{ij} = \hat{z}_{ij} \cdot \hat{u}_{ij}$.
2. Matrix-Matrix Multiplikation von q_{ij} mit x. Dies nimmt die Berechnung der Zeilensumme ab.

Somit ist Gleichung (3.59):

$$\vec{\mu}_j = \frac{q^T \cdot x}{f_i} \tag{3.62}$$

mit

$$f_i = \sum_{i=1}^{n} \hat{z}_{ij}\hat{u}_{ij} . \tag{3.63}$$

Weiterhin können wir die Gleichung (3.57)

$$\Sigma_j = \frac{\sum_{i=1}^{n}\left(\hat{z}_{ij}\hat{u}_{ij}\right)(\vec{x}_i - \vec{\mu}_j)\left(\vec{x}_i - \vec{\mu}_j\right)^T}{\sum_{i=1}^{n} \hat{z}_{ij}\hat{u}_{ij}}$$

so umwandeln, dass es in zwei Schritten berechnet wird:

1. Berechnung von

$$M = \sqrt{\frac{\sum_{i=1}^{n}\left(\hat{z}_{ij}\hat{u}_{ij}\right)}{\sum_{i=1}^{n} \hat{z}_{ij}\hat{u}_{ij}}} \cdot (\vec{x}_i - \vec{\mu}_j), \tag{3.64}$$

wobei die im Zähler vorhandenen $\hat{z}_{ij}\hat{u}_{ij}$ durch q_{ij} und $\left(\vec{x}_i - \vec{\mu}_j\right)$ durch E_1 und somit auch die Zeilensumme durch das Skalarprodukt ersetzt werden können. Die Wurzel muss aufgrund von Schritt (2) gezogen werden.

$$M_j = \sqrt{\frac{q_{ij}}{\sum_{i=1}^{n} \hat{z}_{ij}\hat{u}_{ij}}} \cdot E_1 \tag{3.65}$$

2.

$$\Sigma_j = M_j^T \cdot M_j \tag{3.66}$$

Hier wird deutlich, warum die Wurzel gezogen werden muss. Denn durch das Matrix-Matrix Produkt werden die Matrix Elemente quadriert und die Wurzel hat die Aufgabe, eben dies wieder aufzuheben.
Die weiteren Berechnungen und Zwischenrechnungen werden durch CUDA Kernels erledigt. Dies ist auch nur möglich, wenn alle Daten im Grafikspeicher liegen und somit das mehrmalige Kopieren der Daten gespart wird.

3.5 CUDA Kernel

Wie in den Grundlagen beschrieben wird, handelt es bei einem CUDA Kernel um den Programmteil, der die Berechnung auf der GPU ausführt. Diese werden jedoch nicht direkt aus dem Host-Teil des Programms aufgerufen, sondern um die Portierbarkeit auf andere parallele Plattformen zu erleichtern, in sogenannten „Kernel-Wrapper“ Funktionen abgekapselt.
Durch den Einsatz von expliziter Template Instanziierung, kann erreicht werden, dass die „Kernel-Wrapper“ mit verschiedenen Datentypen aufgerufen werden können. Dafür werden alle Funktionen deklariert beziehungsweise definiert, bevor die Klasse am Ende explizit instanziiert wird [24]. Die „Kernel-Wrapper definieren die Anzahl der *threads* und *threadsblocks* der Kernel und allokieren auch Speicher dafür, wenn sie benötigt werden.
Durch die „Kernel-Wrapper“ schaffen wir somit eine Trennung zwischen GPU und CPU Code, die dem einzelnen Compiler erlaubt, den dazu gehörenden Code zu kompilieren.

3.6 Kernel Implementierung

Bei den meisten hier implementierten *Kernel* handelt es sich um überschaubare, unkomplizierte Funktionen, die die Aufgabe haben, die benötigten Elementzugriffe auf der Grafikhardware auf einfache Weise durchzuführen. Es kommen eindimensionale Threadblöcke zum Einsatz, mit 512 zu startenden *threads* pro threadblock. Die Blockgitter werden entweder eindimensional oder zweidimensional angelegt, je nachdem ob es sich bei den Eingabedaten um ein- beziehungsweise zweidimensionale Vektoren handelt. Durch den Einsatz von zweidimensionalen Blockgittern kann der Zugriff auf Daten in den Eingabematrizen erleichtert werden. So beinhaltet die Gitterdimension so viele threadblöcke, wie es benötigt wird, um alle Datenpunkte abzudecken. Die zweite Dimension ist dafür verantwortlich die Spalten der Eingabematrix zu realisieren. Auf diese *Kernel* werden wir hier nicht weiter eingehen, stattdessen folgt eine Beschreibung der mehr anspruchsvolleren *Kernel* [15][17].

3.6.1 Assemble_z_ij Kernel

Um die Zuweisungen der Datenpunkte zu den Clustermittelpunkten zu berechnen, wird nach dem oben beschriebenen Algorithmus die folgende Gleichung benötigt:

$$\hat{z}_{ij} = \frac{P_{ij}\pi_j}{\sum_{l=1}^{g} P_{il}\pi_l} \tag{3.52}$$

mit P_{ij} als die Wahrscheinlichkeit, dass der Datenpunkt i zum j-ten Clustermittelpunkt gehört, und dem Vektor π_j , der die Proportionen der einzelnen Komponenten beinhaltet.

Der Kernel bekommt folgende Parameter übergeben:

die Matrix P_{ij} und die Vektoren π und c, wobei der Vektor c die Spaltensumme des Nenners der Gleichung (3.55) beinhaltet. Weiterhin werden die Anzahl der Datenpunkte und die Anzahl der gesuchten Clustermittelpunkte übergeben. Der Kernel wird mit zweidimensionalen threadblöcken ausgeführt, die in der x-Dimension ein Mehrfaches von 32 *threads* starten. Hier wurde die Zahl 32 gewählt, da ein *thread*-warp aus genau 32 *threads* besteht. Dabei handelt es sich um eine *thread* Gruppe, die parallel ausgeführt werden kann. In der y-Dimension werden eine Anzahl von *threads* gestartet, die der Anzahl der gesuchten Cluster entspricht, beziehungsweise die Anzahl der Spalten der P_{ij} Matrix. Beim Blockgitter handelt es sich um ein eindimensionales Gitter mit so vielen Blöcken, wie zur Bearbeitung alle Datenpunkte benötigt werden.

Zuerst werden die Vektoren π und c in das *Shared Memory* des Multiprozessors geladen. Dies dient dazu, dass der Zugriff auf diese Daten beschleunigt wird. Denn alle threads können auf das *shared Memory* zugreifen und müssen nicht einzeln die Daten aus dem globalen Speicher laden.

Danach folgt die eigentliche Berechnung der Matrix $\hat{z}_{ij}$ nach der Gleichung (3.52). Dafür übernehmen die gestarteten *threads* das Laden der Matrixelemente (vgl. Anhang A.)

3.6.2 Sum_array Kernel

Dieser Kernel hat die Aufgabe, die Elemente eines Arrays beliebiger Größe aufzusummieren. Dabei ist eine solche Reduktionsoperation in der Theorie mit Hilfe von CUDA einfach zu realisieren, aber praktisch ist sie schwer richtig zu implementieren. Im CUDA SDK findet man verschiedene Implementierungen solcher Operationen mit unterschiedlicher Effizienz.

Bei unserer Implementierung kommen ein-dimensionale Threadblöcke und Gitter zum Einsatz, mit 512 *threads* pro Block und so vielen Threadblöcke, wie zur Erfassung alle Elemente benötigt werden. Innerhalb des Kernels werden die Einzelelemente in das *shared Memory* geladen und während des Ladens direkt aufsummiert und somit auf 32 Elemente reduziert. Diese letzten 32 Elemente werden durch eine *device*-Funktion dann aufsummiert. Das geschieht simultan, denn 32 *threads* bilden ein Warp. Die Reduktion erfolgt für jeden einzelnen Block. Somit ist es noch nötig, die Summen der einzelnen

Blöcke aufzusummieren. Hier wird das hostseitig erledigt, da die Blöcke nicht gegenseitig auf die Daten andere *Blöcke* zugreifen können.

Diesen Kernel verwenden wir an verschiedenen Stellen im Algorithmus, um die Summe von Vektoren, beziehungsweise Matrizen zu berechnen. Um die Spaltensumme einer Matrix zu berechnen, wird der Kernel einfach mehrmals asynchron gestartet, da die Spalten eine Matrix unabhängig voneinander sind.

4 Laufzeit- und Effizienzanalyse der CUDA-Implementierung

Dieses Kapitel soll die Frage beantworten, ob sich der erhöhte Programmieraufwand mit CUDA lohnt. Als objektives Kriterium dient dazu die relative Auslastung der theoretisch möglichen Speichertransferraten und Rechenleistung durch die selbstgeschriebenen Kernel. Die Ergebnisse sind am Ende des Kapitels in einem Übersichtsschaubild zusammengefasst.

Um die Performanz des implementierten Algorithmus zu bewerten, müssen zuerst die allgemeinen Operationen aufgeteilt werden in zwei Kategorien.

Erstens die auf BLAS abbildbaren Operationen, deren Leistung durch die BLAS-Routinen vorgegeben ist. Zum Beispiel Matrix-Matrix beziehungsweise Matrix-Vektor Multiplikationen. Diese Operationen haben folgende Komplexitäten:

Matrix-Matrix Multiplikation: $\boldsymbol{\mathcal{O}(n^3)}$.

Matrix-Vektor Multiplikation: $\boldsymbol{\mathcal{O}(n^2)}$.

Die zweite Kategorie besteht aus Operationen, die einen eigenen CUDA Kernel benötigen, da keine entsprechende BLAS-Funktion existiert. Bei diesen Operationen müssen Performanztests durchgeführt werden, um festzustellen, ob eine CUDA Implementierung dieser Operationen sich lohnt. Natürlich existieren auch Operationen, die eine Mischung aus den beiden Kategorien darstellen.

Desweiteren müssen wir die Hardware genauer betrachten, auf der die Simulation durchgeführt wird. Hierbei handelt es um eine Nvidia Tesla C2070 Karte mit folgender Produktspezifikation [14]:

Tabelle 1: Spezifikation der Testhardware

Peak double precision floating point performance	515 Gigaflops
Peak single precision floating point performance	1030 Gigaflops
CUDA cores	448
Memory size (GDDR5)	6 GigaBytes
Memory bandwidth (ECC off)	144 GBytes/sec
Memory bandwidth (ECC on)	Ca. 120 GBytes/sec
Intel Xeon E5520 Max Memory bandwidth	25,6 GBytes/s

Dabei muss man beachten, dass die Speichertransferrate mit aktiviertem Error-Correcting Code (ECC) ca. 120 GBytes/sec beträgt.

Weiterhin muss man, um die Performanz der eigenen Kernel zu messen, die Kosten der verschiedenen Operationen in Betracht ziehen.

So kosten Zugriffe auf den globalen Speicher ca. 1000 Taktzyklen, was ca. 1 µs entspricht. Rechenoperationen, wie zum Beispiel Addition, Multiplikation etc., kosten gerade mal einen Taktzyklus und Spezialoperationen wie Wurzelziehen entsprechend 8 Taktzyklen [17], da dieses von seperaten „special function units“ übernommen wird. Dieser brauchen zwar pro Operation nur einen Takt, dafür

gibt es aber nur SFUs pro Multiprozessor, wohingegen für die gewöhnlichen Rechenoperationen 32 CUDA-Kerne pro Multiprozessor zur Verfügung stehen.

Für die Messung der Kernel-Laufzeiten wurden cudaEvents verwendet. Dabei handelt es sich um Timer-Objekte, die von der CUDA zur Verfügung gestellt werden, für die genaue Überwachung der Grafikhardware (vgl. CUDA Programming Guide 4.0, Abschn. 3.2.5.6)[15].

Im nächsten Abschnitt betrachten wir den Aufbau der einzelnen Kernel und deren Performanz.

4.1 Subtract Array Kernel

```
template <typename T>
        __global__ void __subtract_array_from_matrix(int n, int p,
                                                      int j, int g,
                                                      const T * X,
                                                      const T * mu_j,
                                                      T * M)
{
    int r = blockIdx.y * blockDim.x + threadIdx.x;
    int c = blockIdx.x;

    if(r < n && c < p){
        int global_index = c*n+r;
        M[global_index] = X[global_index]-mu_j[j+c*g];
    }
```

Dieser Kernel hat die Aufgabe, einen vorgegebenen Vektor von jeder Zeile einer Matrix zu subtrahieren. In diesem Falle den Vektor mu_j von der Matrix X. Das Ergebnis wird in die Matrix M geschrieben.

Die Zugriffe auf den globalen Speicher sind grün und die Rechenoperationen grau markiert. Somit sind es zwei Zugriffe auf den globalen Speicher, da der Zugriff auf der Matrix mu_j nicht mitgezählt wird aufgrund von Cache-Effekten. In der Praxis werden typisch 10 Cluster mit je 100 Komponenten eingesetzt, die dazu führen, dass die mu_j Matrix eine Größe von ca. 8KB hat. Da für jedes SM (Shared Multiprocessor) mindestens 16KB L1 Cache zur verfügung stehen, können wir die Speicherzugriffe der mu_j ignorieren. Weiterhin wird eine Rechenoperation pro Eintrag der Matrix M $\in \mathbb{R}^{n\times p}$ sowie vier FMADs (fused multiply-add Operationen) für die Indexberechnung benötigt.

Daraus folgt folgende Berechnung der Speichertransferrate:

$$\frac{(Anzahl\ der\ Elemente * Anzahl\ der\ Speicherzugriffe * Datentypgröße)}{Zeit\ in\ Sekunden} / (1024^3) \qquad (4.67)$$

Für eine Testrechnung mit einer Matrix mit 1000000 Zeilen und 64 Spalten, die mit Gleitkommazahlen mit doppelter Genauigkeit durchgeführt wird, erhalten wir $76.984\ GBytes/s$.

Ähnlich folgt für die Berechnung der GFlops für die Rechenoperationen:

$$\frac{(Anzahl\ der\ Elemente * Anzahl\ der\ Rechenoperationen)}{Zeit\ in\ Sekunden} / (1024^3)\ , \tag{4.68}$$

wobei man zwischen normalen Rechenoperationen, wie Addition oder Multiplikation und Spezialoperationen wie Division oder das Logarithmieren, unterscheidet.

Durch die normalen Rechenoperationen, die im Kernel enthalten sind, erhalten wir $3.007\ GFlops/s$.

Das liegt daran, dass gerademal vier Rechenoperationen durchgeführt werden, aber mehrere Speicherzugriffe erfolgen.

4.2 Matrix-Matrix Skalar Kernel

```
template <typename T>
        __global__ void __MM_scalar(int n, int p,
                                    const T * A,
                                    const T * B,
                                    T * result)
{
    int r = blockIdx.x * blockDim.x + threadIdx.x;

    T sum = 0.;

    if(r < n){
        for(int j= 0; j<p; j++){
            sum += A[j*n+r]*B[r*p+j];
        }
        result[r] = sum;
    }
}
```

Hierbei handelt es um einen Kernel, der die Aufgabe besitzt, das Skalarprodukt aus den Zeilen der Matrix A mit den Spalten der Matrix B zu berechnen.

Die Problemgröße beträgt bei diesem Test ist n = 1000000, p = 64 und g = 8.

Bei der Berechnung der Speichertransferrate beachten wir zuerst die zwei Speicherzugriffe in der for-Schleife, die über die Dimension der Matrix läuft. Somit erhalten wir 2*p Speicherzugriffe aus der for-Schleife und einen Speicherzugriff, um die Summe in den Ergebnisvektor zu schreiben, der lediglich die Größe n besitzt.

Daraus folgt eine Speichertransferrate von $598.423\ Gbytes/s$. Diese Zahl liegt weit über der maximal möglichen Transferrate der Hardware. Ursache dafür sind Cache-Effekte, die entstehen, wenn Daten schon in einem früheren Schritt geladen wurden und somit nicht nochmal aus dem globalen Speicher geladen werden müssen.

Bei der Berechnung der GFlops erhalten wir $150.172\ GFlops/s$, da es dabei drei FMADs pro Schleifeniteration für die Berechnung der Zwischensumme und eine weitere Addition für die

Schleifenzählerinkrementierung. Die Berechnung des Zeilenindexes erfordert eine weitere FMAD Operation, also insgesamt 4*p+1 normale Rechenoperationen.

4.3 Kernel zur Determinantenberechnung

```
template <typename T>
        __global__ void __Det(int p, int MS,
                              const T * A,
                              T * result)
{
    T det = 1.;

    for(int i = 0; i<p; i++){
        det *= A[i*MS+i];
    }

    *result = det;

}
```

Dieser Kernel besitzt die einfache Aufgabe, das Produkt der Diagonalelemente der angegebenen Matrix zu berechnen. Da es sich bei der Matrix um die Matrix L aus der Cholesky-Zerlegung $A = L^T L$ handelt, resultiert die Multiplikation der Diagonalelemente in der Berechnung der Determinante. Die Problemgröße beträgt p = 64.

Hierbei haben wir p+1 Speicherzugriffe, die aus den p Zugriffen aus der for-Schleife und dem Zugriff zum Ergebnisschreiben resultieren. Dies führt zu einer Speichertransferrate von $38.133\ Gbytes/s$.

Ähnlich haben wir 3*p Rechenoperationen aus der for-Schleife, die zu $4.686\ GFlops/s$ führen. Dabei ist zu beachten, dass die Problemgröße für p 8 beträgt mit nur einem gestarteten *thread*.

4.4 Assemble P_ij Matrix

```
template <typename T>
        __global__ void __P_ij(T gamma,
                               T v,
                               int p, int n, int g,
                               const T * Delta,
                               const T * delta,
                               T * P_ij)
{
    int i = blockIdx.y * blockDim.x + threadIdx.x;
    int j = blockIdx.x;


    if(i < n && j < g){
        int global_index = j*n+i;
        P_ij[global_index] =
        gamma/Delta[j]/(pow((1.+delta[global_index]/v),(v+p)*0.5));
```

Der Kernel berechnet die Matrix P_ij aus der Gleichung (3.50)

$$P_{ij} = \frac{\Gamma(\frac{v+p}{2})}{\Gamma\left(\frac{v}{2}\right)(\pi v)^{p/2}\Delta_j} \frac{1}{\left(1+\frac{\delta_{ij}}{v}\right)^{(v+p)/2}}.$$

Die Problemgröße bei diesem Test ist: n = 1000000, p = 64 und g = 8.

Die Speicherzugriffe begrenzen sich auf die Matrizen P_{ij} , δ_{ij} da die restlichen Werte dem Kernel direkt übergeben werden. Analog zu mu_j aus dem Subtract Array Kernel, wird das Δ_j im L1 Cache landen. Somit haben wir zwei Speicherzugriffe, die zu einer Speichertransferrate von $15.537\ Gbytes/s$ führen.

Weiterhin werden vier normale Rechenoperationen benötigt, die eine Rechenleistung von $3.884\ GFlops/s$ erbringen. Die vier Spezialoperation, nämlich das Potenzieren und die Division haben wiederum eine Rechenleistung von $3.884\ GFLops/s$.

4.5 Assemble u_ij Kernel

```
template <typename T>
        __global__ void __u_ij(T v,
                               int p, int n, int g,
                               const T * delta,
                               T * u_ij)
{
    int i = blockIdx.y * blockDim.x + threadIdx.x;
    int j = blockIdx.x;


    if(i < n && j < g){
        int global_index = j*n+i;
        u_ij[global_index] = (p+2)/(delta[global_index]+v);
    }
```

Hierbei handelt es um den Kernel, der für die Berechnung der Gleichung (3.53) zuständig ist.

$$\hat{u}_{ij} = \frac{p+v}{\delta_{ij}+v}$$

mit der Problemgröße p = 64, n = 1000000 und g = 8.

Der Kernel enthält zwei Zugriffe auf den globalen Speicher und somit eine Speichertransferrate von $77.945\ Gbytes/s$.

Desweiteren sind vier normale Rechenoperationen nötig, um die Berechnung durchzuführen. Diese führen zu $19.486\ GFlops/s$. Weiterhin wird noch eine Spezialoperation benötigt, die eine Rechenleistung von $4.871\ GFlops/s$ erbringt.

4.6 Assemble z_ij Kernel

```
template <typename T>
        __global__ void __z_ij(int n, int g,
                               const T * P_ij,
                               const T * c,
                               const T * pi,
                               T * z_ij)
{
    __shared__ T cs[512];
    __shared__ T pi_s[32];

    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int j = threadIdx.y;

    if (j == 0){
        cs[threadIdx.x] = c[i];
    }

    if(threadIdx.x < g && j == 0){
        pi_s[threadIdx.x] = pi[threadIdx.x];
    }

    __syncthreads();

    if(i < n && j < g){
        int global_index = j*n+i;
        z_ij[global_index] =
      (P_ij[global_index]*pi_s[j])/cs[threadIdx.x];
}
```

Ähnlich wie bei dem vorherigen Kernel berechnet dieser Kernel die Gleichung (3. 52) mit

$$\hat{z}_{ij} = \frac{P_{ij}\pi_j}{\sum_{l=1}^{g} P_{il}\pi_l} \quad .$$

Der Nenner wird dem Kernel als Vektor direkt übergeben und muss nicht vom Kernel berechnet werden. Somit begrenzen sich die Speicherzugriffe auf zwei Zugriffe, um aus der Matrix P_ij zu laden und in die Matrix z_ij zu schreiben sowie die Speicherzugriffe, um die Elemente des Vektors c zu laden. Die Problemgröße hier beträgt n = 1000000 und g = 8.

Insgesamt folgt eine Speichertransferrate von $71.172\ Gbytes/s$.

Weiterhin beträgt die Anzahl der normalen Rechenoperationen drei und somit eine Leistung von $8.859\ GFlops/s$. Die Rechenleistung der Spezialoperationen beträgt wiederrum $2.953\ GFlops/s$, da nur eine Spezialoperation nötig ist.

4.7 Calculate y Kernel

```
template <typename T>
        __global__ void __y(int n, int g, T v, T digamma,
                            const T * z_ij,
                            const T * delta_ij,
                            const T * u_ij,
                            T * results)
{
    __shared__ T sum[256];

    sum[threadIdx.x] = 0.;
    __syncthreads();


    int i = threadIdx.x + blockDim.x * blockIdx.x;

    if(i>=n)
        return;

    for(int j = 0; j<g; j++){
            int global_index = j*n+i;
            sum[threadIdx.x] += z_ij[global_index]*(digamma+
            log(2/(delta_ij[global_index]+v))-u_ij[global_index]);
    }
    __syncthreads();

    reduction(sum, threadIdx.x);

    if(threadIdx.x == 0){
        results[blockIdx.x]= sum[0];
    }
}
```

```
template<typename T>
__device__ void reduction(volatile T * sum, int tid)
{
    for(int k = blockDim.x/2; k>0; k/=2){
        if(tid < k){
            sum[tid] += sum[tid + k];
        }
        __syncthreads();
    }
}
```

Hierbei handelt es sich um den Kernel für die Berechnung der y Variable, die wiederum zur Berechnung des Freiheitsgradparameters v der t-Verteilung benötigt wird, also die Gleichung (3.54):

$$y \equiv \sum_{i=1}^{n}\sum_{j=1}^{g} \hat{z}_{ij}\left[\psi\left(\frac{p+v_{old}}{2}\right)+\log\left(\frac{2}{\delta_{ij}+v_{old}}\right)-\hat{u}_{ij}\right].$$

Weiterhin beträgt die Problemgröße n = 1000000 und g = 8.

Bei den Speicherzugriffen handelt es sich um das Laden aus den Matrizen z_{ij} , u_{ij} und δ_{ij} . Das Schreiben der Ergebnisse in den Ergebnisvektor kann vernachlässigt werden, weil dieser nur die blockweisen Partialsummen aufsammelt, deren Anzahl deutlich kleiner als die Problemgröße ist. Somit erfolgt die

Summe von drei Zugriffen auf den globalen Speicher und eine Speichertransferrate von $58.203\ Gbytes/s$.

Die Anzahl der Rechenoperationen folgt aus den fünf normalen Operationen die durch die for-Schleife so oft ausgeführt werden, wie Cluster vorhanden sind. Hinzu wird die Addition in der *reduction()*-Funktion hinzuaddiert. Diese Operationen erreichen eine Rechenleistung von $126.107\ GFlops/s$.

Desweiteren erreichen die Spezialoperationen des Logarithmierens und der Berechnung des Kehrwerts, die auch in der Schleife ausgeführt werden, eine Leistung von $48.502\ GFlops/s$ für diesen Kernel.

4.8 Sum Array Kernel

```
template <typename T>
        __global__ void __sum_array(int n,
                                    const T * input,
                                    T * result)
{
    __shared__ T sum[512];


    int i = threadIdx.x + blockDim.x * blockIdx.x;

    if(i>=n)
        return;

    sum[threadIdx.x] = input[i];
    __syncthreads();

    reduction(sum, threadIdx.x);

    if(threadIdx.x == 0){
        result[blockIdx.x]= sum[0];
    }

}
```

Der Kernel berechnet die Summe der Elemente eines vorgegebenen Vektors mit n = 1000000.

Die Speicherzugriffe begrenzen sich auf das Laden der Elemente. Aus diesem Speicherzugriff erfolgt eine Speichertransferrate von $11.062\ Gbytes/s$.

Der Kernel erreicht eine Leistung von $2.765\ GFlops/s$, die sich aus der Addition der Einzelelemente plus die Additionsoperation in der *reduction()*-Funktion ergeben.

4.9 Sum Array2D Kernel

```
template <typename T>
        __global__ void __sum_array2D(int n, int g,
                                       const T * z_ij,
                                       const T * u_ij,
                                       T * output)
{
    if(i>=n)
        return;

    __shared__ T sum[512];

    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int j = blockIdx.y;

    int global_index = j*n+i;
    sum[threadIdx.x] = z_ij[global_index]* u_ij[global_index];
    __syncthreads();

    reduction(sum, threadIdx.x);

    if(threadIdx.x == 0){
        output[blockIdx.y*gridDim.x+blockIdx.x]= sum[0];
    }
}
```

Ähnlich dem vorherigen Kernel, berechnet dieser Kernel die Spaltensumme der Matrix, die aus der Multiplikation der Elemente der vorgegebenen z_{ij} und u_{ij} Matrizen entsteht.

Hier beträgt die Problemgröße n = 1000000 und g = 8.

Die Speichertransferrate berechnet sich im Wesentlichen aus den zwei Zugriffen, um die Daten aus den Matrizen zu laden, womit eine Bandbreite von $36.733\ Gbytes/s$ folgt.

Die Anzahl der Rechenoperationen beträgt zwei Operationen für die Indizes, eine für die Multiplikation der Elemente, hinzuaddiert die Additionsoperation aus dem *reduction()*-Funktion. Daraus folgt eine Leistung von $6.887\ GFlops/s$.

4.10 Matrix_Matrix Multiplikation Kernel

```
template <typename T>
        __global__ void __MxM(int n, int g,
                               const T * z_ij,
                               const T * u_ij,
                               T * q_ij)
{
    int i = blockIdx.y * blockDim.x + threadIdx.x;
    int j = blockIdx.x;


    if(i < n && j < g){
        int global_index = j*n+i;
        q_ij[global_index] = z_ij[global_index]*u_ij[global_index];
    }
}
```

Dieser Kernel berechnet die Multiplikation aus den Elementen der vorgegebenen Matrizen z_{ij} und u_{ij} und speichert das Ergebnis in der Matrix q_{ij}. Die Problemgröße ist n = 1000000 und g = 8.

Aus den drei Zugriffen auf den globalen Speicher erhalten wir eine Speichertransferrate von $73.974\ Gbytes/s$.

Die Leistung des Kernel beträgt $9.246\ GFlops/s$, da er drei Rechenoperationen enthält.

4.11 SMX Kernel

```
template <typename T>
        __global__ void __SMX(int n, int p,
                               const T * q_ij,
                               const T inv_C_j,
                               const int j,
                               const T * X,
                               const T * mu_j,
                               T * M)
{
    int r = blockIdx.y * blockDim.x + threadIdx.x;
    int c = blockIdx.x;


    if(r < n && c < p){
        int global_index = c*n+r;
        M[global_index] = (X[global_index]-mu_j[c]) *
      sqrt(q_ij[j*n+r]*inv_C_j);
    }
}
```

Der Kernel übernimmt die Aufgabe der Berechnung der Gleichung (3.65)

$$M = \sqrt{\frac{q_{ij}}{f_j}} \cdot \left(\vec{x}_i - \vec{\mu}_j\right)$$

Bei diesen Test beträgt die Problemgröße n = 1000000 und p = 64.

Der Kernel besitzt drei Zugriffe auf den globalen Speicher. Dies führt zu einer Speichertransferrate von $113.894\ Gbytes/s$.

Des Weiteren führen die sechs normalen Rechenoperationen zu einer Rechenleistung von $28.473\ GFlops/s$. Die Spezialoperation, nämlich das Wurzelziehen, führt zu einer Kernelleistung von $4.745\ GFlops/s$.

4.12 Zusammenfassung der Leistungsergebnisse

Die auf einer Tesla C2070 gemessenen Speichertransferraten und Leistungen der implementierten Kernel sind in Abb. 13 zusammengefasst. Dabei sollte man beachten, dass die maximal erreichbare Bandbreite der Testhardware mit aktivierter Speicherfehlerkorrektur $120\ Gbytes/s$ und die maximale Rechenleistung für doppelte Genauigkeit $515\ GFlobs/s$ betragen[14].

Die Speichertransferrate des Matrix-Matrix Skalar Kernel wurde im Diagramm nicht betrachtet, da es durch die dort auftretenden Cache-Effekte nicht zu einer repräsentativen Messung der Transferrate kommen konnte.

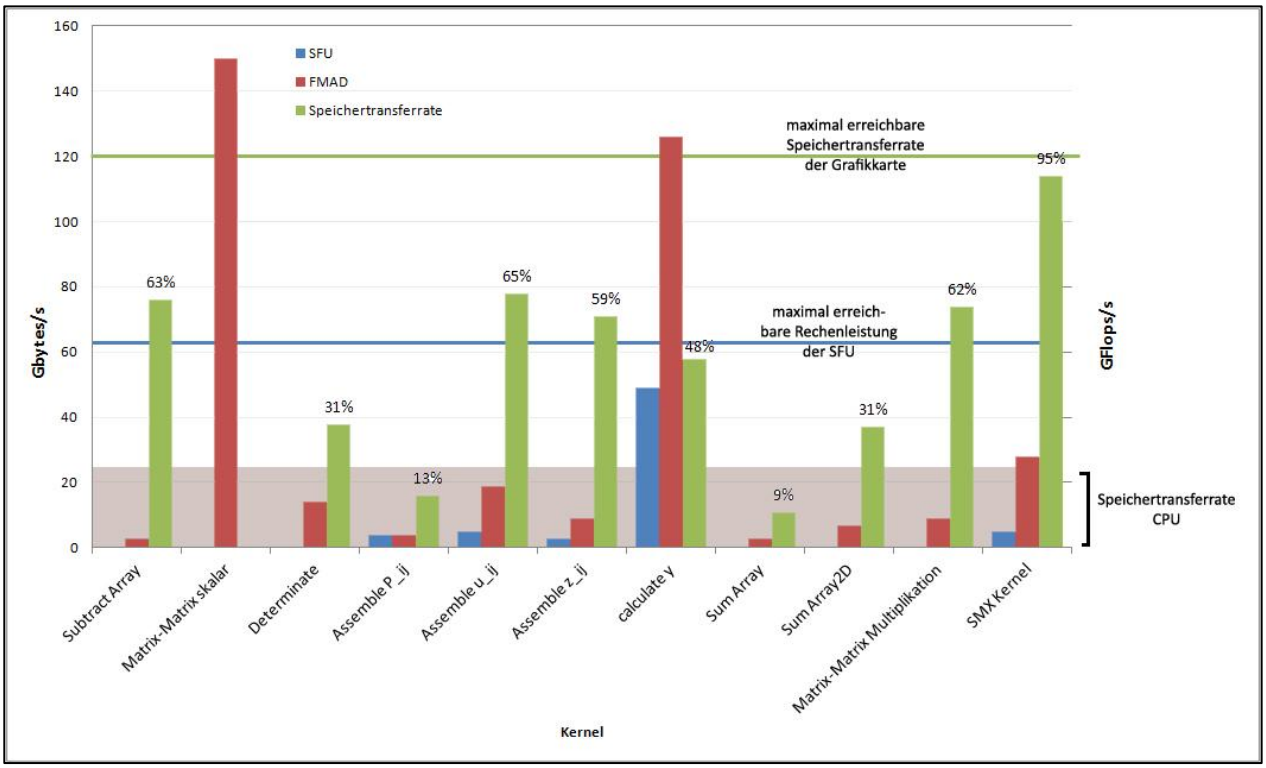

Abbildung 13: Vergleich Speichertransferrate zu Rechenleistungen

In Rot sind die Rechenleistungen, die die Kernel mit den normalen Rechenoperationen erreichen, in Blau die Leistungen der Spezialfunktionen, die „special function units“ (SFUs) zur Berechnung benötigen und in Grün die erreichten Speichertransferraten der Kernel.

Kernel, die meherere Sepicherzugriffe auf dem globalen Speicher benötigen, erreichen eine niedrigere Rechenleistung im Vergleich zu Kernel, die einfachere Berechnungen durchführen.

Hierbei erkennt man, dass für performancekritische Berechnungen, die entweder für große Matrizen durchgeführt werden, wie Subtract Array, Matrix-Matrix Multiplikation oder der SMX Kernel, oder die oft durchgeführt werden, wie die Berechnung der u_{ij} beziehungsweise der z_{ij} Matrix, die maximal erreichbare Bandbreite der Testhardware gut angenähert werden.

Die Berechnungen der Determinante oder auch das Aufsummieren eines Vektors gelten nicht als performancekritisch. Somit sind die erreichten Speichertransferraten für diese Kernel ausreichend. Sie sind mit CUDA implementiert, um einen Datentausch zwischen CPU und GPU während der Berechnung der Einzelteile des Algorithmus zu vermeiden.

Aus diesen Daten können wir ablesen, dass eine CUDA-Implementierung des Algorithmus sich lohnt, da die theoretisch zur Verfügung stehende Speichertransferrate der Grafikkarte oftmals erreicht wird, welche üblicherweise 5-10 mal größer als die aktueller Prozessoren ist (vgl. Tabelle 1). Die Rechenleistungen der einzelnen Kernel sind in derselben Abbildung zu betrachten.

In der folgenden Tabelle sind die Laufzeiten der Kernel ausgeführt, die dazu verwendet wurden, um die Speichertransferraten und Rechenleistungen zu bestimmen.

Tabelle 2: Kernel Laufzeiten in Millisekunde

Kernel	Zeit in Millisekunde
Subtract Array	1.24
Matrix-Matrix Skalar	10
Determinate	0.05
Assemble P_ij	0.96
Assemble u_ij	0.19
Assemble z_ij	0.32
Calculate y	0.38
Sum Array	0.07
Sum Array2D	0.41
Matrix-Matrix Multiplikation	0.30
SMX	1.26

5 Anwendung auf das Spike-Sortingproblem

Führen wir die Simulation mit 25000 Waveforms, die aus drei verschiedenen Spikes produziert werden (vgl. Abb. 14) aus, erhalten wir eine Clusterung dieser Formen. Durch die Projektion dieser Waveforms auf die ersten zwei Hauptkomponenten (vgl. Abb. 15) werden zwei Cluster erkennbar. Somit wird deutlich, dass die Verwendung der Hauptkomponentenanalyse (vgl. Abschn. 2.2) einen verhältnismäßigen Vorteil mit sich bringt, denn die Daten sind durch die Anwendung der PCA vorsortiert.

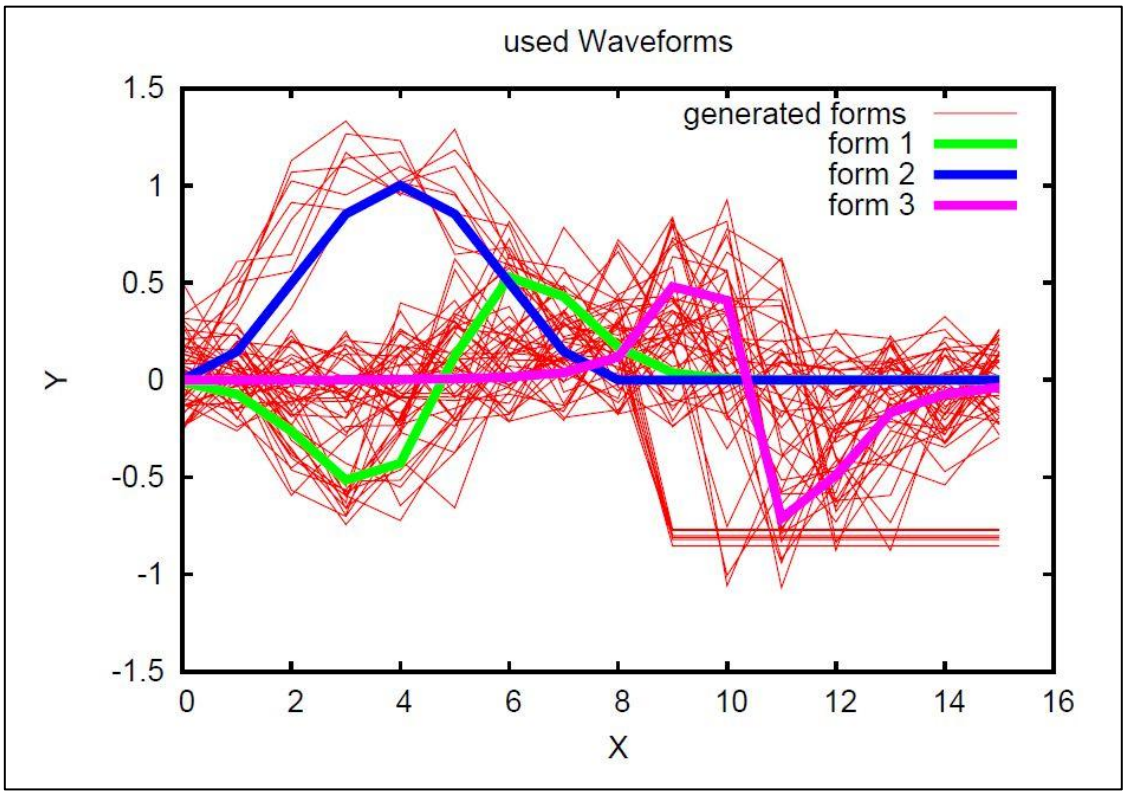

Abbildung 14: eine Auswahl an Waveforms, die für die Simulation eingesetzt wurden

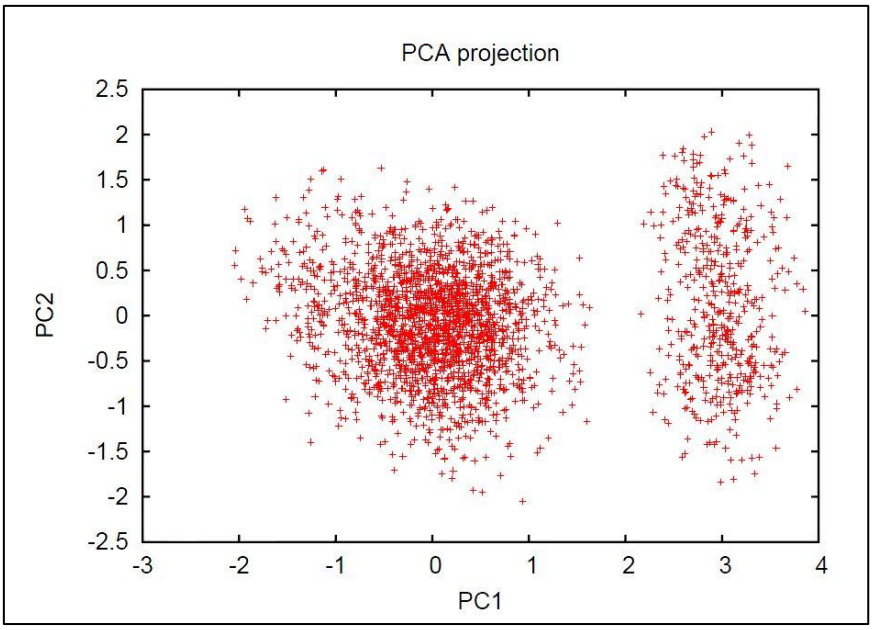

Abbildung 15 : Projektion der Waveforms auf die ersten zwei Hauptkomponenten

Die statistische Korrektheit des Algorithmus können wir überprüfen, indem wir die Verteilung der Mahalanobis Quadrat Abstände der Datenpunkte betrachten (vgl. Abb. 16). Dabei können wir erkennen, dass die erwartete Verteilung, nämlich die Beta-Verteilung nach Anpassung der Verteilungsparameter p und q [1], die empirisch gemessenen Daten vergleichsweise angemessen wiedergibt.

Dabei verwenden wir folgende Wahrscheinlichkeitsdichtefunktion:

$$f(x) = \frac{1}{B(p,q)} x^{p-1}(1-x)^{q-1} \quad \text{mit } p = 6.28 \text{ und } q = 14.07 \tag{5.69}$$

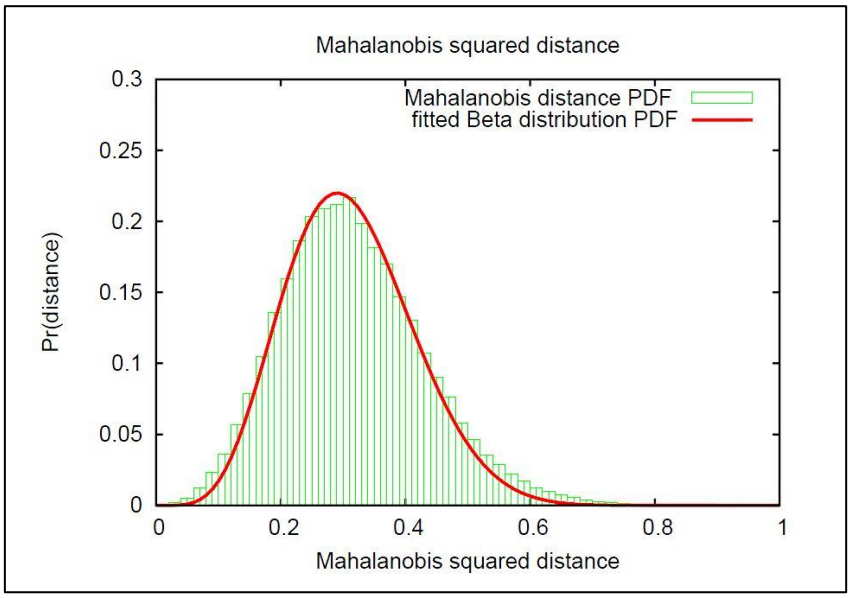

Abbildung 16 : vergleich der erwarteten Beta-verteilung mit der eigentlich beobachteten Verteilung der Mahalanobis Distanzen.

6 Fazit und Ausblick

Im Laufe dieser Arbeit haben wir eine Applikation implementiert, die es erlaubt eine effiziente Clusteranalyse auf einem großen Datensatz zu betreiben. Diese Applikation baut auf dem Shoham-Algorithmus [23] auf, der als Basis die t-Verteilung anwendet, anstelle der allgemein genutzten Gauß-Verteilung. Dabei haben wir gezeigt, dass die additiv verrauschten Daten, t-verteilt sind und von einer Beta-Verteilung besser wiedergegeben werden (vgl. Abschn. 3.7), womit sich die Theorie von Peel und McLachlan [26] bestätigt.

Weiterhin haben wir in Abschnitt 3.7 gezeigt, dass durch das Ausführen eine Hauptkomponentenanalyse die Aufgabe der Clusteranalyse erleichtert werden kann, denn durch die PCA werden die Daten vorgeordnet und somit auch Startmittelpunkte für die Clusteranalyse erschaffen.

Darüber hinaus haben wir ein weiteres Indiz dafür gezeigt, dass eine CUDA-Implementierung von Clustersuchalgorithmen an dem Beispiel des Shoham-Algorithmus [23] einen wesentlichen Vorteil mit sich bringt. Der Algorithmus musste in seine Einzelteile zerlegt und in einzelne Funktionen beziehungsweise Kernel implementiert werden. Dies bringt einen höheren Programmieraufwand mit sich. Dieser Aufwand wird weiterhin durch die Schwierigkeiten des CUDA Programmiermodells verstärkt. Eine Antwort auf die Frage, ob sich der Aufwand lohnt, erhalten wir in Kapitel 4 (vgl. Abschn. 4.12). Die dort gemessenen Speichertransferraten der selbst implementierten Kernel überschreiten die maximal erreichbaren Transferraten der heute verfügbaren CPUs und demonstrieren eindrucksvoll somit den Performancegewinn durch die Parallelisierung mit CUDA.

Nicht alle in dieser Arbeit implementierten Kernel erreichen die maximal erreichbaren Speichertransferraten der Grafikkarte, zum Beispiel der „Assemble P_ij“ Kernel, der gerademal 13% der verfügbaren Transferrate erreicht. Eine weitergehende Aufgabe ist es, diese Kernel zu optimieren und somit die gesamte Simulation noch weiter zu beschleunigen. Weiterhin soll der Algorithmus noch weiter ausgebaut werden, sodass er mit experimentellen Daten arbeiten kann. Die bis jetzt verwendeten Daten werden synthetisch aus Modellprofilen für die Spike-Formen generiert, um die Funktionsweise und Korrektheit der Implementierung zu testen (vgl. Abschn. 3.1).

Die hier gewonnenen Erkenntnisse beziehungsweise Ergebnisse der CUDA-Implementierung werden vorraussichtlich in die bald erscheinende Publikation, „Domain-Specific Embedded Language for Principal Component Analysis“ [10], einfließen.

III. Literaturverzeichnis

[1] Abramowitz, M., & Stegun, I. A. (1965). Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables. Dover Publications.

[2] Arthur, D., & Vassilvitskii, S. (2007). Slides for Presentation of Method: Kmeans++ The Advantage of Careful Seeding. *http://theory.stanford.edu/~sergei/slides/BATS-Means.pdf* . Standford University.

[3] Demmel, J. W. (1997). *Applied Numerical Linear Algebra.* SIAM.

[4] Figeirdo, M. A., & Jain, A. K. (2002). Unsupervised Learning of Finite Mixture Models. *IEEE Transactions on pattern analysis and machine intelligence , 24* (3).

[5] Giuroiu, S. (2011). Von http://serban.org: http://serban.org/software/kmeans/ abgerufen

[6] Göttingen, G.-A.-U. (kein Datum). 20. Abgerufen am 07 2011 von LP Universität Göttingen: QR-Zerlegung für Lineare Ausgleichsprobleme: https://lp.uni-goettingen.de/get/text/1029

[7] Hanke-Bourgeois, M. (2002). *Grundlagen der Numerischen Mathematik und des Wissenschaftlichen Rechnens* (1.Auflage Ausg.). B. G. Teubner.

[8] Harris, K. D., Henze, D. A., Csicsvari, J., Hirase, H., & Buzsáki, G. (2000). *Accuracy of Tetrode Spike Separation as Determined by Simultaneous Intracellular and Extracellular Measurements.* Center for Molecular and Behavioral Neuroscience, Rutgers, The State University of New Jersey, Newark, New Jersey.

[9] Joliffe, I. T. (2002). *Principal Component Analysis* (2. Edition ed.). Springer.

[10] Kramer, S., Witt, A., & Schröbsdorf, H. (2011). *Domain-Specific Embedded Language for Principal Component Analysis.*

[11] Lohninger, H. (07 2011). Grundlagen der Statistik. Virtual Institute of Applied Science. Von statistic for you. abgerufen

[12] Mackay, D. (2003). Information Theory, Inference and Learning Algorithms. Cambridge University Press.

[13] MacQueen, J. B. (1967). Some Methods for classification and Analysis of Multivariate Observations. *5th Berkeley Symposium on Mathematical Statistics and Probability*, (S. 281–297). University of California .

[14] Nvidia Corporation. (September 2011). Von Cuda Personal Supercomputing: http://www.nvidia.com/object/personal-supercomputing.html abgerufen

[15] Nvidia Corporation. (03 2011). CUDA C Programming Guide 4.0.

[16] Nvidia Corporation. (September 2007). CUDA CUBLAS Library.

[17] Nvidia Corporation. (2009). NVIDIA's Next Generation CUDA Compute Architecture: Fermi.

[18] Peel, D., & McLachlan, G. J. (2000). *Robust Mixture Modelling Using the t-Distribution.* Department of Mathematics, University of Queensland, Queensland, Australia.

[19] Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. (2007). *Numerical Recipes: The Art of Scientific Computing* (3. Edition Ausg.). Cambridge University Press.

[20] Raja, Y., & Shaogang, G. (1999). (Q. M. Department of Computer Science, Herausgeber) Von Gaussian Mixture Model: http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/RAJA/CV.html abgerufen

[21] Rana, I. K. (2006). From Geometry to Algebra An Indroduction to Linear Algebra.

[22] Rivisk, H. (2007). *Principal Component Analysis (PCA) & NIPALS algorithm.* University of Oslo.

[23] Shoham, S., Fellows, M. R., & Normann, R. A. (2003). Robust, Automatic Spike Sorting Using Mixture of Multivariate t-Distributions. *Journal of Neuroscience Methods* (127), S. 111-122.

[24] Vandevoorde, D., & Josuttis, N. M. (2003). *C++ Templates* (3rd Printing Ausg.). (Addison-Wesley, Hrsg.)

[25] Wallace, C. S., & Freeman, P. R. (1987). Estimation and Inference by Compact Coding. *Journal of the Royal Statistical Society , 49* (3), S. 240-265.

[26] Weisstein, E. W. (2011). Von MathWorld--A Wolfram Web Resource : K-Means Clustering Algorithm: http://mathworld.wolfram.com/K-MeansClusteringAlgorithm.html abgerufen

[27] Wold, S., Geladi, P., Esbensen, K., & Öhman, J. (Januar 1987). Multi-way Principal Components-and PLS-analysis. *Journal of Chemometrics* (voulme 1), S. 41–56.

IV. Anhang A

Programm mit Dokumentation